Computer Science 686

Spring 2007

Special Topic:

Intel EM64T and VT Extensions

Recent CPU advances

- Intel Corporation’s newest CPUs for the Personal Computer market offer a 64-bit architecture and instructions that support ‘Virtual Machine Management’
- To maintain ‘backward compatibility’ with previous CPUs, these added capabilities are not automatically turned on
- System software must be built to enable them -- and then to utilize them

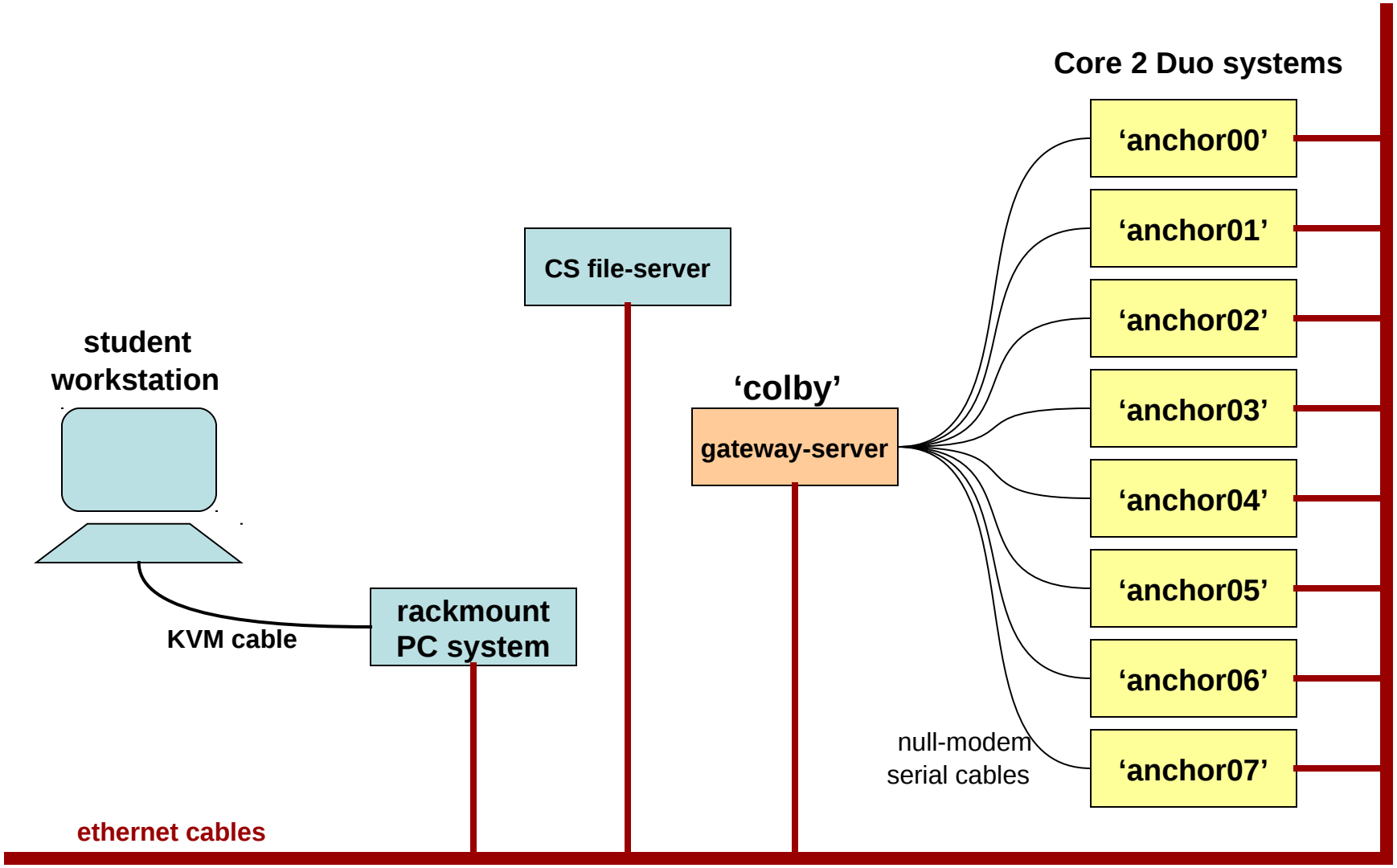
Our course’s purpose

- We want to study these new capabilities, how to activate them and how to utilize them, from a ‘hands-on’ perspective
- Our machines have Core-2 Duo CPUs
- But they are ‘rack-mounted’ boxes (hence no keyboard, mouse, or video display), so we connect with them via the local network
- But the LAN doesn’t work during ‘boot-up’

Alternate access mechanism

- We will need to employ a different scheme for receiving output (or transmitting input) to our remote Core-2 Duo machines when no operating system has yet been loaded
- For this we'll use the PC's serial-port, and a special cable known as a 'null-modem'
- But we will need to write our own software to operate the serial communication link

Our remote-access scheme



Universal Asynchronous Receiver-Transmitter

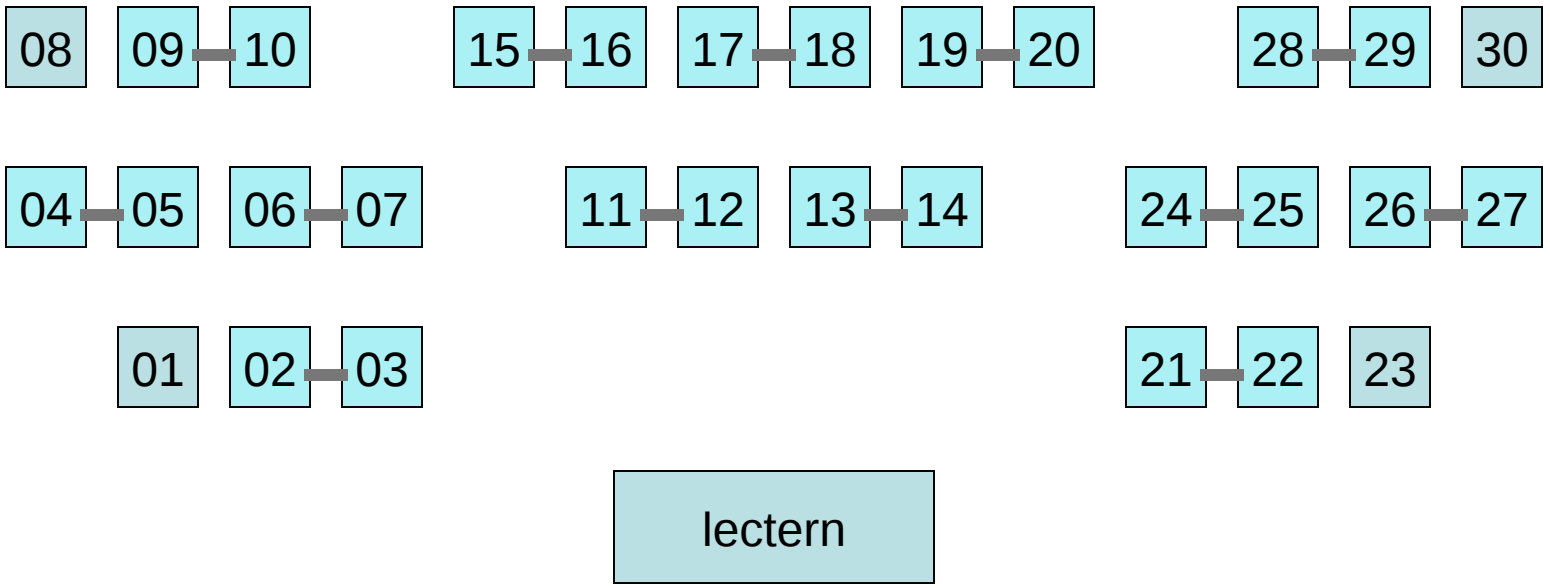
(UART)

See our CS686 course website at:

