Note 1: Interfacing a Matrix Keypad

Introduction. This application note covers the use of matrix-encoded keypads with PIC microcontrollers. It presents an example program in Parallax assembly language for reading a 4 x 4 keypad.

Background. In order to use as few input/output (I/O) pins as possible, most keypads of eight or more switches are wired in a matrix arrangement. Instead of interfacing each set of contacts to an I/O pin, switches are wired to common row and column connections, as shown in the figure. For a given number of switches, this method can save quite a few I/O pins:

No. of Switches	Matrix (rows x columns)	I/O Pins Required
8	2 x 4	6
12	3 x 4	7
16	4 x 4	8
20	5 x 4	9

The disadvantage of matrix encoding is that it requires some additional programming. The listing shows a program that reads a 4 x 4 keypad using an 8-bit I/O port and presents the number of the key pressed on a 4-bit port.

How it works. Code at the beginning of the program sets the lower four bits of port RB (connected to the columns of the keypad) to output, and the upper four bits (rows) to input. It also sets port RA, which will drive LEDs indicating the binary number of the key pressed, to output.

The routine *scankeys* does the actual work of reading the keypad. It performs the following basic steps:

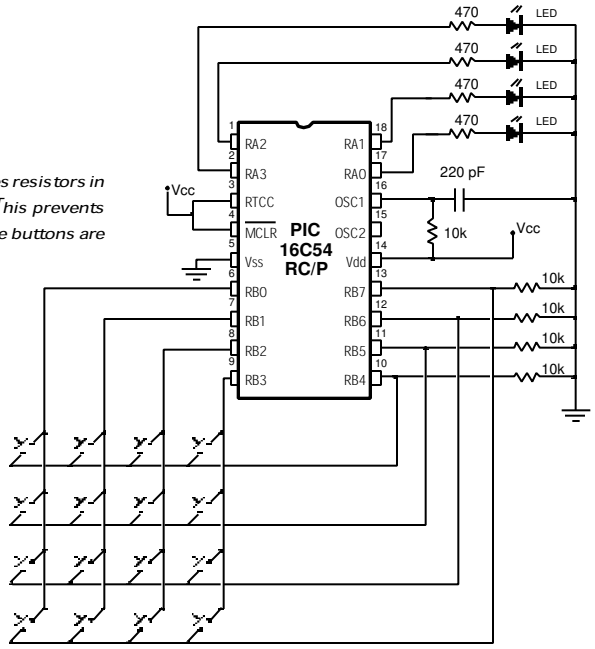
- Assert a "1" on the current column.
- Does the "1" appear on the current row?
  - > No: increment key and try the next row.
  - > Yes: exit the subroutine.
- Try the next column.

If the first key, key 0, is pressed, the routine exits with a 0 in the variable *key*, because it ends before the first increment instruction. If no key is pressed, the variable *key* is incremented 16 (010h) times. Therefore, this

number serves as a flag to the calling program that no key has been pressed.

On the issue of switch debouncing: After a switch is pressed, it may take 20 milliseconds or more for it to settle into its new state. That means that one key press can register as many repeated presses. The easiest way to defeat this problem is to read the switches, and then wait a while before reading them again. That's the purpose of the *:delay* loop in the main program.

You may want to add 1k series resistors in each column (RB0 - RB3). This prevents direct shorting if two or more buttons are pressed at the same time.



Modifications. In circumstances where electrical noise might be a problem, Microchip's data sheet on the PIC indicates that it might be wise to move the port I/O assignments to the beginning of *scankeys*. The reason is that electrostatic discharge (ESD) from the user's fingertips, or presumably any other strong electrical noise, could corrupt an I/O control register and switch an input pin to output. This would prevent the routine from reading one or more rows of the keypad.

Note 1: Interfacing a Matrix Keypad

A milder zap could conceivably cause a false input to the keypad. Some routines check for this condition by comparing two or more consecutive readings of the keys. Unless several readings match, no data is returned to the main program.

