Bit Banging

You don't need a UART to send and receive serial data. Software alone suffices. Here's the code.

Published in Embedded Systems Programming, December 1991

By Jack Ganssle

I love cheap hardware. There's little more intellectually satisfying than replacing a lot of complicated, expensive components with a bit of cleverness. To me it's a way to put a part of myself into a system; a technique to step away from the usual routine of embedded design.

This usually means replacing hardware with software, not always a smart choice unless manufacturing costs are more important than incurring additional engineering expenses. This is a tough decision to make. However, if you can beg, borrow, steal or invent a set of standard library routines that you use over and over, non-recurring engineering costs plummet.

Sometimes it's nice (or vital) to add features to a product for engineering or manufacturing reasons only. In a perfect world we'd all add extensive self-testing capability to every embedded system. A software monitor, invoked by some secret switch combination, can be invaluable for sensor field testing. It might be nice to add some sort of logging feature to a simple system to log long term analog circuit drift. Lots of applications can benefit from the operator interaction that is really only possible when a terminal link exists.

All of these dreams need an unused serial port. With the proliferation of high integration CPUs, serial ports aren't quite as scarce as a few years ago. Still, it is surprising how an application can eat up every available resource, including all of the ports.

Perhaps, we lean a little too heavily on complicated UARTs and other peripheral chips everyone takes for granted. For low speed communications a UART really is not a necessary part of the hardware. An old technique with the nearly scatological name of "bit banging" lets you easily use a pair of parallel I/O pins as a serial port.

But first, a bit of background...

RS-232

RS-232, as has been extended for microcomputer communications, defines signal levels, transfer parameters, and cabling for serial communications over short (under about 50 feet) distances. Of course, different vendors implement various aspects of the standard in different ways, so devices hardly ever work together without some frantic wire swapping.

RS-232 communications takes place one bit at a time. Each of the 8 bits of a byte are sequentially sent out over a single pair of wires.

All communications takes place at a baud rate agreed on by both the driver and receiver. 9600 baud means that each bit of the character stream takes 1/9600 second to transmit.

When the link is idle (no data being sent) it is in the Marking state (the line is more negative than -3 volts). The Start bit, which puts the line into the Spacing state (more positive than +3 volts) for one bit period, is sent first and serves to announce that a character is on the way. The receiver senses the start bit and sets itself up to read the incoming serialized byte.

Data bits follow Start. The least significant (data bit 0) goes first. One at a time, the other bits follow, each being given exactly one bit time on the link.

After the entire character has been transferred the line goes to the marking state for the length of the stop bit - one or two bit times depending on the protocol agreed to by the communicating devices. Stop bits look like an idle line. They give the UART time to recover before the next character starts.

A logic zero data bit is transmitted as a spacing line condition (positive voltage); ones go as marking bits (negative).

So, to send the character "A" (hex 41), the line assumes the following levels:

```
Marking (<-3 volts)     line idle
Spacing (> +3 volts)    Start bit
Marking (<-3 volts)     Data bit 0
Spacing (>+3 volts)     Data bit 1
Spacing (>+3 volts)     Data bit 2
Spacing (>+3 volts)     Data bit 3
Spacing (>+3 volts)     Data bit 4
Spacing (>+3 volts)     Data bit 5
Marking (<-3 volts)     Data bit 6
Spacing (>+3 volts)     Data bit 7
Marking (<-3 volts)     Stop bits
Marking (<-3 volts)     line idle
```

The RS-232 standard defines pinouts for Data Communications equipment (DCE) and Data Terminal Equipment (DTE). Terminals are DTE. Computers seem to be DTE or DCE depending on the whim of the designer. The IBM PC is DTE.

Pin Number DB-9	Pin Number DB-25	Direction	Pin Name
5	7		Ground
1			Frame ground (not used on 9 pin connectors)
3	2	DTE to DCE	Transmitted data from DTE
2	3	DCE to DTE	Received data from DCE
8	5	DCE to DTE	Clear to send (DCE ready)
7	4	DTE to DCE	Request to send (DTE ready)
6	6	DCE to DTE	Data set ready
4	20	DTE to DCE	Data terminal ready

Bit Banging

It takes a bit of hardware to convert the 8 bits of parallel character data to a serial stream, insert start/stop bits, and shift the data out. Receiving is harder, since reliable communication mandates that the bit is sampled in the middle of a bit cell. In addition, some sort of interface circuit must shift the computer's +5 and ground levels to RS-232's bizarre plus and minus voltages.

One of my biggest complaints about this industry is our servitude to RS-232; either terribly expensive power supplies or complex "charge pumps" (DC to DC converters) are used just to satisfy the standard's silly levels. Given that most RS-232 devices are within a few meters of each other, why couldn't the standard givers have settled on a more reasonable +5 & ground?