<http://cs.usfca.edu/~cruse/cs686>

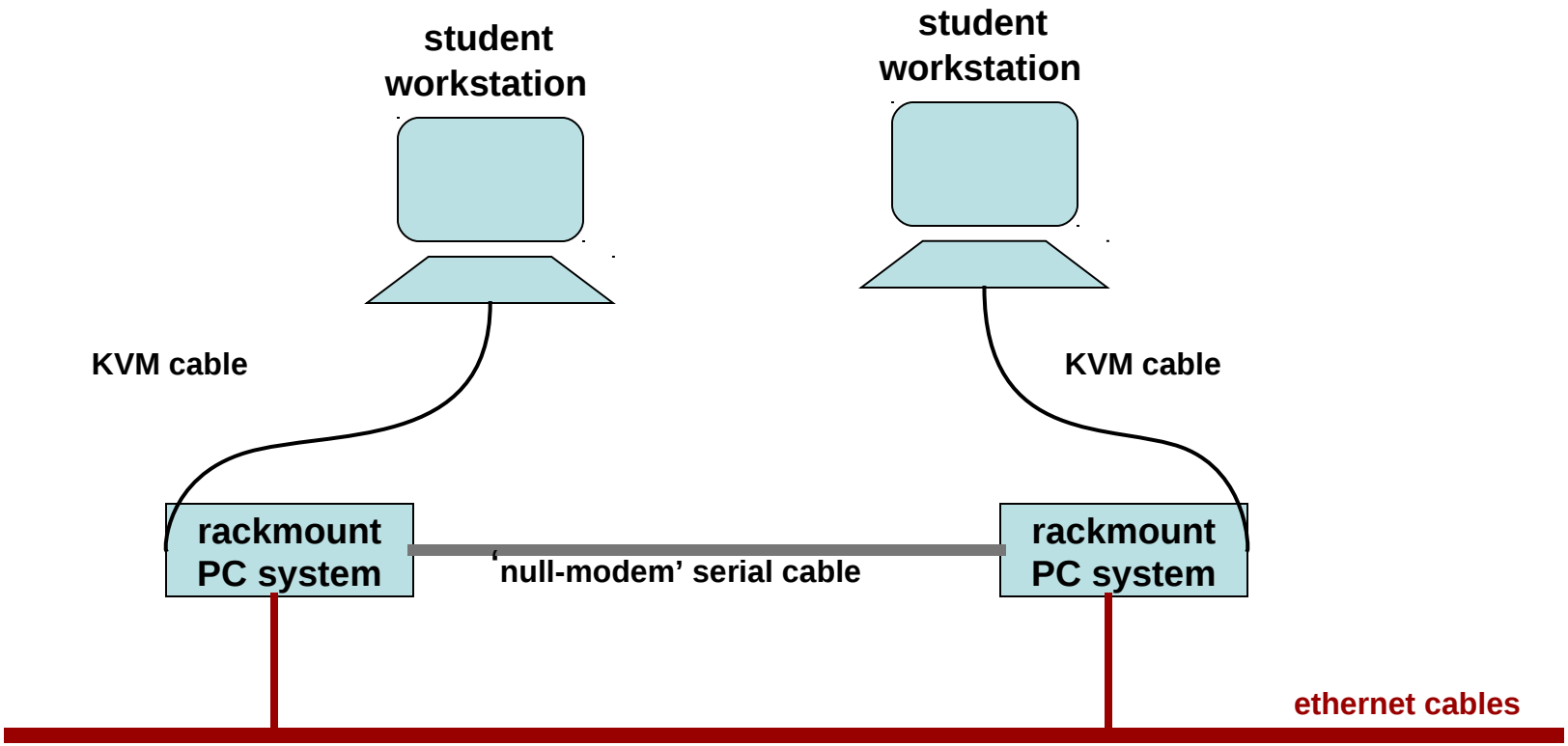
for links to the UART manufacturer's documentation
and to an in-depth online programming tutorial

Kudlick Classroom



— Indicates a “null-modem” PC-to-PC serial cable connection

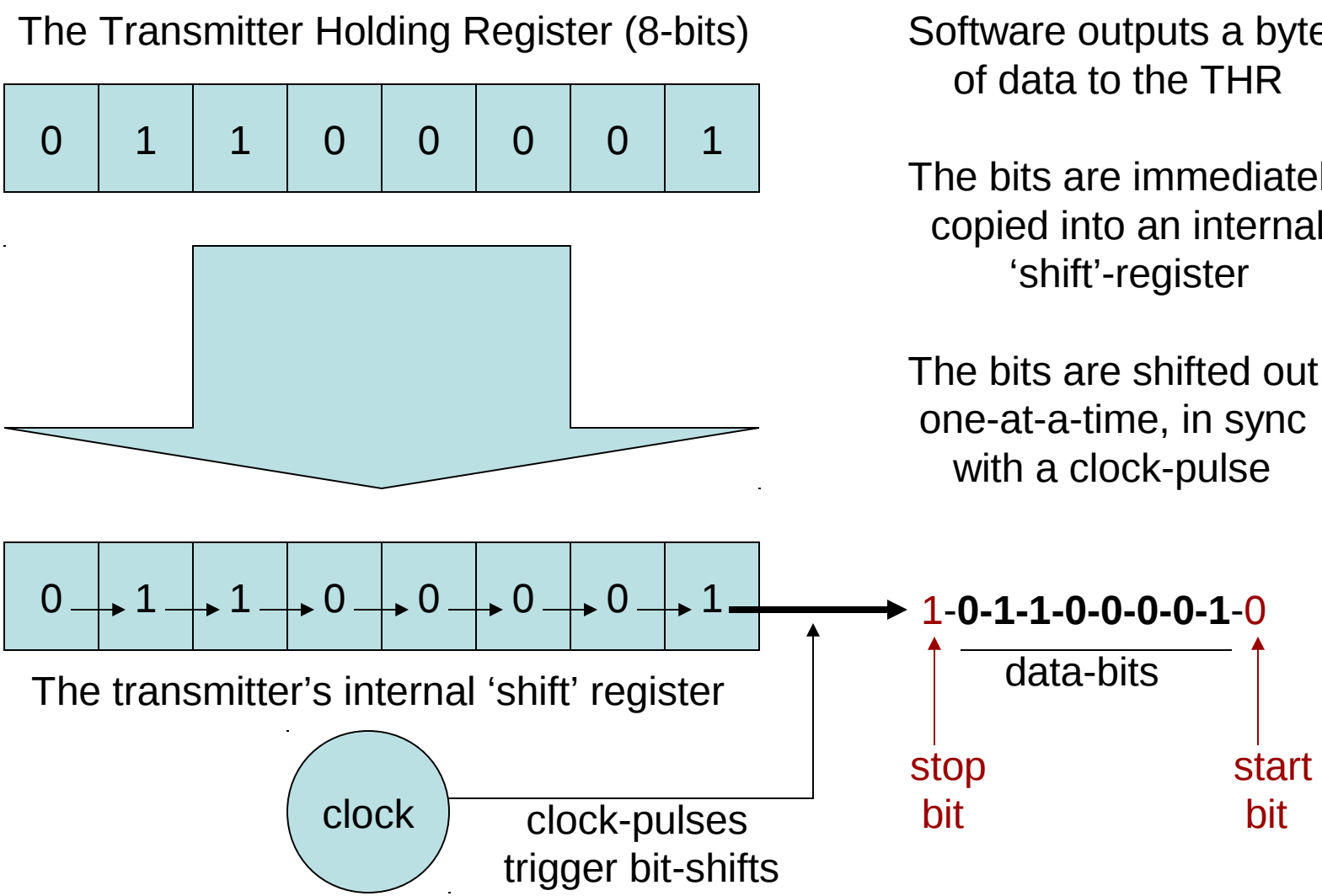
PC-to-PC communications



Tx and Rx

- The UART has a transmission engine, and also a reception engine (they can operate simultaneously)
- Software controls the UART’s operations by accessing several registers, using the CPU’s input and output instructions
- A little history is needed for understanding some of the UART’s terminology

