EL6483: Communication Interfaces in Embedded Systems (e.g., to interface with sensors and actuators or between microcontrollers)
Variety of digital interfaces

- GPIO pins (General Purpose Input/Output)
  - GPIO pins enable reading and writing of raw bits
  - Hardware pins on microcontrollers are often multiplexed (e.g., the same hardware pin can function either as a GPIO or as an I2C data pin, depending on initialization/configuration in software)
    - enable *alternate functions* to provide higher-level functionalities than reading and writing raw bits

- Various serial communication interfaces: UART, SPI, I2C, etc.

- Various communication protocols on top of these interfaces: e.g., register formats, sequences of bytes being read/written
Serial Communication interfaces

- Examples:
  - Universal Asynchronous Receiver/Transmitter (UART): receive and transmit some number of bits (e.g., 8 bits, i.e., 1 byte) at a time; start bit and stop bit(s), optional parity bit
  - I2C (Inter-Integrated Circuit): multiple master devices and slave devices are possible
  - SPI (Serial Peripheral Interface): multiple slave devices are possible; MOSI (Master Output, Slave Input) and MISO (Master Input, Slave Output) pins
In serial port communication, each data packet contains a start bit, a number of data bits (typically 8), an optional parity bit, and one or more stop bits.

| 1 start bit | data bits (5 to 9 bits) | optional parity bit (1 bit) | 1 or 2 stop bits |

The settings for a serial port communication include:

- the bits per second (baud rate), e.g., 115200 bps (bits per second)
- number of bits per data byte, e.g., 8
- parity setting (i.e., parity bit or no parity bit), e.g., even parity
- number of stop bits, e.g., 1
For example, if the settings for a serial port communication are specified as: 115200 8-N-1, then the baud rate (number of bits per second) is 115200, the number of bits per data byte is 8, there is no parity bit being used, and there is 1 stop bit after every data byte.

A very commonly used setting for baud rate in serial communication is 115200 bps (bits per second). Other commonly used settings for baud rate are 9600 bps, 19200 bps, 38400 bps, 57600 bps, 230400 bps, etc.

The most commonly used number of bits per byte in serial communication is 8.

The number of stop bits is typically 1. A setting of 2 stop bits is also used in some devices.
Serial Port (UART)

- The parity bit, if used, is 0 or 1 to indicate if the data byte has even or odd number of 1’s.
  - Parity bit is a simple type of checksum to try to detect any errors in communication of a data byte.
  - Two types of parity bit are possible: even parity and odd parity.
    - In even parity, if the number of 1’s in the data byte is odd, the parity bit is set to 1 (and 0 otherwise).
    - In odd parity, if the number of 1’s in the data byte is even, the parity bit is set to 1 (and 0 otherwise).
Serial communication typically uses separate transmit (TX) and receive (RX) lines.

- When two devices are connected, the TX of the first device is connected to the RX of the second device and the RX of the first device is connected to the TX of the second device.
- With separate TX and RX lines, serial communication is typically full-duplex, i.e., each of the two devices can transmit data at any time (including simultaneously). In half-duplex communication, only one device can transmit at a time.
Serial Port (UART)

- Two commonly used standards in serial communication are RS-232 and TTL.
  - In RS-232, voltage levels such as ±3.3 V or ±5 V are utilized (with a logic 1 represented by a negative voltage and a logic 0 represented by a positive voltage).
  - In TTL (transistor-transistor logic), voltage levels such as 0 to 5 V are used (e.g., logic 1 represented by +5 V and logic 0 represented by 0 V). Typically, any voltage below around 0.5 V is considered logic 0 and any voltage above around 2.7 V is considered logic 1.
- When connecting two serial devices, it is important to check that they are electrically compatible (e.g., both RS232) and that the serial port settings are the same on both sides of the communication (e.g., both devices set to 9600 8-N-1).
Pull-Up and Pull-Down Resistors

- When connecting from a microcontroller to an external device, you might need pull-up or pull-down resistors (depending on the electrical characteristics of the device and circuit structure).
- A pull-up resistor is a resistor connected between an I/O pin and supply voltage (VCC).
- A pull-down resistor is a resistor connected between an I/O pin and ground.
- One function of pull-up and pull-down resistors is to ensure that voltage levels of I/O pins are well-defined when nothing is connected to them (i.e., not floating).
- Pull-up/pull-down/no-pull can often be configured in software in microcontrollers (e.g., look at GPIO_PuPd_UP, GPIO_PuPd_DOWN, GPIO_PuPd_NOPULL, etc., in stm32f4xx_gpio.h).
I2C (Inter-Integrated Circuit)

