Serial and UART Tutorial

Abstract

This article talks about using serial hardware with FreeBSD.

Table of Contents

1. The UART: What it is and how it works

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes.

Serial transmission is commonly used with modems and for non-networked communication between computers, terminals and other devices.

There are two primary forms of serial transmission: Synchronous and Asynchronous. Depending on the modes that are supported by the hardware, the name of the communication sub-system will usually include a A if it supports Asynchronous communications, and a S if it supports Synchronous communications. Both forms are described below.

Some common acronyms are:

UART Universal Asynchronous Receiver/Transmitter

USART Universal Synchronous-Asynchronous Receiver/Transmitter

1.1. Synchronous Serial Transmission

Synchronous serial transmission requires that the sender and receiver share a clock with one another, or that the sender provide a strobe or other timing signal so that the receiver knows when to "read" the next bit of the data. In most forms of serial Synchronous communication, if there is no data available at a given instant to transmit, a fill character must be sent instead so that data is always being transmitted. Synchronous communication is usually more efficient because only data bits are transmitted between sender and receiver, and synchronous communication can be more

costly if extra wiring and circuits are required to share a clock signal between the sender and receiver.

A form of Synchronous transmission is used with printers and fixed disk devices in that the data is sent on one set of wires while a clock or strobe is sent on a different wire. Printers and fixed disk devices are not normally serial devices because most fixed disk interface standards send an entire word of data for each clock or strobe signal by using a separate wire for each bit of the word. In the PC industry, these are known as Parallel devices.

The standard serial communications hardware in the PC does not support Synchronous operations. This mode is described here for comparison purposes only.

1.2. Asynchronous Serial Transmission

Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word which are used to synchronize the sending and receiving units.

When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter. These two clocks must be accurate enough to not have the frequency drift by more than 10% during the transmission of the remaining bits in the word. (This requirement was set in the days of mechanical teleprinters and is easily met by modern electronic equipment.)

After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver "looks" at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a `1` or a `0`. For example, if it takes two seconds to send each bit, the receiver will examine the signal to determine if it is a `1` or a `0` after one second has passed, then it will wait two seconds and then examine the value of the next bit, and so on.

The sender does not know when the receiver has "looked" at the value of the bit. The sender only knows when the clock says to begin transmitting the next bit of the word.

When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity Bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter.

When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver looks for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. The usual cause of a Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted.

Regardless of whether the data was received correctly or not, the UART automatically discards the Start, Parity and Stop bits. If the sender and receiver are configured identically, these bits are not passed to the host.

If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent.

As asynchronous data is "self synchronizing", if there is no data to transmit, the transmission line can be idle.

1.3. Other UART Functions

In addition to the basic job of converting data from parallel to serial for transmission and from serial to parallel on reception, a UART will usually provide additional circuits for signals that can be used to indicate the state of the transmission media, and to regulate the flow of data in the event that the remote device is not prepared to accept more data. For example, when the device connected to the UART is a modem, the modem may report the presence of a carrier on the phone line while the computer may be able to instruct the modem to reset itself or to not take calls by raising or lowering one more of these extra signals. The function of each of these additional signals is defined in the EIA RS232-C standard.

1.4. The RS232-C and V.24 Standards

In most computer systems, the UART is connected to circuitry that generates signals that comply with the EIA RS232-C specification. There is also a CCITT standard named V.24 that mirrors the specifications included in RS232-C.

1.4.1. RS232-C Bit Assignments (Marks and Spaces)

In RS232-C, a value of `1` is called a `Mark` and a value of `0` is called a `Space`. When a communication line is idle, the line is said to be "Marking", or transmitting continuous `1` values.

The Start bit always has a value of `0` (a Space). The Stop Bit always has a value of `1` (a Mark). This means that there will always be a Mark (1) to Space (0) transition on the line at the start of every word, even when multiple word are transmitted back to back. This guarantees that sender and receiver can resynchronize their clocks regardless of the content of the data bits that are being transmitted.

The idle time between Stop and Start bits does not have to be an exact multiple (including zero) of the bit rate of the communication link, but most UARTs are designed this way for simplicity.

In RS232-C, the "Marking" signal (a `1`) is represented by a voltage between -2 VDC and -12 VDC, and a "Spacing" signal (a `0`) is represented by a voltage between 0 and +12 VDC. The transmitter is supposed to send +12 VDC or -12 VDC, and the receiver is supposed to allow for some voltage loss in long cables. Some transmitters in low power devices (like portable computers) sometimes use only +5 VDC and -5 VDC, but these values are still acceptable to a RS232-C receiver, provided that the cable lengths are short.

1.4.2. RS232-C Break Signal

RS232-C also specifies a signal called a `Break`, which is caused by sending continuous Spacing values (no Start or Stop bits). When there is no electricity present on the data circuit, the line is considered to be sending `Break`.

The `Break` signal must be of a duration longer than the time it takes to send a complete byte plus Start, Stop and Parity bits. Most UARTs can distinguish between a Framing Error and a Break, but if the UART cannot do this, the Framing Error detection can be used to identify Breaks.

In the days of teleprinters, when numerous printers around the country were wired in series (such as news services), any unit could cause a `Break` by temporarily opening the entire circuit so that no current flowed. This was used to allow a location with urgent news to interrupt some other location that was currently sending information.

In modern systems there are two types of Break signals. If the Break is longer than 1.6 seconds, it is considered a "Modem Break", and some modems can be programmed to terminate the conversation and go on-hook or enter the modems' command mode when the modem detects this signal. If the Break is smaller than 1.6 seconds, it signifies a Data Break and it is up to the remote computer to respond to this signal. Sometimes this form of Break is used as an Attention or Interrupt signal and sometimes is accepted as a substitute for the ASCII CONTROL-C character.

Marks and Spaces are also equivalent to "Holes" and "No Holes" in paper tape systems.

Breaks cannot be generated from paper tape or from any other byte value, since bytes are always sent with Start and Stop bit. The UART is usually capable of generating the continuous Spacing signal in response to a special command from the host processor.