Serial data-transmission

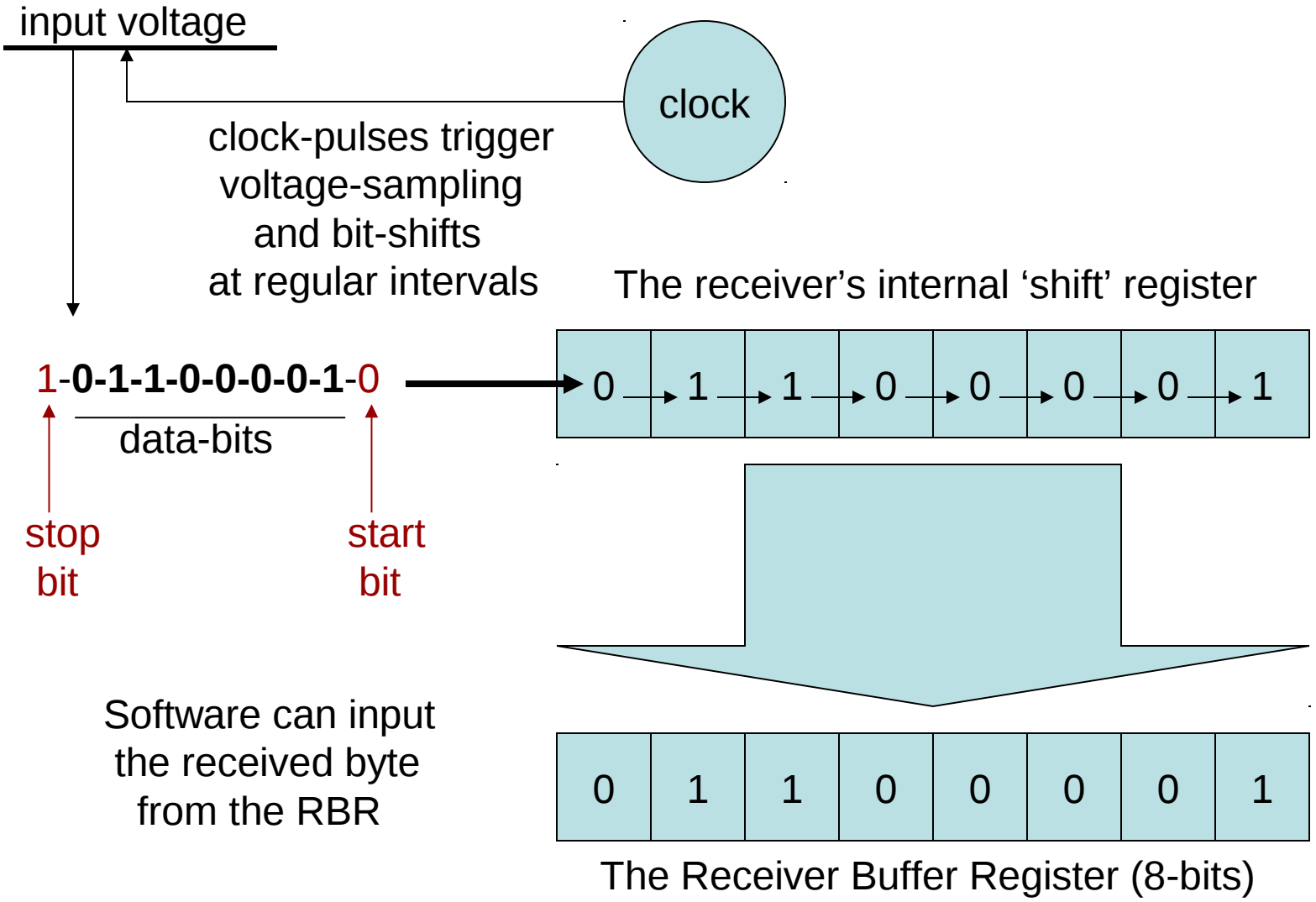


Software outputs a byte of data to the THR

The bits are immediately copied into an internal 'shift'-register

The bits are shifted out, one-at-a-time, in sync with a clock-pulse

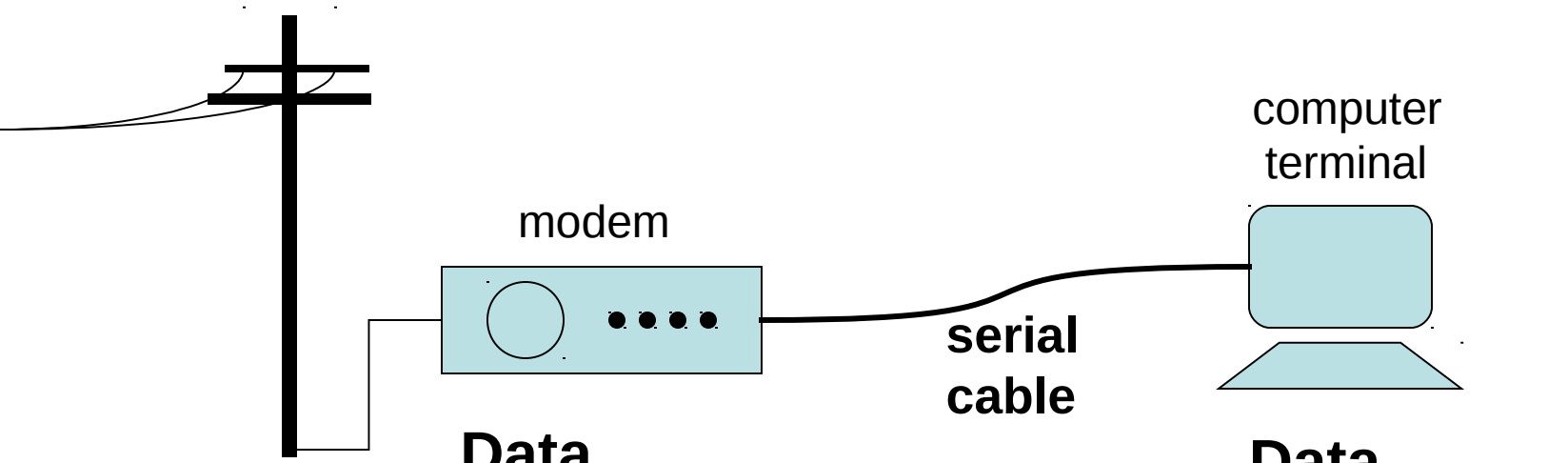
Serial data reception



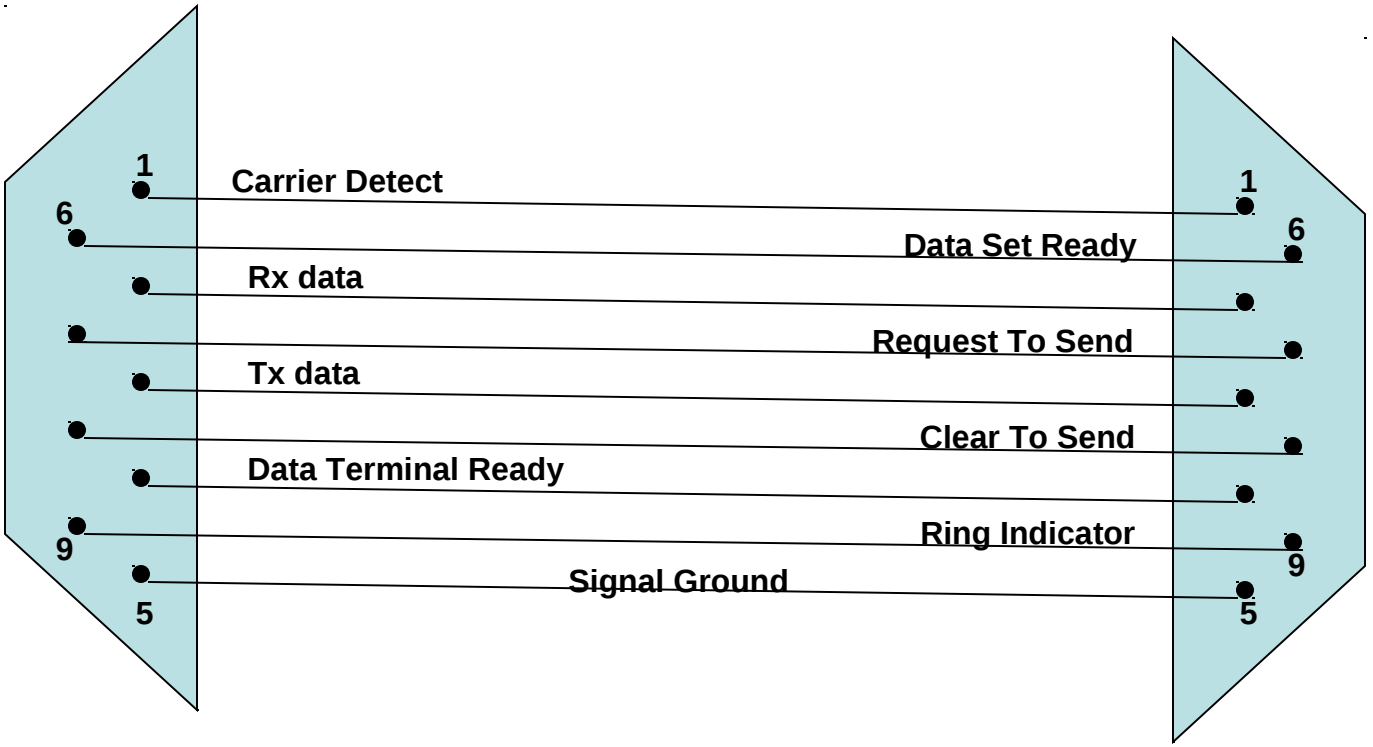
DCE and DTE

- Original purpose of the UART was for PCs to communicate via the telephone network
- Telephones were for voice communication (analog signals) whereas computers need so exchange discrete data (digital signals)
- Special ‘communication equipment’ was needed for doing the signal conversions (i.e. a modulator/demodulator, or **modem**)