- Digital communication protocol that supports multiple masters and multiple slaves
- I2C utilizes a data line and a clock line:
  - SDA: serial data line
  - SCL: serial clock line
- Each device on the I2C bus has an SDA pin and an SCL pin.
  - The SDA pins of all the I2C devices are all connected together (as a common bus); the SCL pins of all the I2C devices are all connected together, i.e., the SDA and SCL lines are shared between all the masters and slaves on the I2C bus.
I2C (Inter-Integrated Circuit)

- Each I2C device has an address.
  - The device address should be unique on the I2C bus (different devices should have different addresses).
  - The address is typically 7 bits. An 8-bit address is formed from the 7-bit address by appending a 1 or 0 depending on whether it is a write operation (0) or read operation (1).
  - For example, if the 7-bit address of a device is 0x50, then the 8-bit address for read would be \((0x50 \ll 1) + 1 = 0xA1\) and the 8-bit address for write would be \((0x50 \ll 1) + 0 = 0xA0\).
  - To enable multiple identical devices (e.g., two identical accelerometer sensors) to be able to operate on the same I2C bus, I2C device manufacturers often provide some configurability of the device address (e.g., enabling one bit of the 7-bit address to be configurable to be 0 or 1 based on connecting a hardware pin to ground or +5 V, etc.).
I2C Communication Protocol

- Typical voltage levels utilized by I2C are +5V and +3.3 V.
- Typical speeds utilized in I2C communication include 100 kbps (standard mode), 400 kbps (fast mode), 1000 kbps (fast mode plus); kbps is “kilo bits per second”
- The master generates the clock (SCL) and starts communication (the communication is always initiated by the master and is always under control of the master).
- The slave device address is used to specify which device is being communicated with. The slave receives the clock signal and responds when addressed by the master (when the address transmitted by the master matches the slave’s address).
I2C Communication Protocol

- There can be multiple masters on the I2C bus. If multiple masters try to communicate at the same time, an arbitration method is used to determine which master should continue communicating and which masters should wait.

- Electrically, the SDA and SCL are open-drain, which means that if one node tries to pull a line up (logic high) and one node tries to pull a line down (logic low), the line will become logic low.

  - One consequence of this design is that the arbitration between multiple masters is very simple. Each master reads the SDA line to check if it corresponds with the value it is trying to write; if it is trying to write 1 and reads 0, it means that another master is writing 0 and, hence, decides to stop its communication until the other master has finished.

  - Another consequence of the open-drain design is that a slave can hold the clock line (SCL) low to get more time to finish processing the bits it has received (or to get time to read a sensor, etc.). This is called clock stretching.
I2C Communication Protocol

- For both read and write operations, the master starts the communication by sending a start bit followed by 8 bits (which is comprised of the 7-bit slave address followed by 0 or 1 to indicate whether the master wants to write to or read from the slave). For example, if the 7-bit address of a device is 0x50, the 8 bits that are transmitted by the master would be 0xA1 for a read operation and 0xA0 for a write operation. The slave responds to the 8-bit byte with an acknowledgement (ACK bit). Only the slave corresponding to the 7-bit address sent by the master will send an acknowledgement.

Master Transmit (transmitting data from master to slave)

- Once the master has sent the start bit followed by 8 bits (7-bit slave address and 0 to indicate write) and the slave has sent an ACK bit, the master sends bytes repeatedly and the slave sends an ACK bit after every byte.

- The master stops the transmission by sending a stop bit or by sending another start bit if it wishes to stop this communication but retain control of the I2C bus (e.g., to write to another slave device or to send a separate message to the same slave device).
I2C Communication Protocol

Master Receive (receiving data from slave to master)

- Once the master has sent the start bit followed by 8 bits (7-bit slave address + 1 to indicate read) and the slave has sent an ACK bit, the slave sends bytes repeatedly and the master sends an ACK bit after every byte except the last one (i.e., slave keeps sending a new byte after receiving an ACK from the master).

- Once the master has received the number of bytes it wished to read, it stops the communication by sending a stop bit or by sending another start bit if it wishes to stop this communication but retain control of the I2C bus for a new communication with the same or different slave device.
Example: Reading/writing registers using I2C

- Utilizing the basic I2C protocol for transmitting and receiving data, a protocol to write to and read from registers on an external I2C-connected device can be defined using, for example, a sequence of reads/writes to indicate the register address and then the data, etc.

