WIKIBOOKS

Serial Programming/8250 UART Programming

Serial Programming: Introduction and OSI Network Model -- RS-232 Wiring and Connections -- Typical RS232 Hardware Configuration -- 8250 UART -- DOS -- MAX232 Driver/Receiver Family -- TAPI Communications In Windows -- Linux and Unix -- Java -- Hayes-compatible Modems and AT Commands -- Universal Serial Bus (USB) -- Forming Data Packets -- Error Correction Methods -- Two Way Communication -- Packet Recovery Methods -- Serial Data Networks -- Practical Application Development -- IP Over Serial Connections

Contents

Introduction

8086 I/O ports
- Software I/O access
- x86 port I/O extensions

x86 Processor Interrupts
- IRQs Explained
- Interrupt handlers
- Software interrupts

8259 PIC (Programmable Interrupt Controller)
- 8259 Registers
- Device Registers
- ISR Cleanup
- PIC Device Masking

Serial COM Port Memory and I/O Allocation

UART Registers
- Transmitter Holding Buffer/Receiver Buffer
- Divisor Latch Bytes
- Interrupt Enable Register
- Interrupt Identification Register
- FIFO Control Register
- Line Control Register
- Modem Control Register
- Line Status Register
- Modem Status Register
- Scratch Register

Software Identification of the UART

External References

Other Serial Programming Articles

Introduction

Finally we are moving away from wires and voltages and hard-core electrical engineering applications, although we still need to know quite a bit regarding computer chip architectures at this level. While the primary focus of this section will concentrate on the 8250 UART, there are really three computer chips that we will be working with here:

- 8250 UART
- 8259 PIC (Programmable Interrupt Controller)
- 8086 CPU (Central Processing Unit)

Keep in mind that these are chip families, not simply the chip part number itself. Computer designs have evolved quite a bit over the years, and often all three chips are put onto the same piece of silicon because they are tied together so much, and to reduce overall costs of the equipment. So when I say 8086, I also mean the successor chips including the 80286, 80386, Pentium, and compatible chips made by manufacturers other than Intel. There are some subtle differences and things you need to worry about for serial data communication between the different chips other than the 8086, but in many cases you could in theory write software for the original IBM PC doing serial communication and it should run just fine on a modern computer you just bought that is running the latest version of Linux or Windows XP.

Modern operating systems handle most of the details that we will be covering here through low-level drivers, so this should be more of a quick understanding for how this works rather than something you might implement yourself, unless you are writing your own operating system. For people who are designing small embedded computer devices, it does become quite a bit more important to understand the 8250 at this level.

Just like the 8086, the 8250 has evolved quite a bit as well, e.g. into the 16550 UART. Further down I will go into how to detect many of the different UART chips on PCs, and some quirks or changes that affect each one. The differences really aren't as significant as the changes to CPU architecture, and the primary reason for updating the UART chip was to make it work with the considerably faster CPUs that are around right now. The 8250 itself simply can't keep up with a Pentium chip.

Remember as well that this is trying to build a foundation for serial programming on the software side. While this can be useful for hardware design as well, quite a bit will be missing from the descriptions here to implement a full system.

8086 I/O ports

We should go back even further than the Intel 8086, to the original Intel CPU, the 4004, and its successor, the 8008. All computer instructions, or op-codes, for the 8008 still function in today's Intel chips, so even port I/O tutorials written 30 years ago are valid today. The newer CPUs have enhanced instructions for dealing with more data more efficiently, but the original instructions are still there.

When the 8008 was released, Intel tried to devise a method for the CPU to communicate with external devices. They chose a method called I/O port architecture, meaning that the chip has a special set of pins dedicated to communicating with external devices. In the 8008, this meant that there were a total of sixteen (16) pins dedicated to communicating with the chip. The exact details varied based on chip design and other factors too detailed for the current discussion, but the general theory is fairly straightforward.

Eight of the pins represent an I/O code that signaled a specific device. This is known as the I/O port. Since this is just a binary code, it represents the potential to hook up 256 different devices to the CPU. It gets a little more complicated than that, but still you can think of it from software like a small-town post-office that has a bank of 256 PO boxes for its customers.

The next set of pins represent the actual data being exchanged. You can think of this as the postcards being put into or removed from the PO boxes.

All the external device has to do is look for its I/O code, and then when it matches what it is "assigned" to look for, it has control over the corresponding port. An pin signals whether the data is being sent to or from the CPU. For those familiar with setting up early PCs, this is also where I/O conflicts happen: when two or more devices try to access the same I/O port at the same time. This was a source of heartburn on those early systems, particularly when adding new equipment.

Incidentally, this is very similar to how conventional RAM works, and some CPU designs mimic this whole process straight in RAM, reserving a block of memory for I/O control. This has some problems,

including the fact that it chews up a portion of potential memory that could be used for software instead. It ends up that with the IBM PC and later PC systems, both Memory-mapped I/O (MMIO) and Port-mapped I/O (PMIO) are used extensively, so it really gets complicated. For serial communication, however, we are going to stick with the port I/O method, as that is how the 8250 chip works.

Software I/O access

When you get down to actually using this in your software, the assembly language instruction to send or receive data to port 9 looks something like this:

```
out 9, al ; sending data from register al out to port 9
in al, 9 ; getting data from port 9 and putting it in register al
```

When programming in higher level languages, it gets a bit simpler. A typical C language Port I/O library is usually written like this:

```
char test;

test = 255;
outp(9,test);
inp(9,&test);
```

For many versions of Pascal, it treats the I/O ports like a massive array that you can access, that is simply named Port:

```
procedure PortIO(var Test: Byte);
begin
  Port[9] := Test;
  Test := Port[9];
end;
```

Warning!! And this really is a warning. By randomly accessing I/O ports in your computer without really knowing what it is connected to can really mess up your computer. At the minimum, it will crash the operating system and cause the computer to not work. Writing to some I/O ports can permanently change the internal configuration of your computer, making a trip to the repair shop necessary just to undo the damage you've done through software. Worse yet, in some cases it can cause actual damage to the computer. This means that some chips inside the computer will no longer work and those components would have to be replaced in order for the computer to work again. Damaged chips are an indication of lousy engineering on the part of the computer, but unfortunately it does happen and you should be aware of it.

Don't be afraid to use the I/O ports, just make sure you know what you are writing to, and you know what equipment is "mapped" to for each I/O port if you intend to use a particular I/O port. We will get into more of the specifics for how to identify the I/O ports for serial communication in a bit. Finally we are starting to write a little bit of software, and there is more to come.

x86 port I/O extensions

There are a few differences between the 8088 CPU and the 8086. The most notable that affects software development is that instead of just 256 port I/O addresses, the 8086 can access 65536 different I/O ports. However, computer configurations may use less than 16 wires for the I/O address bus ; for example on the IBM PC, only 10 wires were used, making only 1024 different ports. Higher bits of the port number being ignored, this made multiple port number aliases for the same port.

In addition, besides simply sending a single character in or out, the 8086 will let you send and receive 16 bits at once. The 16-bit word bytes is read/written in little endian using consecutive port numbers. The 386 chips will even let you send and receive 32-bits simultaneously. The need for more than 65536 different I/O ports has never been a serious problem, and if a device needed a larger piece of memory, the Direct Memory Access (DMA) methods are available. This is where the device writes and reads the RAM of the computer directly instead of going through the CPU. We will not cover that topic here.