PC with a modem



Normal 9-wire serial cable



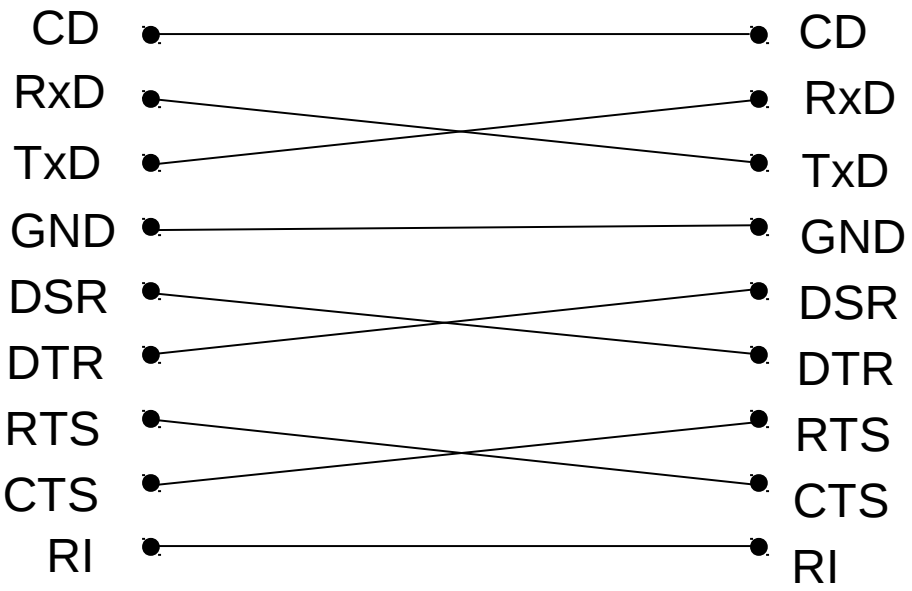
Signal functions

- CD: **Carrier Detect** The modem asserts this signal to indicate that it successfully made its connection to a remote device
- RI: **Ring Indicator** The modem asserts this signal to indicate that the phone is ringing at the other end of its connection
- DSR: **Data Set Ready** Modem to PC
- DTR: **Data Terminal Ready** PC to Modem

Signal functions (continued)

- RTS: **Request To Send** PC is ready for the modem to relay some received data
- CLS: **Clear To Send** Modem is ready for the PC to begin transmitting some data