Program listing. This program may be downloaded from the Parallax BBS as KEYPAD.SRC. You can reach the BBS at (916) 624-7101.

```
; PROGRAM: KEYPAD
; This program demonstrates a simple method for scanning a matrix-encoded
; keypad. The 4 columns of the pad are connected to RB.0 through RB.3; 4 rows to
; RB.4 through RB.7. The scanning subroutine returns the code of key pressed
; (0h—0Fh) in the variable key. If no switch is pressed, the out-of-range value 10h is
; returned in key (this avoids the use of a separate flag variable).

keypad      =       rb
row1        =       rb.4
row2        =       rb.5
row3        =       rb.6
row4        =       rb.7

; Variable storage above special-purpose registers.
            org     8
cols        ds      1
key         ds      1
index       ds      1

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

; Set starting point in program ROM to zero.
            org     0
start       mov     !rb, #11110000b     ; cols out, rows in
            mov     !ra, #0             ; ra all outputs
:keys       call    scankeys
            cje     key, #16, :delay
            mov     ra, key
:delay      nop
            nop
            djnz    index,:delay
            goto    :keys

; Subroutine to scan the keypad. Assumes that the calling routine will delay long
; enough for debounce.
scankeys    clr     key
```

```
                clr     keypad
                mov     cols,#4         ; 4 x 4 keypad
                setb    c               ; put a 1 into carry

; On the first time through the following loop, the carry bit (1) is pulled into keypad.
; On subsequent loops, the lone 1 moves across the column outputs.
:scan           rl      keypad
                clrb    c               ; follow the 1 with zeros
                jb      row1, press

; If a 1 is detected, quit the routine with the current value of key. If row1, column1 is
; pressed, the value of key is 0 (on the first loop). If not, increment key and try the
; next row.
                inc     key
                jb      row2, press
                inc     key
                jb      row3, press
                inc     key
                jb      row4, press
                inc     key
                djnz    cols, :scan     ; Try all 4 columns.
press           ret                     ; Return with value in key.

; If no key is pressed, the value will be 10h due to execution of the final increment
; instruction. The program should interpret this out-of-range value as 'no press.'
```

BLANK PAGE

BLANK PAGE

Note 2: Receiving RS-232 Serial Data

Introduction. This application note presents a simple program for receiving asynchronous serial data with PIC microcontrollers. The example program, written using Parallax assembly language, displays received bytes on a bank of eight LEDs.

Background. Many controller applications involve receiving data or commands from a larger system. The RS-232 serial port is a nearly universal means for this communication. While the PIC lacks the serial receive function found on some more expensive chips, it can readily be programmed to receive serial data.

A byte of serial data is commonly sent as a string of 10 bits; a start bit, eight data bits, and a stop bit, as shown in figure 1 below. The start and stop bits help the receiver to synchronize to the incoming data bits. In some cases, a serial transmitter will lengthen the stop bit to 1.5 or 2 times the duration of the data bits in order to ensure proper sync under noisy conditions.

The speed of a serial transmission is expressed in baud or bits per second (bps). Since a complete transmission is 10 bits long, the number of bytes per second is one-tenth the baud rate. A 1200-baud signal conveys 120 bytes per second. The bit duration is 1 second divided by the baud rate. For instance, each bit of a 1200-baud signal is 833 microseconds long.

RS-232 is an electrical standard for signals used in serial communication. It represents a binary 1 with a level of -5 to -15 volts, and a 0 with +5 to +15 volts. In order for a 5-volt device like the PIC to interface with this signal, additional circuitry must convert the RS-232 signal to logic levels. The 1489 quad line receiver used in the circuit (see schematic, figure 3) can convert four RS-232 signals to 5-volt logic levels; the example circuit uses only one section of the device.

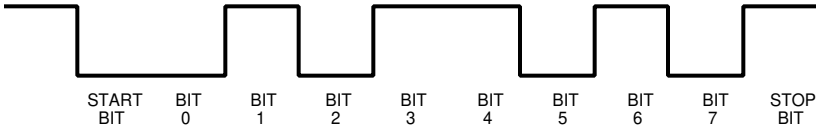


Figure 1. A byte of serial data. The byte depicted is 01011010, the ASCII code for Z.

Note 2: Receiving RS-232 Serial Data

Where cost or space is a problem, the PIC can accept the RS-232 signal through a 22k resistor, as shown in the inset to figure 3. The resistor limits the input current, while the PIC's internal clamping diodes (intended to protect against static electricity) clip the voltage to logic levels. The resistor method does not invert the RS-232 signal, so three minor changes to the program are required as shown in comments to listing 1. The method also gives up the noise rejection built into the 1489, and should not be used in noisy environments or over long cable runs.

