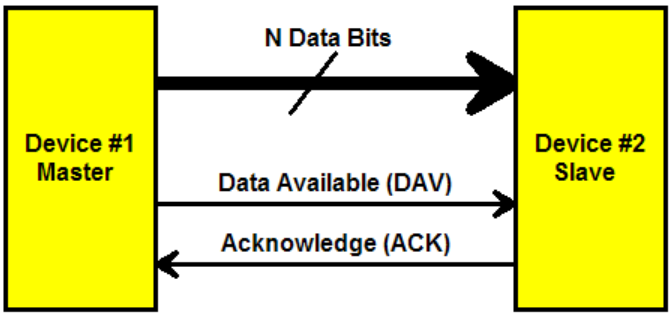
Lecture #28 – April 21, 2004

Serial and Parallel Communication

Parallel Communication Using Handshaking

Suppose we have two devices that need to communicate. At any single point in time only one can send to the other; the sender is the master and the receiver is the slave. The devices may swap master/slave status as needed. The data bits are sent all at the same time, in ***parallel***, with one wire for each data bit. In addition to the data bits there will be an extra line to tell the slave that new data are ready, and another line for the slave to acknowledge the transmission. Early printers (pre-USB) used this approach.



The process of communicating data from one device to another using this approach is called ***handshaking***. The master places new information onto the data lines, and then *raises* the DAV data available line, making it go from 0 to 1. When the slave sees DAV go high, it acknowledges by raising the ACK line and latching the data into its receive buffer. When the master sees ACK go high, it lowers DAV and optionally removes the information from the data lines. When the slave sees DAV go low, it lowers ACK. As soon as the master sees ACK go low it knows that the cycle is complete.

An alternative approach retards the latching of the data by the receiver by a small amount in order to insure that all data lines have had a chance to settle to their final values first. In this approach, the master places the information on the data lines and raises DAV, as before. The slave sees DAV go high and acknowledges by raising ACK. When the master sees ACK go high it lowers DAV, and only when the slave sees DAV go low does it latch the data. Afterwards, it lowers ACK to signal that it is done. The master waits until it sees ACK go low before starting a new communication cycle.

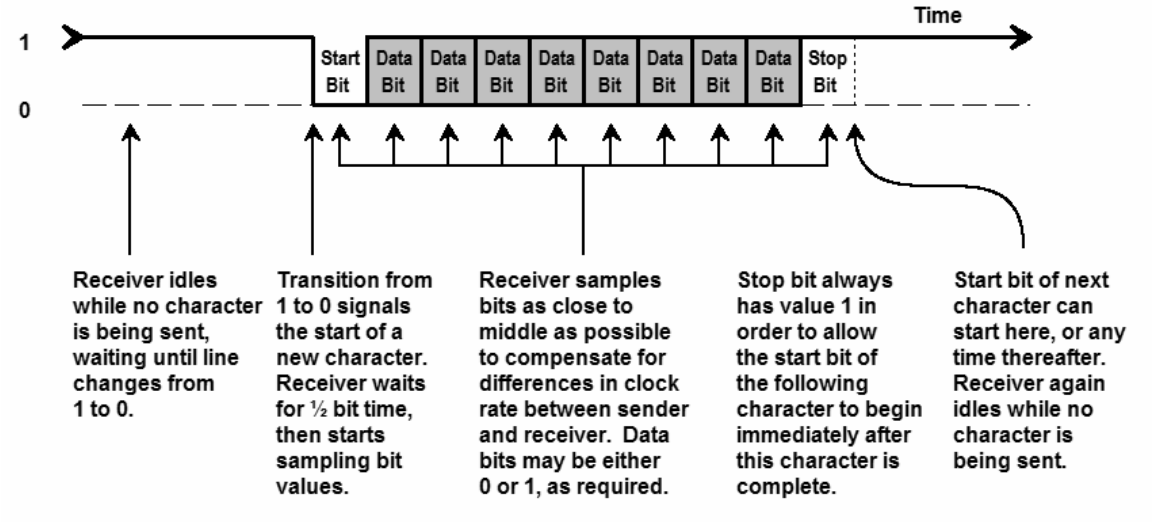
Regardless of the approach, the result is to guarantee that signals traversing a long distance have a chance to settle to their correct values before the receiver grabs them. Remember that the speed of light in a vacuum is between 11 and 12 inches per nanosecond, so two cables that differ in length by only a few inches will delay signals by significantly different amounts of time. Handshaking accommodates these differences.