1.4.3. RS232-C DTE and DCE Devices

The RS232-C specification defines two types of equipment: the Data Terminal Equipment (DTE) and the Data Carrier Equipment (DCE). Usually, the DTE device is the terminal (or computer), and the DCE is a modem. Across the phone line at the other end of a conversation, the receiving modem is also a DCE device and the computer that is connected to that modem is a DTE device. The DCE device receives signals on the pins that the DTE device transmits on, and vice versa.

When two devices that are both DTE or both DCE must be connected together without a modem or a similar media translator between them, a NULL modem must be used. The NULL modem electrically re-arranges the cabling so that the transmitter output is connected to the receiver input on the other device, and vice versa. Similar translations are performed on all of the control signals so that each device will see what it thinks are DCE (or DTE) signals from the other device.

The number of signals generated by the DTE and DCE devices are not symmetrical. The DTE device generates fewer signals for the DCE device than the DTE device receives from the DCE.

1.4.4. RS232-C Pin Assignments

The EIA RS232-C specification (and the ITU equivalent, V.24) calls for a twenty-five pin connector

(usually a DB25) and defines the purpose of most of the pins in that connector.

In the IBM Personal Computer and similar systems, a subset of RS232-C signals are provided via nine pin connectors (DB9). The signals that are not included on the PC connector deal mainly with synchronous operation, and this transmission mode is not supported by the UART that IBM selected for use in the IBM PC.

Depending on the computer manufacturer, a DB25, a DB9, or both types of connector may be used for RS232-C communications. (The IBM PC also uses a DB25 connector for the parallel printer interface which causes some confusion.)

Below is a table of the RS232-C signal assignments in the DB25 and DB9 connectors.

DB25 RS232-C Pin	DB9 IBM PC Pin	EIA Circuit Symbol	CCITT Circuit Symbol	Common Name	Signal Source	Description
1	-	AA	101	PG/FG	-	Frame/Protective Ground
2	3	BA	103	TD	DTE	Transmit Data
3	2	BB	104	RD	DCE	Receive Data
4	7	CA	105	RTS	DTE	Request to Send
5	8	CB	106	CTS	DCE	Clear to Send
6	6	CC	107	DSR	DCE	Data Set Ready
7	5	AV	102	SG/GND	-	Signal Ground
8	1	CF	109	DCD/CD	DCE	Data Carrier Detect
9	-	-	-	-	-	Reserved for Test
10	-	-	-	-	-	Reserved for Test
11	-	-	-	-	-	Reserved for Test
12	-	CI	122	SRLSD	DCE	Sec. Recv. Line Signal Detector
13	-	SCB	121	SCTS	DCE	Secondary Clear to Send

DB25 RS232-C Pin	DB9 IBM PC Pin	EIA Circuit Symbol	CCITT Circuit Symbol	Common Name	Signal Source	Description
14	-	SBA	118	STD	DTE	Secondary Transmit Data
15	-	DB	114	TSET	DCE	Trans. Sig. Element Timing
16	-	SBB	119	SRD	DCE	Secondary Received Data
17	-	DD	115	RSET	DCE	Receiver Signal Element Timing
18	-	-	141	LOOP	DTE	Local Loopback
19	-	SCA	120	SRS	DTE	Secondary Request to Send
20	4	CD	108.2	DTR	DTE	Data Terminal Ready
21	-	-	-	RDL	DTE	Remote Digital Loopback
22	9	CE	125	RI	DCE	Ring Indicator
23	-	CH	111	DSRS	DTE	Data Signal Rate Selector
24	-	DA	113	TSET	DTE	Trans. Sig. Element Timing
25	-	-	142	-	DCE	Test Mode

1.5. Bits, Baud and Symbols

Baud is a measurement of transmission speed in asynchronous communication. Due to advances in modem communication technology, this term is frequently misused when describing the data rates in newer devices.

Traditionally, a Baud Rate represents the number of bits that are actually being sent over the

media, not the amount of data that is actually moved from one DTE device to the other. The Baud count includes the overhead bits Start, Stop and Parity that are generated by the sending UART and removed by the receiving UART. This means that seven-bit words of data actually take 10 bits to be completely transmitted. Therefore, a modem capable of moving 300 bits per second from one place to another can normally only move 30 7-bit words if Parity is used and one Start and Stop bit are present.

If 8-bit data words are used and Parity bits are also used, the data rate falls to 27.27 words per second, because it now takes 11 bits to send the eight-bit words, and the modem still only sends 300 bits per second.

The formula for converting bytes per second into a baud rate and vice versa was simple until error-correcting modems came along. These modems receive the serial stream of bits from the UART in the host computer (even when internal modems are used the data is still frequently serialized) and converts the bits back into bytes. These bytes are then combined into packets and sent over the phone line using a Synchronous transmission method. This means that the Stop, Start, and Parity bits added by the UART in the DTE (the computer) were removed by the modem before transmission by the sending modem. When these bytes are received by the remote modem, the remote modem adds Start, Stop and Parity bits to the words, converts them to a serial format and then sends them to the receiving UART in the remote computer, who then strips the Start, Stop and Parity bits.

The reason all these extra conversions are done is so that the two modems can perform error correction, which means that the receiving modem is able to ask the sending modem to resend a block of data that was not received with the correct checksum. This checking is handled by the modems, and the DTE devices are usually unaware that the process is occurring.

By striping the Start, Stop and Parity bits, the additional bits of data that the two modems must share between themselves to perform error-correction are mostly concealed from the effective transmission rate seen by the sending and receiving DTE equipment. For example, if a modem sends ten 7-bit words to another modem without including the Start, Stop and Parity bits, the sending modem will be able to add 30 bits of its own information that the receiving modem can use to do error-correction without impacting the transmission speed of the real data.

The use of the term Baud is further confused by modems that perform compression. A single 8-bit word passed over the telephone line might represent a dozen words that were transmitted to the sending modem. The receiving modem will expand the data back to its original content and pass that data to the receiving DTE.

