1 Background and Motivation

- We want to transfer data between two endpoints in the same embedded system. That is, we want to talk to the world outside the microcontroller and do something more interesting than blinking a LED or reading a push button. For example, we want to be able to drive a character LCD, so we need to send data to it; or we want to read data from an 3-Axis accelerometer that is found in our board.
- For the sake of these series of lectures, we will assume that the device we want to talk to offers a *digital interface*. Yes, there are devices out there that have *analog interfaces*. A common example is Video Graphics Array (VGA) monitors. A *digital interface* is one that sends and/or receives data as digital signals (logic 1 or logic 0). An *analog interface* is one that sends and/or receives signals as a series of voltage levels.
- To talk to our device, we could build it so that it takes blocks of data at a time. That is, we can send it multiple bits of data at one time and the device will read them all *at the same time*. This is called a *parallel interface*. On a parallel interface:
  - There are multiple data lines.
  - Data is sent and received multiple bits at a time over the data lines.
  - We must guarantee that all the data lines are stable before they are sampled (read) by the receiver.
  - We must also guarantee that all the data is received at the *same time*.
  - We may have to worry about crosstalk between the data lines. Crosstalk happens when an alternating current generates a magnetic field, and the magnetic field in effect induces current in adjacent lines. Since switching digital signals are in effect alternating currents, they may generate quite a bit of noise,

  This limits the usability of parallel interfaces to relatively low bandwidth applications. It is hard and expensive to make high speed parallel interfaces. A very common one is the interface between the CPU and RAM in your computers. Care is taken care when desigining this interface to avoid the pitfalls listed above, and then some more. If you ever look at a motherboard, look at the traces in this bus and notice how they are laid out. You will see unexpected curves, twists and turns, all to avoid the above listed problems.
- Other examples of parallel interfaces:
  - AT Attachment (ATA, now Parallel ATA, or PATA) in the old IBM PC and clones. This interface has been phased out mostly and replaced by the newer Serial AT Attachment (SATA), see below.
  - PCMCIA (Personal Computer Memory Card International Association) Interface and derivatives such as the CompactFlash interface. CompactFlash is rather popular in professional cameras.
  - Peripheral Component Interface (PCI) which was also popular in PC and clones. This interface has mostly been phased out by PCI Express (PCIe).
  - The HD44780 interface, which is a *de-facto* standard interface in character LCDs.
- To avoid the pitfalls of parallel interfaces, we could try sending data over a single data line one bit at a time. This is a *serial interface*. On a serial interface:

- There is (usually) a single data line.
- The data line may be bidirectional, that is, used for sending and receiving. Obviously, we can not send and receive over the same data line at the same time so the devices must know when to transmit or receive.
- There is (often) a clock line which tells devices when to sample the data line(s).
- We do not need to worry about the data arriving at the receiver at different times as much. There are fewer lines to worry about.
- Since we have fewer lines switching, we can shield ourselves better from crosstalk and other effects.

It turns out that serial interfaces can be faster than parallel ones.

- Examples of serial interfaces are:
  - Serial AT Interface (SATA). Currently, this is the bus that is used to connect drives to computers. SATA III can reach speeds of 6Gbit s^{-1}. This is 750MB per second. For comparison, PATA in its standard configuration achieved transfer rates of 133MB s^{-1} at its fastest configuration.
  - Peripheral Component Interface Express (PCI Express, or PCIe). This serial interface has largely replaced PCI and the long defuct AGP (Accelerated Graphics Port) in Personal Computers. PCIe version 4 can achieve transfer rates of 1.969Gbit s^{-1} in x1 mode and of 31.51Gbit s^{-1} in x16 mode. For comparison, PCI could achieve 133MB s^{-1} in its standard configuration and 533MB s^{-1} at its fastest configuration. This is over 7 times slower than PCIe. AGP could achieve up to 2133MB s^{-1}.
  - The Universal Serial Bus (USB) interface. Everybody here has used this interface before to connect flash drives, keyboards, printers, and other peripherals to computers. USB 2.0 interfaces support speeds of up to 480Mbit s^{-1}, while USB 3.0 devices support speeds of up to 5Gbit s^{-1}.
  - Universal Asynchronous Receiver Trasmitter (UART). This serial interface is commonly used in embedded systems between two devices. You have been using this interface in the lab and now in your homework to communicate with your microcontroller. UART is not limited to talking from and to a computer to a microcontroller. There are other peripherals, such as the Bosch Sensortec BNO-055 accelerometer/gyroscope/magnetometer sensor which can be used over a UART interface from a microcontroller. We will look at UART in this class.
  - Serial Peripheral Interface (SPI). This serial interface is commonly used in embedded systems. External EEPROM chips sometimes use this interface. We will look at this interface in detail in class.
  - Inter-Integrated Circuit ($\mathrm{I}^2\mathrm{C}$, sometimes I2C) interface. Also commonly used in embedded system. There is a larger, more encompassing version of this interface called System Management Bus (SMBus, SMB), which is used in personal computers. In this class, we will limit ourselves to $\mathrm{I}^2\mathrm{C}$.