Also, while the 8086 CPU was able to address 65536 different I/O ports, in actual practice it didn't. The chip designers at Intel got cheap and only had address lines for 10 bits, which has implications for software designers having to work with legacy systems. This also meant that I/O port address $1E8 and $19E8 (and others... this is just an example) would resolve to the same I/O port for those early PCs. The Pentium CPUs don't have this limitation, but software written for some of that early hardware sometimes wrote to I/O port addresses that were "aliased" because those upper bits were ignored. There are other legacy issues that show up, but fortunately for the 8250 chip and serial communications in general this isn't a concern, unless you happen to have a serial driver that "took advantage" of this aliasing situation. This issue would generally only show up when you are using more than the typical 2 or 4 serial COM ports on a PC.

x86 Processor Interrupts

The 8086 CPU and compatible chips have what is known as an interrupt line. This is literally a wire to the rest of the computer that can be turned on to let the CPU know that it is time to stop whatever it is doing and pay attention to some I/O situations.

Within the 8086, there are two kinds of interrupts: Hardware interrupts and Software interrupts. There are some interesting quirks that are different from each kind, but from a software perspective they are essentially the same thing. The 8086 CPU allows for 256 interrupts, but the number available for equipment to perform a Hardware interrupt is considerably restricted.

IRQs Explained

Hardware interrupts are numbered IRQ 0 through IRQ 15. IRQ means Interrupt ReQuest. There are a total of fifteen different hardware interrupts. Before you think I don't know how to count or do math, we need to do a little bit of a history lesson here, which we will finish when we move on to the 8259 chip. When the original IBM-PC was built, it only had eight IRQs, labeled IRQ 0 through IRQ 7. At the time it was felt that was sufficient for almost everything that would ever be put on a PC, but very soon it became apparent it wasn't nearly enough for everything that was being added. When the IBM-PC/AT was made (the first one with the 80286 CPU, and a number of enhancements that are commonly found on PCs today), it was decided that instead of a single 8259 chip, they would use two of these same chips, and "chain" them to one another in order to expand the number of interrupts from 8 to 15. One IRQ had to be sacrificed in order to accomplish this task, and that was IRQ 2.

The point here is that if a device wants to notify the CPU that it has some data ready for the CPU, it sends a signal that it wants to stop whatever software is currently running on the computer and instead run a special "little" program called an interrupt handler. Once the interrupt handler is finished, the computer can go back to whatever it was doing before. If the interrupt handler is fast enough, you wouldn't even notice that the handler has even been used.

In fact, if you are reading this text on a PC, in the time that it takes for you to read this sentence several interrupt handlers have already been used by your computer. Every time that you use a keyboard or a mouse, or receive some data over the Internet, an interrupt handler has been used at some point in your computer to retrieve that information.

Interrupt handlers

We will be getting into specific details of interrupt handlers in a little bit, but now I want to explain just what they are. Interrupt handlers are a method of showing the CPU exactly what piece of software should be running when the interrupt is triggered.

The 8086 CPU has a portion of RAM that has been established that "points" to where the interrupt software is located elsewhere in RAM. The advantage of going this route is that the CPU only has to do a simple look-up to find just where the software is, and then transfers software execution to that point in RAM. This also allows you as a programmer to change where the CPU is "pointing" to in RAM, and instead of going to something in the operating system, you can customize the interrupt handler and put something else there yourself.

How this is best done depends largely on your operating system. For a simple operating system like MS-DOS, it actually encourages you to directly write these interrupt handlers, particularly when you are working with external peripherals. Other operating systems like Linux or MS-Windows use the approach of having a "driver" that hooks into these interrupt handlers or service routines, and then the application software deals with the drivers rather than dealing directly with the equipment. How a program actually does this is very dependent on the specific operating system you would be using. If you are instead trying to write your own operating system, you would have to write these interrupt handlers directly, and establish the protocol on how you access these handlers to send and retrieve data.

Software interrupts

Before we move on, I want to hit very briefly on software interrupts. Software interrupts are invoked with the 8086 assembly instruction "int", as in:

```
int $21
```

From the perspective of a software application, this is really just another way to call a subroutine, but with a twist. The "software" that is running in the interrupt handler doesn't have to be from the same application, or even made from the same compiler. Indeed, often these subroutines are written directly in assembly language. In the above example, this interrupt actually calls a "DOS" subroutine that will allow you to perform some sort of I/O access that is directly related to DOS. Depending on the values of the registers, usually the AX register in the 8086 in this case, it can determine just what information you want to get from DOS, such as the current time, date, disk size, and just about everything that normally you would associate with DOS. Compilers often hide these details, because setting up these interrupt routines can be a little tricky.

Now to really make a mess of things. "Hardware interrupts" can also be called from "software interrupts", and indeed this is a reasonable way to make sure you have written your software correctly. The difference here is that software interrupts will only be invoked, or have their portion of software code running in the CPU, if it has been explicitly called through this assembly opcode.

8259 PIC (Programmable Interrupt Controller)

The 8259 chip is the "heart" of the whole process of doing hardware interrupts. External devices are directly connected to this chip, or in the case of the PC-AT compatibles (most likely what you are most familiar with for a modern PC) it will have two of these devices that are connected together. Literally sixteen wires come into this pair of chips, each wire labeled IRQ-0 through IRQ-15.

The purpose of these chips is to help "prioritize" the interrupt signals and organize them in some orderly

fashion. There is no way to predict when a certain device is going to "request" an interrupt, so often multiple devices can be competing for attention from the CPU.

Generally speaking, the lower numbered IRQ gets priority. In other words, if both IRQ-1 and IRQ-4 are requesting attention at the same time, IRQ-1 gets priority and will be triggered first as far as the CPU is concerned. IRQ-4 has to wait until after IRQ-1 has completed its "Interrupt Service Routine" or ISR.

If the opposite happens however, with IRQ-4 doing its ISR (remember, this is software, just like any computer program you might normally write as a computer application), IRQ-1 will "interrupt" the ISR for IRQ-4 and push through its own ISR to be run instead, returning to the IRQ-4 ISR when it has finished. There are exceptions to this as well, but let's keep things simple at the moment.

Let's return for a minute to the original IBM-PC. When it was built, there was only one 8259 chip on the motherboard. When the IBM-AT came out the engineers at IBM decided to add a second 8259 chip to add some additional IRQ signals. Since there was still only 1 pin on the CPU (at this point the 80286) that could receive notification of an interrupt, it was decided to grab IRQ-2 from the original 8259 chip and use that to chain onto the next chip. IRQ-2 was re-routed to IRQ-9 as far as any devices that depended on IRQ-2. The nice thing about going with this scheme was that software that planned on something using IRQ-2 would still be "notified" when that device was used, even though seven other devices were now "sharing" this interrupt. These are IRQ-8 through IRQ-15.

What this means in terms of priorities, however, is that IRQ-8 through IRQ-15 have a higher priority than IRQ-3. This is mainly of concern when you are trying to sort out which device can take precedence over another, and how important it would be to notified when a piece of equipment is trying to get your attention. If you are dealing with software running a specific computer configuration, this priority level is very important.

It should be noted here that COM1 (serial communication channel one) usually uses IRQ-4, and COM2 uses IRQ-3, which has the net effect of making COM2 to be a higher priority for receiving data over COM1. Usually the software really doesn't care, but on some rare occasions you really need to know this fact.

8259 Registers

The 8259 has several "registers" that are associated with I/O port addresses. We will visit this concept a little bit more when we get to the 8250 chip. For a typical PC Computer system, the following are typical primary port addresses associated with the 8259:

Interrupt Controller Port I/O Addresses

Register Name	I/O Port
Master Interrupt Controller	$0020
Slave Interrupt Controller	$00A0

This primary port address is what we will use to directly communicate with the 8259 chip in our software. There are a number of commands that can be sent to this chip through these I/O port addresses, but for our purposes we really don't need to deal with them. Most of these are used to do the initial setup and configuration of the computer equipment by the Basic Input Output System (BIOS) of the computer, and unless you are rewriting the BIOS from scratch, you really don't have to worry about this. Also, each computer is a little different in its behavior when you are dealing with equipment at this level, so this is

something more for a computer manufacturer to worry about rather than something an application programmer should have to deal with, which is exactly why BIOS software is written at all.

