All the I/O examples we have discussed use the system bus to send data between the CPU, main memory, and I/O controllers. The system bus runs at a slower clock speed than the CPU because of greater distances involved between the various components, but there is still a clock governing the reading and writing to the bus. The sender and receiver of any bits written to the system bus use the same system bus clock, e.g. writes are performed at the end of the clock pulse. For this reason, the system bus is called *synchronous*.

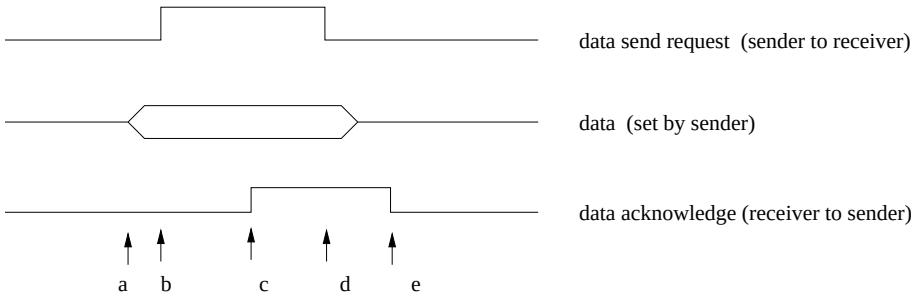
Synchronous buses don't work well when the distances between the sender and receiver are too large, however, such as when data is being sent from an I/0 controller to a peripheral device. The clock signals themselves need to travel, and so the assumption of events happening "at a given time" becomes problematic. When the distances between sender and receiver are too large (say 1 meter), the precise timing of the clock cannot be relied on in the same way as before. When the bus itself is not controlled by a clock, one says the bus is *asynchronous*. Timing events are still important though. Today we'll look at two ways to achieve timing on such buses.

Handshaking

How can data be sent on a bus without the sender and receiver synchronizing on the same clock? Let's take the case of an I/O controller (e.g. a printer controller, i.e. inside the computer case) that communicates data to an I/0 peripheral device (i.e. the printer itself, attached to a cable several metres long). The asynchronous transfer can be implemented with a method called *handshaking*.

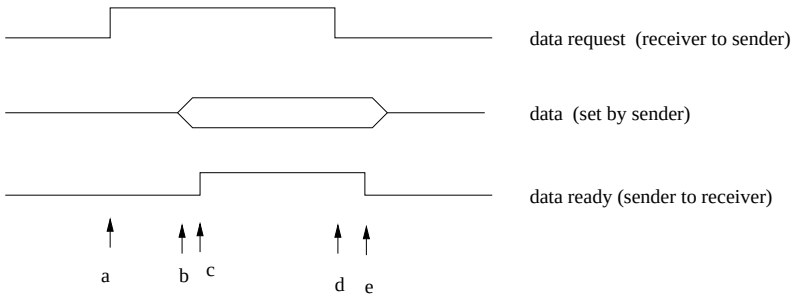
In general, there are two types of handshaking, depending on whether the data source (sender) or destination (receiver) initiates the transfer. For the printer example, the source/sender is the I/0 controller and the destination/receiver is the printer.

In *source initiated* handshaking, the sender (a) puts data on data line and shortly afterwards (b) sets a control signal called *data request*, and holds both signals. The receiver reads the control signal, then reads the data, and then (c) responds with a *data acknowledge* control signal. The sender receives the acknowledgement and (d) stops writing the data, and resets the data request line to 0. The receiver then (e) sets the data acknowledge to 0.



The hexagonal symbol in the middle row indicates the time that the data that is put on the line. This symbol is used here indicates that it doesn't matter whether the data is a 1 or 0 (but it has to be one them during that interval). Also note that this data line can consist of multiple parallel lines. It doesn't have to be just a single line as indicated in the figure.

Alternatively, the receiver (destination) can initiate a data transfer. It does so by (a) setting its *data request* control signal. The sender reads this signal, then (b) puts the data on the data line,



and shortly afterwards – to make sure the data signal can be read – it (c) sets a data ready signal. The receiver reads the data ready signal and then reads the data, then (d) the receiver resets the data request signal (off). The sender then sees that the request has been turned off so it (e) turns the data ready signal off and stops writing the data.