2 The Universal Asynchronous Receiver Transmitter Interface

- The Universal Asynchronous Receiver Transimitter Interface is a point to point serial protocol that generally utilizes two lines: a dedicated transmit line (`TX`) and a dedicated receive line

(`RX`). Sometimes two extra lines are present: request to send (`RTS`) and clear to send (`CTS`).

- The `TX` line of one endpoint must be connected to the `RX` line of the other endpoint for communication to occur.
- Note the lack of a clock, data is sent asynchronously. Both endpoints must be configured to know when to expect the data.
- With one transmit and one receive line, data is sent one bit at a time. Figure 1 shows a typical UART frame.

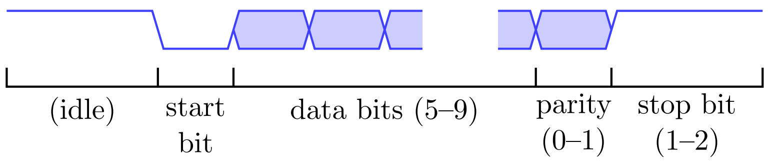


Figure 1: Typical UART frame

  - When the lines are idle (no data is being transmitted or received), the lines are held high.
  - To start transmission, the transmitter drives the line low. This signals the start bit.
  - After this point, the transmitter will start driving the line low or high, sending from 5 to 9 data bits. In most cases, 8 data bits are sent. Data is usually sent least significant bit (LSB) first.
  - Optionally, a parity bit is sent. To compute parity, we must first decide whether we want even or odd parity. If we wish to have even parity, then the number of bits that are set including the parity bit must be even. If we wish to have odd parity, the number of bits that are set including the parity bit must be odd. For example if we wish to have even parity on the 8bit message `11000110`, we count that 4 bits are set, so our parity bit must be `0`. If we wish to have odd parity in that same message, we must make the number of bits that are set an odd number. Since we can not change the message, we must set the parity bit to `1`, giving us a total of 5 bits set. We use parity bits to be able to detect any errors in transmission. With one parity bit, we can detect a 1bit error. If an error occurs in the parity computation, this results in a *parity error*.
  - Finally, one or two stop bits are sent. This is done by the transmitter driving the line high for one or two bits. If the receiver does not see the appropriate number of bits set during the expected stop bits, then a *framing error* occurs.

- It is imperative that both endpoints are properly configured for UART to work, otherwise, *framing errors* and *parity errors* will occur.
- The `RTS` and `CTS` signals are used for hardware flow control. If using hardware flow control the transmitter will assert the `RTS` line of the receiver, requesting to send data. If the receiver is capable of accepting the data, it will assert the `CTS` line, letting the transmitter know that the data can be safely sent.
- Since we have dedicated `TX` and `RX` lines in UART, there is no possibility of contention on the data bus. We effectively have two data busses, one for transmitting and one for receiving.

- It is still the responsibility of the endpoints to negotiate when to send data or to retrieve the sent data whenever it is available.

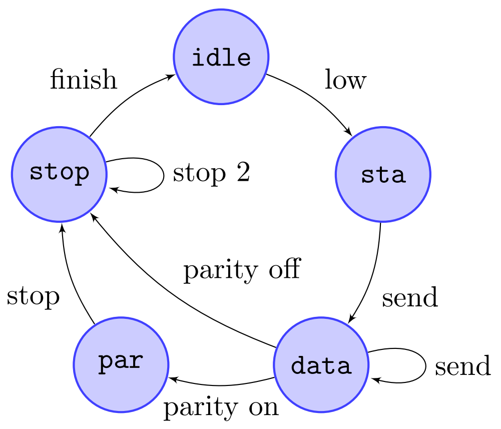
2.1 Typical Implementation of a UART module

The following is an overview of how you would implement a UART module in hardware. We will not go into a detailed implementation in hardware, as that is better served as an exercise to the student.