Keep in mind that this is the "typical" Port I/O address for most PC-compatible type computer systems, and can vary depending on what the manufacturer is trying to accomplish. Generally you don't have to worry about incompatibility at this level, but when we get to Port I/O addresses for the serial ports this will become a much larger issue.

Device Registers

I'm going to spend a little time here to explain the meaning of the word register. When you are working with equipment at this level, the electrical engineers who designed the equipment refer to registers that change the configuration of the equipment. This can happen at several levels of abstraction, so I want to clear up some of the confusion.

A register is simply a small piece of RAM that is available for a device to directly manipulate. In a CPU like the 8086 or a Pentium, these are the memory areas that are used to directly perform mathematical operations like adding two numbers together. These usually go by names like AX, SP, etc. There are very few registers on a typical CPU because access to these registers is encoded directly into the basic machine-level instructions.

When we are talking about device register, keep in mind these are not the CPU registers, but instead memory areas on the devices themselves. These are often designed so they are connected to the Port I/O memory, so when you write to or read from the Port I/O addresses, you are directly accessing the device registers. Sometimes there will be a further level of abstraction, where you will have one Port I/O address that will indicate which register you are changing, and another Port I/O address that has the data you are sending to that register. How you deal with the device is based on how complex it is and what you are going to be doing.

In a real sense, they are registers, but keep in mind that often each of these devices can be considered a full computer in its own right, and all you are doing is establishing how it will be communicating with the main CPU. Don't get hung up here and get these confused with the CPU registers.

ISR Cleanup

One area that you have to interact on a regular basis when using interrupt controllers is to inform the 8259 PIC controller that the interrupt service routine is completed. When your software is performing an interrupt handler, there is no automated method for the CPU to signal to the 8259 chip that you have finished, so a specific "register" in the PIC needs to be set to let the next interrupt handler be able to access the computer system. Typical software to accomplish this is like the following:

```
   Port[$20] := $20;
```

This is sending the command called "End of Interrupt" or often written as an abbreviation simply "EOI". There are other commands that can be sent to this register, but for our purposes this is the only one that we need to concern ourselves with.

Now this will clear the "master" PIC, but if you are using a device that is triggered on the "slave" PIC, you also need to inform that chip as well that the interrupt service has been completed. This means you need to send "EOI" to that chip as well in a manner like this:

```
   Port[$A0] := $20;
```

```
   Port[$20] := $20;
```

There are other things you can do to make your computer system work smoothly, but let's keep things simple for now.

PIC Device Masking

Before we leave the subject of the 8259 PIC, I'd like to cover the concept of device masking. Each one of the devices that are attached to the PIC can be "turned on" or "turned off" from the viewpoint of how they can interrupt the CPU through the PIC chip. Usually as an application developer all we really care about is if the device is turned on, although if you are trying to isolate performance issues you might turn off some other devices. Keep in mind that if you turn a device "off", the interrupt will not work until it is turned back on. That can include the keyboard or other critical devices you may need to operate your computer.

The register to set this mask is called "Operation Control Word 1" or "OCW1". This is located at the PIC base address + 1, or for the "Master" PIC at Port I/O Address $21. This is where you need to go over bit manipulation, which I won't cover in detail here. The following tables show the related bits to change in order to enable or disable each of the hardware interrupt devices:

Master OCW1 ($21)

Bit	IRQ Enabled	Device Function
7	IRQ7	Parallel Port (LPT1)
6	IRQ6	Floppy Disk Controller
5	IRQ5	Reserved/Sound Card
4	IRQ4	Serial Port (COM1)
3	IRQ3	Serial Port (COM2)
2	IRQ2	Slave PIC
1	IRQ1	Keyboard
0	IRQ0	System Timer

Slave OCW1 ($A1)

Bit	IRQ Enabled	Device Function
7	IRQ15	Reserved
6	IRQ14	Hard Disk Drive
5	IRQ13	Math Co-Processor
4	IRQ12	PS/2 Mouse
3	IRQ11	PCI Devices
2	IRQ10	PCI Devices
1	IRQ9	Redirected IRQ2 Devices
0	IRQ8	Real Time Clock

Assuming that we want to turn on IRQ3 (typical for the serial port COM2), we would use the following software:

```
Port[$21] := Port[$21] and $F7; {Clearing bit 3 for enabling IRQ3}
```

And to turn it off we would use the following software:

```
Port[$21] := Port[$21] or $08; {Setting bit 3 for disabling IRQ3}
```

If you are having problems getting anything to work, you can simply send this command in your software:

```
Port[$21] := 0;
```

which will simply enable everything. This may not be a good thing to do, but will have to be something for you to experiment with depending on what you are working with. Try not to take short cuts like this as not only is it a sign of a lazy programmer, but it can have side effects that your computer may behave different than you intended. If you are working with the computer at this level, the goal is to change as little as possible so you don't cause damage to any other software you are using.

Serial COM Port Memory and I/O Allocation

Now that we have pushed through the 8259 chip, lets move on to the UART itself. While the Port I/O addresses for the PICs are fairly standard, it is common for computer manufacturers to move stuff around for the serial ports themselves. Also, if you have serial port devices that are part of an add-in card (like an ISA or PCI card in the expansion slots of your computer), these will usually have different settings than something built into the main motherboard of your computer. It may take some time to hunt down these settings, and it is important to know what these values are when you are trying to write your software. Often these values can be found in the BIOS setup screens of your computer, or if you can pause the messages when your computer turns on, they can be found as a part of the boot process of your computer.

For a "typical" PC system, the following are the Port I/O addresses and IRQs for each serial COM port:

Common UART IRQ and I/O Port Addresses

COM Port	IRQ	Base Port I/O address
COM1	IRQ4	$3F8
COM2	IRQ3	$2F8
COM3	IRQ4	$3E8
COM4	IRQ3	$2E8

If you notice something interesting here, you can see that COM3 and COM1 share the same interrupt. This is not a mistake but something you need to keep in mind when you are writing an interrupt service routine. The 15 interrupts that were made available through the 8259 PIC chips still have not been enough to allow all of the devices that are found on a modern computer to have their own separate hardware interrupt, so in this case you will need to learn how to share the interrupt with other devices. I'll cover more of that later when we get into the actual software to access the serial data ports, but for now remember not to write your software strictly for one device.

The Base Port I/O address is important for the next topic we will cover, which is directly accessing the UART registers.

UART Registers

The UART chip has a total of 12 different registers that are mapped into 8 different Port I/O locations. Yes, you read that correct, 12 registers in 8 locations. Obviously that means there is more than one register that uses the same Port I/O location, and affects how the UART can be configured. In reality, two of the registers are really the same one but in a different context, as the Port I/O address that you transmit the characters to be sent out of the serial data port is the same address that you can read in the characters that are sent to the computer. Another I/O port address has a different context when you write data to it than when you read data from it... and the number will be different after writing the data to it than when you read data from it. More on that in a little bit.

One of the issues that came up when this chip was originally being designed was that the designer needed to be able to send information about the baud rate of the serial data with 16 bits. This actually takes up two different "registers" and is toggled by what is called the "Divisor Latch Access Bit" or "DLAB". When the DLAB is set to "1", the baud rate registers can be set and when it is "0" the registers have a different context.