9-wire null-modem cable

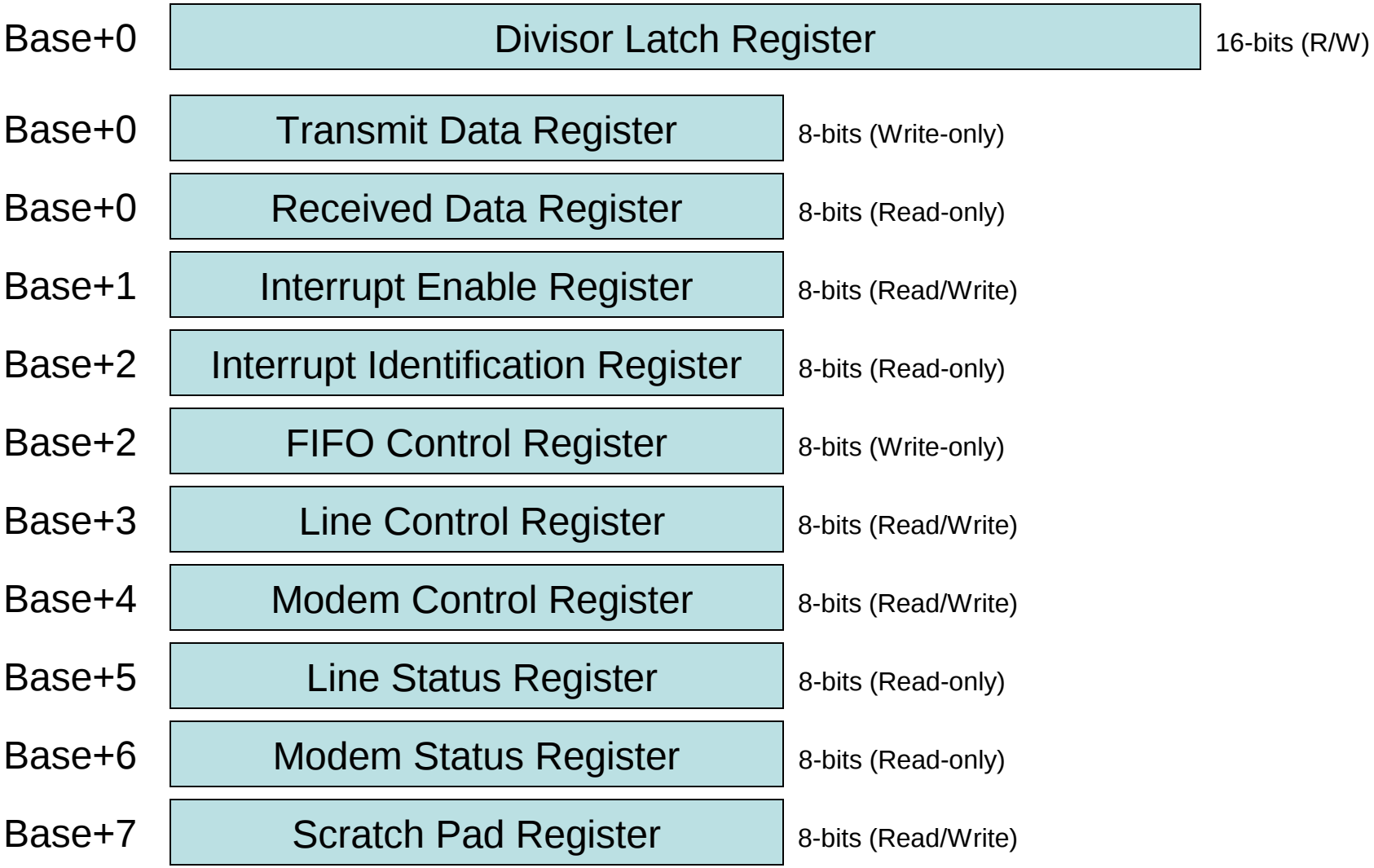


Data Terminal Equipment

Data Terminal Equipment

no modems

The 16550 UART registers



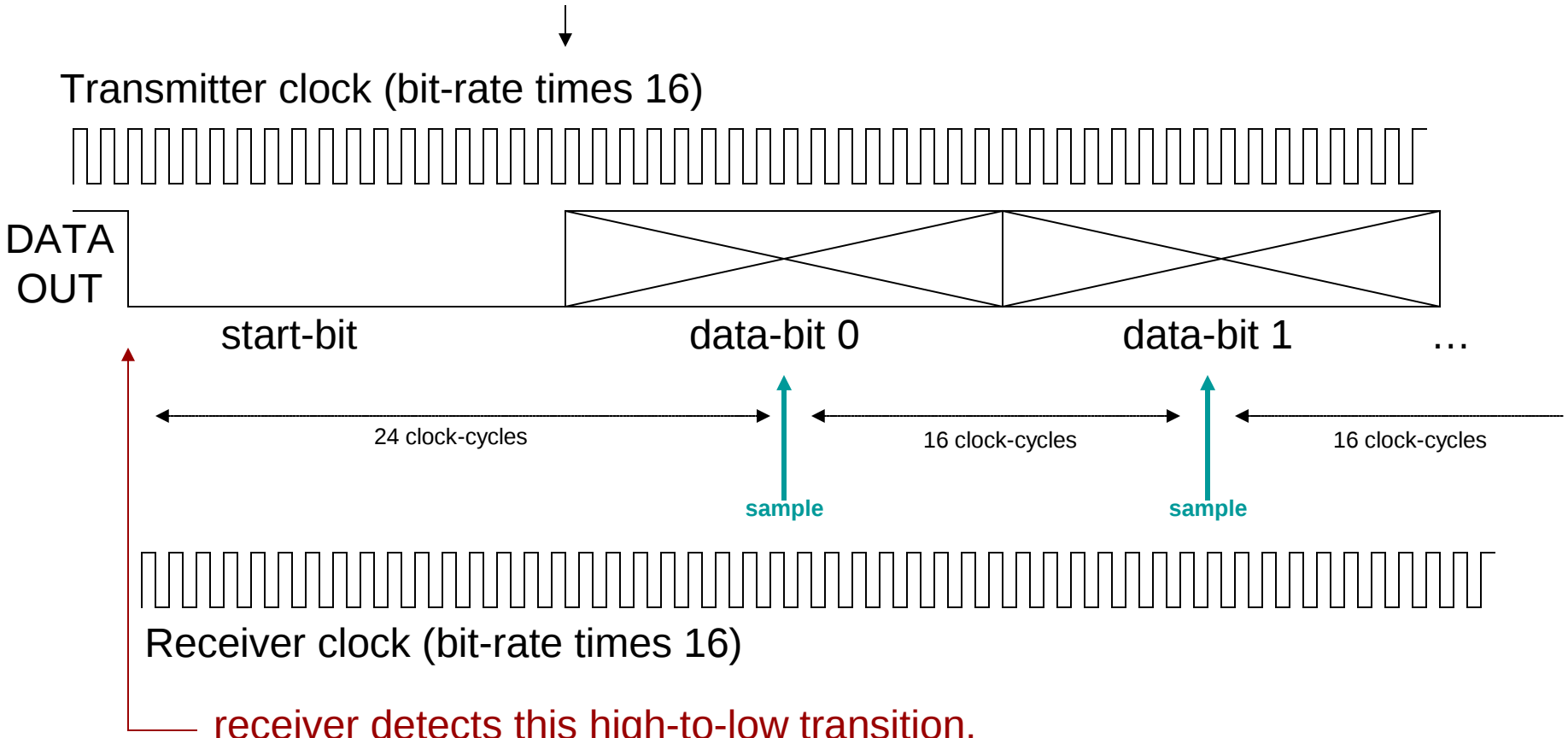
Rate of data-transfer

- The standard UART clock-frequency for PCs equals 1,843,200 cycles-per-second
- Each data-bit consumes 16 clock-cycles
- So the fastest serial bit-rate in PCs would be 1843200/16 = 115200 bits-per-second
- With one ‘start’ bit and one ‘stop’ bit, ten bits are required for each ‘byte’ of data
- Rate is too fast for ‘teletype’ terminals

Divisor Latch

- The ‘Divisor Latch’ may be used to slow down the UART’s rate of data-transfer
- Clock-frequency gets divided by the value programmed in the ‘Divisor Latch’ register
- Older terminals often were operated at a ‘baud rate’ of 300 bits-per-second (which translates into 30 characters-per-second)
- So Divisor-Latch was set to 0x0180

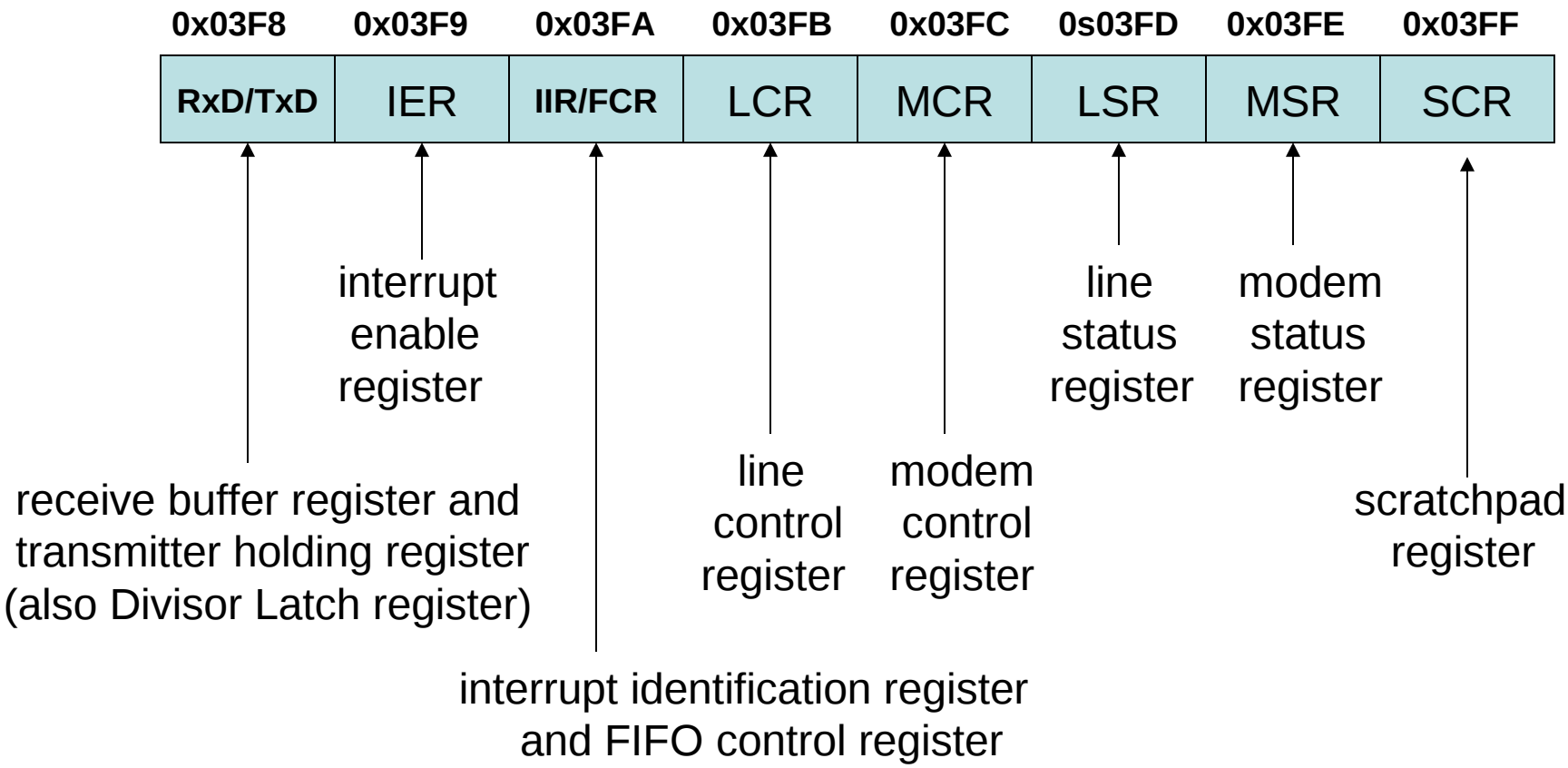
How timing works



receiver detects this high-to-low transition,
so it waits 24 clock-cycles,
then samples the data-line's voltage
every 16 clock-cycles afterward

Programming interface

The PC uses eight consecutive I/O-ports to access the UART's registers



Modem Control Register

7	6	5	4	3	2	1	0
0	0	0	LOOP BACK	OUT2	OUT1	RTS	DTR

Legend:

DTR = Data Terminal Ready (1=yes, 0=no)

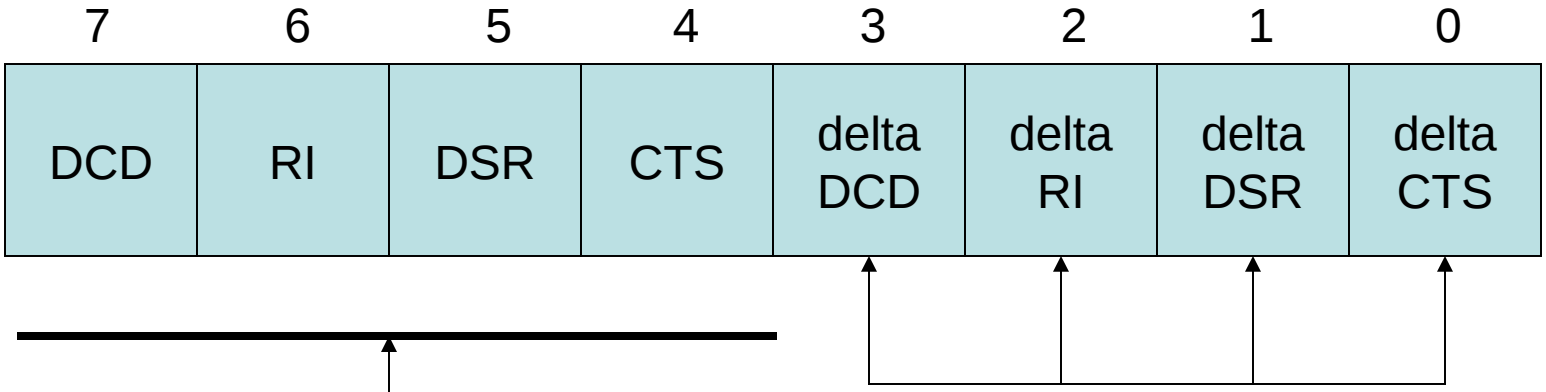
RTS = Request To Send (1=yes, 0=no)

OUT1 = not used (except in loopback mode)

OUT2 = enables the UART to issue interrupts

LOOPBACK-mode (1=enabled, 0=disabled)