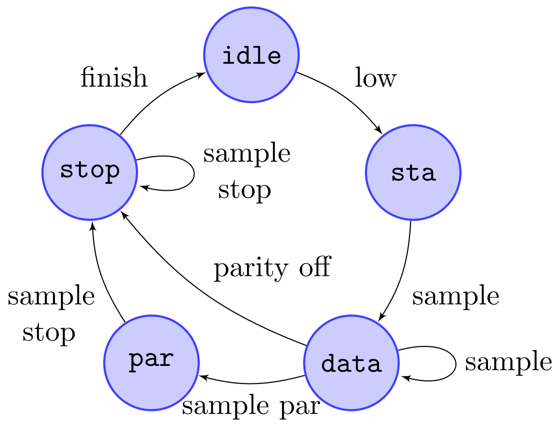
- Implementing a UART module requires a few components:
    1. a parallel-in serial-out shift register to implement the transmitter,
    2. a serial-in parallel-out shift register to implement the receiver,
    3. two optional parity generators (one for the receiver and one for the transmitter),
    4. a state machine to drive the receiver,
    5. a modulator to oversample the receiver line,
    6. a state machine to drive the transmitter, and
    7. a modulator to drive the transmit line high or low at the proper times.
- *Oversampling* refers to reading the a signal at a much higher frequency than the maximum frequency of the signal. In our case, we usually oversample the receive line at 16× its frequency.
- The receiver will sample the `RX` line until the start bit is detected. It will use the modulator to count until about halfway in the start bit. The receiver state machine will then wait for the modulator to count until about halfway in the middle of the first data bit. The receiver state machine will then sample the data bit. It will repeat this process until all data bits have been received. The state machine will then transition to receive the parity bit if configured to do so. Otherwise, it will transition to receive the stop bit(s). At this point, if it was configured to receive a parity bit, it will use the parity generator to compare the received parity with the one calculated. If the computed parity differs from the received one, a *parity error* is raised. After the receiver state machine reads the stop bits, it will determine whether it is necessary to raise a *framing error*.
- The transmitter works much of the same way. It will drive the `TX` line low for the duration of a start bit, then send each data bit, followed by the optional parity bit, then the stop bit(s).
- Figures 2a and 2b show the state machines for a UART transmitter and receiver, respectively. The implementation of these state machines is beyond the scope of this course and left to the reader as an exercise.

3 The Serial Peripheral Interface

- The Serial Peripheral Interface (SPI), unlike UART, allows for multiple devices to be connected to the same bus. For this purpose, a bus controller, called the *master*, communicates with multiple peripheral devices, called *slaves*. Furthermore, unlike UART, SPI is a *synchronous protocol*, as it has a dedicated clock line.
- There are three main signals in SPI:



(a) Transmit FSM



(b) Receive FSM

Figure 2: Finite State Machines used in a typical UART module

- `MOSI`: The *master out, slave in* signal. This is one of the two data lines in SPI. It is the one used by the master to transmit data and the slave receivers are connected to it. TI calls this signal `SIMO` in their datasheets.
- `MISO`: The *master in, slave out* signal. This is the second of the two data lines in SPI. It is the one used by the master to receive data and for the slaves to send data. TI calls this signal `SOMI` in their datasheets.
- `SCLK`: The *serial clock*. Depending on the configuration, a rising edge of the clock will shift data out of the master and slave devices and a falling edge of the clock will sample data on the master and slave devices. The master always drives the clock signal. Slave devices are allowed to transmit at the same time the master is transmitting. The clock is allowed to idle high or low, depending on the configuration of the devices.

- Being a multi-slave protocol, we need a way to select which device we are talking to, otherwise many slaves can decide to transmit data at the same time, causing bus contention. This is a *bad thing* since it invites magic smoke to come out of our parts. For this purpose, a *slave select* (`SS`) signal is introduced. This signal is typically active low. We use as many slave select signals as we have slaves in the bus, one per slave.
- When the master device wishes to talk to a slave, it asserts that particular slave's `SS` signal then it starts communication by driving the clock.
- We show a sample SPI frame in Figure 3 and a sample SPI topology in Figure 4.