I emphasize that no clock is involved here i.e. it is asynchronous. One side waits until the other side is ready. The sender and receiver might each have their own clocks, but there is no *shared* clock used for the bus.

Example: parallel port (printer)

An example of the above scheme was used by the standard "parallel port"(called Centronics) which is a D-shaped port that was used to connect a printer to a computer. See lecture slides. It has 25 holes (1-13 and 14-25) into which 25 wires can fit. For example, the source ready (#1) and data (#2-#9) pins are output only, i.e. from the controller to the printer. The ACK (#10) and Busy (#11) pins are input only, i.e. from the printer to the controller. Another input pin is the "Paper Out" pin (#12), indicating there is no paper left!

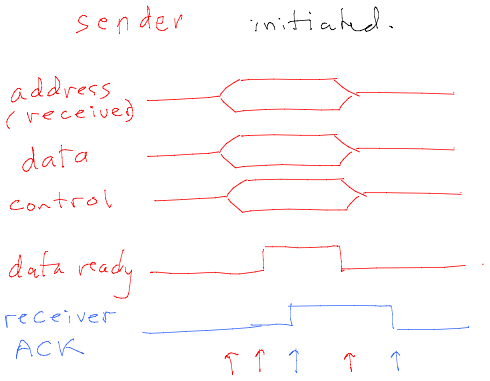
Centronics used a source-initiated handshaking method. Suppose the printer controller tries to write data to the printer. It turns on wires #1 for 100 ns (a relatively long time) and then puts a byte of data on wires #2-#9. The printer observes the source ready signal on wire #1. If the printer is ready, it reads the data (from wires #2-#9). After completing the read, the printer sets the ACK wire (#10) for 400 ns (a long time). Once the controller sees the ACK wire is set, it stops sending the data and sets wire #1 to 0. If, however, if the printer is not yet ready for the data i.e. if it cannot keep up with successive data ready signals, it sends a BUSY signal (wire #11), and the printer controller waits for some time before trying again.

asynchronous shared bus

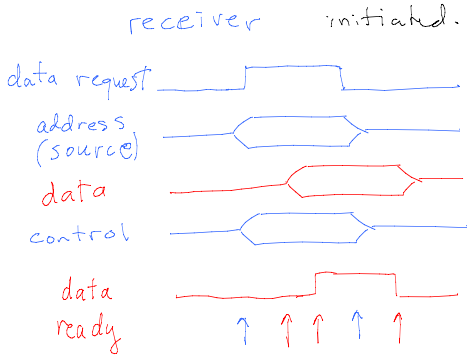
Although I described this asynchronous scheme for the case of an I/O controller communicating with a single I/O device, it is possible in principle to use the same scheme for a bus where several devices could be sharing lines.

Consider the sender initiated case. In the figure below, I have indicated when the sender uses part of the bus in red and when the receiver uses part of the bus in blue. First, the sender puts the data, address *i.e* the targeted receiver, and command (control) on the appropriate lines. It then waits a short time and turns on the data ready signal. The devices read these signals. The receiver, in particular, sees from the address bus that it is the target for this signal. Once it has read the

information, it turns on a 'receiver acknowledge' (ACK) signal (some other line). This signals that it has read the information off the bus. The sender sees this, then stops writing on the bus (and the bus then has unspecified values). The receiver eventually drops its ACK signal to 0.



In the receiver initiated case, the receiver puts address *i.e* the targeted source and the command (control) on the appropriate lines. Then, the receiver waits and gives the signal a chance to stabilize over the whole line. The devices decode the signals. The source/sender, reading the address line, sees that it is the target. So it puts the requested data on the bus and shortly after it sets the 'data ready' signal. The receiver sees the ready signal and reads the data. When the receiver is done, it removes the data request line and stops writing on the address and control lines (and so the values on these lines is unspecified). When the sender sees that the 'data request' signal has been turned off, the sender stops putting the data on the line, and it turns off the data ready signal.



Notice that when the bus is shared, there needs to be some scheme to avoid that multiple devices write on the bus at the same. This could be achieved using a similar scheme to what we saw with the (clocked) system bus e.g. dedicated IRQ/IACK lines with one party (the CPU in that case) controlling who is allowed to write.