Modem Status Register

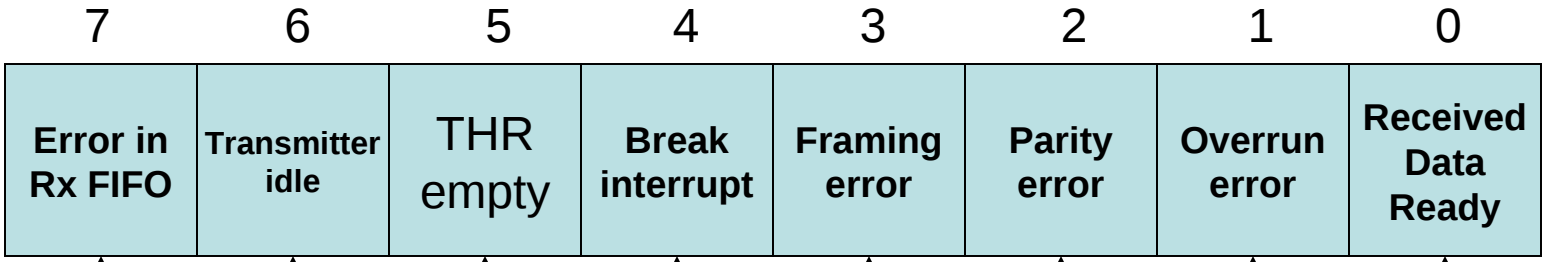


Legend:

CTS = Clear To Send (1=yes, 0=no) [bit 0 in Modem Control]
DSR = Data Set Ready (1=yes, 0=no) [bit 1 in Modem Control]
RI = Ring Indicator (1=yes,0=no) [bit 2 in Modem Control]
DCD = Data Carrier Detected (1=yes,0=no) [bit 3 in Modem Control]

[---- loopback-mode ----]

Line Status Register



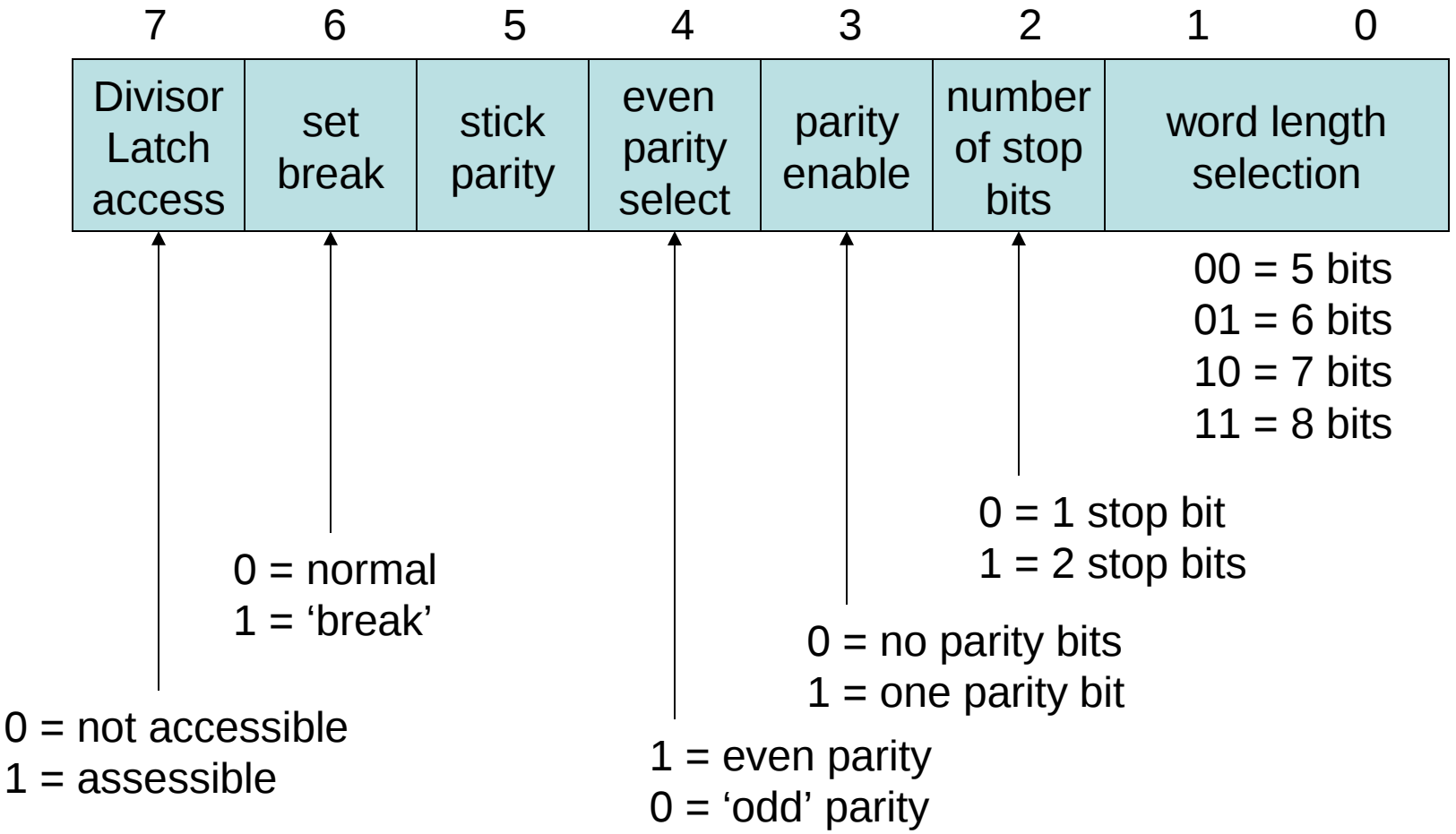
These status-bits indicate errors in the received data

This status-bit indicates that the data-transmission has been completed

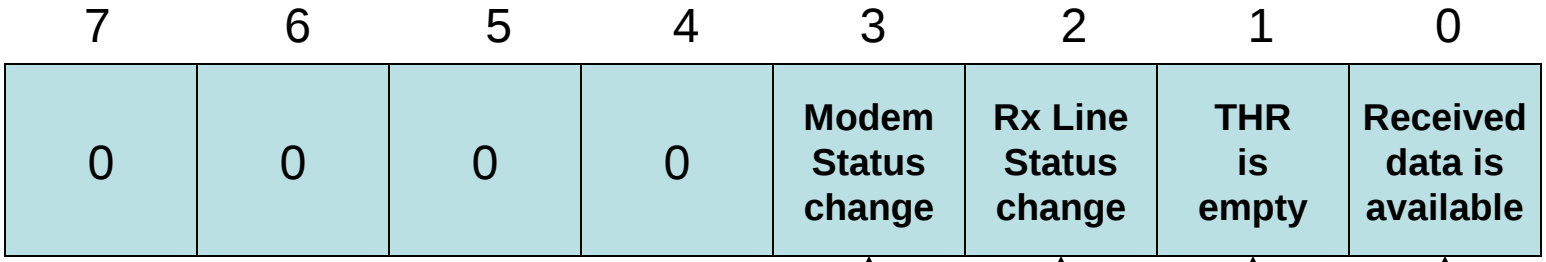
This status-bit indicates that the Transmitter Holding Register is ready to accept a new data byte

This status-bit indicates that a new byte of data has arrived (or, in FIFO-mode, that the receiver-FIFO has reached its threshold)

Line Control Register



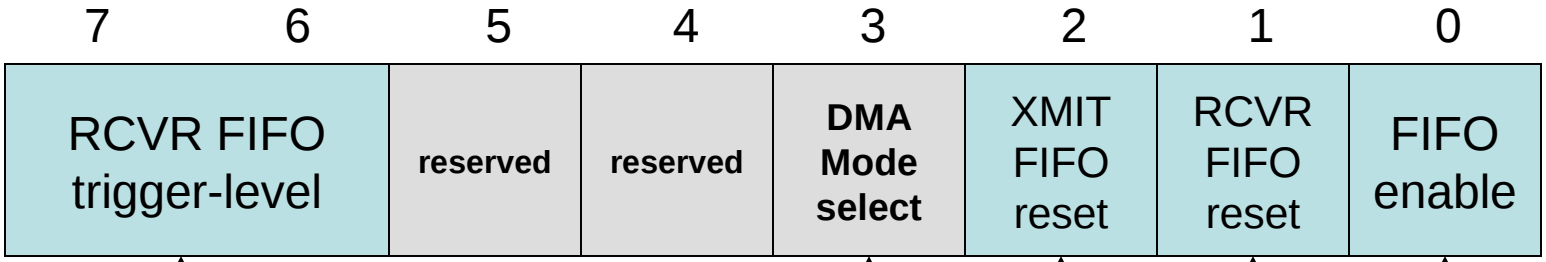
Interrupt Enable Register



If **enabled** (by setting the bit to 1),
the UART will generate an interrupt:
(bit 3) whenever modem status changes
(bit 2) whenever a receive-error is detected
(bit 1) whenever the transmit-buffer is empty
(bit 0) whenever the receive-buffer is nonempty