Does all this sound confusing? It can be, but lets take it one simple little piece at a time. The following is a table of each of the registers that can be found in a typical UART chip:

UART Registers

Base Address	DLAB	I/O Access	Abbrv.	Register Name
+0	0	Write	THR	Transmitter Holding Buffer
+0	0	Read	RBR	Receiver Buffer
+0	1	Read/Write	DLL	Divisor Latch Low Byte
+1	0	Read/Write	IER	Interrupt Enable Register
+1	1	Read/Write	DLH	Divisor Latch High Byte
+2	x	Read	IIR	Interrupt Identification Register
+2	x	Write	FCR	FIFO Control Register
+3	x	Read/Write	LCR	Line Control Register
+4	x	Read/Write	MCR	Modem Control Register
+5	x	Read	LSR	Line Status Register
+6	x	Read	MSR	Modem Status Register
+7	x	Read/Write	SR	Scratch Register

The "x" in the DLAB column means that the status of the DLAB has no effect on what register is going to be accessed for that offset range. Notice also that some registers are Read only. If you attempt to write data to them, you may end up with either some problems with the modem (worst case), or the data will simply be ignored (typically the result). As mentioned earlier, some registers share a Port I/O address where one register will be used when you write data to it and another register will be used to retrieve data from the same address.

Each serial communication port will have its own set of these registers. For example, if you wanted to access the Line Status Register (LSR) for COM1, and assuming the base I/O Port address of $3F8, the I/O Port address to get the information in this register would be found at $3F8 + $05 or $3FD. Some example code would be like this:

```
const
  COM1_Base = $3F8;
  COM2_Base = $2F8;
  LSR_Offset = $05;

function LSR_Value: Byte;
begin
  Result := Port[COM1_Base+LSR_Offset];
end;
```

There is quite a bit of information packed into each of these registers, and the following is an explanation for the meaning of each register and the information it contains.

Transmitter Holding Buffer/Receiver Buffer

Offset: +0 . The Transmit and Receive buffers are related, and often even use the very same memory. This is also one of the areas where later versions of the 8250 chip have a significant impact, as the later models incorporate some internal buffering of the data within the chip before it gets transmitted as serial data. The base 8250 chip can only receive one byte at a time, while later chips like the 16550 chip will hold up to 16 bytes either to transmit or to receive (sometimes both... depending on the manufacturer) before you have to wait for the character to be sent. This can be useful in multi-tasking environments where you have a computer doing many things, and it may be a couple of milliseconds before you get back to dealing with serial data flow.

These registers really are the "heart" of serial data communication, and how data is transferred from your software to another computer and how it gets data from other devices. Reading and Writing to these registers is simply a matter of accessing the Port I/O address for the respective UART.

If the receive buffer is occupied or the FIFO is full, the incoming data is discarded and the Receiver Line Status interrupt is written to the IIR register. The Overrun Error bit is also set in the Line Status Register.

Divisor Latch Bytes

Offset: +0 and +1 . The Divisor Latch Bytes are what control the baud rate of the modem. As you might guess from the name of this register, it is used as a divisor to determine what baud rate that the chip is going to be transmitting at.

In reality, it is even simpler than that. This is really a count-down clock that is used each time a bit is transmitted by the UART. Each time a bit is sent, a count-down register is reset to this value and then counts down to zero. This clock is running typically at 115.2 kHz. In other words, at 115 thousand times per second a counter is going down to determine when to send the next bit. At one time during the design process it was anticipated that some other frequencies might be used to get a UART working, but with the large amount of software already written for this chip this frequency is pretty much standard for almost all UART chips used on a PC platform. They may use a faster clock in some portion (like a 1.843 MHz clock), but some fraction of that frequency will then be used to scale down to a 115.2 kHz clock.

Some more on UART clock speeds (advanced coverage): For many UART chips, the clock frequency that is driving the UART is 1.8432 MHz. This frequency is then put through a divider circuit that drops the frequency down by a factor of 16, giving us the 115.2 KHz frequency mentioned above. If you are doing some custom equipment using this chip, the National Semiconductor spec sheets allow for a 3.072 MHz clock and 18.432 MHz clock. These higher frequencies will allow you to communicate at higher baud rates, but require custom circuits on the motherboard and often new drivers in order to deal with these new frequencies. What is interesting is that you can still operate at 50 baud with these higher clock frequencies, but at the time the original IBM-PC/XT was manufactured this wasn't a big concern as it is

now for higher data throughput.

If you use the following mathematical formula, you can determine what numbers you need to put into the Divisor Latch Bytes:

$$DivisorLatchValue = \frac{115200}{BaudRate}$$

That gives you the following table that can be used to determine common baud rates for serial communication:

Divisor Latch Byte Values (common baud rates)

Baud Rate	Divisor (in decimal)	Divisor Latch High Byte	Divisor Latch Low Byte
50	2304	$09	$00
110	1047	$04	$17
220	524	$02	$0C
300	384	$01	$80
600	192	$00	$C0
1200	96	$00	$60
2400	48	$00	$30
4800	24	$00	$18
9600	12	$00	$0C
19200	6	$00	$06
38400	3	$00	$03
57600	2	$00	$02
115200	1	$00	$01

One thing to keep in mind when looking at the table is that baud rates 600 and above all set the Divisor Latch High Byte to zero. A sloppy programmer might try to skip setting the high byte, assuming that nobody would deal with such low baud rates, but this is not something to always presume. Good programming habits suggest you should still try to set this to zero even if all you are doing is running at higher baud rates.

Another thing to notice is that there are other potential baud rates other than the standard ones listed above. While this is not encouraged for a typical application, it would be something fun to experiment with. Also, you can attempt to communicate with older equipment in this fashion where a standard API library might not allow a specific baud rate that should be compatible. This should demonstrate why knowledge of these chips at this level is still very useful.

When working with these registers, also remember that these are the only ones that require the Divisor Latch Access Bit to be set to "1". More on that below, but I'd like to mention that it would be useful for application software setting the baud rate to set the DLAB to "1" just for the immediate operation of changing the baud rate, then putting it back to "0" as the very next step before you do any more I/O access to the modem. This is just a good working habit, and keeps the rest of the software you need to write for accessing the UART much cleaner and easier.

One word of caution: Do not set the value "0" for both Divisor Latch bytes. While it will not (likely)

damage the UART chip, the behavior on how the UART will be transmitting serial data will be unpredictable, and will change from one computer to the next, or even from one time you boot the computer to the next. This is an error condition, and if you are writing software that works with baud rate settings on this level you should catch potential "0" values for the Divisor Latch.

Here is some sample software to set and retrieve the baud rate for COM1:

```
const
  COM1_Base = $3F8;
  COM2_Base = $2F8;
  LCR_Offset = $03;
  Latch_Low = $00;
  Latch_High = $01;

procedure SetBaudRate(NewRate: Word);
var
  DivisorLatch: Word;
begin
  DivisorLatch := 115200 div NewRate;
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] or $80; {Set DLAB}
  Port[COM1_Base + Latch_High] := DivisorLatch shr 8;
  Port[COM1_Base + Latch_Low] := DivisorLatch and $FF;
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] and $7F; {Clear DLAB}
end;

function GetBaudRate: Integer;
var
  DivisorLatch: Word;
begin
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] or $80; {Set DLAB}
  DivisorLatch := (Port[COM1_Base + Latch_High] shl 8) + Port[COM1_Base + Latch_Low];
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] and $7F; {Clear DLAB}
  Result := 115200 div DivisorLatch;
end;
```

Interrupt Enable Register

Offset: +1 . This register allows you to control when and how the UART is going to trigger an interrupt event with the hardware interrupt associated with the serial COM port. If used properly, this can enable an efficient use of system resources and allow you to react to information being sent across a serial data line in essentially real-time conditions. Some more on that will be covered later, but the point here is that you can use the UART to let you know exactly when you need to extract some data. This register has both read- and write-access.

