Running at 32 kHz

Introduction.
Example programs show how to receive RS-232 serial data at 300 and 1200 baud with a PIC running at 32.768 kHz.

Background. One of the PIC's greatest virtues is its tremendous speed. With instruction cycles as short as 200 nanoseconds and an inherently efficient design, the PIC leaves comparably priced micros in the dust.

This makes it easy to forget about the LP-series PICs, whose virtue is their ability to go very slow. At 32 kHz, these devices draw as little as 15µA; good news for battery- or solar-powered applications.

Life in the slow lane requires some slightly different techniques than multi-megahertz PIC design. First of all, you'll need to get acquainted with the 32.768-kHz quartz crystal. This is the crystal of choice for LP PICs. Why 32.768 kHz? It's the standard timing reference for electronic watches. It resonates exactly 2^{15} times per second, making it easy and inexpensive to build a chain of counters to generate 1-pulse-per-second ticks. Because of common use in watches, 32.768-kHz crystals are inexpensive (often less than a buck in single quantities) and accurate [±20 parts per million (ppm), compared to the ±50 ppm common in garden-variety clock crystals, or ±3000 ppm of ceramic resonators).

At a clock rate of 32.768 kHz, the PIC executes 8192 instructions per second, with instruction cycles taking about 122.1 microseconds each. Whether or not this is slow depends on your perspective—many applications spend most of their time in delay loops anyway.

The first circuit (figure 1a) is normally constructed with a comparator, because of the fast switching requirements imposed by a clock oscillator. But for the pokey 32-kHz crystal, even an internally compensated op-amp like the CA5160 works just fine. If you substitute another part for the '5160, you will have to play with the values of the feedback resistor (10k) and capacitor (0.001µF) to get the circuit to oscillate. If you use a comparator with an open-collector output, don't forget to add a pullup resistor.

The second oscillator (figure 1b) uses a CMOS inverter as its active element. In this case, the role of the inverter is played by one of the NOR gates of a 4001 chip. Adjusting the values of the 20-pF capacitors slightly will fine-tune the frequency of oscillation. Do not omit the 47-k resistor on the output. The capacitance of the OSC1 pin, or other connected logic (say a CMOS inverter or buffer) will pull the oscillator off frequency. It may even jump to a multiple of 32.768 kHz. This will throw your timing calculations way off.

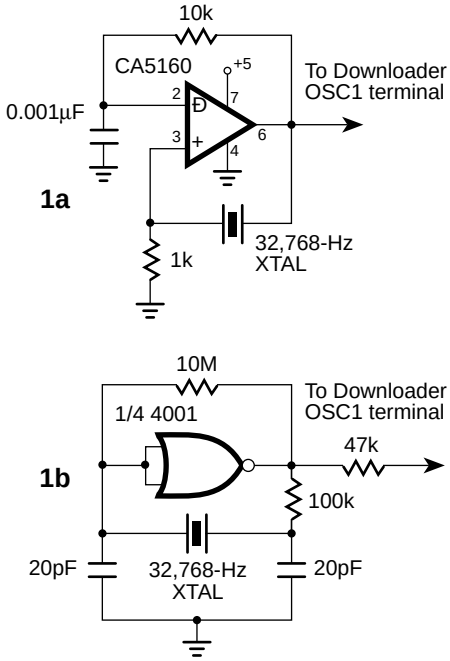


Figure 1. 32.768-kHz clocks.

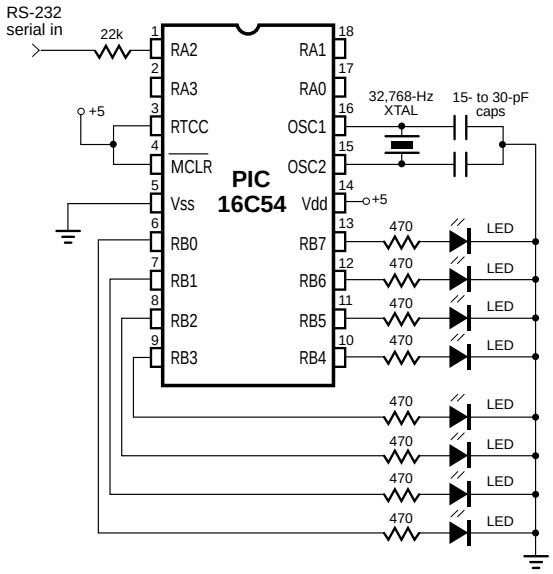
With the help of one of these oscillators, you can have Downloader convenience in the development of LP applications.

