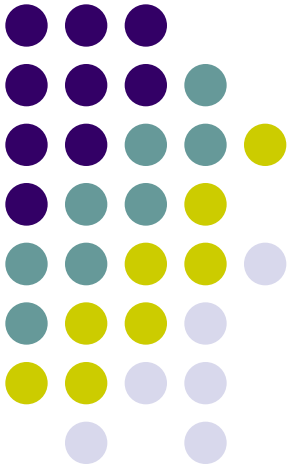
Chapter 8: Serial Communication

EE2405

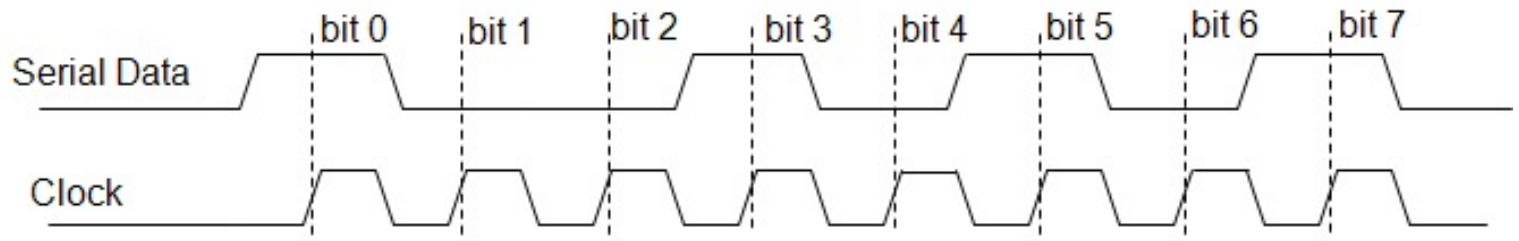
嵌入式系統與實驗

Embedded System Lab



Serial Data Communication

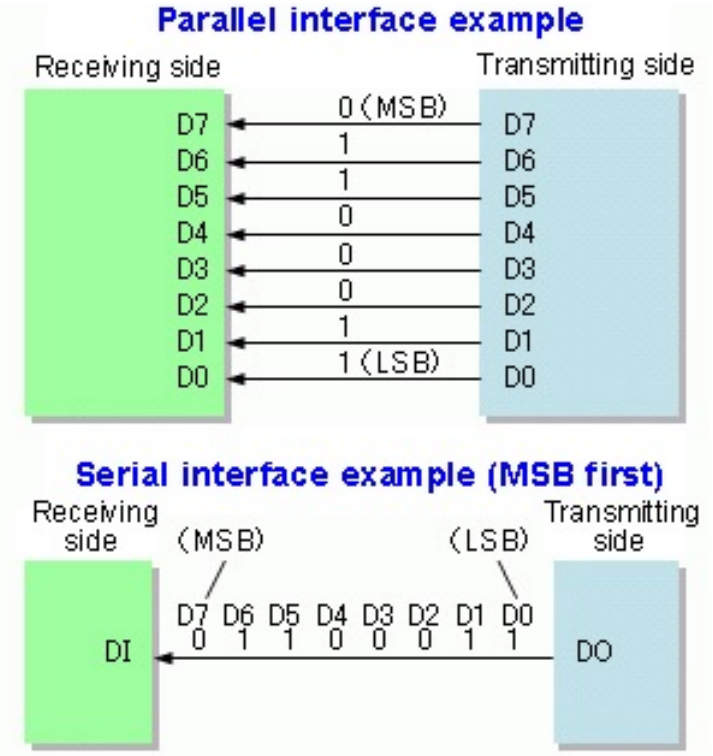
- In serial data transfer, one bit of the data is transmitted at a time, along a single interconnection.
  - Less wire lead to less PCB tracks and IC pins. Also less noise.
- How does the receiver know when each bit begins and ends, and how does it know when each word begins and ends?
  - Synchronization.
- Synchronous serial communication --- The data is synchronized to the clock
  - To send a clock signal alongside the data, with one clock pulse per data bit.



Synchronous serial data communication

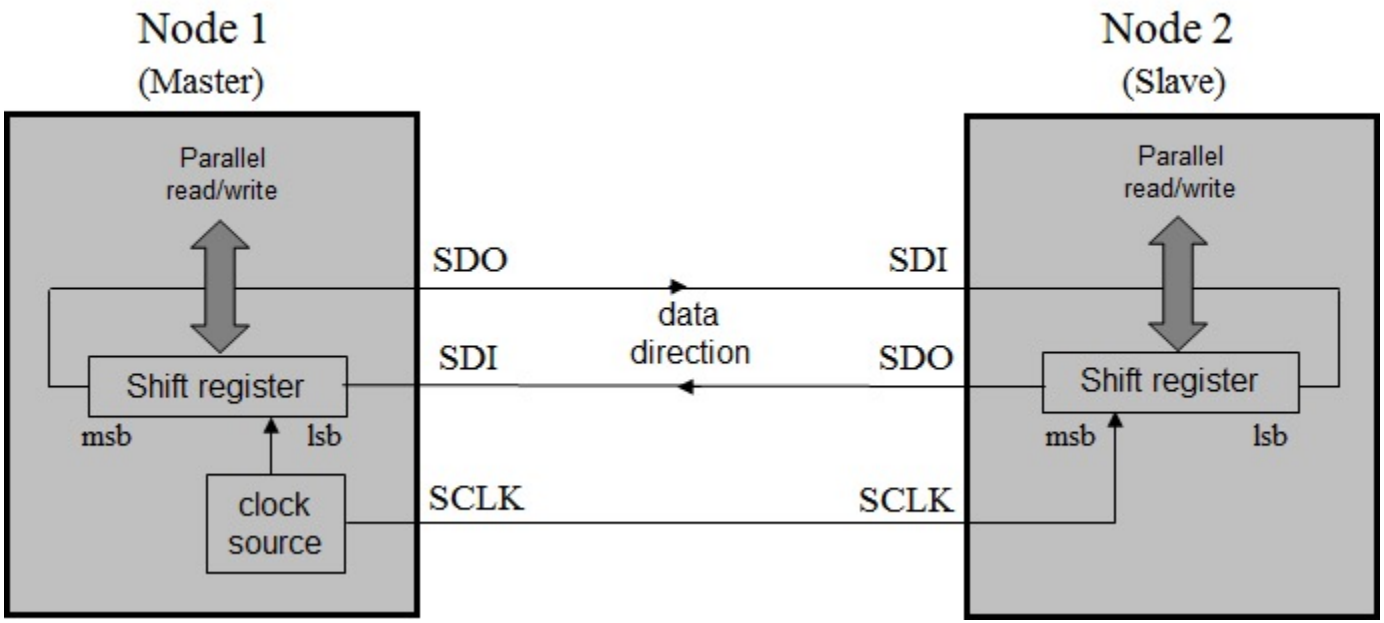
Serial vs. Parallel

- Often serial links can be clocked considerably faster than parallel links to achieve a higher data rate:
  - Less clock skew between different channels
  - Less crosstalk with fewer conductors in proximity.