The following is a table showing each bit in this register and what events that it will enable to allow you check on the status of this chip:

Interrupt Enable Register (IER)

Bit	Notes
7	Reserved
6	Reserved
5	Enables Low Power Mode (16750)
4	Enables Sleep Mode (16750)
3	Enable Modem Status Interrupt
2	Enable Receiver Line Status Interrupt
1	Enable Transmitter Holding Register Empty Interrupt

0	Enable Received Data Available Interrupt

The Received Data interrupt is a way to let you know that there is some data waiting for you to pull off of the UART. This is probably the one bit that you will use more than the rest, and has more use.

The Transmitter Holding Register Empty Interrupt is to let you know that the output buffer (on more advanced models of the chip like the 16550) has finished sending everything that you pushed into the buffer. This is a way to streamline the data transmission routines so they take up less CPU time.

The Receiver Line Status Interrupt indicates that something in the LSR register has probably changed. This is usually an error condition, and if you are going to write an efficient error handler for the UART that will give plain text descriptions to the end user of your application, this is something you should consider. This is certainly something that takes a bit more advanced knowledge of programming.

The Modem Status Interrupt is to notify you when something changes with an external modem connected to your computer. This can include things like the telephone "bell" ringing (you can simulate this in your software), that you have successfully connected to another modem (Carrier Detect has been turned on), or that somebody has "hung up" the telephone (Carrier Detect has turned off). It can also help you to know if the external modem or data equipment can continue to receive data (Clear to Send). Essentially, this deals with the other wires in the RS-232 standard other than strictly the transmit and receive wires.

The other two modes are strictly for the 16750 chip, and help put the chip into a "low power" state for use on things like a laptop computer or an embedded controller that has a very limited power source like a battery. On earlier chips you should treat these bits as "Reserved", and only put a "0" into them.

Interrupt Identification Register

Offset: +2 . This register is to be used to help identify what the unique characteristics of the UART chip that you are using has. This register has two uses:

- Identification of why the UART triggered an interrupt.
- Identification of the UART chip itself.

Of these, identification of why the interrupt service routine has been invoked is perhaps the most important.

The following table explains some of the details of this register, and what each bit on it represents:

Interrupt Identification Register (IIR)

Bit	Notes		
7 and 6	**Bit 7**	**Bit 6**	
	0	0	No FIFO on chip
	0	1	Reserved condition
	1	0	FIFO enabled, but not functioning
	1	1	FIFO enabled
5	64 Byte FIFO Enabled (16750 only)		
4	Reserved		

	Bit 3	Bit 2	Bit 1		Reset Method
3, 2 and 1	0	0	0	Modem Status Interrupt	Reading Modem Status Register(MSR)
	0	0	1	Transmitter Holding Register Empty Interrupt	Reading Interrupt Identification Register(IIR) or Writing to Transmit Holding Buffer(THR)
	0	1	0	Received Data Available Interrupt	Reading Receive Buffer Register(RBR)
	0	1	1	Receiver Line Status Interrupt	Reading Line Status Register(LSR)
	1	0	0	Reserved	N/A
	1	0	1	Reserved	N/A
	1	1	0	Time-out Interrupt Pending (16550 & later)	Reading Receive Buffer Register(RBR)
	1	1	1	Reserved	N/A
0	Interrupt Pending Flag				

When you are writing an interrupt handler for the 8250 chip (and later), this is the register that you need to look at in order to determine what exactly was the trigger for the interrupt.

As explained earlier, multiple serial communication devices can share the same hardware interrupt. The use of "Bit 0" of this register will let you know (or confirm) that this was indeed the device that caused the interrupt. What you need to do is check on all serial devices (that are in separate port I/O address spaces), and get the contents of this register. Keep in mind that it is at least possible for more than one device to trigger an interrupt at the same time, so when you are doing this scanning of serial devices, make sure you examine all of them, even one of the first devices did in fact need to be processed. Some computer systems may not require this to occur, but this is a good programming practice anyway. It is also possible that due to how you processed the UARTs earlier, that you have already dealt with all of the UARTs for a given interrupt. When this bit is a "0", it identifies that the UART is triggering an interrupt. When it is "1", that means the interrupt has already been processed or this particular UART was not the triggering device. I know that this seems a little bit backward for a typical bit-flag used in computers, but this is called digital logic being asserted low, and is fairly common with electrical circuit design. This is a bit more unusual through for this logic pattern to go into the software domain.

Bits 1, 2 & 3 help to identify exactly what sort of interrupt event was used within the UART to invoke the hardware interrupt. These are the same interrupts that were earlier enabled with the IER register. In this case, however, each time you process the registers and deal with the interrupt it will be unique. If multiple "triggers" occur for the UART due to many things happening at the same time, this will be invoked through multiple hardware interrupts. Earlier chip sets don't use bit 3, but this is a reserved bit on those UART systems and always set to logic state "0", so programming logic doesn't have to be different when trying to decipher which interrupt has been used.

To explain the FIFO timeout Interrupt, this is a way to check for the end of a packet or if the incoming data stream has stopped. Generally the following conditions must exist for this interrupt to be triggered: Some data needs to be in the incoming FIFO and has not been read by the computer. Data transmissions being sent to the UART via serial data link must have ended with no new characters being received. The CPU processing incoming data must not have retrieved any data from the FIFO before the timeout has occurred. The timeout will occur usually after the period it would take to transmit or receive at least 4

characters. If you are talking about data sent at 1200 baud, 8 data bits, 2 stop bits, odd parity, that would take about 40 milliseconds, which is almost an eternity in terms of things that your computer can accomplish on a 4 GHz Pentium CPU.

The "Reset Method" listed above describes how the UART is notified that a given interrupt has been processed. When you access the register mentioned under the reset method, this will clear the interrupt condition for that UART. If multiple interrupts for the same UART have been triggered, either it won't clear the interrupt signal on the CPU (triggering a new hardware interrupt when you are done), or if you check back to this register (IIR) and query the Interrupt Pending Flag to see if there are more interrupts to process, you can move on and attempt to resolve any new interrupt issue that you may have to deal with, using appropriate application code.

Bits 5, 6 & 7 are reporting the current status of FIFO buffers being used for transmitting and receiving characters. There was a bug in the original 16550 chip design when it was first released that had a serious flaw in the FIFO, causing the FIFO to report that it was working but in fact it wasn't. Because some software had already been written to work with the FIFO, this bit (Bit 7 of this register) was kept, but Bit 6 was added to confirm that the FIFO was in fact working correctly, in case some new software wanted to ignore the hardware FIFO on the earlier versions of the 16550 chip. This pattern has been kept on future versions of this chip as well. On the 16750 chip an added 64-byte FIFO has been implemented, and Bit 5 is used to designate the presence of this extended buffer. These FIFO buffers can be turned on and off using registers listed below.

FIFO Control Register

Offset: +2 . This is a relatively "new" register that was not a part of the original 8250 UART implementation. The purpose of this register is to control how the First In/First Out (FIFO) buffers will behave on the chip and to help you fine-tune their performance in your application. This even gives you the ability to "turn on" or "turn off" the FIFO.

Keep in mind that this is a "write only" register. Attempting to read in the contents will only give you the Interrupt Identification Register (IIR), which has a totally different context.