- To write to a register on the external device:
  - the master first starts the Master Transmit mode (i.e., sends start bit, sends slave device address with 0, gets ACK from slave)
  - then, the master writes the address of the register it wants to write to
  - after that, the master sends the data to write to the register.

- To read from a register on the external device:
  - the master first starts the Master Transmit mode (i.e., sends start bit, sends slave device address with 0, gets ACK from slave)
  - then, the master writes the address of the register it wants to read from
  - after that, the master switches to the Master Receive mode (i.e., send another start bit, send slave device address with 1, get ACK from slave)
  - then, the slave device sends the data from the specified register.
Example of a Sensor Interface Using I2C

![Diagram of I2C communication](image)

Figure from the datasheet (page 13) for the ADXL313 accelerometer sensor ([http://www.analog.com/static/imported-files/data_sheets/ADXL313.pdf](http://www.analog.com/static/imported-files/data_sheets/ADXL313.pdf)). The sequence of steps for single-byte write, multiple-byte write, single-byte read, and multiple-byte read are shown above. For each of these read/write operations, the figure shows the communications from both the master and the slave. Note that the only difference between the single-byte and multiple-byte operations is in when the master sends the STOP bit (i.e., master decides how many bytes are read/written by sending a STOP bit after the required number of bytes).
SPI (Serial Peripheral Interface)

- Single master, multiple slaves; separate Slave Select (SS) line for each slave device; on the master, SS is simply a digital pin that can be set to 0 or 1 (e.g., can be a GPIO pin)
- Each slave device has a separate Slave Select (SS) line, also sometimes called a Chip Select (CS) line.
  - The slave select line is active low (i.e., writing 0 to the line tells the corresponding slave device that the master wants to communicate with it); to indicate that the slave select line is active low, it is usually written as $\overline{SS}$
- In addition to the slave select lines, SPI utilizes three lines that are shared between the master and slave devices
  - serial clock (SCLK) : the serial clock is generated by the master
  - master out, slave in (MOSI) : for data transmission from the master to a slave
  - master in, slave out (MISO) : for data transmission from a slave to the master
The MOSI and MISO pins are sometimes also called Slave Data In (SDI) and Slave Data Out (SDO), respectively.

If a single master device is connected to multiple slaves, separate Slave Select lines are connected from the master to each of the slave devices.

For example, if a microcontroller (SPI master) is connected to three sensors (slave devices) through an SPI bus, then there will be SCLK, MOSI, and MISO pins from the master device connected to each of the slave devices and there will be three separate SS pins, one for each slave device.
SPI (Serial Peripheral Interface)

- Full-duplex communication; both sides transmit at the same time.
  - In fact, SPI requires full-duplex, i.e., for each bit sent from one side (master or slave), the other side (master or slave) has to send a bit (which can be a dummy bit if there is no specific data to be sent in that direction at that time).

- Slave select lines are used in SPI instead of using device addresses; hence, multiple identical sensors can be easily connected to the same SPI bus by just utilizing separate slave select lines.

- Some SPI devices support a daisy chaining interconnection wherein the MISO of the first slave device is connected to the MOSI of the second slave device, etc.
Communicating bits over SPI

- The master starts the communication with a slave device by starting the clock (SCLK) signal (with a clock frequency less than or equal to the maximum frequency supported by the slave device) and selecting a slave device (by setting the corresponding SS line to 0).

- Then, in each clock cycle (as long as the clock pulses continue and the SS line is held as 0), the master and slave both send one bit at a time as follows:
  - The master sends one bit on the MOSI line; the slave reads it from MOSI
  - The slave sends one bit on the MISO line; the master reads it from MISO
The exchange of bytes between the master and a slave device during SPI communication can be visualized in terms of two shift registers.

- actually, SPI is often implemented in hardware using a shift register on each device
- one shift register on the master device and one shift register on the slave device (each shift register being typically 8 bits wide).
SPI communication can be visualized as two connected shift registers:

- While the master shifts out bits from its shift register (typically shifting out the most significant bit first) onto the MOSI line, the slave device shifts out bits from its shift register onto the MISO line. The bit arriving on the MOSI line is copied into the slave shift register while the bit arriving from the MISO line is copied into the master shift register. At the end of 8 shifts, the initial contents of the master and slave shift registers will have been exchanged. Thus, SPI can be thought of as a simple way to exchange bytes between the master and slave, with the master and the slave both simultaneously getting a byte from the other.