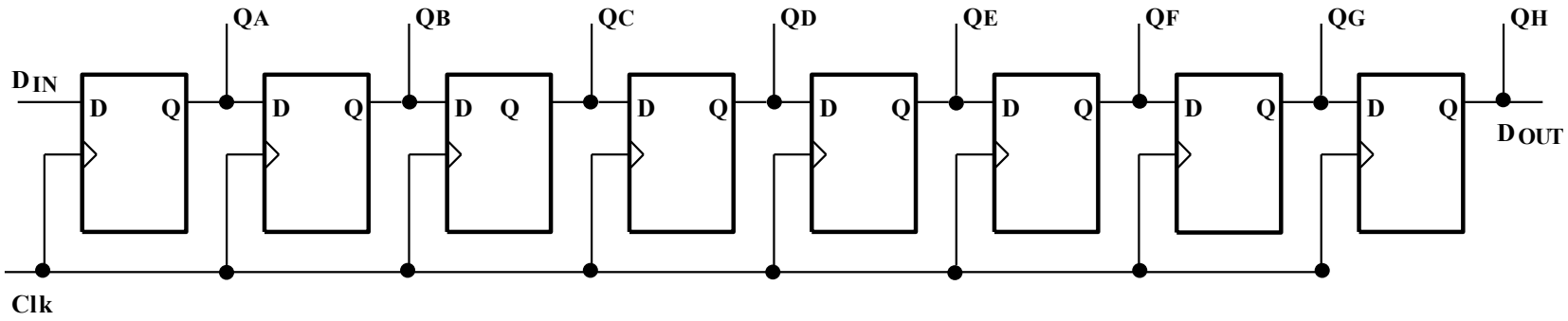
A Simple Serial Link With Clock

- Write from Node 1 to Node 2:
  - Set data at SR at Node 1; Control clock for the length of SR
- For each clock, each flip-flop passes its bit on to its neighbor on one side, and receives a new bit from its other neighbor.
  - Master SDO → Slave SDI
  - Slave SDO → Master SDI



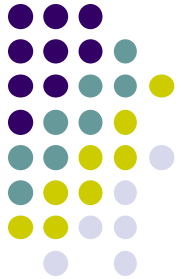
The Shift Register

- An essential feature of most serial links is a shift register. This is made up of a string of flip-flops, connected so that the output of one is connected to the input of the next. Each flip-flop holds one bit of information.



An 8-Bit Shift Register – a Possible Receiver and/or Transmitter of Serial Data

Serial Peripheral Interface (SPI)

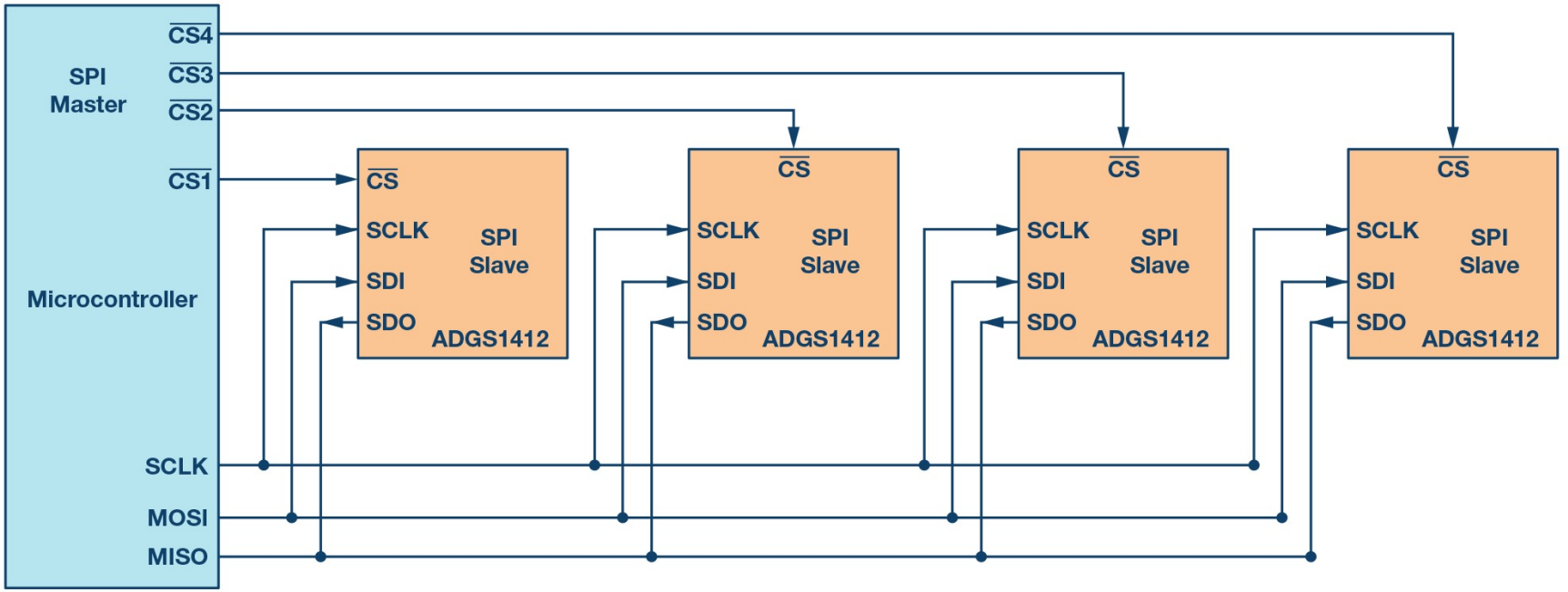


- In the early days of microcontrollers, both National Semiconductors and Motorola started introducing simple serial communication, based on the previous slides.

- Each formulated a set of rules (**protocols on electrical and signals**) which governed how these links worked, and allowed others to develop devices which could interface correctly.

- These became de facto standards. Motorola called its standard Serial Peripheral Interface (**SPI**), and National Semiconductors called theirs Microwire. They're very similar to each other.

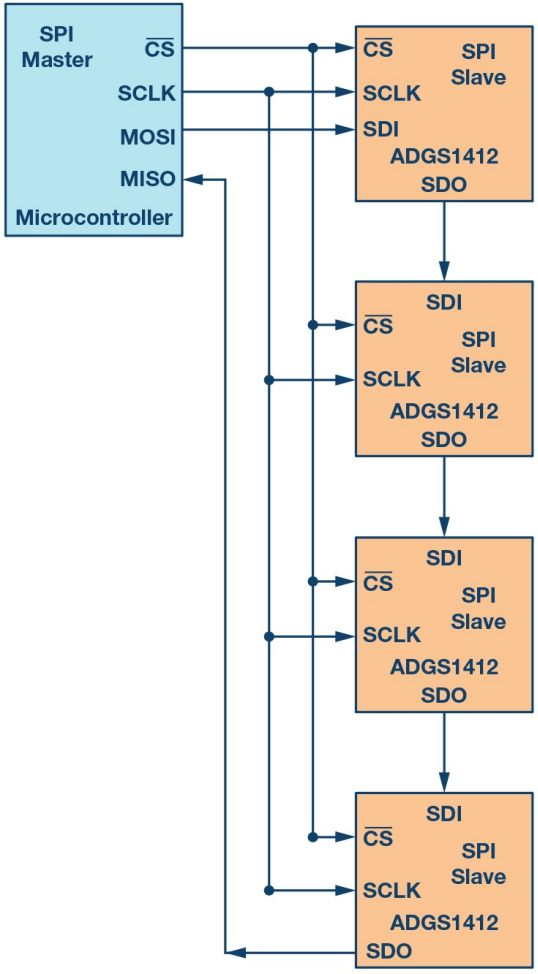
SPI with Parallel CS Signals



SPI interconnections for multiple Slaves

https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html#

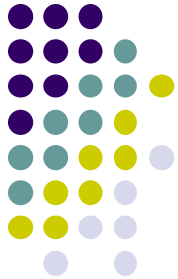
SPI with Daisy Chain Signals



SPI interconnections
for multiple Slaves in daisy chain.

https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html#

SPI mbed API



- The micro-controller has several SPI ports, each can be configured as Master or Slave.
- On the micro-controller, the same pin is used for SDI if in Master mode, or SDO if Slave. Hence this pin gets to be called MISO, **M**aster **i**n, **S**lave **o**ut. Its partner pin is MOSI.