FIFO Control Register (FCR)

Bit	Notes			
7 & 6	**Bit 7**	**Bit 6**	**Interrupt Trigger Level (16 byte)**	**Trigger Level (64 byte)**
	0	0	1 Byte	1 Byte
	0	1	4 Bytes	16 Bytes
	1	0	8 Bytes	32 Bytes
	1	1	14 Bytes	56 Bytes
5	Enable 64 Byte FIFO (16750)			
4	Reserved			
3	DMA Mode Select			
2	Clear Transmit FIFO			
1	Clear Receive FIFO			
0	Enable FIFOs			

Writing a "0" to bit 0 will disable the FIFOs, in essence turning the UART into 8250 compatibility mode.

In effect this also renders the rest of the settings in this register to become useless. If you write a "0" here it will also stop the FIFOs from sending or receiving data, so any data that is sent through the serial data port may be scrambled after this setting has been changed. It would be recommended to disable FIFOs only if you are trying to reset the serial communication protocol and clearing any working buffers you may have in your application software. Some documentation suggests that setting this bit to "0" also clears the FIFO buffers, but I would recommend explicit buffer clearing instead using bits 1 and 2.

Bits 1 and 2 are used to clear the internal FIFO buffers. This is useful when you are first starting up an application where you might want to clear out any data that may have been "left behind" by a previous piece of software using the UART, or if you want to reset a communications connection. These bits are "automatically" reset, so if you set either of these to a logical "1" state you will not have to go and put them back to "0" later. Sending a logical "0" only tells the UART not to reset the FIFO buffers, even if other aspects of FIFO control are going to be changed.

Bit 3 is in reference to how the DMA (Direct Memory Access) takes place, primarily when you are trying to retrieve data from the FIFO. This would be useful primarily to a chip designer who is trying to directly access the serial data, and store this data in an internal buffer. There are two digital logic pins on the UART chip itself labeled RXRDY and TXRDY. If you are trying to design a computer circuit with the UART chip this may be useful or even important, but for the purposes of an application developer on a PC system it is of little use and you can safely ignore it.

Bit 5 allows the 16750 UART chip to expand the buffers from 16 bytes to 64 bytes. Not only does this affect the size of the buffer, but it also controls the size of the trigger threshold, as described next. On earlier chip types this is a reserved bit and should be kept in a logical "0" state. On the 16750 it make that UART perform more like the 16550 with only a 16 byte FIFO.

Bits 6 and 7 describe the trigger threshold value. This is the number of characters that would be stored in the FIFO before an interrupt is triggered that will let you know data should be removed from the FIFO. If you anticipate that large amounts of data will be sent over the serial data link, you might want to increase the size of the buffer. The reason why the maximum value for the trigger is less than the size of the FIFO buffer is because it may take a little while for some software to access the UART and retrieve the data. Remember that when the FIFO is full, you will start to lose data from the FIFO, so it is important to make sure you have retrieved the data once this threshold has been reached. If you are encountering software timing problems in trying to retrieve the UART data, you might want to lower the threshold value. At the extreme end where the threshold is set to 1 byte, it will act essentially like the basic 8250, but with the added reliability that some characters may get caught in the buffer in situations where you don't have a chance to get all of them immediately.

Line Control Register

Offset: +3 . This register has two major purposes:

- Setting the Divisor Latch Access Bit (DLAB), allowing you to set the values of the Divisor Latch Bytes.
- Setting the bit patterns that will be used for both receiving and transmitting the serial data. In other words, the serial data protocol you will be using (8-1-None, 5-2-Even, etc.).

Line Control Register (LCR)

Bit	Notes

7	Divisor Latch Access Bit			
6	Set Break Enable			
3, 4 & 5	**Bit 5**	**Bit 4**	**Bit 3**	**Parity Select**
	0	0	0	No Parity
	0	0	1	Odd Parity
	0	1	1	Even Parity
	1	0	1	Mark
	1	1	1	Space
2	0	One Stop Bit		
	1	1.5 Stop Bits or 2 Stop Bits		
0 & 1	**Bit 1**	**Bit 0**	**Word Length**	
	0	0	5 Bits	
	0	1	6 Bits	
	1	0	7 Bits	
	1	1	8 Bits	

The first two bits (Bit 0 and Bit 1) control how many data bits are sent for each data "word" that is transmitted via serial protocol. For most serial data transmission, this will be 8 bits, but you will find some of the earlier protocols and older equipment that will require fewer data bits. For example, some military encryption equipment only uses 5 data bits per serial "word", as did some TELEX equipment. Early ASCII teletype terminals only used 7 data bits, and indeed this heritage has been preserved with SMTP format that only uses 7-bit ASCII for e-mail messages. Clearly this is something that needs to be established before you are able to successfully complete message transmission using RS-232 protocol.

Bit 2 controls how many stop bits are transmitted by the UART to the receiving device. This is selectable as either one or two stop bits, with a logical "0" representing 1 stop bit and "1" representing 2 stop bits. In the case of 5 data bits, the UART instead sends out "1.5 stop bits". Remember that a 'bit' in this context is actually a time interval: at 50 baud (bits per second) each bit takes 20 ms. So "1.5 stop bits" would have a minimum of 30 ms between characters. This is tied to the "5 data bits" setting, since only the equipment that used 5-bit Baudot rather than 7- or 8-bit ASCII used "1.5 stop bits".

Another thing to keep in mind is that the RS-232 standard only specifies that at least one data bit cycle will be kept a logical "1" at the end of each serial data word (in other words, a complete character from start bit, data bits, parity bits, and stop bits). If you are having timing problems between the two computers but are able to in general get the character sent across one at a time, you might want to add a second stop bit instead of reducing baud rate. This adds a one-bit penalty to the transmission speed per character instead of halving the transmission speed by dropping the baud rate (usually).

Bits 3, 4, and 5 control how each serial word responds to parity information. When Bit 3 is a logical "0", this causes no parity bits to be sent out with the serial data word. Instead it moves on immediately to the stop bits, and is an admission that parity checking at this level is really useless. You might still gain a little more reliability with data transmission by including the parity bits, but there are other more reliable and practical ways that will be discussed in other chapters in this book. If you want to include parity checking, the following explains each parity method other than "none" parity:

Odd Parity

Each bit in the data portion of the serial word is added as a simple count of the number of logical "1" bits. If this is an odd number of bits, the parity bit will be transmitted as a logical "0". If the count is even, the parity bit will be

transmitted as a logical "1" to make the number of "1" bits odd.

Even Parity
Like Odd Parity, the bits are added together. In this case, however, if the number of bits end up as an odd number it will be transmitted as a logical "1" to make the number of "1" bits even, which is the exact opposite of odd parity.

Mark Parity
In this case the parity bit will always be a logical "1". While this may seem a little unusual, this is put in for testing and diagnostics purposes. If you want to make sure that the software on the receiving end of the serial connection is responding correctly to a parity error, you can send a Mark or a Space parity, and send characters that don't meet what the receiving UART or device is expecting for parity. In addition for Mark Parity only, you can use this bit as an extra "stop bit". Keep in mind that RS-232 standards are expecting a logical "1" to end a serial data word, so a receiving computer will not be able to tell the difference between a "Mark" parity bit and a stop bit. In essence, you can have 3 or 2.5 stop bits through the use of this setting and by appropriate use of the stop bit portion of this register as well. This is a way to "tweak" the settings on your computer in a way that typical applications don't allow you to do, or at least gain a deeper insight into serial data settings.

Space Parity
Like the Mark parity, this makes the parity bit "sticky", so it doesn't change. In this case it puts in a logical "0" for the parity bit every time you transmit a character. There are not many practical uses for doing this other than a crude way to put in 9 data bits for each serial word, or for diagnostics purposes as described above.