Modern modems also include buffers that allow the rate that bits move across the phone line (DCE to DCE) to be a different speed than the speed that the bits move between the DTE and DCE on both ends of the conversation. Normally the speed between the DTE and DCE is higher than the DCE to DCE speed because of the use of compression by the modems.

As the number of bits needed to describe a byte varied during the trip between the two machines plus the differing bits-per-seconds speeds that are used present on the DTE-DCE and DCE-DCE links, the usage of the term Baud to describe the overall communication speed causes problems and can misrepresent the true transmission speed. So Bits Per Second (bps) is the correct term to use to describe the transmission rate seen at the DCE to DCE interface and Baud or Bits Per Second are

acceptable terms to use when a connection is made between two systems with a wired connection, or if a modem is in use that is not performing error-correction or compression.

Modern high speed modems (2400, 9600, 14,400, and 19,200bps) in reality still operate at or below 2400 baud, or more accurately, 2400 Symbols per second. High speed modem are able to encode more bits of data into each Symbol using a technique called Constellation Stuffing, which is why the effective bits per second rate of the modem is higher, but the modem continues to operate within the limited audio bandwidth that the telephone system provides. Modems operating at 28,800 and higher speeds have variable Symbol rates, but the technique is the same.

1.6. The IBM Personal Computer UART

Starting with the original IBM Personal Computer, IBM selected the National Semiconductor INS8250 UART for use in the IBM PC Parallel/Serial Adapter. Subsequent generations of compatible computers from IBM and other vendors continued to use the INS8250 or improved versions of the National Semiconductor UART family.

1.6.1. National Semiconductor UART Family Tree

There have been several versions and subsequent generations of the INS8250 UART. Each major version is described below.

```
INS8250  -> INS8250B
  \
   \
    \-> INS8250A -> INS82C50A
              \
               \
                \-> NS16450 -> NS16C450
                          \
                           \
                            \-> NS16550 -> NS16550A -> PC16550D
```

INS8250

This part was used in the original IBM PC and IBM PC/XT. The original name for this part was the INS8250 ACE (Asynchronous Communications Element) and it is made from NMOS technology.

The 8250 uses eight I/O ports and has a one-byte send and a one-byte receive buffer. This original UART has several race conditions and other flaws. The original IBM BIOS includes code to work around these flaws, but this made the BIOS dependent on the flaws being present, so subsequent parts like the 8250A, 16450 or 16550 could not be used in the original IBM PC or IBM PC/XT.

INS8250-B

This is the slower speed of the INS8250 made from NMOS technology. It contains the same problems as the original INS8250.

INS8250A

An improved version of the INS8250 using XMOS technology with various functional flaws

corrected. The INS8250A was used initially in PC clone computers by vendors who used "clean" BIOS designs. Due to the corrections in the chip, this part could not be used with a BIOS compatible with the INS8250 or INS8250B.

INS82C50A

This is a CMOS version (low power consumption) of the INS8250A and has similar functional characteristics.

NS16450

Same as NS8250A with improvements so it can be used with faster CPU bus designs. IBM used this part in the IBM AT and updated the IBM BIOS to no longer rely on the bugs in the INS8250.

NS16C450

This is a CMOS version (low power consumption) of the NS16450.

NS16550

Same as NS16450 with a 16-byte send and receive buffer but the buffer design was flawed and could not be reliably be used.

NS16550A

Same as NS16550 with the buffer flaws corrected. The 16550A and its successors have become the most popular UART design in the PC industry, mainly due to its ability to reliably handle higher data rates on operating systems with sluggish interrupt response times.

NS16C552

This component consists of two NS16C550A CMOS UARTs in a single package.

PC16550D

Same as NS16550A with subtle flaws corrected. This is revision D of the 16550 family and is the latest design available from National Semiconductor.

1.6.2. The NS16550AF and the PC16550D are the same thing

National reorganized their part numbering system a few years ago, and the NS16550AFN no longer exists by that name. (If you have a NS16550AFN, look at the date code on the part, which is a four digit number that usually starts with a nine. The first two digits of the number are the year, and the last two digits are the week in that year when the part was packaged. If you have a NS16550AFN, it is probably a few years old.)

The new numbers are like PC16550DV, with minor differences in the suffix letters depending on the package material and its shape. (A description of the numbering system can be found below.)

It is important to understand that in some stores, you may pay $15(US) for a NS16550AFN made in 1990 and in the next bin are the new PC16550DN parts with minor fixes that National has made since the AFN part was in production, the PC16550DN was probably made in the past six months and it costs half (as low as $5(US) in volume) as much as the NS16550AFN because they are readily available.

As the supply of NS16550AFN chips continues to shrink, the price will probably continue to

increase until more people discover and accept that the PC16550DN really has the same function as the old part number.

1.6.3. National Semiconductor Part Numbering System

The older NSnnnnnrqp part numbers are now of the format PCnnnnnrgp.

The r is the revision field. The current revision of the 16550 from National Semiconductor is D.

The p is the package-type field. The types are:

"F"	QFP	(quad flat pack) L lead type
"N"	DIP	(dual inline package) through hole straight lead type
"V"	LPCC	(lead plastic chip carrier) J lead type

The *g* is the product grade field. If an I precedes the package-type letter, it indicates an "industrial" grade part, which has higher specs than a standard part but not as high as Military Specification (Milspec) component. This is an optional field.

So what we used to call a NS16550AFN (DIP Package) is now called a PC16550DN or PC16550DIN.

1.7. Other Vendors and Similar UARTs

Over the years, the 8250, 8250A, 16450 and 16550 have been licensed or copied by other chip vendors. In the case of the 8250, 8250A and 16450, the exact circuit (the "megacell") was licensed to many vendors, including Western Digital and Intel. Other vendors reverse-engineered the part or produced emulations that had similar behavior.

In internal modems, the modem designer will frequently emulate the 8250A/16450 with the modem microprocessor, and the emulated UART will frequently have a hidden buffer consisting of several hundred bytes. Due to the size of the buffer, these emulations can be as reliable as a 16550A in their ability to handle high speed data. However, most operating systems will still report that the UART is only a 8250A or 16450, and may not make effective use of the extra buffering present in the emulated UART unless special drivers are used.