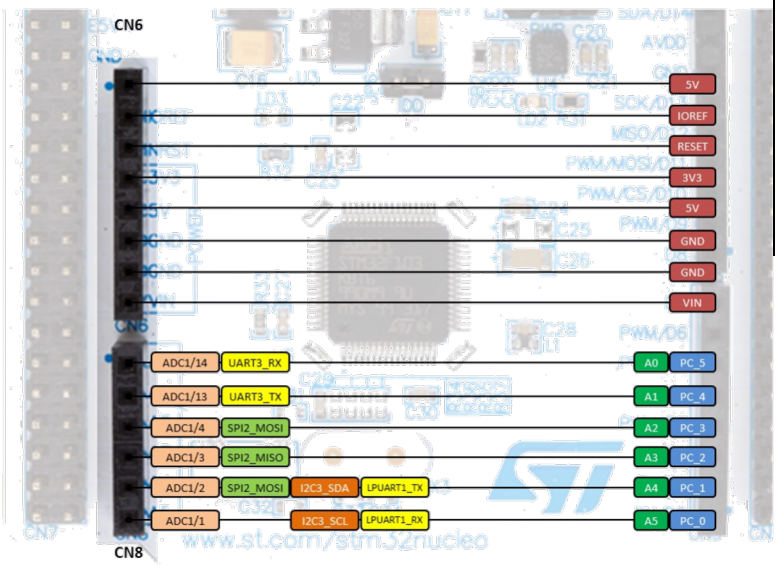
Functions	Usage
SPI	Create a SPI master connected to the specified pins
format	Configure the data transmission mode and data length
frequency	Set the SPI bus clock frequency
write	Write to the SPI Slave and return the response

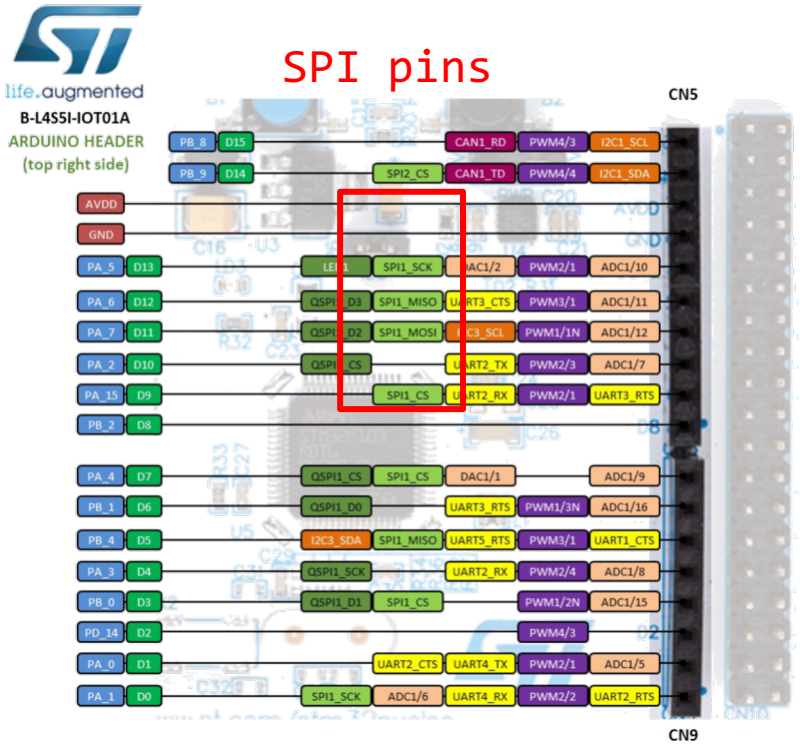
SPI Pins

PinNames.h

```
// SPI signals aliases
SPI_CS   = D10,
SPI_MOSI = D11,
SPI_MISO = D12,
SPI_SCK  = D13,
```

All light-green background labels are SPI pins

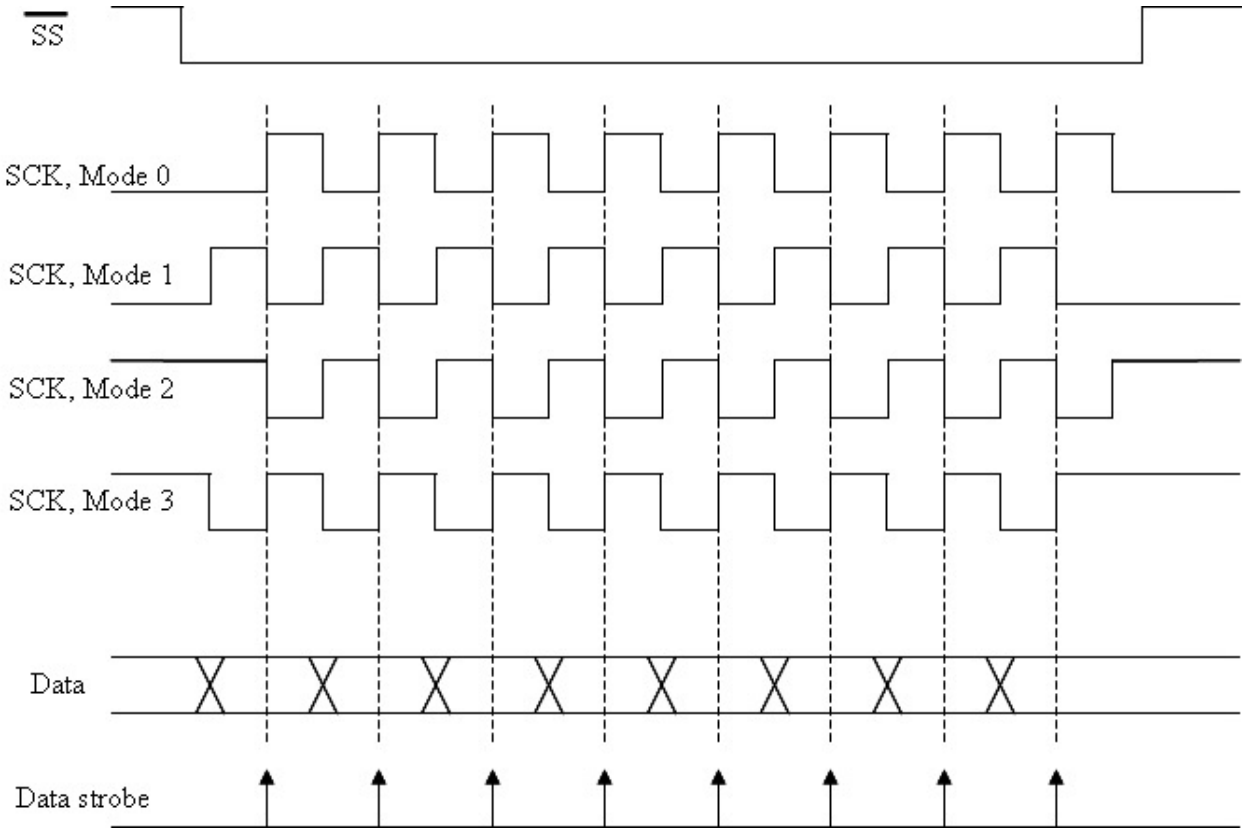




SPI mode

- The mode is a feature of SPI which allows choice of which clock edge is used to clock data into the shift register (indicated as “Data strobe” in the diagram), and whether the clock idles high or low. For most applications the default mode, i.e. Mode 0, is acceptable.