The example presented here does not use any of the RS-232 handshaking lines. These lines help when a fast computer must communicate with (for instance) a slow printer. When the receiving device does not use the handshaking lines, it is necessary to loop them back as shown in figure 2. That way, when the computer asks for permission to send, the signal appears at its own clear-to-send pin. In effect, it answers its own question.

Although it is the most common, RS-232 is not the only serial-signaling standard. RS-422 is becoming increasingly popular because of its resistance to noise, high speed, long allowable wire runs, and ability to operate from a single-ended 5-volt power supply. Since the only difference between RS-232 and RS-422 is the electrical interface, conversion requires only the substitution of an RS-422 line receiver chip.

How it works. The example program in listing 1 is a no-frills algorithm

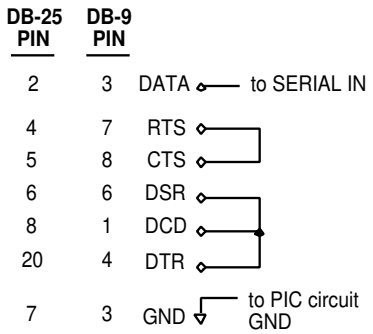


Figure 2. Hookups for standard 9- and 28-pin connectors. Connecting RTS to CTS disables normal handshaking, which is not used here.

for receiving serial data in the popular N81 format; i.e., no parity bit, eight data bits, and one stop bit. Listing 2 is a BASIC program for sending individual bytes to the circuit.

Listing 1 begins by setting up the input and output bits. It then enters a loop waiting to detect the start bit. Once the start bit is detected, the program waits one-half bit time and checks to see whether the start bit is still present. This helps ensure that the program isn't fooled by a noise burst into trying to receive a nonexistent transmission. It also makes sure that subsequent bits are read during the middle of their time slots; another precaution against noise.

Once it detects and verifies the start bit, the program enters another loop that does the actual job of receiving the data. It works like this:

- Wait one bit time.
- Copy input bit to carry bit.
- Rotate the receive byte right.
- Decrement the bit counter.
- Is the counter zero?
  > No, loop again.
  > Yes, exit the loop.

If you are unfamiliar with the rotate right (rr) instruction, you may not see how the input bit gets from the carry bit into the receive byte. Performing an rr on a byte moves its bits one space to the right. Bit 7 goes to bit 6, bit 6 to bit 5, and so on. Bit 0 is moved into the carry bit. The carry bit moves into bit 7.

Once the byte is received, the program waits a final bit delay (until the middle of the stop bit), copies the received byte to the output port to which the LEDs are connected, and goes back to the beginning to await another start bit.

The program can be set up for most standard data rates. The table lists PIC clock speeds and values of the bit time constant *bit_K* (declared at the beginning of listing 1) for a wide range of common rates. For other combinations of clock speed and data rate, just replace the delay routines with ones that provide the appropriate timing. The footnote to the table gives general guidance.

Note 2: Receiving RS-232 Serial Data

One final hardware note: Although timing isn't overly critical for receiving this type of serial data, resistor/capacitor timing circuits are inadequate. The PIC's RC clock is specified to fairly loose tolerances (up to ±28 percent) from one unit to another. The values of common resistors and capacitors can vary substantially from their marked values, and can change with temperature and humidity. Always use a ceramic resonator or crystal in applications involving serial communication.