Some modem makers are driven by market forces to abandon a design that has hundreds of bytes of buffer and instead use a 16550A UART so that the product will compare favorably in market comparisons even though the effective performance may be lowered by this action.

A common misconception is that all parts with "16550A" written on them are identical in performance. There are differences, and in some cases, outright flaws in most of these 16550A clones.

When the NS16550 was developed, the National Semiconductor obtained several patents on the design and they also limited licensing, making it harder for other vendors to provide a chip with similar features. As a result of the patents, reverse-engineered designs and emulations had to avoid

infringing the claims covered by the patents. Subsequently, these copies almost never perform exactly the same as the NS16550A or PC16550D, which are the parts most computer and modem makers want to buy but are sometimes unwilling to pay the price required to get the genuine part.

Some of the differences in the clone 16550A parts are unimportant, while others can prevent the device from being used at all with a given operating system or driver. These differences may show up when using other drivers, or when particular combinations of events occur that were not well tested or considered in the Windows® driver. This is because most modem vendors and 16550-clone makers use the Microsoft drivers from Windows® for Workgroups 3.11 and the Microsoft® MS-DOS® utility as the primary tests for compatibility with the NS16550A. This over-simplistic criteria means that if a different operating system is used, problems could appear due to subtle differences between the clones and genuine components.

National Semiconductor has made available a program named COMTEST that performs compatibility tests independent of any OS drivers. It should be remembered that the purpose of this type of program is to demonstrate the flaws in the products of the competition, so the program will report major as well as extremely subtle differences in behavior in the part being tested.

In a series of tests performed by the author of this document in 1994, components made by National Semiconductor, TI, StarTech, and CMD as well as megacells and emulations embedded in internal modems were tested with COMTEST. A difference count for some of these components is listed below. Since these tests were performed in 1994, they may not reflect the current performance of the given product from a vendor.

It should be noted that COMTEST normally aborts when an excessive number or certain types of problems have been detected. As part of this testing, COMTEST was modified so that it would not abort no matter how many differences were encountered.

Vendor	**Part Number**	**Errors (aka "differences" reported)**
National	(PC16550DV)	0
National	(NS16550AFN)	0
National	(NS16C552V)	0
TI	(TL16550AFN)	3
CMD	(16C550PE)	19
StarTech	(ST16C550J)	23
Rockwell	Reference modem with internal 16550 or an emulation (RC144DPi/C3000-25)	117
Sierra	Modem with an internal 16550 (SC11951/SC11351)	91

To date, the author of this document has not found any non-National parts that report zero differences using the COMTEST program. It should also be noted that National has had five versions of the 16550 over the years and the newest parts behave a bit differently than the classic NS16550AFN that is considered the benchmark for functionality. COMTEST appears to turn a blind eye to the differences within the National product line and reports no errors on the National parts (except for the original 16550) even when there are official erratas that describe bugs in the A, B and C revisions of the parts, so this bias in COMTEST must be taken into account.

It is important to understand that a simple count of differences from COMTEST does not reveal a lot about what differences are important and which are not. For example, about half of the differences reported in the two modems listed above that have internal UARTs were caused by the clone UARTs not supporting five- and six-bit character modes. The real 16550, 16450, and 8250 UARTs all support these modes and COMTEST checks the functionality of these modes so over fifty differences are reported. However, almost no modern modem supports five- or six-bit characters, particularly those with error-correction and compression capabilities. This means that the differences related to five- and six-bit character modes can be discounted.

Many of the differences COMTEST reports have to do with timing. In many of the clone designs, when the host reads from one port, the status bits in some other port may not update in the same amount of time (some faster, some slower) as a *real* NS16550AFN and COMTEST looks for these differences. This means that the number of differences can be misleading in that one device may only have one or two differences but they are extremely serious, and some other device that updates the status registers faster or slower than the reference part (that would probably never affect the operation of a properly written driver) could have dozens of differences reported.

COMTEST can be used as a screening tool to alert the administrator to the presence of potentially incompatible components that might cause problems or have to be handled as a special case.

If you run COMTEST on a 16550 that is in a modem or a modem is attached to the serial port, you need to first issue a ATE0&W command to the modem so that the modem will not echo any of the test characters. If you forget to do this, COMTEST will report at least this one difference:

```
Error (6)...Timeout interrupt failed: IIR = c1  LSR = 61
```

1.8. 8250/16450/16550 Registers

The 8250/16450/16550 UART occupies eight contiguous I/O port addresses. In the IBM PC, there are two defined locations for these eight ports and they are known collectively as COM1 and COM2. The makers of PC-clones and add-on cards have created two additional areas known as COM3 and COM4, but these extra COM ports conflict with other hardware on some systems. The most common conflict is with video adapters that provide IBM 8514 emulation.

COM1 is located from 0x3f8 to 0x3ff and normally uses IRQ 4. COM2 is located from 0x2f8 to 0x2ff and normally uses IRQ 3. COM3 is located from 0x3e8 to 0x3ef and has no standardized IRQ. COM4 is located from 0x2e8 to 0x2ef and has no standardized IRQ.

A description of the I/O ports of the 8250/16450/16550 UART is provided below.

I/O Port	Access Allowed	Description
+0x00	write (DLAB==0)	Transmit Holding Register (THR). Information written to this port are treated as data words and will be transmitted by the UART.
+0x00	read (DLAB==0)	Receive Buffer Register (RBR). Any data words received by the UART form the serial link are accessed by the host by reading this port.
+0x00	write/read (DLAB==1)	Divisor Latch LSB (DLL) This value will be divided from the master input clock (in the IBM PC, the master clock is 1.8432MHz) and the resulting clock will determine the baud rate of the UART. This register holds bits 0 thru 7 of the divisor.
+0x01	write/read (DLAB==1)	Divisor Latch MSB (DLH) This value will be divided from the master input clock (in the IBM PC, the master clock is 1.8432MHz) and the resulting clock will determine the baud rate of the UART. This register holds bits 8 thru 15 of the divisor.