SPI modes

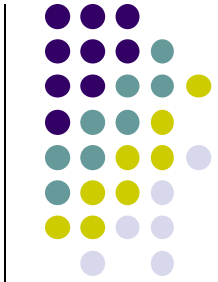


Mode	Polarity	Phase
0	0	0
1	0	1
2	1	0
3	1	1

SPI Slave API

Functions	Usage
SPISlave	Create a SPI slave connected to the specified pins
format	Configure the data transmission format
frequency	Set the SPI bus clock frequency
receive	Polls the SPI to see if data has been received
read	Retrieve data from receive buffer as slave
reply	Fill the transmission buffer with the value to be written out as slave on the next received message from the master.

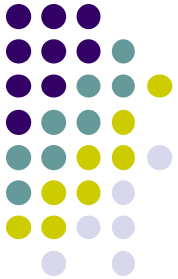
Example of SPI Master and Slave



```
//master definition

SPI spi(D11, D12, D13); // mosi, miso, sclk
DigitalOut cs(D9);

//Slave definition

SPISlave device(PD_4, PD_3, PD_1, PD_0); //mosi,
miso, sclk, cs; PMOD pins
```

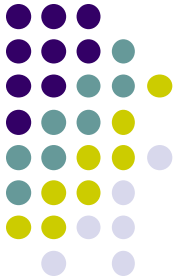
- Note that “Chip Select” pin can be any digital output pin from a SPI master.
- We use the pins on PMODs for SPI2 slave devices.
- Master ”mosi” pin is connected to slave “mosi” pin.
  - `D11 to PD_4`

SPI Master



```
cs = 1;                              // Deselect the device
ThisThread::sleep_for(100ms); //Wait for debug print
printf("First response from slave = %d\n", response);

// Select the device by setting chip select low
cs = 0;
printf("Send number = %d\n", number);

spi.write(number); //Send number to slave
ThisThread::sleep_for(100ms); //Wait for debug print

response = spi.write(number); //Read slave reply
ThisThread::sleep_for(100ms); //Wait for debug print
printf("Second response from slave = %d\n", response);
cs = 1; // Deselect the device
```

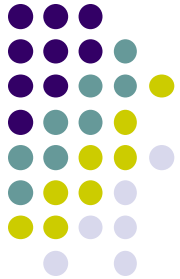
SPI Slave



```
v = device.read(); // Read another byte from master
printf("Second Read from master: v = %d\n", v);
v = v + 10;
device.reply(v); // Make this the next reply
v = device.read(); // Read again to allow master read back
```

- The first read() gets from master an integer (by a master write())
- After slave reply(), the master should write() again to read back the replied data and slave has to read() the written data to clear it.

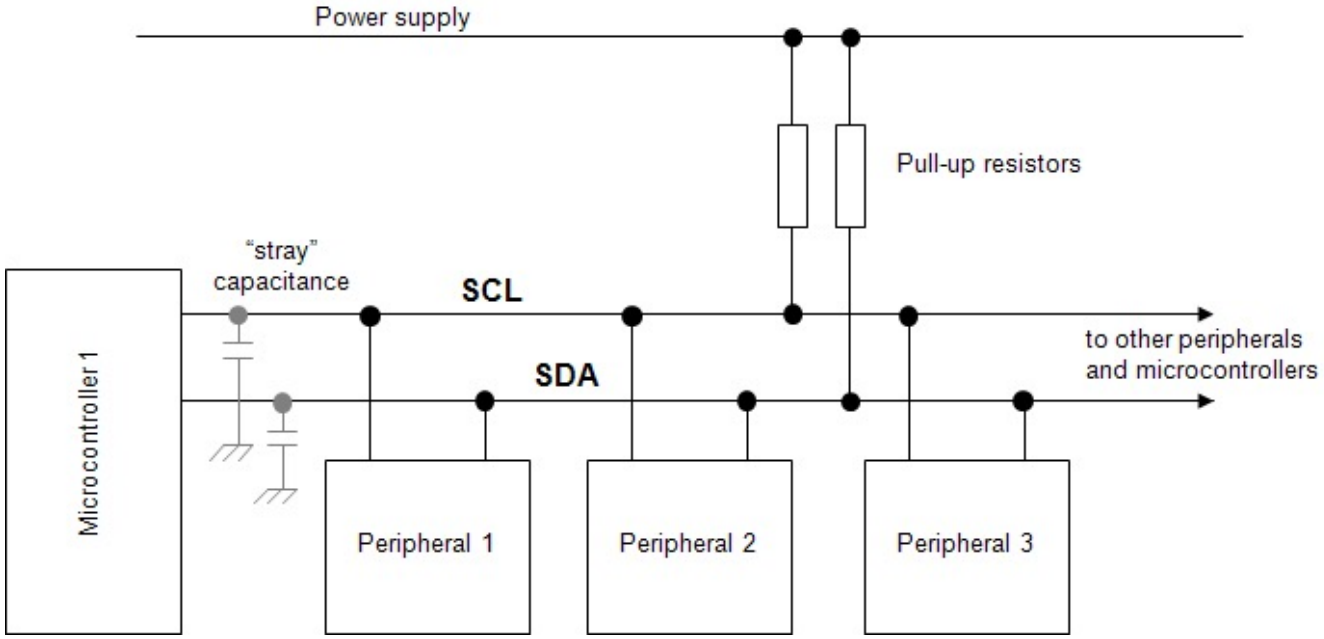
SPI Comments



- The SPI is simple and therefore cheap.
- There are disadvantages.
  - There is **no acknowledgement** from the receiver, so in a simple system the Master cannot be sure that data has been received.
  - There is **no addressing**. In a system where there are multiple slaves, a separate CS (chip select) line must be run to each Slave.
    - Even in daisy chain mode, we still need at least one CS signal to route to all slaves. All slaves operate in serial.
  - There is **no error-checking**. Suppose some electromagnetic interference was experienced in a long data link, data or clock would be corrupted, but the system would have no way of detecting this, or correcting for it.

I2C

- **I**nter-**I**ntegrated **C**ircuit bus.
- Use two physical wires: serial data (SDA) and serial clock (SCL).
  - This means that data only travels in one direction at a time.
- Any node can only pull down the SCL or SDA line to Logic 0; it cannot force the line up to Logic 1. This role is played by a single pull-up resistor connected to each line.