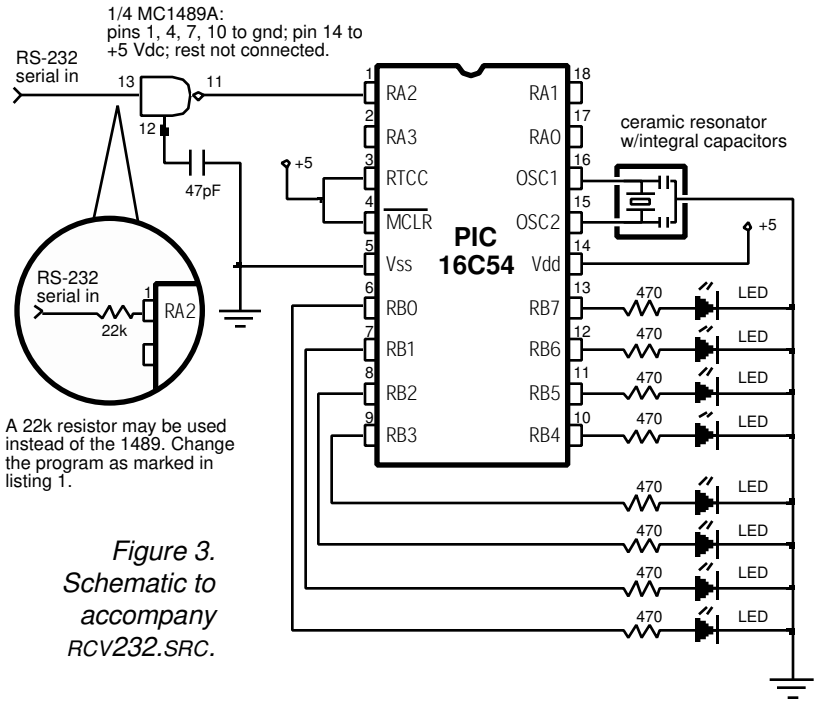


Figure 3. Schematic to accompany RCV232.SRC.

Program listing. This program may be downloaded from the Parallax BBS as RCV232.SRC. You can reach the BBS at (916) 624-7101.

Note 2: Receiving RS-232 Serial Data

Values of Timing Constant Bit_K for Various Clock Speeds and Bit Rates

Clock Frequency	Serial Bit Rate (bit time)						
	300 (3.33 ms)	600 (1.66 ms)	1200 (833 µs)	2400 (417 µs)	4800 (208 µs)	9600 (104 µs)	19,200 (52 µs)
1 MHz	206	102	50	24	—	—	—
2 MHz	—	206	102	50	24	—	—
4 MHz	—	—	206	102	50	24	—
8 MHz	—	—	—	206	102	50	24

Other combinations of clock speed and bit rate can be supported by changing the bit_delay and start_delay subroutines. The required bit delay is $\frac{1}{\text{bit rate}}$. For example, at 1200 baud the bit delay is $\frac{1}{1200}$ = 833µs. The start delay is half of the bit delay; 416µs for the 1200-baud example. Calculate the time delay of a subroutine by adding up its instruction cycles and multiplying by $\frac{4}{\text{clock speed}}$. At 2 MHz, the time per instruction cycle is $\frac{4}{2,000,000}$ = 2µs.

```
; PROGRAM: RCV232
; This program receives a byte of serial data and displays it on eight LEDs
; connected to port RB. The receiving baud rate is determined by the value of the
; constant bit_K and the clock speed of the PIC. See the table in the application note
; (above) for values of bit_K. For example, with the clock running at 4 MHz and a
; desired receiving rate of 4800 baud, make bit_K 50.

bit_K       =       24              ; Change this value for desired
                                    ; baud rate as shown in table.
half_bit    =       bit_K/2
serial_in   =       ra.2
data_out    =       rb

; Variable storage above special-purpose registers.
            org     8
delay_cntr  ds      1               ; counter for serial delay routines
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

; Remember to change device info if programming a different PIC. Do not use RC
; devices. They are not sufficiently accurate or stable for serial communication.

            device  pic16c54,xt_osc,wdt_off,protect_off
            reset   begin

; Set up I/O ports.

begin       mov     !ra, #00000100b ; Use RA.2 for serial input.
```