Asynchronous Serial Communication

In a case where there is only a single wire to communicate between one device and another (or a single wire in each direction), we must turn to ***serial*** techniques that transmit one bit at a time. Since we lack a data-available line and an acknowledge line, handshaking isn't possible. Hence, each character is sent ***asynchronously***, or at unpredictable times. A classic example of this is that of the modem, which must transmit data as successive tones over a telephone line; each tone represents one bit value (or a very small number of bit values). The computer on the other end of the telephone line has no way of knowing when the user will type the next key, and it has no DAV or ACK lines to help out.

After removing the modem, telephone line, and acoustic tones from the picture we still have the same problem of transmitting many bits one at a time over a single data line. Sender and receiver must be configured to agree on transmission rates ahead of time, so both know how long a bit is *supposed* to last. They must also be configured to agree on how many data bits are to be sent at any one time.

When no information is being transmitted the data line "idles" at a stable value, usually 1. Sending 1 bits would cause no change in the data line's value, so we need to signal the start of a new character by bringing the data line to 0 for one bit time. This is called the ***start bit***. The receiver initially waits for half of a bit time to synchronize measurements with the middle of each bit, and then samples the values at whole bit times thereafter. Since the data bits are a mixture of 0s and 1s, there is no guarantee that the data line will go back to the idle state at the end of the transmission unless we append a 1 bit to the end of the data bits. This is called the ***stop bit***. After the stop bit the data line can be idle for an arbitrary amount of time, and will resynchronize with the next character upon seeing the beginning of the next start bit (which may occur immediately after the previous character's stop bit is complete).

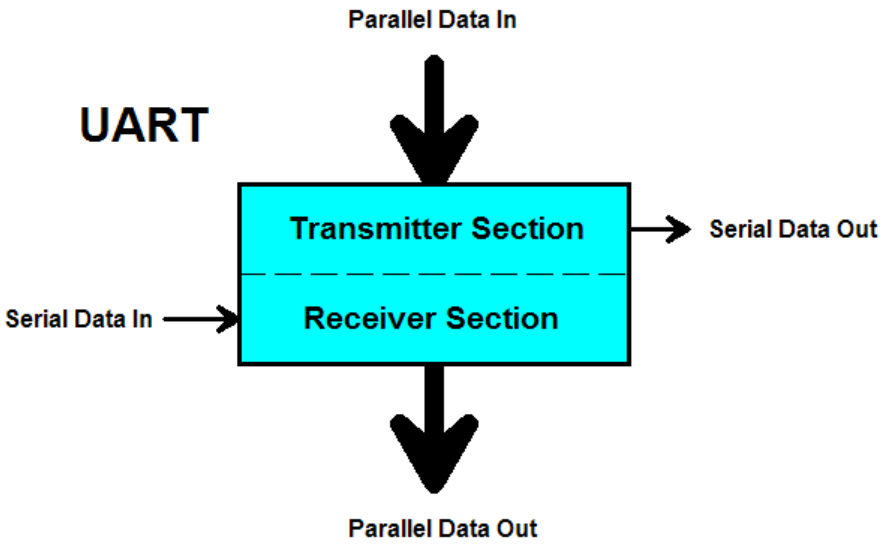


The receiver samples as close to the middle of a bit as possible to avoid measuring the line as a bit is changing from 0 to 1 or from 1 to 0. This also compensates for slight differences in clock rates between sender and receiver. For 8-bit characters a difference of a few percent can be tolerated without dropping data.

One of the data bits may be used as a ***parity bit***, to detect when transmission errors occur. ***Odd parity*** means that the total number of 1 bits in the character must add up to an odd number, so upon transmission the parity bit of a character is set to 1 or 0 to guarantee this condition. Similarly, ***even parity*** means that the total number of 1 bits must add up to an even number. Upon receipt, the character is checked to see if it still has the correct parity. The receiver can detect if any single bit is in error, but it doesn't know which one was changed. The sender and receiver must be configured to agree on which parity model to use; otherwise every character will be flagged as an error.

Universal Asynchronous Receiver Transmitter (UART)

A ***Universal Asynchronous Receiver Transmitter***, or ***UART***, is a single chip that contains both a parallel-to-serial converter (the transmitter) and a serial-to-parallel converter (the receiver). While the two sections may share the master clock signal and some configuration lines (such as which parity model to use), the two sections are largely independent from one another.



The transmitter section may contain a buffer (i.e., a hardware queue) so that a communications program can dispatch a bunch of characters all at once. The UART will then take care of sending each one in order without intervention by the software. Similarly, a buffer on the receiver section will hold several characters until the software gets a chance to read them in. Characters will be lost if the software does not empty the buffer in time. Early UART chips on the IBM-PC could buffer only a single character; data loss was common in such cases. Later UARTs could buffer up as many as 16 characters, drastically reducing the occurrence of data loss.