Serial Ports

From OSDev Wiki

Serial ports are a legacy communications port which has pretty much been succeeded by USB and other modern communications technology. However, it is much easier to program than USB, and it is still found in a lot of computers (especially older ones such as the ones the financially limited amateur OS writer might use for testing). Also, a lot of phone modems (which are still used to access the Internet quite often, even though broadband is very accessible and affordable) connect via the serial interface. Furthermore it is possible to use the serial ports for debbuging, since a lot of emulators allow the redirection of their output into a file.

Contents

Wires, Pins, Connectors and the like

The Wikipedia page on Serial ports has a lot of information, and it is summarised here. The serial interface is very simple. There are actually two kinds of serial port: 25-pin and 9-pin. 25-pin ports are not any better, they just have more pins and are bigger. 9-pin is smaller and is used more often though in the past the 25-pin ones were used more often. The 9-pin ones are called DE-9 (or more commonly, DB-9 even though DE-9 is it's technical name) and the 25-pin ones are called DB-25. They plug in to your computer using a female plug (unless your computer is odd and has a female port, in which case your cable will need a male plug). This Wikipedia page has more information on the plug used.

Both have the same basic types of pins. A DB-25 has most of the pins as ground pins, whereas a DE-9 has only a few ground pins. There is a transmitting pin (for sending information away) and a receiving pin (for getting information). Some serial ports can have a duplex mode--that is, they can send and receive simultaneously. There are a few other pins, used for hardware handshaking. In the past, there was no duplex mode, so if a computer wanted to send something it had to tell the other device or

computer that it was about to transmit, using one of the hardware handshaking pins. The other thing would then use another handshaking pin to tell it to send whatever it wanted to send. Today there is duplex mode, but the handshaking pins are still used.

If you want to connect two computers, you need two things in your cable:

1. The cable needs to have two female plugs so it can plug into both computers.
2. The cable needs to have it's transmit-receive wires and it's handshaking wires switched. This can be done in the cable itself, or as an extension called a Null Modem

For serial devices, you don't need to setup the cable this way. The receiving end of the device has the wires switched and it has a female port, which means you can plug a male plug into it.

Why Use a Serial Port?

During the early stages of kernel development, you might wonder why you would bother writing a serial driver. There are several reasons why you might:

GDB debugging
: You can use the serial port to connect to a host computer, and use the GDB debugger to debug your operating system. This involves writing a stub for GDB within your OS. Helpful information might be found at http://developer.apple.com/documentation/DeveloperTools/gdb/gdb/gdb_18.html.

Headless console
: You can operate the computer without a monitor, keyboard or mouse and instead use the serial port as a console using a protocol such as TTY or VT100.

External logging
: When the system itself is in danger of potentially crashing at times, it's nice to get debugging outputs safe to another computer before the test system triple-faults.

Networking and File transfers
: Serial ports are useful for transferring information between systems when other more traditional methods are unavailable.

Programming the Serial Communications Port

If you want to use the serial port for communications, you first have to initialize it. You tell it how fast your connection speed between the other computer or device will be (this is called the baud rate)--you must have the same speed as the other device or computer is setup to use, or you will have problems. It is probably safer to use the slower speeds unless you need the faster speeds for some reason, for example if you are playing a multi-player game over a serial connection. You also need to setup the parity type and the number of bits in a character. Once again, your computer must be setup with the same values for these things as the other computer or device has, or communication will not work.

Once you have setup these things, you still need to setup the interrupt handlers. You can poll the port to see if any new things have arrived, or if it's time to send another character, but this slows things down and will not work very well in most real-time applications or multithreaded environments. In the case of a game, this is not a good idea at all.

You use IRQ #4 for COM ports 1 or 3, and IRQ #3 for COM ports 2 or 4 (you can tell which port sent the interrupt when you receive the interrupt). The IRQ handlers check if you are receiving something, and if so they receive the character and handle it somehow, such as placing it into a buffer. They also

check if the other side is ready to receive something from you, and if you have something to send, it is sent.

Port Addresses

The addresses for COM ports can vary depending on how they are connected to the machine and how the BIOS is configured. Some BIOS configuration utilities allow you to see and set what these are, so if you in doubt for a test machine, this might be a good place to look to get you started.

For the most part, the first two COM ports will be at the addresses specified, the addresses for further COM ports is less reliable.

COM Port	IO Port
COM1	3F8h
COM2	2F8h
COM3	3E8h
COM4	2E8h

You should be able to find the IO port addresses of the COM ports in the BIOS Data Area

Once you have the base address of your COM port, you add an offset value to get to one of the data registers. One of the registers hold what is termed the DLAB or Divisor Latch Access Bit. When this bit is set, offsets 0 and 1 are mapped to the low and high bytes of the Divisor register for setting the baud rate of the port. When this bit is clear, offsets 0 and 1 are mapped to their normal registers. The DLAB bit only affects port offsets 0 and 1, the other offsets ignore this setting.

IO Port Offset	Setting of DLAB	Register mapped to this port
+0	0	Data register. Reading this registers read from the Receive buffer. Writing to this register writes to the Transmit buffer.
+1	0	Interrupt Enable Register.
+0	1	With DLAB set to 1, this is the least significant byte of the divisor value for setting the baud rate.
+1	1	With DLAB set to 1, this is the most significant byte of the divisor value.
+2	-	Interrupt Identification and FIFO control registers
+3	-	Line Control Register. The most significant bit of this register is the DLAB.
+4	-	Modem Control Register.
+5	-	Line Status Register.
+6	-	Modem Status Register.
+7	-	Scratch Register.

Line Protocol

The serial data transmitted across the wire can have a number of different parameters set. As a rule, the sending device and the receiving device require the same protocol parameter values written to each serial controller in order for communication to be successful.