Note 2: Receiving RS-232 Serial Data

```
                mov         !rb, #0                 ; Output to LEDs.

:start_bit      snb         serial_in               ; Detect start bit. Change to sb
                                                    ; serial_in  if using 22k resistor
                                                    ; input.

                jmp         :start_bit              ; No start bit yet? Keep watching.
                call        start_delay             ; Wait one-half bit time to the
                                                    ; middle of the start bit.

                jb          Serial_in, :start_bit   ; If the start bit is still good,
                                                    ; continue. Otherwise, resume
                                                    ; waiting.

                                                    ; Change to jnb Serial_in, :start_bit
                                                    ; if using 22k resistor input.

                mov         bit_cntr, #8            ; Set the counter to receive 8 data
                                                    ; bits.

                clr         rcv_byte                ; Clear the receive byte to get
                                                    ; ready for new data.

:receive        call        bit_delay               ; Wait one bit time.
                movb        c,Serial_in             ; Put the data bit into carry.
                                                    ; Change to movb  c,/Serial_in if
                                                    ; using 22k resistor input.

                rr          rcv_byte                ; Rotate the carry bit into the
                                                    ; receive byte.

                djnz        bit_cntr,:receive       ; Not eight bits yet? Get next bit.
                call        bit_delay               ; Wait for stop bit.
                mov         data_out, rcv_byte      ; Display data on LEDs.

                goto        begin:start_bit         ; Receive next byte.

; This delay loop takes four instruction cycles per loop, plus eight instruction cycles
; for other operations (call, mov, the final djnz, and ret). These extra cycles become
; significant at higher baud rates. The values for bit_K in the table take the time
; required for additional instructions into account.

bit_delay       mov         delay_cntr,#bit_K
:loop           nop
                djnz        delay_cntr, :loop
                ret

; This delay loop is identical to bit_delay above, but provides half the delay time.

start_delay     mov         delay_cntr,#half_bit
```

```
:loop       nop
            djnz        delay_cntr, :loop
            ret
```

BASIC Program for Transmitting Bytes via COM1

```
10  REM  Open the serial port com1. Substitute the desired baud rate
20  REM  for 9600 in this line. The parameters CD0, CS0, DS0, and OP0
30  REM  serve to disable hardware handshaking. They may be omitted if
40  REM  these lines are looped back as shown in figure 2.
50  OPEN “com1:9600,N,8,1,CD0,CS0,DS0,OP0” FOR OUTPUT AS #1
60  CLS
70  REM  At the prompt, enter a value between 0 and 255 representing a
80  REM  byte of data to send out the serial port.
90  INPUT “ASCII code to send:  “, A%
100 PRINT #1, CHR$(A%);
110 GOTO 60
120 END
```

BLANK PAGE

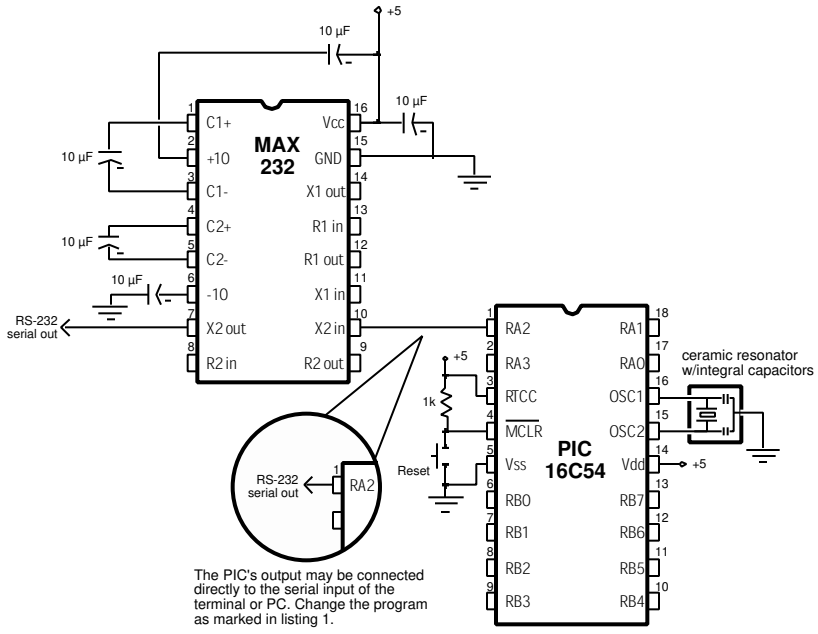
Note 3: Sending RS-232 Serial Data

Introduction. This application note covers transmission of asynchronous serial data using PIC microcontrollers. It presents an example program in Parallax assembly language that transmits a text string serially via RS-232 at speeds up to 19,200 bits per second. Hardware examples demonstrate the use of a popular serial line driver, as well as a method for using the PIC's output to directly drive a serial line.