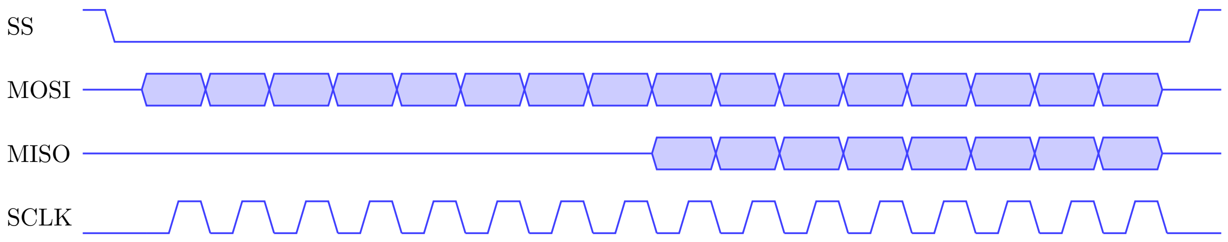


Figure 3: Example SPI frame

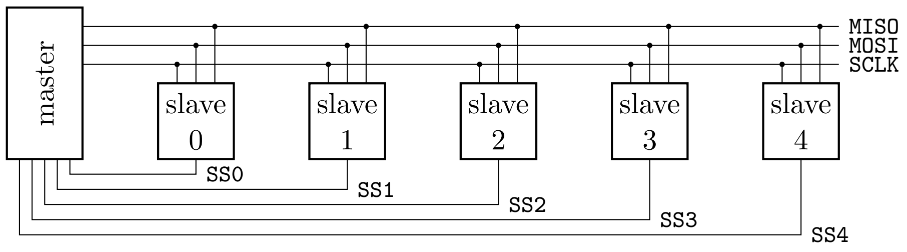


Figure 4: Example SPI topology. Although all slave devices utilize a shared `MISO`, `MOSI`, and `SCLK`, each device has its own dedicated `SS` line.

4 The Inter-Integrated Circuit Interface

- The Inter-Integrated Circuit (I^{2}C) Interface also allows multiple devices to be connected to the same bus. For this purpose, a bus controller, called the *master*, communicates with multiple peripheral devices, called *slaves*. Much like SPI, I^{2}C is a *synchronous protocol*, as it has a dedicated clock line, however I^{2}C allows for multiple masters to reside on the bus.
- I^{2}C was originally designed by Philips Semiconductor, now a part of NXP Semiconductors. Patents used to be required to implement the I^{2}C protocol, but these expired in 2006.
- There are two signals in I^{2}C:
  - `SDA`: The *serial data* signal. This is the data line in I^{2}C. It is used to transmit data to and from peripheral devices. As such, the `SDA` signal is *bidirectional*.
  - `SCL`: The *serial clock* signal. This is the clock signal in I^{2}C. Generally speaking, the master drives the clock to send and receive data from the slaves. Some slave devices may require the clock signal to be stretched. This is implemented by having the slave device hold the clock low, signaling the master device it is not ready to send or receive any more data. This may require extra configuration in the I^{2}C master.

  Signals in I^{2}C are open drain. This means that I^{2}C peripherals can only drive the signals low. As such, the signals must be pulled up by resistors. As both the master and slave devices will relinquish control of these signals at time, a signal may enter into a floating state, making it prone to noise and changing the state in the I^{2}C state machines. We show a typical I^2C topology in Figure 5.

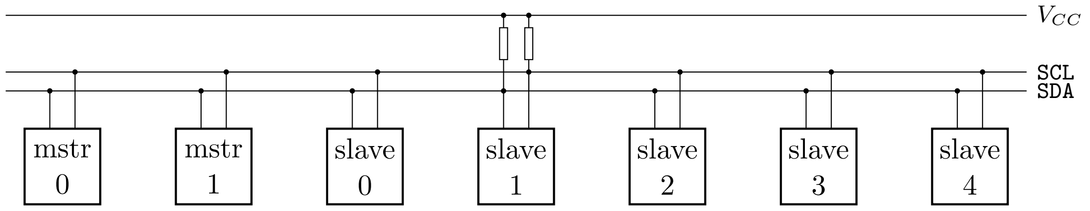


Figure 5: Example I^{2}C topology. Two pullup resistors are added, one to `SDA` and the other one to `SCL`.