Serial bus

For all of the buses we have discussed up to now, we have thought of the signals on the bus as being constant over each line. That is, each wire of the bus held one value and a device only read from the bus once that value had stabilized over the whole bus. A *serial bus* is completely different. With a serial bus, you should think of a sequence of signal *pulses* travelling down a wire. To read from the serial bus, you need to time the reads with the pulses as they pass. A clock is still necessary since the pulses must be all teh same duration and this duration needs to be agreed upon in advance.

The "speed" (or frequency) of a serial bus is measured in bits per second (baud). This is the number of bits or pulses that passes any given point on the bus per second. If data is sent at a certain speed (e.g. 56,000 bps in the case of a very old modem) then each bit has a duration of 1/56,000 seconds, which is about 20,000 ns. If you have a particular physical cable, then you could think of each bit as occupying a certain spatial extent on the cable. Or, if you measure the signal at a particular point on the bus, then a particular value can be measured at that point for this duration.

For a 56 Kbps modem, the speed is much less (by a factor of several thousand!) than the clock speed of your system bus. So, you should think of the clock speed of the system bus as being a very fine sampling relative to the bits on the serial bus. (And the CPU clock speed another factor of ten faster than the system bus clock speed.)

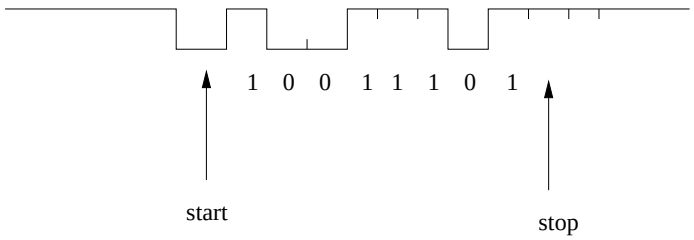
How does the receiver read from a serial bus? Assume for the moment that the receiver and sender have agreed on:

- the duration T of one bit
- number of bits to be transmitted (say 8, i.e. one byte)

The sender puts a 1 signal on the wire for the normal state (non-transmitting). Then the sender switches to 0 to indicate that a sequence of bits is to follow. This initial 0 has the same duration T as each of the bits that follow, and is called *start bit*. The start bit indicates that a byte is about to be sent, but otherwise contains no information. The receiver waits .5 T and checks to make sure that the signal is still 0. This is a check that that the switch to 0 wasn't some electronice noise. The receiver then waits 1 T and samples the signal, and repeatedly samples every one T for as many bits as are expected (8). Note that by waiting .5 T and regularly sampling after that every T, it samples from the middle of the pulse i.e. avoiding the transition at the boundaries of the pulse. When it has all 8 bits of data, it reads a "stop bit" (value 1). At least one stop bit is there to ensure some spacing between bytes. In particular, the receiver can only read in the next byte once it notices a transition from 1 to 0, that is, when it notices the next start bit. To guarentee this happens, the line must be in the 1 state.

Note that the sender and receiver each their own clock, typically of different speeds, and the pulse duration of these clocks is much shorter than T.

For example, if the byte to be sent is 10011101, then the sequence looks like as follows. (Here the wave of pulses is travelling to the left. In the slides, I showed it travelling to the right.) For this example, the bits are sent from MSB to LSB (though one could send in the other order too). The receiver reads the start bit first, followed by the eight bits of the sequence, followed by the stop bit.



How do messages get started? Messages often have headers and text. To indicate boundaries, the sender sends a special ASCII value:

- 0x01, called `SOH` (start of header)
- 0x02, called `STX` (start of text)
- 0x03, called `ETX` (end of text)
- 0x04 called `EOT` (end of transmission)

In some systems there is an extra mechanism for checking if error have occured in the transmission. Errors can occur because lines are physical and sometimes have noise. One simple way to check for errors is for the sender to append an extra bit to the character – called a *parity* bit – such that the total number of 1 bits in the code is even. This is called *even parity*. If the character has an odd number of 1 bits, then the parity bit is set to 1, so that the total number of 1's is even. If the ASCII character has an even number of 1 bits, then the parity bit is set to 0. Note it could happen that there are two errors (two of the ASCII bits are incorrect), and in this case, the errors would not be detectable.