Background. Many PIC applications require sending data to a larger computer system. The common RS-232 serial port is almost ideal for this communication. While the PIC lacks the onboard serial communication hardware available with some more expensive controllers, it can readily be programmed to add this capability.

A previous application note (#2, Receiving RS-232 Serial Data) covered the data format and RS-232 signals in some detail. Here are the highlights:

A byte of serial data is commonly sent as 10 bits: a start bit, eight data bits, and a stop bit. The duration of these bits is calculated by dividing



the rate in bits per second (commonly baud) into 1 second. The stop bit can be stretched to any desired duration.

Under the RS-232 standards, a high (1) is represented by a negative voltage in the range of -5 to -15 volts. A low (0) is a voltage from +5 to +15 volts. In addition to connections for data input, output, and ground, most RS-232 ports include handshaking lines that help devices turn the flow of data on and off when necessary (for instance, when a printer is receiving data faster than it can store and process it). Many applications avoid the use of these hardware signals, instead embedding flow-control commands in the data stream itself.

A traditional method of sending serial data is to use a parallel-in serial-out shift register and a precise clock. The start, data, and stop bits are loaded into the register in parallel, and then shifted out serially with each tick of the bit-rate clock. You can easily use a software version of this method in a PIC program. Really, the most difficult part of transmitting an RS-232-compatible signal from the PIC is achieving the signaling voltages. However, the hardware example presented here shows that:

- There are devices that generate the RS-232 voltages from a single-ended 5-volt supply.
- It is possible to use the PIC's output to directly drive an RS-232 input, if the cable is kept short.

How it works. The PIC program in listing 1 sends a short text string as a serial data stream whenever the PIC is reset. To reset the PIC, push and release the reset button shown in the schematic (if you are using the Parallax Downloader, omit S1 and use the built-in reset switch). If you lack appropriate terminal software for your computer, listing 2 is a BASIC program that will help get you started. If you use other terminal software and get "device timeout" errors, try cross-connecting the handshaking lines as shown in figure 2 below. This has the effect of disabling handshaking. The program in listing 2 does this in software.

The PIC program starts by setting port RA to output. Serial data will be transmitted through pin RA.2. The program consists of two major loops, defined by the labels *:again* and *:xmit*. The first loop, *:again*, initializes the

bit counter, and then gets a byte from the subroutine/table called *string*. Once *string* returns a byte in the *w* register, the program puts this byte into *xmt_byte*. After a start bit is sent by clearing the serial-out pin, *:xmit* takes over. It performs the following steps:

- Rotate the transmit byte right (into carry).
- Move the carry bit to serial output.
- Wait one bit time (set by *bit_K*).
- Decrement the bit counter.
- Is the counter zero?
  > No, loop again.
  > Yes, exit the loop.

With the *:xmit* loop finished, the program sets the serial-out pin to send a stop bit, and checks to see whether it has reached the end of the string. If not, it goes back to *:again* to retrieve another byte. If it is done with the string, the program enters an endless loop.

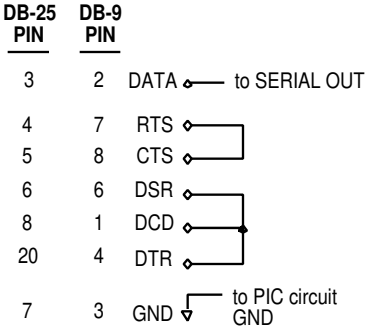


Figure 2. Hookups for standard 9- and 28-pin connectors. Connecting RTS to CTS disables normal handshaking, which is not used here.

The hardware portion of the application is fairly straightforward. The PIC produces logic-level signals that the MAX232 chip converts to acceptable RS-232 levels, approximately ±10 volts. The MAX232 has onboard charge-pumps that use the external capacitors to build up the required signaling voltages. There are other chips that provide addi-

tional features, such as additional I/O channels, higher signaling rates, lower power consumption, and the ability to work without external capacitors.

Modifications. The MAX232 draws more current and costs more (in single-part quantities) than the PIC it supports. Most of the time, this is still a bargain compared to adding a bipolarity power supply to a device just to support RS-232 signaling. In fact, the MAX232 has excess current capacity that can be used to power other small bipolarity devices, such as low-power op-amps. Depending on the application, additional voltage regulation may be required.