I/O Port	Access Allowed	Description
+0x01	write/read (DLAB==0)	Interrupt Enable Register (IER) The 8250/16450/16550 UART classifies events into one of four categories. Each category can be configured to generate an interrupt when any of the events occurs. The 8250/16450/16550 UART generates a single external interrupt signal regardless of how many events in the enabled categories have occurred. It is up to the host processor to respond to the interrupt and then poll the enabled interrupt categories (usually all categories have interrupts enabled) to determine the true cause(s) of the interrupt. Bit 7 → Reserved, always 0. Bit 6 → Reserved, always 0. Bit 5 → Reserved, always 0. Bit 4 → Reserved, always 0. Bit 3 → Enable Modem Status Interrupt (EDSSI). Setting this bit to "1" allows the UART to generate an interrupt when a change occurs on one or more of the status lines. Bit 2 → Enable Receiver Line Status Interrupt (ELSI) Setting this bit to "1" causes the UART to generate an interrupt when the an error (or a BREAK signal) has been detected in the incoming data. Bit 1 → Enable Transmitter Holding Register Empty Interrupt (ETBEI) Setting this bit to "1" causes the UART to generate an interrupt when the UART has room for one or more additional characters that are to be transmitted. Bit 0 → Enable Received Data Available Interrupt (ERBFI) Setting this bit to "1" causes the UART to generate an interrupt when the UART has received enough characters to exceed the trigger level of the FIFO, or the FIFO timer has expired (stale data), or a single character has been received when the FIFO is disabled.

I/O Port	Access Allowed	Description
+0x02	write	FIFO Control Register (FCR) (This port does not exist on the 8250 and 16450 UART.) Bit 7 → Receiver Trigger Bit #1 Bit 6 → Receiver Trigger Bit #0 These two bits control at what point the receiver is to generate an interrupt when the FIFO is active. 7 6 How many words are received before an interrupt is generated 0 0 1 0 1 4 1 0 8 1 1 14 Bit 5 → Reserved, always 0. Bit 4 → Reserved, always 0. Bit 3 → DMA Mode Select. If Bit 0 is set to "1" (FIFOs enabled), setting this bit changes the operation of the -RXRDY and -TXRDY signals from Mode 0 to Mode 1. Bit 2 → Transmit FIFO Reset. When a "1" is written to this bit, the contents of the FIFO are discarded. Any word currently being transmitted will be sent intact. This function is useful in aborting transfers. Bit 1 → Receiver FIFO Reset. When a "1" is written to this bit, the contents of the FIFO are discarded. Any word currently being assembled in the shift register will be received intact. Bit 0 → 16550 FIFO Enable. When set, both the transmit and receive FIFOs are enabled. Any contents in the holding register, shift registers or FIFOs are lost when FIFOs are enabled or disabled.

I/O Port	Access Allowed	Description
+0x02	read	Interrupt Identification Register Bit 7 → FIFOs enabled. On the 8250/16450 UART, this bit is zero. Bit 6 → FIFOs enabled. On the 8250/16450 UART, this bit is zero. Bit 5 → Reserved, always 0. Bit 4 → Reserved, always 0. Bit 3 → Interrupt ID Bit #2. On the 8250/16450 UART, this bit is zero. Bit 2 → Interrupt ID Bit #1 Bit 1 → Interrupt ID Bit #0.These three bits combine to report the category of event that caused the interrupt that is in progress. These categories have priorities, so if multiple categories of events occur at the same time, the UART will report the more important events first and the host must resolve the events in the order they are reported. All events that caused the current interrupt must be resolved before any new interrupts will be generated. (This is a limitation of the PC architecture.) 2 1 0 Priority Description 0 1 1 First Received Error (OE, PE, BI, or FE) 0 1 0 Second Received Data Available 1 1 0 Second Trigger level identification (Stale data in receive buffer) 0 0 1 Third Transmitter has room for more words (THRE) 0 0 0 Fourth Modem Status Change (-CTS, -DSR, -RI, or -DCD) Bit 0 → Interrupt Pending Bit. If this bit is set to "0", then at least one interrupt is pending.

I/O Port	Access Allowed	Description
+0x03	write/read	Line Control Register (LCR) Bit 7 → Divisor Latch Access Bit (DLAB). When set, access to the data transmit/receive register (THR/RBR) and the Interrupt Enable Register (IER) is disabled. Any access to these ports is now redirected to the Divisor Latch Registers. Setting this bit, loading the Divisor Registers, and clearing DLAB should be done with interrupts disabled. Bit 6 → Set Break. When set to "1", the transmitter begins to transmit continuous Spacing until this bit is set to "0". This overrides any bits of characters that are being transmitted. Bit 5 → Stick Parity. When parity is enabled, setting this bit causes parity to always be "1" or "0", based on the value of Bit 4. Bit 4 → Even Parity Select (EPS). When parity is enabled and Bit 5 is "0", setting this bit causes even parity to be transmitted and expected. Otherwise, odd parity is used. Bit 3 → Parity Enable (PEN). When set to "1", a parity bit is inserted between the last bit of the data and the Stop Bit. The UART will also expect parity to be present in the received data. Bit 2 → Number of Stop Bits (STB). If set to "1" and using 5-bit data words, 1.5 Stop Bits are transmitted and expected in each data word. For 6, 7 and 8-bit data words, 2 Stop Bits are transmitted and expected. When this bit is set to "0", one Stop Bit is used on each data word. Bit 1 → Word Length Select Bit #1 (WLSB1) Bit 0 → Word Length Select Bit #0 (WLSB0) Together these bits specify the number of bits in each data word. 1 0 Word Length 0 0 5 Data Bits 0 1 6 Data Bits 1 0 7 Data Bits 1 1 8 Data Bits