How it works. The application this time is a variation on application note number 2, receiving RS-232 data. The circuit shown in figure 2 receives RS-232 data at 300 or 1200 baud and displays it in binary form on eight LEDs connected to port rb. The baud rate depends on the program used. Listing 1 runs at 300 baud while listing 2 runs at 1200.

The previous fast-clock RS-232 application used a counter variable to determine how many trips through a *djnz* loop the PIC should take. Each loop burns up three instruction cycles, so the best resolution possible with this type of delay is 3 (oscillator period/4). When the oscillator is running at 4 MHz, resolution is a respectable 3 microseconds. However, at 32.768 kHz, the resolution of a *djnz* loop is 366.3 microseconds! There's another method that provides 1-instruction-cycle resolution: the

nop table. To use this approach, you create code like this:

```
mov     w,delay         ;Put length of delay into w.
jmp     pc+w            ;Jump w nops into the table
nop                     ;The nops. Each does nothing
```




```
nop                     ;for 1 program cycle.
...
nop
; The program continues here.
```

The number in *w* represents the number of *nop*s to be skipped, so the larger the number the shorter the delay. If you need delays of varying sizes at several points in the program, set the table up as a callable subroutine. Don't forget to account for the program cycles used by call and return; two cycles each. For long delays, it would also make sense to use a two-step approach that creates most of the delay using loops and then pads the result with nops. Otherwise, you could end up filling most of your code space with nops.

The RS-232 reception routine in listing 1 uses *nop*-table delays of 0, 12, and 15 *nop*s. If the program were expanded, other code could make use of the *nop* table, too. When writing such a program, make sure not to jump completely over the table!

Listing 2 uses the same circuit to receive 1200-baud serial data. A bit delay at 1200 baud is 833.33 µs, which is 6.8 instruction cycles at 32.768 kHz. There's no instruction that offers a 0.8-cycle delay, so the routine gets bit 0 as early as possible after the start bit is detected (7 instruction cycles) and then uses a 7-cycle delay between subsequent bits. The program is written so that it samples the first bit early, and spreads the timing error over the rest of the reception of the byte.

This approach is a little risky. It probably would produce errors if the line were noisy or the timing of the serial transmitter's clock were off. Still, it's a useful example of straight-line programming. In a real-world application, the serial transmitter would have to be set for 1.5 or 2 stop bits in order to give the program time to do anything useful with the received data. During the time afforded by an extra stop bit, the PIC could stuff the received by into a file register for later processing.