Also, in FIFO mode, a ‘timeout’ interrupt will be generated if neither FIFO has been ‘serviced’ for at least four character-clock times

FIFO Control Register



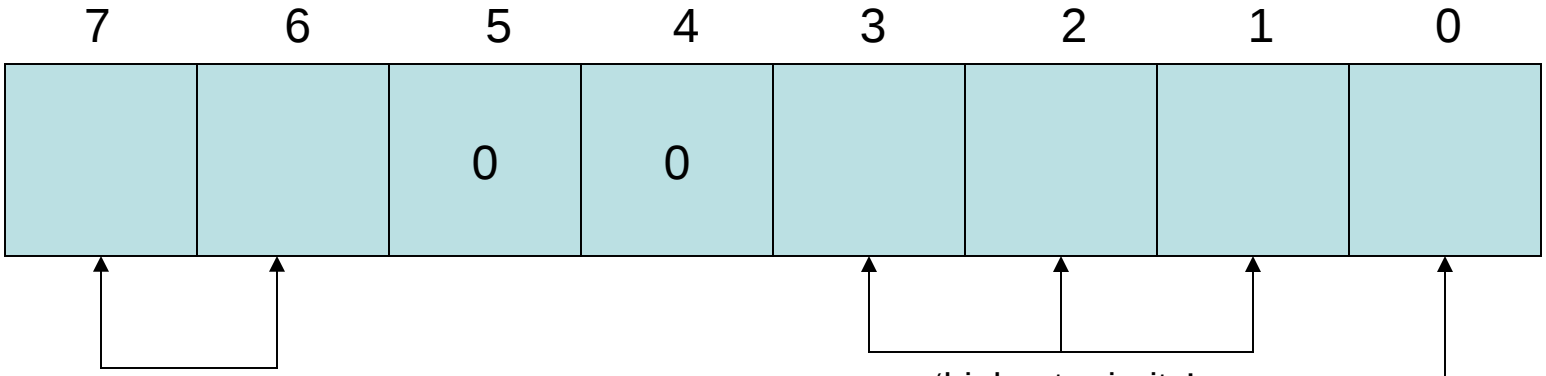
00 = 1 byte
01 = 4 bytes
10 = 8 bytes
11 = 14 bytes

NOTE: DMA is unsupported for the UART on our systems

Writing 1 empties the FIFO, writing 0 has no effect

Writing 0 will disable the UART's FIFO-mode, writing 1 will enable FIFO-mode

Interrupt Identification Register



00 = FIFO-mode has not been enabled
11 = FIFO-mode is currently enabled

‘highest priority’
UART interrupt
still pending

highest
011 = receiver line-status
010 = received data ready
100 = character timeout
001 = Tx Holding Reg empty
000 = modem-status change
lowest

1 = No UART interrupts are pending
0 = At least one UART interrupt is pending

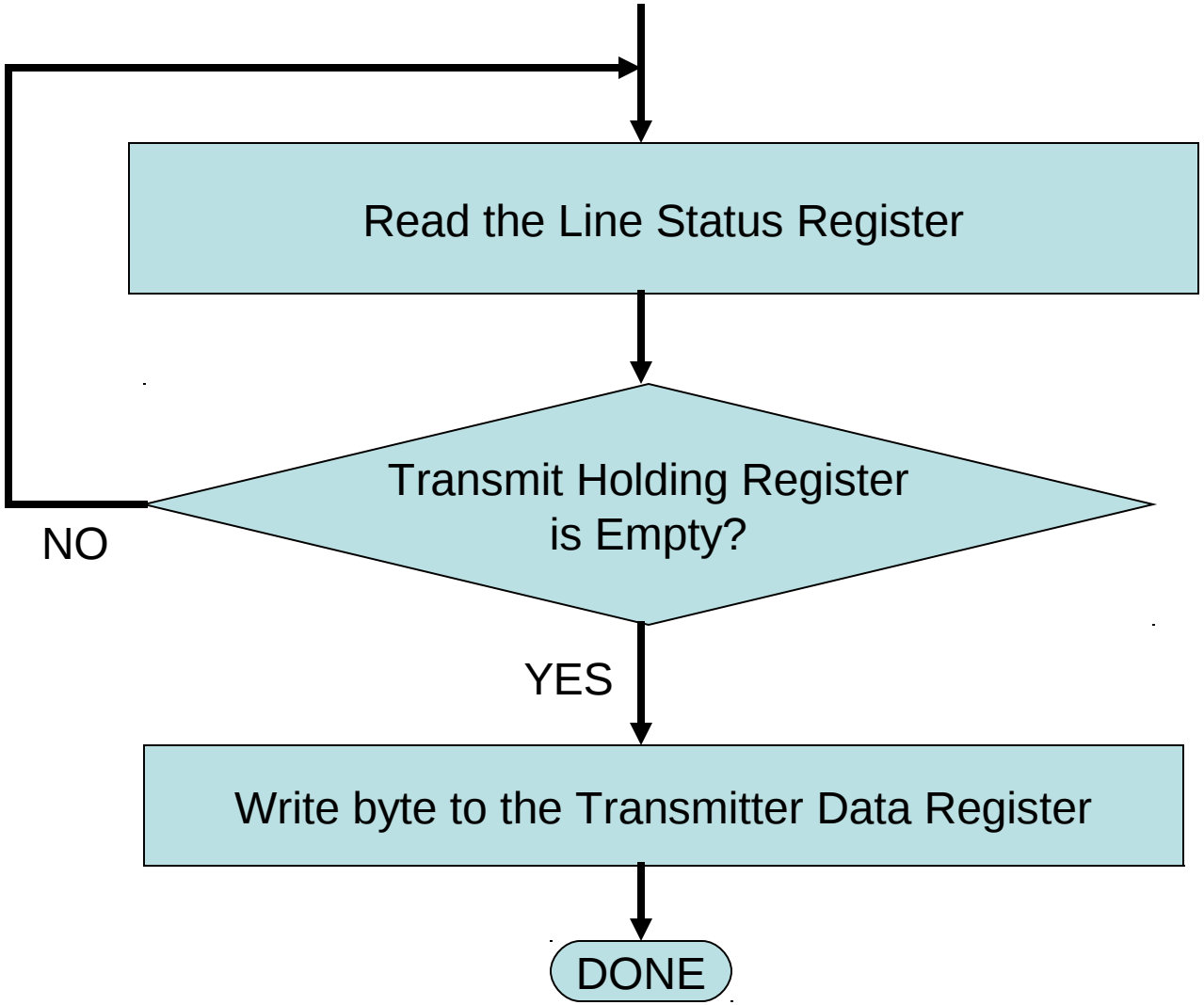
Responding to interrupts

- You need to ‘clear’ a reported interrupt by taking some action -- depending on which condition was the cause of the interrupt:
  - Line-Status: read the Line Status Register
  - Rx Data Ready: read Receiver Data Register
  - Timeout: read from Receiver Data Register
  - THRE: read Interrupt Identification Register or write to Transmitter Data Register (or both)
  - Modem-Status: read Modem Status Register