I/O Port	Access Allowed	Description
+0x04	write/read	Modem Control Register (MCR) Bit 7 → Reserved, always 0. Bit 6 → Reserved, always 0. Bit 5 → Reserved, always 0. Bit 4 → Loop-Back Enable. When set to "1", the UART transmitter and receiver are internally connected together to allow diagnostic operations. In addition, the UART modem control outputs are connected to the UART modem control inputs. CTS is connected to RTS, DTR is connected to DSR, OUT1 is connected to RI, and OUT 2 is connected to DCD. Bit 3 → OUT 2. An auxiliary output that the host processor may set high or low. In the IBM PC serial adapter (and most clones), OUT 2 is used to tri-state (disable) the interrupt signal from the 8250/16450/16550 UART. Bit 2 → OUT 1. An auxiliary output that the host processor may set high or low. This output is not used on the IBM PC serial adapter. Bit 1 → Request to Send (RTS). When set to "1", the output of the UART -RTS line is Low (Active). Bit 0 → Data Terminal Ready (DTR). When set to "1", the output of the UART -DTR line is Low (Active).
+0x05	write/read	Line Status Register (LSR) Bit 7 → Error in Receiver FIFO. On the 8250/16450 UART, this bit is zero. This bit is set to "1" when any of the bytes in the FIFO have one or more of the following error conditions: PE, FE, or BI. Bit 6 → Transmitter Empty (TEMT). When set to "1", there are no words remaining in the transmit FIFO or the transmit shift register. The transmitter is completely idle. Bit 5 → Transmitter Holding Register Empty (THRE). When set to "1", the FIFO (or holding register) now has room for at least one additional word to transmit. The transmitter may still be transmitting when this bit is set to "1". Bit 4 → Break Interrupt (BI). The receiver has detected a Break signal. Bit 3 → Framing Error (FE). A Start Bit was detected but the Stop Bit did not appear at the expected time. The received word is probably garbled. Bit 2 → Parity Error (PE). The parity bit was incorrect for the word received. Bit 1 → Overrun Error (OE). A new word was received and there was no room in the receive buffer. The newly-arrived word in the shift register is discarded. On 8250/16450 UARTs, the word in the holding register is discarded and the newly- arrived word is put in the holding register. Bit 0 → Data Ready (DR) One or more words are in the receive FIFO that the host may read. A word must be completely received and moved from the shift register into the FIFO (or holding register for 8250/16450 designs) before this bit is set.

I/O Port	Access Allowed	Description
+0x06	write/read	Modem Status Register (MSR) Bit 7 → Data Carrier Detect (DCD). Reflects the state of the DCD line on the UART. Bit 6 → Ring Indicator (RI). Reflects the state of the RI line on the UART. Bit 5 → Data Set Ready (DSR). Reflects the state of the DSR line on the UART. Bit 4 → Clear To Send (CTS). Reflects the state of the CTS line on the UART. Bit 3 → Delta Data Carrier Detect (DDCD). Set to "1" if the -DCD line has changed state one more time since the last time the MSR was read by the host. Bit 2 → Trailing Edge Ring Indicator (TERI). Set to "1" if the -RI line has had a low to high transition since the last time the MSR was read by the host. Bit 1 → Delta Data Set Ready (DDSR). Set to "1" if the -DSR line has changed state one more time since the last time the MSR was read by the host. Bit 0 → Delta Clear To Send (DCTS). Set to "1" if the -CTS line has changed state one more time since the last time the MSR was read by the host.
+0x07	write/read	Scratch Register (SCR). This register performs no function in the UART. Any value can be written by the host to this location and read by the host later on.

1.9. Beyond the 16550A UART

Although National Semiconductor has not offered any components compatible with the 16550 that provide additional features, various other vendors have. Some of these components are described below. It should be understood that to effectively utilize these improvements, drivers may have to be provided by the chip vendor since most of the popular operating systems do not support features beyond those provided by the 16550.

ST16650

By default this part is similar to the NS16550A, but an extended 32-byte send and receive buffer can be optionally enabled. Made by StarTech.

TIL16660

By default this part behaves similar to the NS16550A, but an extended 64-byte send and receive buffer can be optionally enabled. Made by Texas Instruments.

Hayes ESP

This proprietary plug-in card contains a 2048-byte send and receive buffer, and supports data rates to 230.4Kbit/sec. Made by Hayes.

In addition to these "dumb" UARTs, many vendors produce intelligent serial communication boards. This type of design usually provides a microprocessor that interfaces with several UARTs, processes and buffers the data, and then alerts the main PC processor when necessary. As the UARTs are not directly accessed by the PC processor in this type of communication system, it is not necessary for the vendor to use UARTs that are compatible with the 8250, 16450, or the 16550 UART. This leaves the designer free to components that may have better performance characteristics.

2. Configuring the sio driver

The sio driver provides support for NS8250-, NS16450-, NS16550 and NS16550A-based EIA RS-232C (CCITT V.24) communications interfaces. Several multiport cards are supported as well. See the sio(4) manual page for detailed technical documentation.

2.1. Digi International (DigiBoard) PC/8

Contributed by Andrew Webster <awebster@pubnix.net>. *26 August 1995.*

Here is a config snippet from a machine with a Digi International PC/8 with 16550. It has 8 modems connected to these 8 lines, and they work just great. Do not forget to add `options COM_MULTIPORT` or it will not work very well!

```
device          sio4    at isa? port 0x100 flags 0xb05
device          sio5    at isa? port 0x108 flags 0xb05
device          sio6    at isa? port 0x110 flags 0xb05
device          sio7    at isa? port 0x118 flags 0xb05
device          sio8    at isa? port 0x120 flags 0xb05
device          sio9    at isa? port 0x128 flags 0xb05
device          sio10   at isa? port 0x130 flags 0xb05
device          sio11   at isa? port 0x138 flags 0xb05 irq 9
```

The trick in setting this up is that the MSB of the flags represent the last SIO port, in this case 11 so flags are 0xb05.

2.2. Boca 16

Contributed by Don Whiteside <whiteside@acm.org>. *26 August 1995.*

The procedures to make a Boca 16 port board with FreeBSD are pretty straightforward, but you will need a couple things to make it work:

1. You either need the kernel sources installed so you can recompile the necessary options or you will need someone else to compile it for you. The 2.0.5 default kernel does *not* come with multiport support enabled and you will need to add a device entry for each port anyways.
2. Two, you will need to know the interrupt and IO setting for your Boca Board so you can set these options properly in the kernel.