Most systems use a UART to convert parallel bytes of data from the program into a serial stream of bits and vice versa. Any UART handles a lot of other RS-232 interface chores like double buffering the data, automatic handshaking, etc. A UART is almost a tiny external co-processor that sends a character, or that accepts input data and reassembles it, signalling the CPU only when the task is done. The UART is by far the most complex part of a serial port.

You can replace the UART entirely with software if the demands placed on the port aren't too severe. A software UART replacement tediously serializes and deserializes data bytes, and so demands the full attention of the CPU. A "bit banger" software UART won't automatically assemble characters for you while the code is off doing something else; if the code isn't listening when a character comes, data will be lost.

This is not a problem for maintenance ports or back doors into a system. Usually these are invoked rarely, and drive the system into a funny mode which just replies to commands from a terminal.

A surprising number of systems that use RS-232 as a primary communications interface also use bit bangers. Where interrupts are not a problem, and where multitasking doesn't exist, a simple bit banger can be a fairly efficient main comm link.

Why is interrupt service a problem? As will be seen shortly, all of the character's timing is derived from the execution of software loops. During transmission, if the code goes off to service an interrupt the length of a bit time will be shifted. DMA can cause a similar problem, especially if the DMA timing varies.

Figure 1 shows the three subroutines needed to transmit, receive, and initialize the baud rate. This code is written in Z80 mnemonics, but will run equally well on the Z80, 64180, 8085 and NSC800 processors. It is easily adaptable to other processors, but be sure to balance the timing between subroutines.

The routines transmit and receive data through two parallel lines you must supply. It's not too hard to come up with one input and one input bit on most systems; reserve them early in the design for a serial port "back door".

BRID, the baud rate detection routine, loops for the user to type a space character. After BRID detects a start bit it waits for Start to go away (i.e., for the line to return to a logic 0) and then counts loop iterations during the six zero periods before the logic one (space is hex 20) occurs. This count is then transformed into a bit time, the basis of all timing in the transmit and receive routines.

The space character is particularly good to establish bit time, start bit and a one. An '@' might be marginally better, since its hex 40 code has 7 bits before the 1. Using '@' would require some adjustment of the timing calculations in the code.

Routine COUT sends a character by toggling the serial line high (a start bit), delaying for one bit time, and then sending data bits one at a time. It loops for a bit period between each data bit to ensure that the character's timing is correct. The routine transmits two stop bits at the end of the character.

Receiving is trickier. When routine CIN detects a start bit it delays for half a bit time to the start's center. This improves timing margins - we always sample the data stream in the center of each bit cell. Why? The line could be a little noisy, or capacitive effects may smear the exact starting edge of the bit.

The code then delays one bit time and reads data bit 0. It repeats the delay and inputs the rest of the character, shifting it into a register as it goes to reverse COUT's parallel-to-serial translation.

The principle works with any processor. Balancing execution times between the three subroutines is the most difficult part of a software UART. Intel's "Using the Intel 8085 Serial I/O Lines" application note (AP-29) is the best reference for the math behind the computations. It was published in August of 1977, and at that time carried the publication number 9800684A.

Notice that all of the operations are independent of baud rate and processor clock speeds. That is, nothing in this code knows either of these parameters. Instead, BRID just measures bit time in terms of counts through a loop, an arbitrary, relative measurement that is indeed a function of both the CPU's speed (i.e., number of loop iterations per second) and the bit rate. If you know your processor's clock rate you could dispense with the BRID routine altogether; just compute (or better, measure) the bit time and plug it into the code.

I find that on a Z80 or 64180 the code will support 9600 baud transmission if the processor runs at more than about 6 Mhz. A very slow clock will necessitate using reduced baud rates. This is due to the loop count computed in routine BRID that forms the basic unit of timing. If the count falls below 1, as it will with very fast baud rates or slow processors, then the routines' counts will be meaningless.

Level Shifting

Though the three routines replace the UART hardware, you'll still need some sort of level shifter to convert the computer's +5 and zero volt logic levels to RS-232 levels.

It's easy to run the two parallel bits though a commercial level shifter like the MAX232 chip. This bit of technology magic will automatically change to voltages meeting the RS-232 specification. You'll need PC board space for the chip itself and four capacitors.

Where cost or board space is a concern, consider driving the RS-232 lines with regular logic levels. The +5 volt logic one meets RS-232 parameters for a spacing condition. A marking condition that provides zero volts does violate the spec, but most RS-232 devices will accept it happily.

The one exception I know of is IBM's original asyncronous card (serial port) for the PC. Cut the receiver's hysterisis pin free on the board to make it work properly with logic levels as well as true RS-232.