- Some of the bits sent by master/slave might be dummy values (if only data communication in the other direction is required at that time).
Example: Reading/writing registers using SPI

- Utilizing the basic SPI protocol to transmit and receive data, a protocol to write to and read from registers on an external SPI-connected device can be defined, for example, as described below:

- To write to a register on an external slave device:
  - the master first has to start the clock signal (SCLK) and select that particular slave device (by setting its SS line to 0)
  - then, the master sends a bit set to a particular value (e.g., 0) to indicate that it wants to write to a register
  - then the master sends a bit (0 or 1) to indicate if it wants to write multiple bytes (e.g., data bytes for multiple consecutive registers)
  - then, the master sends the address of the register it wants to write to
  - then, the master writes the data byte(s) to write to the register (or data bytes for multiple consecutive registers in multiple-write mode).
  - For each of these bits being sent by the master, the slave device sends a dummy bit (one dummy bit for each bit sent by the master). Once the master has written out the data bytes, it unselects the device (by setting its SS line to 1) and stops the clock signal (SCLK).
To read from a register on an external device:

- the master first has to start the clock signal (SCLK) and select that particular slave device (by setting its SS line to 0)
- then, the master sends a bit set to a particular value (e.g., 1) to indicate that it wants to read from a register
- then the master sends a bit (0 or 1) to indicate if it wants to read multiple bytes (e.g., data bytes from multiple successive registers)
- then, the master sends the address of the register it wants to read from.
- For each of these bits being sent by the master, the slave device sends a dummy bit (one dummy bit for each bit sent by the master). Once the register address has been written out by the master, the slave device starts transmitting the data byte(s) from the register(s). During the data transmission from the slave device, for each of the bits being sent by the slave, the master sends a dummy bit (one dummy bit for each bit sent by the slave). Once the master has received the data bytes it needs, it unselects the device (by setting its SS line to 1) and stops the clock signal (SCLK).
Example of a Sensor Interface Using SPI

Example of reading from a device register using SPI. Figure from the datasheet (page 12) for the ADXL313 accelerometer sensor (http://www.analog.com/static/imported-files/data_sheets/ADXL313.pdf). SDI is the MOSI line and SDO is the MISO line. Here, \( R \) denotes a bit to indicate that the master wants to read from a register and \( MB \) indicates a bit to indicate if the master wants to read multiple bytes.
Example of writing to a device register using SPI. Figure from the datasheet (page 12) for the ADXL313 accelerometer sensor (http://www.analog.com/static/imported-files/data_sheets/ADXL313.pdf). SDI is the MOSI line and SDO is the MISO line. Here, $\bar{w}$ denotes a bit to indicate that the master wants to write to a register and $\text{MB}$ indicates a bit to indicate if the master wants to write multiple bytes.
SDI-12

- SDI-12 (Serial Digital Interface at 1200 bits per second)
- Commonly used for remotely situated sensors in environmental monitoring applications
- A single data line that is connected to all devices on the SDI-12 bus
- Each device on the SDI-12 bus should have a unique address
- Half-duplex communication (i.e., only one device can transmit data at a time)
- Master (e.g., a microcontroller or computer) and multiple slaves (remotely situated sensor devices); the master initiates communication and a slave responds only when master requests data from it
- The sensors are typically in low-power mode (sleep) and wake up when a sensor reading is requested by the master.
- 1200 bps, 7 data bits per data byte and an even parity bit
- To start communication with a device, the master transmits the address of the slave device it wants to communicate with. The device with that address will respond while all other devices will ignore it.
CAN Bus

- CAN (Controller Area Network)
  - commonly used in vehicle applications (e.g., cars, ships, aircraft)
- The electrical specifications of CAN Bus are designed to enable utilization over longer distances (e.g., across the length of a large ship). The electrical communication over the CAN Bus utilizes CAN-High (CAN+) and CAN-Low (CAN−) lines. To convert from the voltage levels of the microcontroller’s digital pins to the voltage levels (and electrical protocol) utilized by the CAN Bus (CAN-High and CAN-Low), a CAN transceiver (transmitter + receiver) is used. On the microcontroller side, the CAN transceiver is connected through TX and RX pins (transmit and receive pins) through which the microcontroller can send data and receive data, respectively. When the microcontroller sends data over the TX pin, the CAN transceiver converts it to the CAN Bus electrical protocol and transmits over the CAN Bus (CAN-High and CAN-Low) lines. When the CAN transceiver receives data over the CAN Bus (CAN-High and CAN-Low) lines, the CAN transceiver sends the data to the microcontroller RX pin with the voltage level of the microcontroller pin.
All the devices (nodes) on the CAN Bus are connected through CAN transceivers to the same CAN-High and CAN-Low lines. Each node can send and receive messages, but not simultaneously (since all data is sent over the same CAN-High and CAN-Low lines). All messages sent over the CAN Bus are received by all the nodes.