One important note - the actual UART chips for the Boca 16 are in the connector box, not on the internal board itself. So if you have it unplugged, probes of those ports will fail. I have never tested booting with the box unplugged and plugging it back in, and I suggest you do not either.

If you do not already have a custom kernel configuration file set up, refer to Kernel Configuration chapter of the FreeBSD Handbook for general procedures. The following are the specifics for the Boca 16 board and assume you are using the kernel name MYKERNEL and editing with vi.

1. Add the line

   ```
   options COM_MULTIPORT
   ```

   to the config file.

2. Where the current `device sion` lines are, you will need to add 16 more devices. The following example is for a Boca Board with an interrupt of 3, and a base IO address 100h. The IO address for Each port is +8 hexadecimal from the previous port, thus the 100h, 108h, 110h... addresses.

   ```
   device sio1 at isa? port 0x100 flags 0x1005
   device sio2 at isa? port 0x108 flags 0x1005
   device sio3 at isa? port 0x110 flags 0x1005
   device sio4 at isa? port 0x118 flags 0x1005
   ...
   device sio15 at isa? port 0x170 flags 0x1005
   device sio16 at isa? port 0x178 flags 0x1005 irq 3
   ```

   The flags entry *must* be changed from this example unless you are using the exact same sio assignments. Flags are set according to 0x`MYY` where *M* indicates the minor number of the master port (the last port on a Boca 16) and *YY* indicates if FIFO is enabled or disabled(enabled), IRQ sharing is used(yes) and if there is an AST/4 compatible IRQ control register(no). In this example,

   ```
   flags
           0x1005
   ```

   indicates that the master port is sio16. If I added another board and assigned sio17 through sio28, the flags for all 16 ports on *that* board would be 0x1C05, where 1C indicates the minor number of the master port. Do not change the 05 setting.

3. Save and complete the kernel configuration, recompile, install and reboot. Presuming you have successfully installed the recompiled kernel and have it set to the correct address and IRQ, your boot message should indicate the successful probe of the Boca ports as follows: (obviously the sio numbers, IO and IRQ could be different)

```
sio1 at 0x100-0x107 flags 0x1005 on isa
sio1: type 16550A (multiport)
sio2 at 0x108-0x10f flags 0x1005 on isa
sio2: type 16550A (multiport)
sio3 at 0x110-0x117 flags 0x1005 on isa
sio3: type 16550A (multiport)
sio4 at 0x118-0x11f flags 0x1005 on isa
sio4: type 16550A (multiport)
sio5 at 0x120-0x127 flags 0x1005 on isa
sio5: type 16550A (multiport)
sio6 at 0x128-0x12f flags 0x1005 on isa
sio6: type 16550A (multiport)
sio7 at 0x130-0x137 flags 0x1005 on isa
sio7: type 16550A (multiport)
sio8 at 0x138-0x13f flags 0x1005 on isa
sio8: type 16550A (multiport)
sio9 at 0x140-0x147 flags 0x1005 on isa
sio9: type 16550A (multiport)
sio10 at 0x148-0x14f flags 0x1005 on isa
sio10: type 16550A (multiport)
sio11 at 0x150-0x157 flags 0x1005 on isa
sio11: type 16550A (multiport)
sio12 at 0x158-0x15f flags 0x1005 on isa
sio12: type 16550A (multiport)
sio13 at 0x160-0x167 flags 0x1005 on isa
sio13: type 16550A (multiport)
sio14 at 0x168-0x16f flags 0x1005 on isa
sio14: type 16550A (multiport)
sio15 at 0x170-0x177 flags 0x1005 on isa
sio15: type 16550A (multiport)
sio16 at 0x178-0x17f irq 3 flags 0x1005 on isa
sio16: type 16550A (multiport master)
```

If the messages go by too fast to see,

```
# dmesg | more
```

will show you the boot messages.

4. Next, appropriate entries in /dev for the devices must be made using the /dev/MAKEDEV script. This step can be omitted if you are running FreeBSD 5.X with a kernel that has devfs(5) support compiled in.

   If you do need to create the /dev entries, run the following as `root`:

```
# cd /dev
# ./MAKEDEV tty1
# ./MAKEDEV cua1

(everything in between)
# ./MAKEDEV ttyg
# ./MAKEDEV cuag
```

If you do not want or need call-out devices for some reason, you can dispense with making the cua* devices.

5. If you want a quick and sloppy way to make sure the devices are working, you can simply plug a modem into each port and (as root)

```
# echo at > ttyd*
```

for each device you have made. You *should* see the RX lights flash for each working port.

2.3. Support for Cheap Multi-UART Cards

Contributed by Helge Oldach hmo@sep.hamburg.com, September 1999

Ever wondered about FreeBSD support for your 20$ multi-I/O card with two (or more) COM ports, sharing IRQs? Here is how:

Usually the only option to support these kind of boards is to use a distinct IRQ for each port. For example, if your CPU board has an on-board COM1 port (aka sio0-I/O address 0x3F8 and IRQ 4) and you have an extension board with two UARTs, you will commonly need to configure them as COM2 (aka sio1-I/O address 0x2F8 and IRQ 3), and the third port (aka sio2) as I/O 0x3E8 and IRQ 5. Obviously this is a waste of IRQ resources, as it should be basically possible to run both extension board ports using a single IRQ with the `COM_MULTIPORT` configuration described in the previous sections.

Such cheap I/O boards commonly have a 4 by 3 jumper matrix for the COM ports, similar to the following:

```
          o  o  o  *
Port A                |
          o  *  o  *
Port B      |
          o  *  o  o
IRQ       2  3  4  5
```

Shown here is port A wired for IRQ 5 and port B wired for IRQ 3. The IRQ columns on your specific board may vary-other boards may supply jumpers for IRQs 3, 4, 5, and 7 instead.