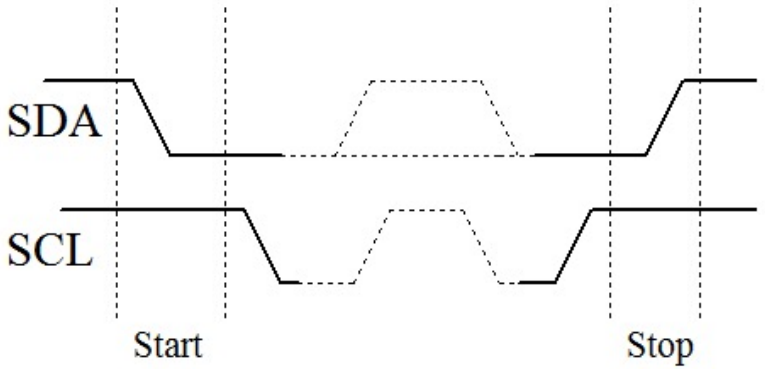


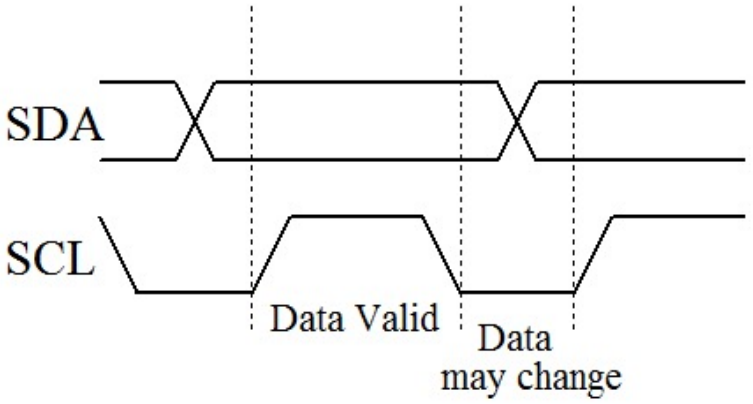
I2C Communications

- The device that initiates communication is termed the ‘master’.
- A device being addressed by the master is called a ‘slave’.

- A data transfer is started by the master signaling 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

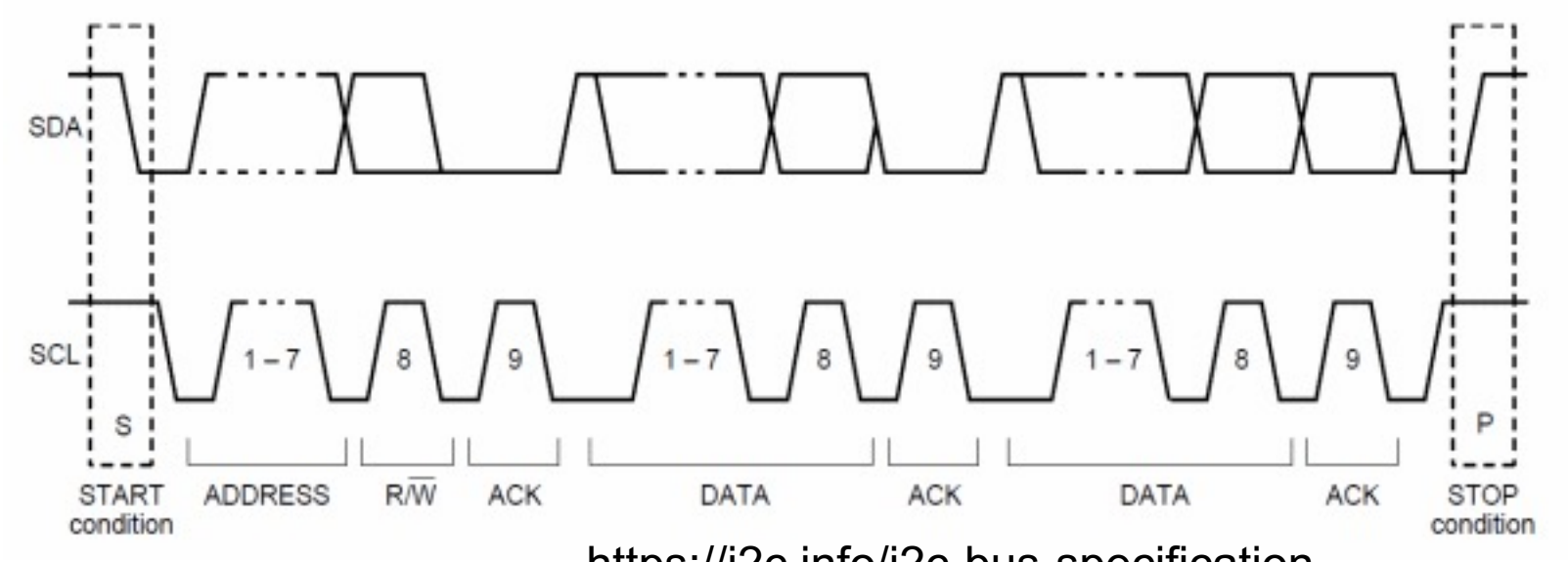
- One SCL clock pulse is generated for each SDA data bit, and data may only change when the clock is low.





I2C Data Transfer Protocol

- I2C has a built-in addressing scheme.
  - Each slave has a predefined address. They monitor the bus and respond only to data and commands associate with their own address.
- 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.



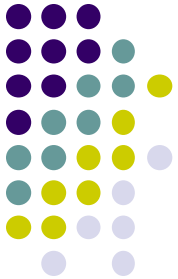
https://i2c.info/i2c-bus-specification

I2C Master and Slave API

Functions	Usage
I2C	Create an I²C Master interface, connected to the specified pins
frequency	Set the frequency of the I²C interface
read	Read from an I²C slave
write	Write to an I²C slave
start	Creates a start condition on the I²C bus
stop	Creates a stop condition on the I²C bus

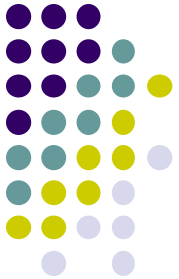
Functions	Usage
I2CSlave	Create an I²C Slave interface, connected to the specified pins
frequency	Set the frequency of the I²C interface
receive	Check to see if this slave has been addressed
read	Read from an I²C master
write	Write to an I²C master
address	Set the the I²C slave address
stop	Reset the slave to a known ready receiving state

I2C Master Example



```
#include "mbed.h"

I2C m_i2c(D14, D15);
char m_addr = 0x90;
int main()
{
    while (1)
    {
        const char tempRegAddr = 0x00;

        m_i2c.write(m_addr, &tempRegAddr, 1);
        //Set pointer to the temperature register

        char reg[2] = {0, 0};
        m_i2c.read(m_addr, reg, 2); //Read

        unsigned short res = (reg[0] << 4) | (reg[1] >> 4);
        float temp = (float)((float)res * 0.0625);
        printf("Temp code=(%d, %d)\r\n", reg[0], reg[1]);
        printf("Temp = %f.\r\n", temp);
        ThisThread::sleep_for(1s);
    }
}
```