Each message sent over the CAN Bus includes a message ID (identifier), which specifies the type of the message. Also, message identifiers are used to specify the priority of the message (some message types have higher priorities). Message identifiers must be unique on a CAN Bus.

The message identifier is either 11 bits (CAN 2.0A) or 29 bits (CAN 2.0B). In CAN 2.0B, the 29-bit message identifier includes an 11-bit base identifier and an 18-bit identifier extension.
A message (or frame) sent over the CAN Bus contains the message identifier and a set of data bytes (up to eight data bytes). The complete message sent over the CAN Bus also includes a CRC (cyclic redundancy check) used to detect any errors in communication and an acknowledge (ACK) slot.

The receiving of messages from the CAN Bus is often organized in software in terms of message boxes into which messages are received.

For any device that provides a CAN bus interface, an important part of the device datasheet is a list of message identifiers (also called object identifiers) supported by the device and the corresponding sets of data bytes (and the physical meanings of the data bytes). This is also sometimes called the CAN object dictionary of the device.
Ethernet is a quite complex technology. A few of the basic concepts utilized in Ethernet-based interfacing in embedded systems are described below.

The IP (Internet Protocol) address of each device is of form A1.A2.A3.A4 (with each of the numbers A1, A2, A3, and A4 being in the range 0 to 255). The network interface of each device also has a MAC (media access control) address which is utilized in the link layer of the networking protocol to indicate the source and destination for a packet.

Communications between ethernet-connected devices most commonly utilize either the TCP or the UDP protocol:
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)
TCP (Transmission Control Protocol): reliable, guarantees *ordered* communication, i.e., packets are guaranteed to be received in the same order in which they are transmitted.

- TCP utilizes various error checking and error recovery methods to achieve reliable and ordered communication.
UDP (User Datagram Protocol): does not guarantee reliable or ordered communication, e.g., some packets that are transmitted might not be received at the destination device (packet loss), some packets that are transmitted later might be received earlier.

However, by not having the elaborate error checking and error recovery components of TCP, UDP is more lightweight and can provide higher throughput (more number of packets or bytes per second) and lower latency (less delay between transmitting of a packet and receiving of the packet at the destination device).

UDP is a good choice for applications where some packet loss is okay, but higher throughput and reduced latency are more important. To handle some out-of-order packet receiving, message IDs included within the packet data are sometimes used (so that the destination device can, in software, interchange the order of received packets if required based on message IDs). UDP is commonly used in audio or video applications (e.g., voice over IP, audio streaming). UDP is also sometimes used for sensor interfaces in embedded systems wherein increasing number of sensor data samples and reducing delay is more important and it is okay to miss a few samples or to get some samples out of order.
USB (Universal Serial Bus)

USB (Universal Serial Bus) is a quite complex technology. A few of the basic concepts utilized in USB-based interfacing in embedded systems are described below.

In USB, devices are connected in a tree structure starting from the host (e.g., a computer or embedded microcontroller): devices connected to the host (one of these devices might be a USB hub), devices connected to a USB hub that is connected to the host (one of these new devices might again be a hub and additional devices can be connected to it), etc.

Each device can provide a set of endpoints with which the host can communicate (as a serial communication channel). An endpoint is simply a logical entity in software (i.e., the device can provide multiple endpoints with only one physical USB hardware connection). A device can provide multiple endpoints depending on the functionalities in the device; although up to 32 endpoints are allowed by the USB specification, most devices would typically only use a few endpoints.
The host can communicate with an endpoint through a pipe (a separate pipe for each endpoint). A pipe is a logical channel (through which bytes can be sent and received as if it were a separate serial communication channel). Two types of pipes are used:

- **Stream** pipes: uni-directional communication (communication in only one direction), e.g., the channel through which a sensor device sends back data as a continuous stream.
- **Message** pipes: bi-directional communication (communication in both directions); a bi-directional communication pipe is often used for configuration commands (from the host to the device) and status responses.

In USB communication, the first step is to find the devices on the USB bus and the endpoints provided by each device. This is called *enumeration*. Once the host has the information on the connected devices and their endpoints, it can open pipes to communicate with endpoints. To communicate with an endpoint of a device, the host would then specify (device_address, endpoint_number).