; Listing 1: SLOW1.SRC (300-baud serial receive for 32-kHz clocks)

```
; This program receives a byte of serial data and displays it on eight LEDs
; connected to port rb. Special programming techniques (careful counting of
; instruction cycles, use of a nop table) allow a PIC running at 32.768kHz to receive
; data at 300 baud.

; Remember to change device info if programming a different PIC.
                device  pic16c54,lp_osc,wdt_off,protect_off
                reset   begin

half_bit        =       14              ; Executes 1 nop in table.
bit             =       3               ; Executes 12 nops in table.
stop            =       0               ; Executes 15 nops in table
serial_in       =        ra.2
data_out        =        rb

; Variable storage above special-purpose registers.
                org     8
bit_cntr        ds      1               ; number of received bits
rcv_byte        ds      1               ; the received byte
```

```
; Org 0 sets ROM origin to beginning for program.
                org         0

                ; Set up I/O ports.
begin           mov         !ra, #4             ; Use ra.2 for serial input.
                mov         !rb, #0             ; Output to LEDs.
:start_bit      sb          serial_in           ; Detect start bit.
                jmp         :start_bit          ; No start bit? Keep watching.
                mov         w,#half_bit         ; Wait 1 nop (plus mov, call, and
                                                ; ret).
                call        nop_delay           ; Wait half bit to the middle of start
                                                ; bit.
                jnb         Serial_in, :start_bit ; Continue if start bit good.
                mov         bit_cntr, #8        ; Set counter to receive 8 data
                                                ; bits.

:receive        mov         w,#bit              ; Wait one bit time (12 nops).
                call        nop_delay
                movb        c,/Serial_in        ; Put the data bit into carry.
                rr          rcv_byte            ; Rotate carry bit into the receive
                                                ; byte.
                djnz        bit_cntr,:receive   ; Not eight bits yet? Get next bit.
                mov         w,#stop             ; Wait 1 bit time (15 nops) for stop
                                                ; bit.
                call        nop_delay
                mov         data_out, rcv_byte  ; Display data on LEDs.
                goto        begin:start_bit     ; Receive next byte.

nop_delay       jmp         pc+w
                nop                             ; w = 0—executes 15 nops.
                nop                             ; w = 1—executes 14 nops.
                nop                             ; w = 2—executes 13 nops.
                nop                             ; w = 3—executes 12 nops.
                nop                             ; w = 4—executes 11 nops.
                nop                             ; w = 5—executes 10 nops.
                nop                             ; w = 6—executes 9 nops.
                nop                             ; w = 7—executes 8 nops.
                nop                             ; w = 8—executes 7 nops.
                nop                             ; w = 9—executes 6 nops.
                nop                             ; w = 10—executes 5 nops.
                nop                             ; w = 11—executes 4 nops.
                nop                             ; w = 12—executes 3 nops.
                nop                             ; w = 13—executes 2 nops.
                nop                             ; w = 14—executes 1 nop.
                ret                             ; w = 15—executes 0 nops

; If w > 15, the program will jump into unprogrammed code memory,
; causing a reset.
```

; Listing 2: SLOW2.SRC (1200-baud serial receive for 32-kHz clocks)

```
; This program receives a byte of serial data and displays it on eight LEDs
; connected to port rb. Straight-line programming allows a PIC running at 32.768kHz
; to receive data at 1200 baud. Since timing is so critical to the operation of this
; program, the number of instruction cycles required for each instruction appears in
; () at the beginning of most comments.

; Remember to change device info if programming a different PIC.
                device  pic16c54,lp_osc,wdt_off,protect_off
                reset   begin

serial_in       =       ra.2                    ; RS-232 via a 22k resistor.
data_out        =       rb                      ; LED anodes.

; Variable storage above special-purpose registers.
                org     8
rcv_byte        ds      1                       ; The received byte.

; Org 0 sets ROM origin to beginning for program.
                org     0

; Set up I/O ports.
begin           mov     !ra, #4                 ; Use ra.2 for serial input.
                mov     !rb, #0                 ; Output to LEDs.

:start_bit      sb      serial_in               ; (2) Detect start bit.
                jmp     :start_bit              ; (2) No start bit? Keep watching.
                nop                             ; (1)
                sb      serial_in               ; (2) Confirm start bit.
                jmp     :start_bit              ; (2) False alarm? back to loop.
                nop                             ; (1)
                nop                             ; (1)
                movb    rcv_byte.0,/serial_in   ; (4) Get bit 0.
                nop                             ; (1)
                nop                             ; (1)
                nop                             ; (1)
                movb    rcv_byte.1,/serial_in   ; (4) Get bit 1.
                nop                             ; (1)
                nop                             ; (1)
                nop                             ; (1)
                movb    rcv_byte.2,/serial_in   ; (4) Get bit 2.
                nop                             ; (1)
                nop                             ; (1)
                nop                             ; (1)
                movb    rcv_byte.3,/serial_in   ; (4) Get bit 3.
                nop                             ; (1)
```

```
nop                             ; (1)
nop                             ; (1)
movb    rcv_byte.4,/serial_in   ; (4) Get bit 4.
nop                             ; (1)
nop                             ; (1)
nop                             ; (1)
movb    rcv_byte.5,/serial_in   ; (4) Get bit 5.
nop                             ; (1)
nop                             ; (1)
nop                             ; (1)
movb    rcv_byte.6,/serial_in   ; (4) Get bit 6.
nop                             ; (1)
nop                             ; (1)
nop                             ; (1)
movb    rcv_byte.7,/serial_in   ; (4) Get bit 7.
mov     data_out,rcv_byte       ; (2) Data to LEDs.
jmp     :start_bit              ; (2) Do it again
```