If you plan to use the serial port infrequently, consider powering the MAX chip through one of the PIC's I/O pins. Your program could turn the MAX on shortly before it needed to send a serial message, and turn it back off afterward. This option is especially attractive for uses that require wringing the most life out of a set of batteries.

A sample MAX232 that we tested drew 15 mA while driving two simulated serial-port loads. This is well within the PIC's drive capability. A further test showed that the MAX232's output voltages rose to their full ±10-volt levels about 1.5 milliseconds after power was applied to the chip. If your program switches the MAX232 on and off, program a 1.5-ms delay between turning the device on and sending the first bit.

In some cases, you may be able to dispense with the serial line driver altogether. Make the changes marked in listing 1 "for direct connection" and wire RA.2 directly to the serial receive connection of your terminal or PC. The changes in the program invert the logic of the serial bits, so that a 1 is represented by an output of 0 volts and a 0 is a 5-volt output. According to RS-232, the receiving device should expect voltages of at least -3 volts for a 1 and +3 volts for a 0. Voltages lying between +3 and -3 are undefined, meaning they could go either way and still comply with the standard.

As a practical matter, though, it wouldn't be smart to let 0 volts be interpreted as a 0, since this is a serial start bit. Any time the serial input was at ground potential, the terminal or PC would attempt to receive incoming serial data. This would cause plenty of false receptions. Serial-port designers apparently take this into account and set the 0 threshold somewhere above ground.

Note 3: Sending RS-232 Serial Data

That's why this cheap trick works. Don't count on it to provide full RS-232 performance, especially when it comes to sending data rapidly over long cables. Keep cables short and you shouldn't have any problems (19,200 baud seems to work error-free through 6 feet of twisted pair).

If your application will have access to the handshaking pins, you may be able to steal enough power from them to eliminate batteries entirely. According to the RS-232 specifications, all signals must be capable of operating into a 3000-ohm load. Since many computers use ±12 volts for their RS-232 signals, each line should be capable of delivering 4 mA. In practice, most can provide more, up to perhaps 15 mA. The only problem with exploiting this free power is that the software running on the PC or terminal must be written or modified to keep the handshaking lines in the required state.

One final hardware note: Although timing isn't overly critical for transmitting serial data, resistor/capacitor timing circuits are inadequate. The PIC's RC clock is specified to fairly loose tolerances (up to ±28 percent) from one unit to another. The values of common resistors and capacitors can vary substantially from their marked values, and can change with temperature and humidity. Always use a ceramic resonator or crystal in applications involving serial communication.

Program listing. This program may be downloaded from the Parallax BBS as XMIT232.SRC. You can reach the BBS at (916) 624-7101.

Values of Timing Constant Bit_K for Various Clock Speeds and Bit Rates

Clock Frequency	Serial Bit Rate (bit time)						
	300 (3.33 ms)	600 (1.66 ms)	1200 (833 µs)	2400 (417 µs)	4800 (208 µs)	9600 (104 µs)	19,200 (52 µs)
1 MHz	206	102	50	24	—	—	—
2 MHz	—	206	102	50	24	—	—
4 MHz	—	—	206	102	50	24	—
8 MHz	—	—	—	206	102	50	24

Other combinations of clock speed and bit rate can be supported by changing the bit_delay and start_delay subroutines. The required bit delay is $\frac{1}{\text{bit rate}}$. For example, at 1200 baud the bit delay is $\frac{1}{1200}$ = 833µs. The start delay is half of the bit delay; 416µs for the 1200-baud example. Calculate the time delay of a subroutine by adding up its instruction cycles and multiplying by $\frac{4}{\text{clock speed}}$. At 2 MHz, the time per instruction cycle is $\frac{4}{2,000,000}$ = 2µs.