I2C Slave Example



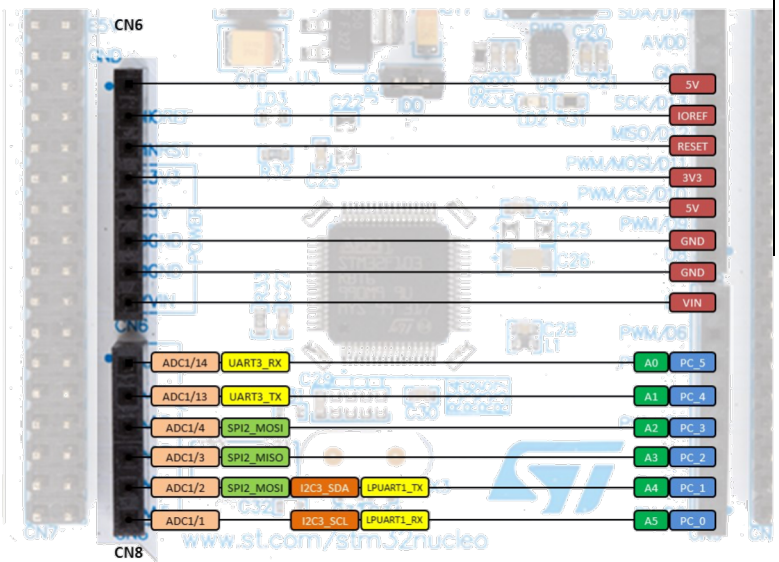
```
I2CSlave slave(D14, D15);

int main() {
    char buf[10];
    char msg[] = "Slave!";

    slave.address(0xA0);
    while (1) {
        int i = slave.receive();
        switch (i) {
            case I2CSlave::ReadAddressed:
                slave.write(msg, strlen(msg) + 1); // Includes null char
                break;
            case I2CSlave::WriteGeneral:
                slave.read(buf, 10);
                printf("Read G: %s\n", buf);
                break;
            case I2CSlave::WriteAddressed:
                slave.read(buf, 10);
                printf("Read A: %s\n", buf);
                break;
        }
        for(int i = 0; i < 10; i++) buf[i] = 0;    // Clear buffer
    }
}
```

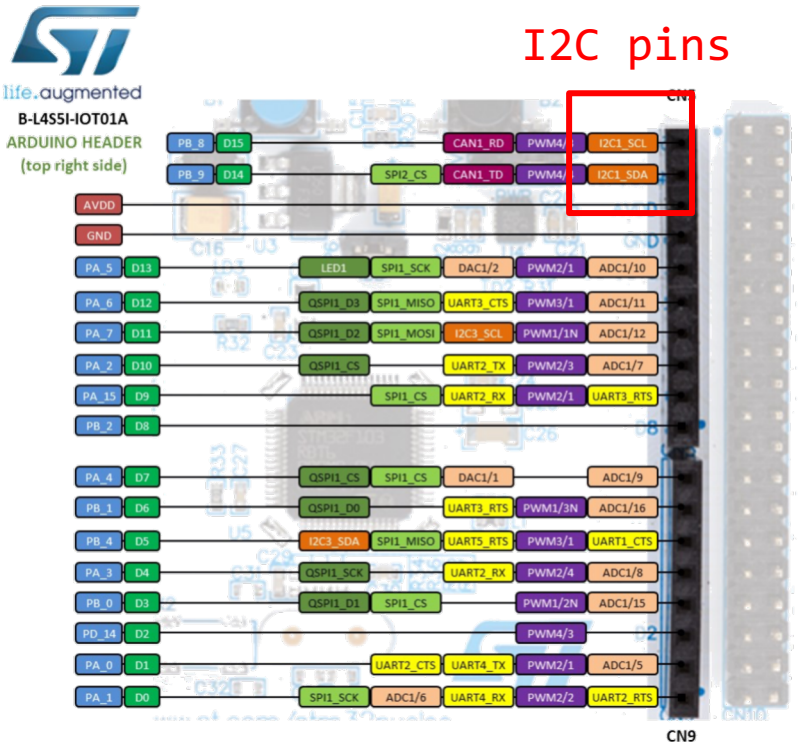
I2C Pins

PinNames.h

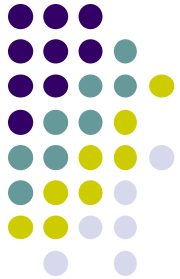
```
// I2C signals aliases
   I2C_SDA = D14,
   I2C_SCL = D15,
```

All orange background labels are I2C pins



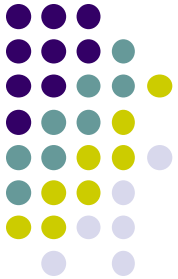


I2C Comments



- An extra (clock) line needs to go to every data node.
- The bandwidth needed for the clock is always twice the bandwidth needed for the data; therefore, it is the demands of the clock which limit the overall data rate.
- Over long distances, clock and data themselves could lose synchronization.
- No error checking.

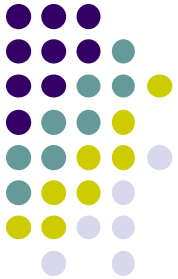
TextLCD I2C main.cpp



```
#include "LCD.h"

int main()
{
    LCD_init();             // call the initialise function
    display_to_LCD(0x48); // ‘H’
    display_to_LCD(0x45); // ‘E’
    display_to_LCD(0x4C); // ‘L’
    display_to_LCD(0x4C); // ‘L’
    display_to_LCD(0x4F); // ‘O’
    for (char x = 0x30; x <= 0x39; x++)
    {
        display_to_LCD(x); // display numbers 0–9
    }
}
```

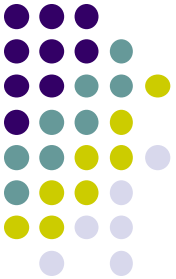
This is the same code as the non-i2c version.

TextLCD I2C lcd.cpp
display_to_LCD()



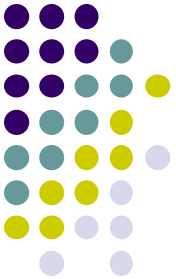
```
#include "LCD.h"

I2C _i2c(D14, D15);
char _slaveAddress = 0x4E;

void display_to_LCD(int value)
{
    char bus_data = 0;
    bus_data |= LCD_BUS_I2C_RS; // Set RS bit
    _i2c.write(_slaveAddress, &bus_data, 1);
    ThisThread::sleep_for(1us);

    _writeByte(value);
    ThisThread::sleep_for(40us);
}
```

TextLCD I2C lcd.cpp
LCD_init(void)



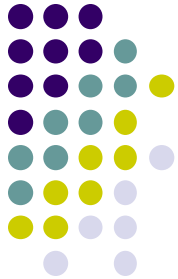
```
void LCD_init(void)
{
    _i2c.frequency(100000);
    ThisThread::sleep_for(20ms);

    _writeCommand(0x02); // Controller is now in 4-bit mode
    _writeCommand(0x28); // Function set 001 DL N F - -
                         //  DL(Data Length)=0 (4 bits bus)
                         //  N=1 (2 lines)
                         //  F=0 (5x7 dots font, only option for 2 line)
                         //  -  (Don't care)

    ThisThread::sleep_for(10ms);

    // display mode
    _writeCommand(0x0F); // display on, cursor on,  blink on
    //Display Ctrl 0000 1 D C B

    _writeCommand(0x01); // cls, and set cursor to 0
    ThisThread::sleep_for(20ms);
}
```