On an ultra low cost system leave the two parallel pins unconnected. Make a little board with an RS-232 adaptor, perhaps using the MAX232 chip, that you can clip into the system whenever a terminal is needed for diagnostics.

Conclusion

I've often wondered how well an interrupt driven version of the bit banger would work. Given a free timer on some high integration CPU, why not derive all serial periods from the timer? Run the serial device as a task. Each timer interrupt could signal the software to read the input bit.

This would require synchronizing the timer to the middle of a bit cell. Run the serial stream into an interrupt input as well as parallel input. Let the first bit, the start bit, interrupt the CPU to start off the timing. Then disable that one interrupt until the character is fully assembled.

Though this requires the use of more resources, it could support DMA and some limited (depending on baud rate) background interrupt servicing. Let me know if you have tried it.

```
;
;  BRID - Determine the baud rate of the terminal. This routine
; actually finds the proper divisors BITTIM and HALFBT to run CIN
; and COUT properly.
;
;   The routine expects a space. It looks at the 6 zeroes in the
; 20h stream from the serial port and counts time from the start
; bit to the first 1.
;
;  serial_port is the port address of the input data. data_bit
; is the bit mask.
;
brid:
        in      a,(serial_port)
        and     data_bit
        jp      z,brid          ; loop till serial not busy
bri1:   in      a,(serial_port)
        and     data_bit
        jp      nz,bri1         ; loop till start bit comes
        ld      hl,-7           ; bit count
bri3:   ld      e,3
bri4:   dec     e               ; 42 machine cycle loop
        jp      nz,bri4
        nop                     ; balance cycle counts
        inc     hl              ; inc counter every 98 cycles
```

```
                                ; while serial line is low
        in      a,(serial_port)
        and     data_bit
        jp      z,bri3          ; loop while serial line low
        push    hl              ; save count for halfbt computation
        inc     h
        inc     l               ; add 101h w/o doing internal carry
        ld      (bittim),hl     ; save bit time
        pop     hl              ; restore count
        or      a               ; clear carry
        ld      a,h             ; compute hl/2
        rra
        ld      h,a
        ld      a,l
        rra
        ld      l,a             ; hl=count/2
        ld      (halfbt),hl
        ret
;
; Output the character in C
;
;  Bittime has the delay time per bit, and is computed as:
;
;  <HL>' = ((freq in Hz/baudrate) - 98 )/14
;  BITTIM = <HL>'+101H  (with no internal carry prop between bytes)
;
; and OUT to serial_high sets the serial line high; an OUT
; to serial_low sets it low, regardless of the contents set to the
; port.
;
cout:   ld      b,11            ; # bits to send
                                ; (start, 8 data, 2 stop)
        xor     a               ; clear carry for start bit
co1:    jp      nc,cc1          ; if carry, will set line high
        out     (serial_high),a ; set serial line high
        jp      cc2
cc1:    out     (serial_low),a  ; set serial line low
        jp      cc2             ; idle; balance # cycles with those
                                ; from setting output high
cc2:    ld      hl,(bittim)     ; time per bit
co2:    dec     l
        jp      nz,co2          ; idle for one bit time
        dec     h
        jp      nz,co2          ; idle for one bit time
        scf                     ; set carry high for next bit
        ld      a,c             ; a=character
        rra                     ; shift it into the carry
        ld      c,a
        dec     b               ; --bit count
        jp      nz,co1          ; send entire character
        ret
;
;  CIN - input a character to C.
;
;  HALFBT is the time for a half bit transition on the serial input
; line. It is calculated as follows:
;   (BITTIM-101h)/2 +101h
;
cin:    ld      b,9             ; bit count (start + 8 data)
ci1:    in      a,(serial_port) ; read serial line
        and     data_bit        ; isolate serial bit
        jp      nz,ci1          ; wait till serial data comes
        ld      hl,(halfbt)     ; get 1/2 bit time
```

```
ci2:    dec     l
        jp      nz,ci2          ; wait till middle of start bit
        dec     h
        jp      nz,ci2
ci3:    ld      hl,(bittim)     ; bit time
ci4:    dec     l
        jp      nz,ci4          ; now wait one entire bit time
        dec     h
        jp      nz,ci4
        in      a,(serial_port) ; read serial character
        and     data_bit        ; isolate serial data
        jp      z,ci6           ; j if data is 0
        inc     a               ; now register A=serial data
ci6:    rra                     ; rotate it into carry
        dec     b               ; dec bit count
        jp      z,ci5           ; j if last bit
        ld      a,c             ; this is where we assemble char
        rra                     ; rotate it into the character from carry
        ld      c,a
        nop                     ; delay so timing matches that in output
                                ; routine
        jp      ci3             ; do next bit
ci5:    ret
```

FIGURE 1: Bit Banger UART