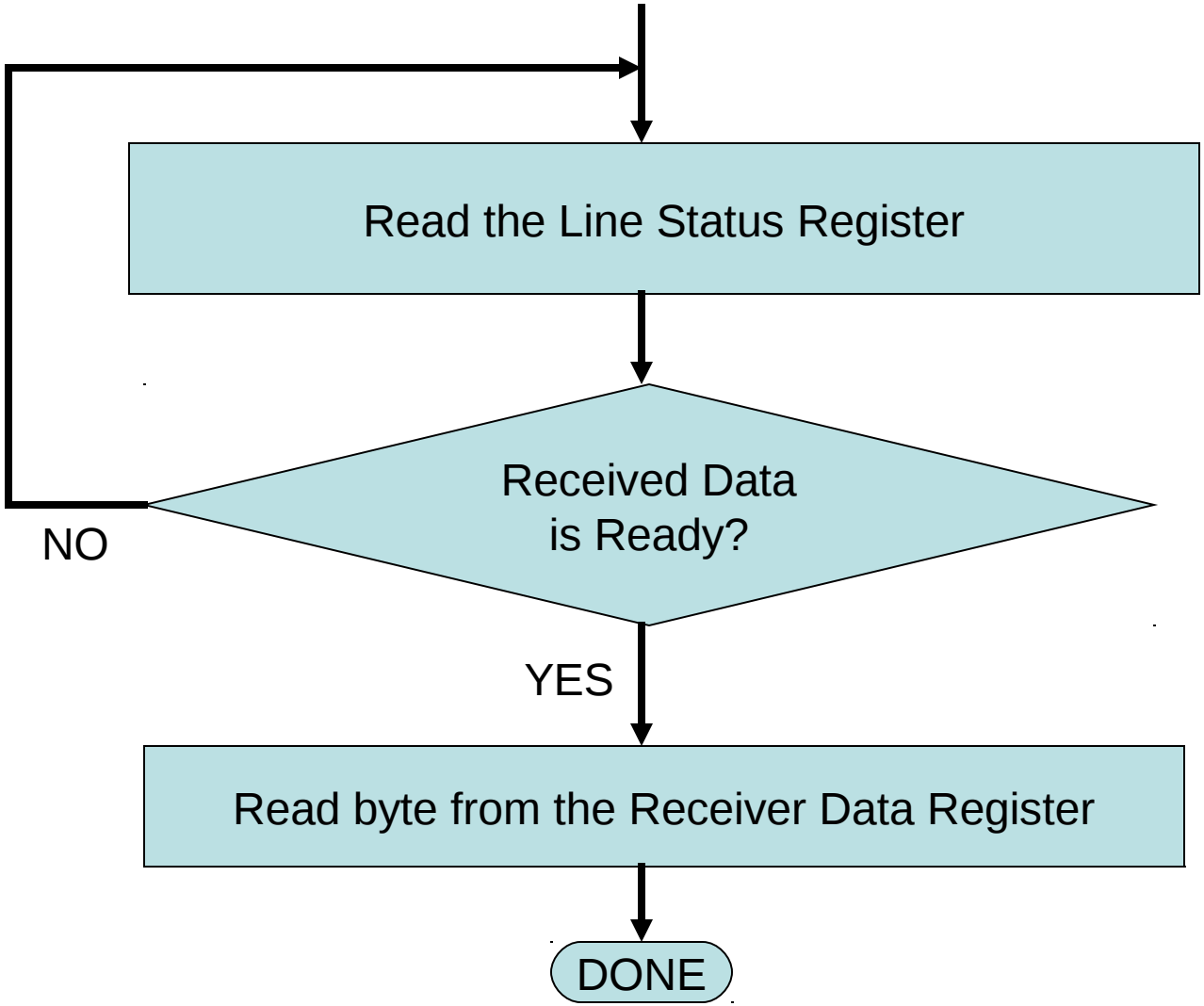
Usage flexibility

- A UART can be programmed to operate in “polled” mode or in “interrupt-driven” mode
- While “Polled Mode” is simple to program (as we shall show on the following slides), it does not make efficient use of the CPU in situations that require ‘multitasking’ (as the CPU is kept busy doing “polling” of the UART’s status instead of useful work

How to transmit a byte



How to receive a byte



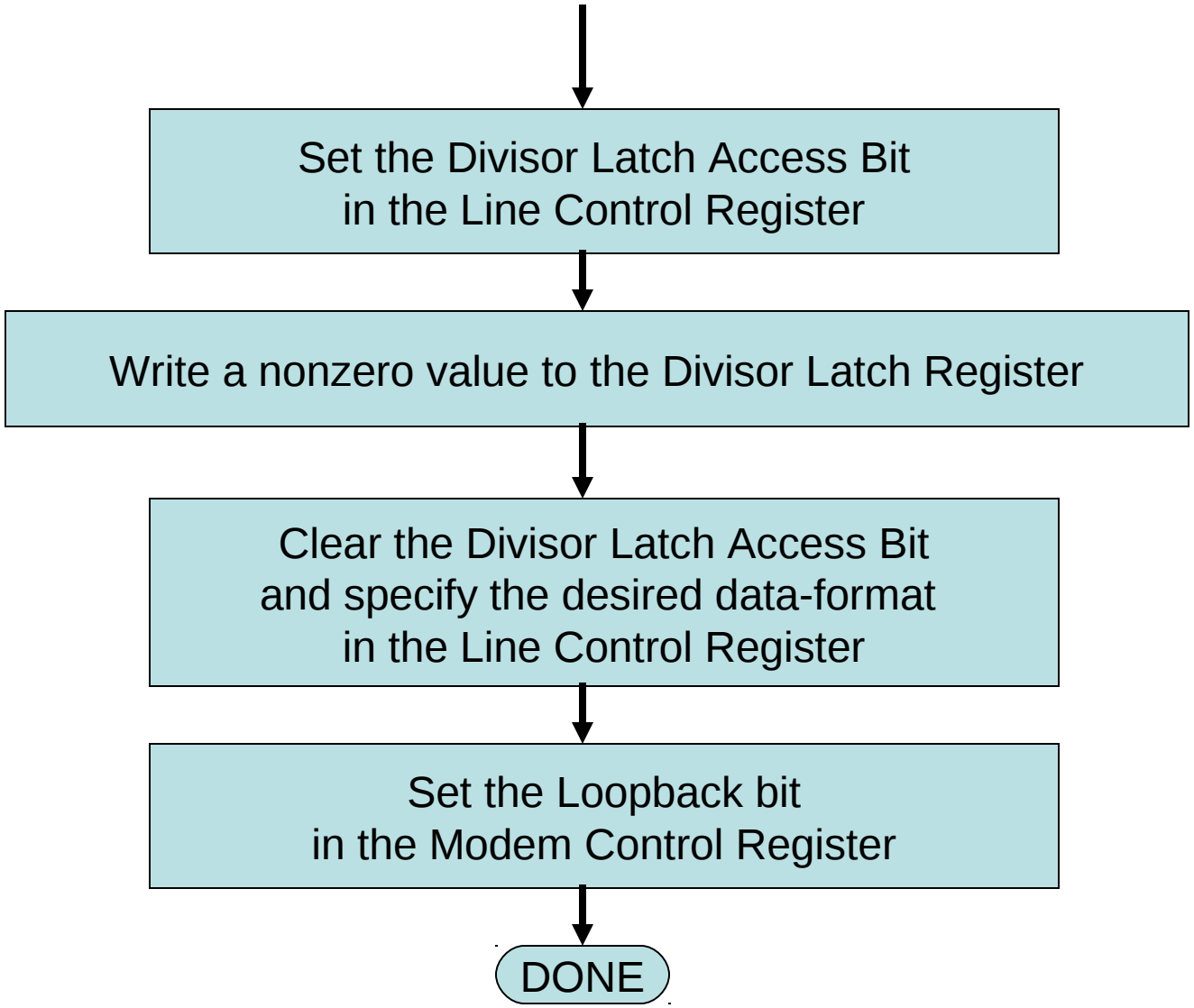
How to implement in C/C++

```
// declare the program's variables and constants
char       inch, outch = 'A';

// ---------------------- Transmitting a byte -------------------
// wait until the Transmitter Holding Register is empty,
// then output the byte to the Transmit Data Register
do { } while ( (inb( LINE_STATUS) & 0x20) == 0 );
outb( data, TRANSMIT_DATA_REGISTER );

// ---------------------- Receiving a byte ----------------------
// wait until the Received Data Ready bit becomes true,
// then input a byte from the Received Data Register
do { } while ( (inb( LINE_STATUS ) & 0x01 ) == 0 );
inch = inb( RECEIVED_DATA_REGISTER );
```

How to initialize ‘loopback’ mode



How to adjust the cpu’s IOPL

- Linux provides a system-call (to privileged programs) that need to access I/O ports
- The <sys/io.h> header-file prototypes it, and the ‘iopl()’ library-function invokes it
- The kernel will modify the CPU’s current I/O Permission Level in cpu’s EFLAGS (if the program’s owner has ‘root’ privileges)
- So you first execute the ‘iopl3’ command

In-class exercise 1

- Modify the ‘testuart.cpp’ demo-program by commenting out the instruction that places the UART into ‘loopback’ mode
- Apply the ideas presented in this lesson to create a program (named ‘uartecho.cpp’) that simply transmits each byte it receives
- Execute those two programs on a pair of PCs that are connected by a null-modem

In-class exercise 2

- Add a pair of counters to ‘testuart.cpp’:
  - Declare two integer variables (initialized to 0)

```
int       txwait = 0, rxwait = 0;
```

  - Increment these in the body of your do-loops

```
do { ++txwait; } while ( /* Transmitter is busy */ );
do { ++rxwait; } while ( /* Receiver not ready */ );
```

  - Display their totals at the demo’s conclusion

```
printf( “txwait=%d  rxwait=%d \n”, txwait, rxwait );
```

In-class exercise 3

- Modify the ‘testuart.cpp’ demo-program to experiment with using a different baud rate and a different data-format
- For example, use 300 baud and 7-N-2:
  - output 0x0180 to the Divisor Latch register
  - output 0x06 to the Line Control register
- Then, to better observe the effect, add the statement ‘fflush(stdout);’ in the program loop immediately after ‘printf(“%c”, data);’