- Being a multi-slave protocol, we need a way to select which device we are talking to. This is accomplished with the use of I^{2}C addresses. Slave devices implementing the I^{2}C protocol must pay fees to NXP Semiconductor to obtain a slave address.

- I^2C addresses are 7bit in size. As such, up to 128 slave devices may reside in an I^2C bus. The limitation is that each address must be unique. A further limitation is that adding multiple devices to the bus results in more parasitic capacitance being introduced, which has the effect of introducing extra latencies in the speed the lines can switch resulting in a requirement to use a slower speed grade.
- We show a typical I^2C frame in Figure 6.
  - Transmission is always started by an I^2C master. Because this is a multi-master protocol, a master must check whether the bus is available to transmit lest bus contention occur. If it is, it sends a start condition, which is generated by driving the `SDA` signal low while `SCL` is high. Other masters in the bus can detect the start condition and wait until the bus becomes ready again before attempting to transmit.
  - The transmitting master then sends the 7bit slave address. The most significant bit of the address is transmitted first. Then it sends a bit designating whether a read or write transaction is to take place. If a read transaction is to take place a 1 is sent, for a write transaction a 0 is sent.
  - If a slave device exists on the bus with that particular address, it replies with an `ACK` bit. This is generated by the slave driving the `SDA` line low. It should be noted that this requires the master to relinquish control over the `SDA` line.
  - What happens after this point depends on the type of transaction that was initiated by the master and the device that replied. Either the slave or the master will transmit data. Each time a byte is transferred, the other endpoint must reply with an `ACK`.
  - Transmission ends with a stop condition. This is done by the master releasing the `SDA` line and the `SCL` line. At this point a new transmission can start.

  The nice thing about having a hardware peripheral that implements the I^2C protocol is that the low level details of communication are left to the hardware.

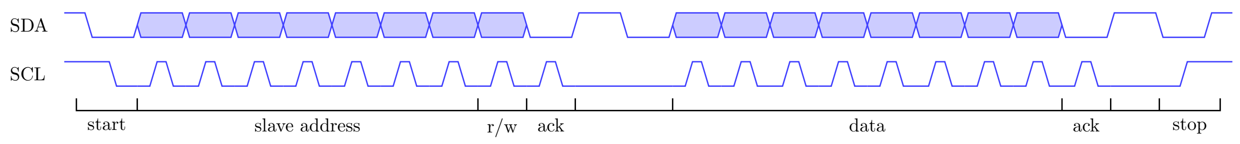


Figure 6: Example I^2C frame.

5 The MSP430 Universal Serial Interface

- Texas Instruments provides four versions of their Universal Serial Communication Interface.
  - `USCI_A`: Implements UART, SPI, and I^2C. You have used this peripheral in UART mode for your homeworks and your laboratory assignments. To configure the device in SPI or I^2C mode, you must set it in synchronous mode using bit `UCSYNC` in register `UCAxCTL0`. SPI or I^2C functionality is selected with the `UCMODEx` bits in `UCAxCTL0`. This peripheral is also capable of providing automatic baud rate detection and pulse shaping for IrDA (Infrared Data Association) transmission. This peripheral is described in detail in chapter 15 of document number `slau144` for UART mode, and chapters 16 and 17 of the same document for SPI and I^2C modes, respectively.

- `USCI_B`: This peripheral is a reduced version of the `USCI_A`, providing only I^{2}C and SPI functionality. This peripheral is described in detail in chapters 16 and 17 of document number `slau144` for SPI and I^{2}C modes, respectively.
- `eUSCI_A`: This peripheral is largely equivalent to the `USCI_A`. However, it offers a more resilient I^{2}C state machine and better baud generation capabilities, among other improved functions. Refer to document number `slaa522` for differences between this peripheral and the `USCI_A`. Newer TI microcontrollers have this peripheral instead.
- `eUSCI_B`: This peripheral is largely equivalent to the `USCI_B`. However, it offers a more resilient I^{2}C engine. Newer TI microcontrollers have this peripheral instead. Document number `slaa522` provides the difference between these peripherals.