One could conclude that wiring both ports for IRQ 3 using a handcrafted wire-made jumper covering all three connection points in the IRQ 3 column would solve the issue, but no. You cannot duplicate IRQ 3 because the output drivers of each UART are wired in a "totem pole" fashion, so if one of the UARTs drives IRQ 3, the output signal will not be what you would expect. Depending on the implementation of the extension board or your motherboard, the IRQ 3 line will continuously stay up, or always stay low.

You need to decouple the IRQ drivers for the two UARTs, so that the IRQ line of the board only goes up if (and only if) one of the UARTs asserts a IRQ, and stays low otherwise. The solution was proposed by Joerg Wunsch j@ida.interface-business.de: To solder up a wired-or consisting of two diodes (Germanium or Schottky-types strongly preferred) and a 1 kOhm resistor. Here is the schematic, starting from the 4 by 3 jumper field above:

```
                        Diode
               +---------->|-------+
              /                    |
          o  *  o  o               |     1 kOhm
Port A                             +----|######|-------+
          o  *  o  o               |                    |
Port B         `-------------------+                ==+==
          o  *  o  o               |                Ground
               \                   |
                +--------->|-------+
IRQ       2  3  4  5    Diode
```

The cathodes of the diodes are connected to a common point, together with a 1 kOhm pull-down resistor. It is essential to connect the resistor to ground to avoid floating of the IRQ line on the bus.

Now we are ready to configure a kernel. Staying with this example, we would configure:

```
# standard on-board COM1 port
device          sio0    at isa? port "IO_COM1" flags 0x10
# patched-up multi-I/O extension board
options         COM_MULTIPORT
device          sio1    at isa? port "IO_COM2" flags 0x205
device          sio2    at isa? port "IO_COM3" flags 0x205 irq 3
```

Note that the `flags` setting for sio1 and sio2 is truly essential; refer to sio(4) for details. (Generally, the `2` in the "flags" attribute refers to sio`2` which holds the IRQ, and you surely want a `5` low nibble.) With kernel verbose mode turned on this should yield something similar to this:

```
sio0: irq maps: 0x1 0x11 0x1 0x1
sio0 at 0x3f8-0x3ff irq 4 flags 0x10 on isa
sio0: type 16550A
sio1: irq maps: 0x1 0x9 0x1 0x1
sio1 at 0x2f8-0x2ff flags 0x205 on isa
sio1: type 16550A (multiport)
sio2: irq maps: 0x1 0x9 0x1 0x1
sio2 at 0x3e8-0x3ef irq 3 flags 0x205 on isa
sio2: type 16550A (multiport master)
```

Though /sys/i386/isa/sio.c is somewhat cryptic with its use of the "irq maps" array above, the basic idea is that you observe `0x1` in the first, third, and fourth place. This means that the corresponding IRQ was set upon output and cleared after, which is just what we would expect. If your kernel does not display this behavior, most likely there is something wrong with your wiring.

3. Configuring the cy driver

Contributed by Alex Nash. 6 June 1996.

The Cyclades multiport cards are based on the cy driver instead of the usual sio driver used by other multiport cards. Configuration is a simple matter of:

1. Add the cy device to your kernel configuration (note that your irq and iomem settings may differ).

   ```
   device cy0 at isa? irq 10 iomem 0xd4000 iosiz 0x2000
   ```

2. Rebuild and install the new kernel.
3. Make the device nodes by typing (the following example assumes an 8-port board):

   ```
   # cd /dev
   # for i in 0 1 2 3 4 5 6 7;do ./MAKEDEV cuac$i ttyc$i;done
   ```

4. If appropriate, add dialup entries to /etc/ttys by duplicating serial device (`ttyd`) entries and using `ttyc` in place of `ttyd`. For example:

   ```
   ttyc0   "/usr/libexec/getty std.38400"  unknown on insecure
   ttyc1   "/usr/libexec/getty std.38400"  unknown on insecure
   ttyc2   "/usr/libexec/getty std.38400"  unknown on insecure
   ...
   ttyc7   "/usr/libexec/getty std.38400"  unknown on insecure
   ```

5. Reboot with the new kernel.

4. Configuring the si driver

Contributed by Nick Sayer <nsayer@FreeBSD.org>. *25 March 1998.*

The Specialix SI/XIO and SX multiport cards use the si driver. A single machine can have up to 4 host cards. The following host cards are supported:

- ISA SI/XIO host card (2 versions)
- EISA SI/XIO host card
- PCI SI/XIO host card
- ISA SX host card
- PCI SX host card

Although the SX and SI/XIO host cards look markedly different, their functionality are basically the same. The host cards do not use I/O locations, but instead require a 32K chunk of memory. The factory configuration for ISA cards places this at `0xd0000-0xd7fff`. They also require an IRQ. PCI cards will, of course, auto-configure themselves.

You can attach up to 4 external modules to each host card. The external modules contain either 4 or 8 serial ports. They come in the following varieties:

- SI 4 or 8 port modules. Up to 57600 bps on each port supported.
- XIO 8 port modules. Up to 115200 bps on each port supported. One type of XIO module has 7 serial and 1 parallel port.
- SXDC 8 port modules. Up to 921600 bps on each port supported. Like XIO, a module is available with one parallel port as well.

To configure an ISA host card, add the following line to your kernel configuration file, changing the numbers as appropriate:

```
device si0 at isa? iomem 0xd0000 irq 11
```

Valid IRQ numbers are 9, 10, 11, 12 and 15 for SX ISA host cards and 11, 12 and 15 for SI/XIO ISA host cards.

To configure an EISA or PCI host card, use this line:

```
device si0
```

After adding the configuration entry, rebuild and install your new kernel.

The following step, is not necessary if you are using devfs(5) in FreeBSD 5.*X*.

After rebooting with the new kernel, you need to make the device nodes in /dev. The MAKEDEV script will take care of this for you. Count how many total ports you have and type:

```
# cd /dev
# ./MAKEDEV ttyAnn cuaAnn
```

(where *nn* is the number of ports)

If you want login prompts to appear on these ports, you will need to add lines like this to /etc/ttys:

```
ttyA01  "/usr/libexec/getty std.9600"   vt100   on insecure
```

Change the terminal type as appropriate. For modems, `dialup` or `unknown` is fine.