UART

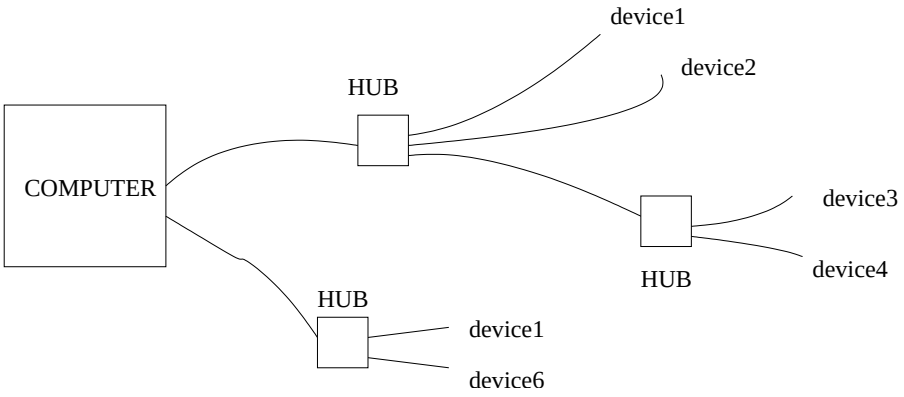
What sort of hardware is needed by the sender and receiver? The sender needs to convert bytes to bit sequences. The receiver needs to convert bit sequences into bytes. Thus, the sender needs a parallel-in serial-out shift register. This is just what it sounds like. For example, consider a I/O controller for a printer which uses a serial line. It takes one byte in parallel off the system bus, and then writes the bits one by one on a serial line. It could write either the least significant bits first (in the case of shift right) or most significant bits first (in the case of shift left). Similarly, the printer (receiver) needs a serial-in parallel-out shift register. It reads the bits one by one into a register, and then outputs the bytes as a unit into another register which is used for processing the byte.

The *UART* (Universal Asynchronous Receiver/Transmitter) is an I/O controller that converts serial to parallel (input) or parallel to serial (output). A UART is more than just the shift registers mentioned above, though. It has several registers so that the CPU or I/O device can communicate with it. It also needs special circuitry to remove the start and stop bits and to check for parity.

USB (Universal serial bus)

As you know, computers that are sold these days do not have dedicated ports for the printer, mouse, keyboard, scanner, etc. Instead, the devices typically use a standard bus called USB.

Many USB devices can be connected to a computer. Physically the connections define a tree structure. The root of the tree is the USB controller, inside the computer, which is also called the USB *host*. You may have two ports (slots where you plug the USB connector) which are children of the root. You can connect USB hubs which serve as internal tree nodes. And you can connect hubs to hubs. In total, you can connect up to 127 devices to a single USB host.



Although the above describes a tree like structure, in fact each of the USB devices sees everything that it put on the bus, i.e. the hubs (internal nodes) do not block the signals being sent. The host communicates by polling. It sends messages to the devices, and they respond only when prompted. "Device 15, do you have anything to say". (Response: "no)". "Device 24, do you have anything to say". (Response: "no"). "Device 15, do you have anything to say". (Response: "yes, bla bla bla"). etc.

The USB standard specifies various formats for *packets* of data that are used to communicate between the host and a USB device. Each packet consists of fields including a synch which is used to synchronize the devices to some extent (details omitted), packet ID which says what type of packet it is ('ready'), address (which device is this packet for?), data, error correctio information. Details (many!) are omitted here.

Many USB drivers come with typical operating systems (Windows 7, Linux Ubuntu, Max OS X, ...). If a driver doesn't come preinstalled then it can be downloaded (sometimes automatically). When you connect a USB device (such as a new mouse, or printer) to your computer, the USB controller (host) interrogates the device to find out what it is. It then sends this information to the CPU, which checks if the OS has the driver on disk. If it does, it loads the driver. This automatic system is known as "plug and play".[1]

Other I/O ports

In class, I briefly discussed a few other I/O ports.

- VGA (a 15 pin parallel connector – what I use to connect my laptop to the projector), invented in 1980s, the pins contain R,G,B data values, plus various control lines (end of row, end of frame)

[1]When the system was first being introduced about a decade ago and didn't work as well as it does now, it was known as "plug and pray".

- DisplayPort, miniDP
- HDMI (high definition, multimedia (video and audio)

I also briefly discussed how an optical mouse works. This material is for your interest only. It is not on the final exam and so I do not elaborate here.