TextLCD I2C lcd.cpp _setDataBits()



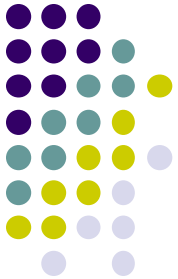
```
void _setDataBits(char &bus_data, int value){
    //Clear all databits
    bus_data &= ~LCD_BUS_I2C_MSK;

    // Set 4 data bits
    if (value & 0x01){
        bus_data |= LCD_BUS_I2C_D4;
    } // Set D4 Databit

    if (value & 0x02){
        bus_data |= LCD_BUS_I2C_D5;
    } // Set D5 Databit

    if (value & 0x04){
        bus_data |= LCD_BUS_I2C_D6;
    } // Set D6 Databit

    if (value & 0x08){
        bus_data |= LCD_BUS_I2C_D7;
    } // Set D7 Databit
}
```

TextLCD I2C lcd.cpp _writeByte(int value)



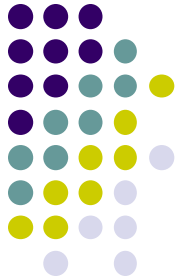
```
void _writeByte(int value)
{
    char data[4] = "";
    data[0] |= LCD_BUS_I2C_E;           // Set E bit
    _setDataBits(data[0], value >> 4); // set data with 4 MSB bits

    data[1] &= ~LCD_BUS_I2C_E; // clear E

    data[2] |= LCD_BUS_I2C_E;     // Set E bit
    _setDataBits(data[2], value); // set data with 4 LSB bits

    data[3] &= ~LCD_BUS_I2C_E; // clear E

    // write the packed data to the I2C port
    _i2c.write(_slaveAddress, data, 4);
}
```

TextLCD I2C lcd.cpp _writeCommand(int)



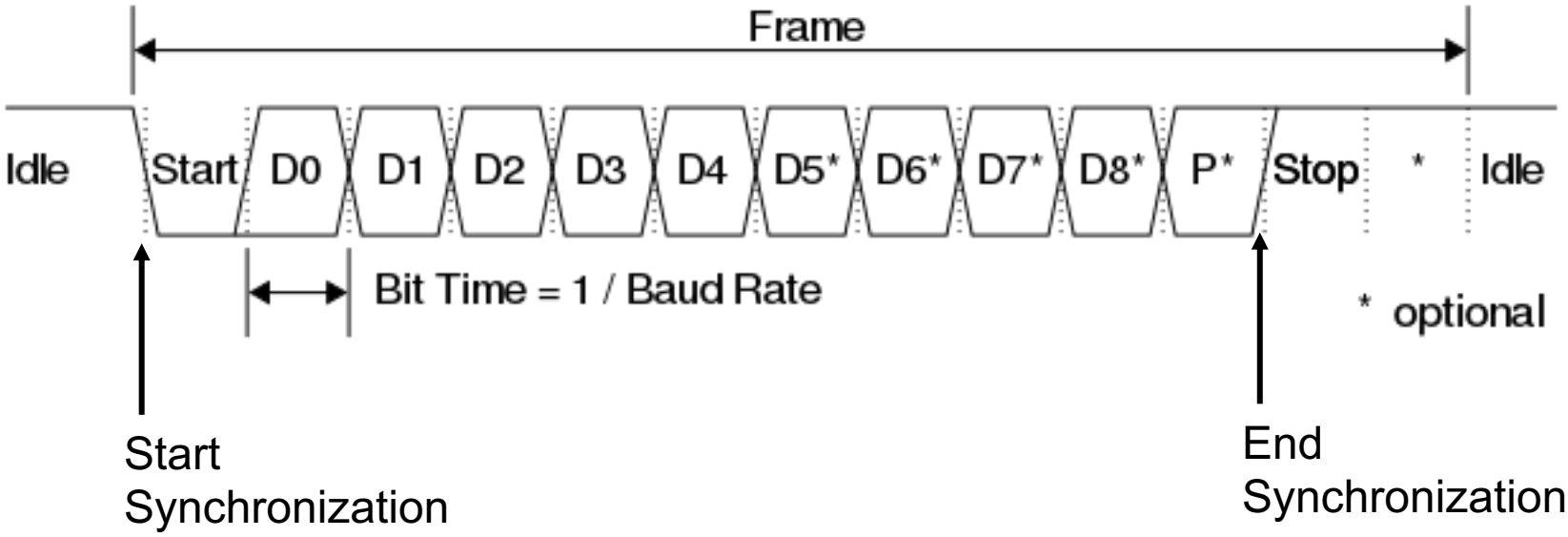
```
void _writeCommand(int command)
{
    char bus_data = 0;
    bus_data &= ~LCD_BUS_I2C_RS; // Reset RS bit
    _i2c.write(_slaveAddress, &bus_data, 1);
    ThisThread::sleep_for(1us);
    _writeByte(command);
    ThisThread::sleep_for(40us); // most instructions take 40us
}
```

Asynchronous Serial Data Communication



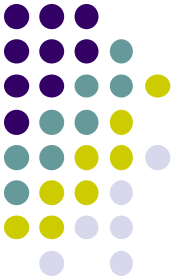
- Asynchronous communication doesn't require the clock to be connected between nodes.
- Data rate (baud rate) is predetermined – both transmitter and receiver are pre-set to recognize the same data rate. Hence each node needs an accurate and stable clock source, from which the data rate (internal clock) can be generated.
- Each byte or word is framed with a Start and Stop bit. These allow synchronization to be initiated before the data starts to flow.
- After "Start" bit is received, a double-rate clock starts to sample data to avoid wrong data capture.
- An asynchronous serial port is generally called a UART, Universal Asynchronous Receiver/ Transmitter. A UART has one connection for transmitted data, called TX, and another for received data, called RX.

UART Frame Format



D0, D1, D2, D3, D4, D5, D6, D7, D8 are data bits.
P is parity bit.