Bit 6, when set to 1, causes TX wire to go logical "0" and stay that way, which is interpreted as long stream of "0" bits by the receiving UART - the "break condition". To end the "break", set bit 6 back to 0.

Modem Control Register

Offset: +4 . This register allows you to do "hardware" flow control, under software control. Or in a more practical manner, it allows direct manipulation of four different wires on the UART that you can set to any series of independent logical states, and be able to offer control of the modem. It should also be noted that most UARTs need Auxiliary Output 2 set to a logical "1" to enable interrupts.

Modem Control Register (MCR)

Bit	Notes
7	Reserved
6	Reserved
5	Autoflow Control Enabled (16750)
4	Loopback Mode
3	Auxiliary Output 2
2	Auxiliary Output 1
1	Request To Send

0	Data Terminal Ready

Of these outputs on a typical PC platform, only the Request to Send (RTS) and Data Terminal Ready (DTR) are actually connected to the output of the PC on the DB-9 connector. If you are fortunate to have a DB-25 serial connector (more commonly used for parallel communications on a PC platform), or if you have a custom UART on an expansion card, the auxiliary outputs might be connected to the RS-232 connection. If you are using this chip as a component on a custom circuit, this would give you some "free" extra output signals you can use in your chip design to signal anything you might want to have triggered by a TTL output, and would be under software control. There are easier ways to do this, but in this case it might save you an extra chip on your layout.

The "loopback" mode is primarily a way to test the UART to verify that the circuits are working between your main CPU and the UART. This seldom, if ever, needs to be tested by an end user, but might be useful for some initial testing of some software that uses the UART. When this is set to a logical state of "1", any character that gets put into the transmit register will immediately be found in the receive register of the UART. Other logical signals like the RTS and DTS listed above will show up in the modem status register just as if you had put a loopback RS-232 device on the end of your serial communication port. In short, this allows you to do a loopback test using just software. Except for these diagnostics purposes and for some early development testing of software using the UART, this will never be used.

On the 16750 there is a special mode that can be invoked using the Modem Control Register. Basically this allows the UART to directly control the state of the RTS and DTS for hardware character flow control, depending on the current state of the FIFO. This behavior is also affected by the status of Bit 5 of the FIFO Control Register (FCR). While this is useful, and can change some of the logic on how you would write UART control software, the 16750 is comparatively new as a chip and not commonly found on many computer systems. If you know your computer has a 16750 UART, have fun taking advantage of this increased functionality.

Line Status Register

Offset: +5 . This register is used primarily to give you information on possible error conditions that may exist within the UART, based on the data that has been received. Keep in mind that this is a "read only" register, and any data written to this register is likely to be ignored or worse, cause different behavior in the UART. There are several uses for this information, and some information will be given below on how it can be useful for diagnosing problems with your serial data connection:

Line Status Register (LSR)

Bit	Notes
7	Error in Received FIFO
6	Empty Data Holding Registers
5	Empty Transmitter Holding Register
4	Break Interrupt
3	Framing Error
2	Parity Error
1	Overrun Error
0	Data Ready

Bit 7 refers to errors that are with characters in the FIFO. If any character that is currently in the FIFO has

had one of the other error messages listed here (like a framing error, parity error, etc.), this is reminding you that the FIFO needs to be cleared as the character data in the FIFO is unreliable and has one or more errors. On UART chips without a FIFO this is a reserved bit field.

Bits 5 and 6 refer to the condition of the character transmitter circuits and can help you to identify if the UART is ready to accept another character. Bit 6 is set to a logical "1" if all characters have been transmitted (including the FIFO, if active), and the "shift register" is done transmitting as well. This shift register is an internal memory block within the UART that grabs data from the Transmitter Holding Buffer (THB) or the FIFO and is the circuitry that does the actual transformation of the data to a serial format, sending out one bit of the data at a time and "shifting" the contents of the shift register down one bit to get the value of the next bit. Bit 5 merely tells you that the UART is capable of receiving more characters, including into the FIFO for transmitting.

The Break Interrupt (Bit 4) gets to a logical state of "1" when the serial data input line has received "0" bits for a period of time that is at least as long as an entire serial data "word", including the start bit, data bits, parity bit, and stop bits, for the given baud rate in the Divisor Latch Bytes. (The normal state of a serial line is to send "1" bits when idle, or send start bit which is always one "0" bit, then send variable data and parity bits, then stop bit which is "1", continued into more "1"s if line goes idle.) A long sequence of "0" bits instead of the normal state usually means that the device that is sending serial data to your computer has stopped for some reason. Often with serial communications this is a normal condition, but in this way you have a way to monitor just how the other device is functioning. Some serial terminals have a key which make them generate this "break condition" as an out-of-band signaling method.

Framing errors (Bit 3) occur when the last bit is not a stop bit. Or to be more precise the stop bit is a logical "0". There are several causes for this, including that you have the timing between the two computer mismatched. This is usually caused by a mismatch in baud rate, although other causes might be involved as well, including problems in the physical cabling between the devices or that the cable is too long. You may even have the number of data bits off, so when errors like this are encountered, check the serial data protocol very closely to make sure that all of the settings for the UART (data bit length, parity, and stop bit count) are what should be expected.

Parity errors (Bit 2) can also indicate a mismatched baud rate like the framing errors (particularly if both errors are occurring at the same time). This bit is raised when the parity algorithm that is expected (odd, even, mark, or space) has not been found. If you are using "no parity" in the setup of the UART, this bit should always be a logical "0". When framing errors are not occurring, this is a way to identify that there are some problems with the cabling, although there are other issues you may have to deal with as well.

Overrun errors (Bit 1) are a sign of poor programming or an operating system that is not giving you proper access to the UART. This error condition occurs when there is a character waiting to be read, and the incoming shift register is attempting to move the contents of the next character into the Receiver Buffer (RBR). On UARTs with a FIFO, this also indicates that the FIFO is full as well.

Some things you can do to help get rid of this error include looking at how efficient your software is that is accessing the UART, particularly the part that is monitoring and reading incoming data. On multi-tasking operating systems, you might want to make sure that the portion of the software that reads incoming data is on a separate thread, and that the thread priority is high or time-critical, as this is a very important operation for software that uses serial communications data. A good software practice for applications also includes adding in an application specific "buffer" that is done through software, giving your application more opportunity to be able to deal with the incoming data as necessary, and away from the time critical subroutines needed to get the data off of the UART. This buffer can be as small as 1KB to as large as 1MB, and depends substantially on the kind of data that you are working with. There are other more exotic buffering techniques as well that apply to the realm of application development, and that will be covered in later modules.

If you are working with simpler operating systems like MS-DOS or a real-time operating system, there is a distinction between a poll-driven access to the UART vs. interrupt driven software. Writing an interrupt driver is much more efficient, and there will be a whole section of this book that will go into details of how to write software for UART access.

Finally, when you can't seem to solve the problems of trying to prevent overrun errors from showing up, you might want to think about reducing the baud rate for the serial transmission. This is not always an option, and really should be the option of last choice when trying to resolve this issue in your software. As a quick test to simply verify that the fundamental algorithms are working, you can start with a slower baud rate and gradually go to higher speeds, but that should only be done during the initial development of the software, and not something that gets released to a customer or placed as publicly distributed software.

The Data Ready Bit (Bit 0) is really the simplest part here. This is a way to simply inform you that there is data available for your software to extract from the UART. When this bit is a logical "1", it is time to read the Receiver Buffer (RBR). On UARTs with a FIFO that is active, this bit will remain in a logical "1" state until you have read all of the contents of the FIFO.

Modem Status Register

Offset: +6 . This register is another read-only register that is here to inform your software about the current status of the modem. The modem accessed in this manner can either be an external modem, or an internal modem that uses a UART as an interface to the computer.