These days you could consider 8N1 (8 bits, no parity, one stop bit) pretty much the default.

Baud Rate

The serial controller (UART) has an internal clock which runs at 115200 ticks per second and a clock divisor which is used to control the baud rate. This is exactly the same type of system used by the Programmable Interrupt Timer (PIT).

In order to set the speed of the port, calculate the divisor required for the given baud rate and program that in to the divisor register. For example, a divisor of 1 will give 115200 baud, a divisor of 2 will give 57600 baud, 3 will give 38400 baud, etc.

Do not be tempted to use a divisor of 0 to try to get an infinite baud rate, it won't work. Most serial controllers will generate a unspecified and unpredictable baud rate (and anyway infinite baud would mean infinite transmission errors as they are proportional.)

To set the divisor to the controller:

1. Set the most significant bit of the Line Control Register. This is the DLAB bit, and allows access to the divisor registers.
2. Send the least significant byte of the divisor value to [PORT + 0].
3. Send the most significant byte of the divisor value to [PORT + 1].
4. Clear the most significant bit of the Line Control Register.

Data Bits

The number of bits in a character is variable. Having fewer bits is, of course, faster, but they store less information. If you are only sending ASCII text, you probably only need 7 bits.

Set this value by writing to the two least significant bits of the Line Control Register [PORT + 3].

Bit 1	Bit 0	Character Length (bits)
0	0	5
0	1	6
1	0	7
1	1	8

Stop bits

The serial controller can be configured to send a number of bits after each character of data. These reliable bits can be used to by the controller to verify that the sending and receiving devices are in phase.

If the character length is specifically 5 bits, the stop bits can only be set to 1 or 1.5. For other character lengths, the stop bits can only be set to 1 or 2.

To set the number of stop bits, set bit 2 of the Line Control Register [PORT + 3].

Bit 2	Stop bits
0	1
1	1.5 / 2 (depending on character length)

Parity

The controller can be made to add or expect a parity bit at the end of each character of data transmitted. With this parity bit, if a single bit of data is inverted by interference, a parity error can be raised. The parity type can be NONE, EVEN, ODD, MARK or SPACE.

If parity is set to NONE, no parity bit will be added and none will be expected. If one is sent by the transmitter and not expected by the receiver, it will likely cause an error.

If the parity is MARK or SPACE, the parity bit will be expected to be always set to 1 or 0 respectively.

If the parity is set to EVEN or ODD, the controller calculates the accuracy of the parity by adding together the values of all the data bits and the parity bit. If the port is set to have EVEN parity, the result must be even. If it is set to have ODD parity, the result must be odd.

To set the port parity, set bits 3, 4 and 5 of the Line Control Register [PORT + 3].

Bit 5	Bit 4	Bit 3	Parity
-	-	0	NONE
0	0	1	ODD
0	1	1	EVEN
1	0	1	MARK
1	1	1	SPACE

Interrupt enable register

To communicate with a serial port in interrupt mode, the interrupt-enable-register (see table above) must be set correctly. To determine which interrupts should be enabled, a value with the following bits (0 = disabled, 1 = enabled) must be written to the interrupt-enable-register:

Bit	Interrupt
0	Data available
1	Transmitter empty
2	Break/error
3	Status change
4-7	Unused

Example Code

Initialization

```
#define PORT 0x3f8   /* COM1 */

void init_serial() {
   outb(PORT + 1, 0x00);    // Disable all interrupts
   outb(PORT + 3, 0x80);    // Enable DLAB (set baud rate divisor)
   outb(PORT + 0, 0x03);    // Set divisor to 3 (lo byte) 38400 baud
   outb(PORT + 1, 0x00);    //                  (hi byte)
   outb(PORT + 3, 0x03);    // 8 bits, no parity, one stop bit
   outb(PORT + 2, 0xC7);    // Enable FIFO, clear them, with 14-byte thre
   outb(PORT + 4, 0x0B);    // IRQs enabled, RTS/DSR set
}
```

Notice that the initialization code above writes to [PORT + 1] twice with different values. This is once to write to the Divisor register along with [PORT + 0] and once to write to the Interrupt register as detailed in the previous section. The second write to the Line Control register [PORT + 3] clears the DLAB again as well as setting various other bits.

Receiving data

```
int serial_received() {
   return inb(PORT + 5) & 1;
}

char read_serial() {
   while (serial_received() == 0);

   return inb(PORT);
}
```

Sending data

```
int is_transmit_empty() {
   return inb(PORT + 5) & 0x20;
}

void write_serial(char a) {
   while (is_transmit_empty() == 0);

   outb(PORT,a);
}
```

Glossary

Baud Rate

is the speed at which the serial line switches between it's two states. This is not equivalent to bps, due to the fact there are start and stop bits. On an 8/N/1 line, 10 baud = 1 byte. Modems are more complex than plain serial lines due to having multiple waveforms, but for the purposes of OSDev this is irrelevant.

baud rate divisor
: fastest rate a serial port can run, the number 115200.

stop bits
: the NULL bit(s) sent between each character to synchronize the transmitter and the receiver.

UART
: for Universal Asynchronous Receiver/Transceiver: the chip that picks a byte a send it bit per bit on the serial line and vice versa.

Related Links

- OSRC's "communication" section (http://www.nondot.org/sabre/os/articles/CommunicationDevices/)
- A summary of the PC serial port (with programming info). (http://www.sci.muni.cz/docs/pc/serport.txt) , An archive of that document (http://www.webcitation.org/5ugQv5JOw)
- A Wikibook about programming the serial port (http://en.wikibooks.org/wiki/Serial_Programming)

Categories: Network Hardware | Common Devices