mbed BufferedSerial API



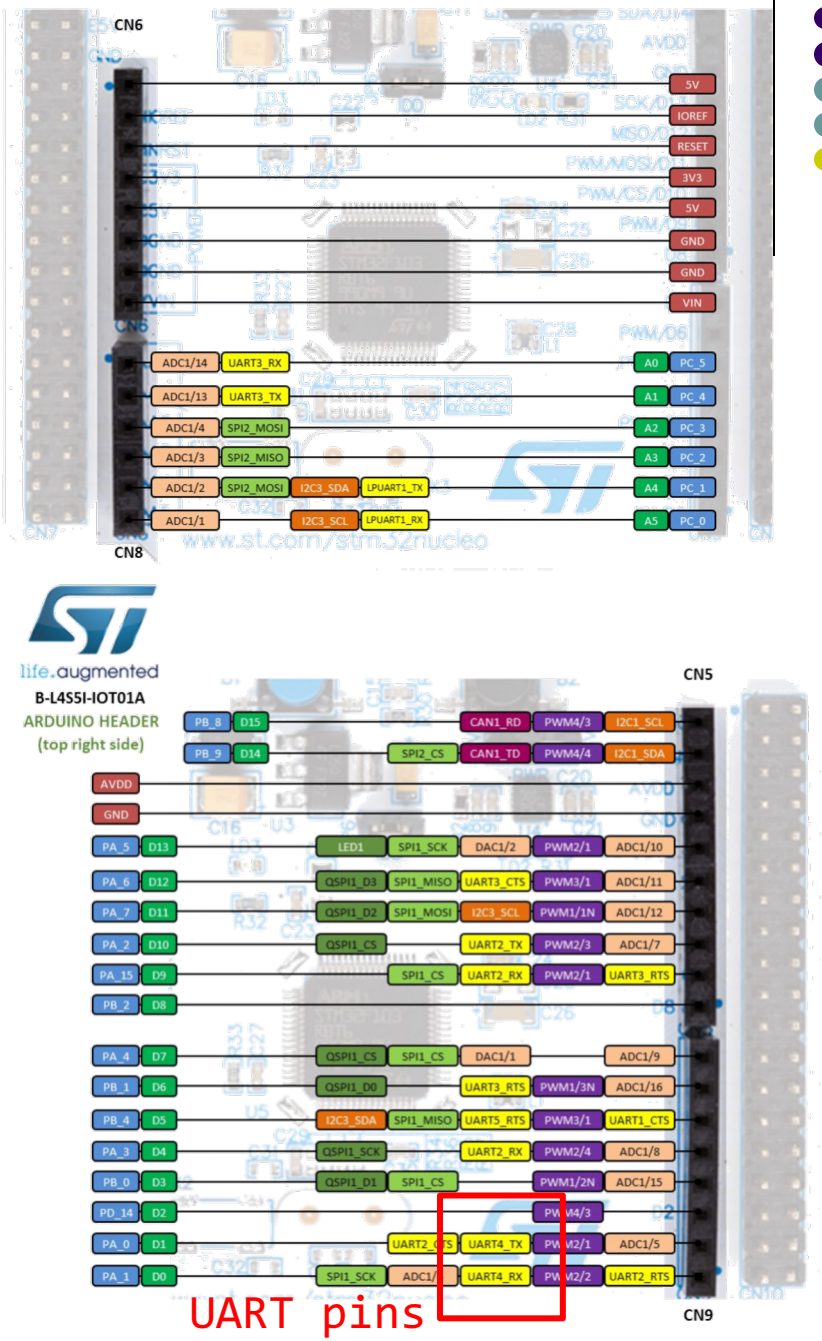
Functions	Usage
BufferedSerial	Create a Serial port, connected to the specified transmit and receive pins with a particular baud rate.
poll	Poll multiplex input/output over a set of file handles at POSIX poll.
read	Read the contents into a buffer with a length.
write	write the contents of a buffer with a length.
readable	Determine if there is a character available to read
writeable	Determine if there is space available to write a character
set_baud	Set the baud rate of the serial port
set_format	Set the transmission format used by the serial port
set_flow_control	Set the flow control type on the serial port: software, CTS/RTS
sigio()	Register a callback on state change of the file.

Note Serial API is deprecated after mbed OS 6.
BufferedSerial supports more API interfaces: filehandle, sigio, poll, etc.

UART Pins

All yellow background labels are UART pins.

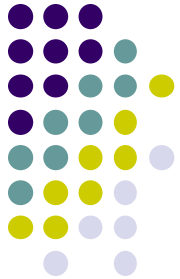
Note there are also RTS/CTS pins, which supports hardware flow control (may be required by some hardware interface).



UART Example

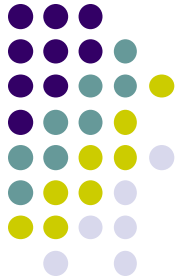
```
#include "mbed.h"

// Maximum number of element the application buffer can contain
#define MAXIMUM_BUFFER_SIZE 32
// Create a DigitalOutput object to toggle an LED whenever data is received.
static DigitalOut led(LED1);
// Create a BufferedSerial object with a default baud rate.
static BufferedSerial serial_port(USBTX, USBRX);

int main(void) {
  // Set desired properties (9600-8-N-1).
  serial_port.set_baud(9600);
  serial_port.set_format(
      /* bits */ 8,
      /* parity */ BufferedSerial::None,
      /* stop bit */ 1);

  // Application buffer to receive the data
  char buf[MAXIMUM_BUFFER_SIZE] = {0};

  while (1) {
    if (uint32_t num = serial_port.read(buf, sizeof(buf))) {
      // Toggle the LED.
      led = !led;
      // Echo the input back to the terminal.
      serial_port.write(buf, num);
    }
  }
}
```

UART with FileHandle Example



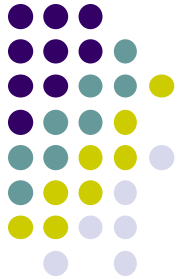
```
#include "mbed.h"

static DigitalOut led2(LED2);

// BufferedSerial derives from FileHandle
static BufferedSerial device(STDIO_UART_TX, STDIO_UART_RX);

int main()
{
  // Once set up, access through the C library
  FILE *devin = fdopen(&device, "r");

  while (1)
  {
    putchar(fgetc(devin));
    led2 = !led2;
  }
}
```

UART with sigio Example



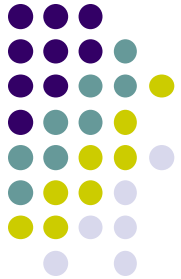
```
#include "mbed.h"
static DigitalOut led1(LED1);
static DigitalOut led2(LED4);
static BufferedSerial device(STDIO_UART_TX, STDIO_UART_RX);
static void callback_ex() {
  // always read until data is exhausted - we may not get another
  // sigio otherwise
  while (1) {
    char c;
    if (device.read(&c, 1) != 1) { break; }
    putchar(c);
    putchar('\n');
    led2 = !led2;
  }
}
int main() {
  // Ensure that device.read() returns -EAGAIN when out of data
  device.set_blocking(false);

  // sigio callback is deferred to event queue, as we cannot in general
  // perform read() calls directly from the sigio() callback.
  device.sigio(mbed_event_queue()->event(callback_ex));
  while (1) {
    led1 = !led1;
    ThisThread::sleep_for(500ms);
  }
}
```

Serial Pass-through between USB and an External UART



```
#include  "mbed.h"
static BufferedSerial pc(STDIO_UART_TX, STDIO_UART_RX);
static BufferedSerial device(D1, D0);
int main()
{
  char c;
  while (1) {
    if (pc.readable()) {
      device.write(pc.read(&c, 1), 1);
      }
    if (device.readable()) {
      pc.write(device.read(&c, 1), 1);
    }
  }
}
```

Chapter Review



- Serial data links provide a simple and economical means of communication between microcontroller and peripherals, and/or between microcontrollers.
- SPI is a simple synchronous standard with serial input, output, clock and enable pins.
  - Limited by the enable pins.
- The I2C has a more sophisticated protocol than SPI
  - I2C runs on a 2-wire bus which includes addressing and acknowledgement in the protocol.
  - I2C allows multi-master configurations.
  - Lack of error correction standards.
- Many sensors communicate through SPI and I2C.
- UART is an asynchronous protocol.
  - We have used an virtual UART communication link with the host computer via USB.
- The USB protocol is even more versatile, yet it requires complex protocols on both hardware and software drivers.