Modem Status Register (MSR)

Bit	Notes
7	Carrier Detect
6	Ring Indicator
5	Data Set Ready
4	Clear To Send
3	Delta Data Carrier Detect
2	Trailing Edge Ring Indicator
1	Delta Data Set Ready
0	Delta Clear To Send

Bits 7 and 6 are directly related to modem activity. Carrier Detect will stay in a logical state of "1" while the modem is "connect" to another modem. When this goes to a logical state of "0", you can assume that the phone connection has been lost. The Ring Indicator bit is directly tied to the RS-232 wire also labeled "RI" or Ring Indicator. Usually this bit goes to a logical state of "1" as a result of the "ring voltage" on the telephone line is detected, like when a conventional telephone will be ringing to inform you that somebody is trying to call you.

When we get to the section of AT modem commands, there will be other methods that can be shown to inform you about this and other information regarding the status of a modem, and instead this information will be sent as characters in the normal serial data stream instead of special wires. In truth, these extra bits are pretty worthless, but have been a part of the specification from the beginning and comparatively easy for UART designers to implement. It may, however, be a way to efficiently send some additional

information or allow a software designer using the UART to get some logical bit signals from other devices for other purposes.

The "Data Set Ready" and "Clear To Send" bits (Bits 4 and 5) are found directly on an RS-232 cable, and are matching wires to "Request To Send" and "Data Terminal Ready" that are transmitted with the "Modem Control Register (MCR). With these four bits in two registers, you can perform "hardware flow control", where you can signal to the other device that it is time to send more data, or to hold back and stop sending data while you are trying to process the information. More will be written about this subject in another module when we get to data flow control.

A note regarding the "delta" bits (Bits 0, 1, 2, and 3). In this case the word "delta" means change, as in a change in the status of one of the bits. This comes from other scientific areas like rocket science where delta-vee means a change in velocity. For the purposes of this register, each of these bits will be a logical "1" the next time you access this Modem Status register if the bit it is associated with (like Delta Data Carrier Detect with Carrier Detect) has changed its logical state from the previous time you accessed this register. The Trailing Edge Ring Indicator is pretty much like the rest, except it is in a logical "1" state only if the "Ring Indicator" bit went from a logical "1" to a logical "0" condition. There really isn't much practical use for this knowledge, but there is some software that tries to take advantage of these bits and perform some manipulation of the data received from the UART based on these bits. If you ignore these 4 bits you can still make a very robust serial communications software.

Scratch Register

Offset: +7 . The Scratch Register is an interesting enigma. So much effort was done to try and squeeze a whole bunch of registers into all of the other I/O port addresses that the designers had an extra "register" that they didn't know what to do with. Keep in mind that when dealing with computer architecture, it is easier when dealing with powers of 2, so they were "stuck" with having to address 8 I/O ports. Allowing another device to use this extra I/O port would make the motherboard design far too complicated.

On some variants of the 8250 UART, any data written to this scratch register will be available to software when you read the I/O port for this register. In effect, this gives you one extra byte of "memory" that you can use in your applications in any way that you find useful. Other than a virus author (maybe I shouldn't give any ideas), there isn't really a good use for this register. Of limited use is the fact that you can use this register to identify specific variations of the UART because the original 8250 did not store the data sent to it through this register. As that chip is hardly ever used anymore on a PC design (those companies are using more advanced chips like the 16550), you will not find that "bug" in most modern PC-type platforms. More details will be given below on how to identify through software which UART chip is being used in your computer, and for each serial port.

Software Identification of the UART

Just as it is possible to identify many of the components on a computer system through just software routines, it is also possible to detect which version or variant of the UART that is found on your computer as well. The reason this is possible is because each different version of the UART chip has some unique qualities that if you do a process of elimination you can identify which version you are dealing with. This can be useful information if you are trying to improve performance of the serial I/O routines, know if there are buffers available for transmitting and sending information, as well as simply getting to know the equipment on your PC better.

One example of how you can determine the version of the UART is if the Scratch Register is working or not. On the first 8250 and 8250A chips, there was a flaw in the design of those chip models where the Scratch Register didn't work. If you write some data to this register and it comes back changed, you know

that the UART in your computer is one of these two chip models.

Another place to look is with the FIFO control registers. If you set bit "0" of this register to a logical **1**, you are trying to enable the FIFOs on the UART, which are only found in the more recent version of this chip. Reading bits "6" and "7" will help you to determine if you are using either the 16550 or 16550A chip. Bit "5" will help you determine if the chip is the 16750.

Below is a full pseudo code algorithm to help you determine the type of chip you are using:

```
Set the value "0xE7" to the FCR to test the status of the FIFO flags.
Read the value of the IIR to test for what flags actually got set.
If Bit 6 is set Then
  If Bit 7 is set Then
    If Bit 5 is set Then
      UART is 16750
    Else
      UART is 16550A
    End If
  Else
    UART is 16550
  End If
Else you know the chip doesn't use FIFO, so we need to check the scratch register
  Set some arbitrary value like 0x2A to the Scratch Register.
  You don't want to use 0xFF or 0x00 as those might be returned by the Scratch Register instead for a
false postive result.
  Read the value of the Scratch Register
  If the arbitrary value comes back identical
    UART is 16450
  Else
    UART is 8250
  End If
End If
```

When written in Pascal, the above algorithm ends up looking like this:

```
const
  COM1_Addr = $3F8;
  FCR = 2;
  IIR = 2;
  SCR = 7;

function IdentifyUART: String;
var
  Test: Byte;
begin
  Port[COM1_Addr + FCR] := $E7;
  Test := Port[COM1_Addr + IIR];
  if (Test and $40) > 0 then
    if (Test and $80) > 0 then
      if (Test and $20) > 0 then
        IdentifyUART := '16750'
      else
        IdentifyUART := '16550A'
    else
      IdentifyUART := '16550'
  else begin
    Port[COM1_Addr + SCR] := $2A;
    if Port[COM1_Addr + SCR] = $2A then
      IdentifyUART := '16450'
    else
      IdentifyUART := '8250';
  end;
end;
```

We still havn't identified between the 8250, 8250A, or 8250B; but that is rather pointless anyway on most current computers as it is very unlikely to even find one of those chips because of their age.

A very similar procedure can be used to determine the CPU of a computer, but that is beyond the scope of this book.

External References

- History of Interrupt Programming (http://www.cs.clemson.edu/~mark/interrupts.html)
- 8259 Chip Information with other registers explained (http://satyap.csoft.net/8259.html) (dead link?)
- Interfacing the Serial / RS232 Port (http://www.beyondlogic.org/serial/serial.htm)

While the 8250 is by far the most popular UART on desktop computers, other popular UARTs include:

- the UART inside the Atmel AVR: ... Embedded_Systems/Atmel_AVR#Serial_Communication
- the UART inside the Microchip PIC: "Microchip AN774: Asynchronous Communications with the PICmicro® USART" (http://microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012073)
- the UART inside the Apple Macintosh: ...
- "bit-banging" a UART: ... [1] (http://microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012058)

Other Serial Programming Articles

Serial Programming: Introduction and OSI Network Model -- RS-232 Wiring and Connections -- Typical RS232 Hardware Configuration -- 8250 UART -- DOS -- MAX232 Driver/Receiver Family -- TAPI Communications In Windows -- Linux and Unix -- Java -- Hayes-compatible Modems and AT Commands -- Universal Serial Bus (USB) -- Forming Data Packets -- Error Correction Methods -- Two Way Communication -- Packet Recovery Methods -- Serial Data Networks -- Practical Application Development -- IP Over Serial Connections