Note 3: Sending RS-232 Serial Data

```
; PROGRAM: XMIT232
; This program transmits a string of serial data. The baud rate is determined by the
; value of the constant bit_K and the clock speed of the PIC. See the table in the
; application note (above) for values of bit_K. For example, with the clock running at
; 4 MHz and a desired transmitting rate of 4800 baud, make bit_K 50.

bit_K           =       24      ; 24 for 19,200-baud operation @ 8 MHz
serial_out      =       ra.2

; Variable storage above special-purpose registers.
                org     8

delay_cntr      ds      1                       ; counter for serial delay routines
bit_cntr        ds      1                       ; number of transmitted bits
msg_cntr        ds      1                       ; offset in string
xmt_byte        ds      1                       ; the transmitted byte

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

; Remember to change device info if programming a different PIC. Do not use RC
; devices. They are not sufficiently accurate or stable for serial communication.
                device  pic16c54,xt_osc,wdt_off,protect_off
                reset   begin

begin           mov     !ra, #00000000b         ; Set port to output.
                mov     msg_cntr, #0            ; Message string has nine
                                                ; characters; 0 through 8.
:again          mov     bit_cntr,#8             ; Eight bits in a byte.
                mov     w,msg_cntr              ; Point to position in the string.
                call    string                  ; Get next character from string.
                mov     xmt_byte,w              ; Put character into transmit byte.
                clrb    serial_out              ; Change to setb serial_out for
                                                ; direct connection.
                call    bit_delay               ; Start bit.
:xmit           rr      xmt_byte                ; Rotate right moves data bits into
                                                ; carry, starting with bit 0.
                movb    serial_out,c            ; Change to movb serial_out,/c for
                                                ; direct connection.
                call    bit_delay               ; Data bit.
                djnz    bit_cntr,:xmit          ; Not eight bits yet? Send next bit.
                setb    serial_out              ; Change to clrb serial_out for
                                                ; direct connection
                call    bit_delay               ; Stop bit.
                inc     msg_cntr                ; Add one to the string pointer.
                cjbe    msg_cntr,#8, :again     ; More characters to send? Go to
                                                ; the top.

:endless        goto    :endless                ; Endless loop. Reset controller to
                                                ; run program.
```

Note 3: Sending RS-232 Serial Data

```
; To change the baud rate, substitute a new value for bit_K at the beginning of this
; program.

bit_delay       mov     delay_cntr,#bit_K
:loop           nop
                djnz    delay_cntr, :loop
                ret

string          jmp     pc+w                    ; Message string consisting of
                                                ; 'Parallax' followed by a linefeed.
                retw    'P','a','r','a','l','l','a','x',10
```

BASIC Program for Receiving Text String via COM1

```
10 CLS
15 REM Substitute desired baud rate for 19200 in the line below.
20 OPEN "com1:19200,N,8,1,CD0,CS0,DS0,OP0" FOR INPUT AS #1
30 IF NOT EOF(1) THEN GOSUB 200
40 GOTO 30
200 Serial$ = INPUT$(LOC(1), #1)
210 PRINT Serial$;
220 RETURN
```

BLANK PAGE

Note 4: Reading Rotary Encoders

Introduction. This application note covers the use of incremental rotary encoders with PIC microcontrollers. It presents an example program in Parallax assembly language for reading a typical encoder and displaying the results as an up/down count on a seven-segment LED display.

Background. Incremental rotary encoders provide a pair of digital signals that allow a microcontroller to determine the speed and direction of a shaft's rotation. They can be used to monitor motors and mechanisms, or to provide a control-knob user interface. The best-known application for rotary encoders is the mouse, which contains two encoders that track the x- and y-axis movements of a ball in the device's underside.

Rotary encoders generally contain a simple electro-optical mechanism consisting of a slotted wheel, two LEDs, and two light sensors. Each

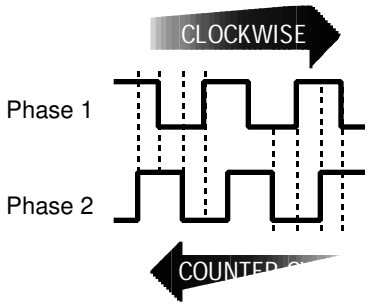


Figure 1. Quadrature waveforms from a rotary encoder contain directional information.

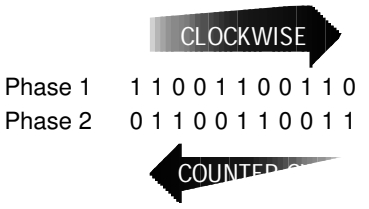
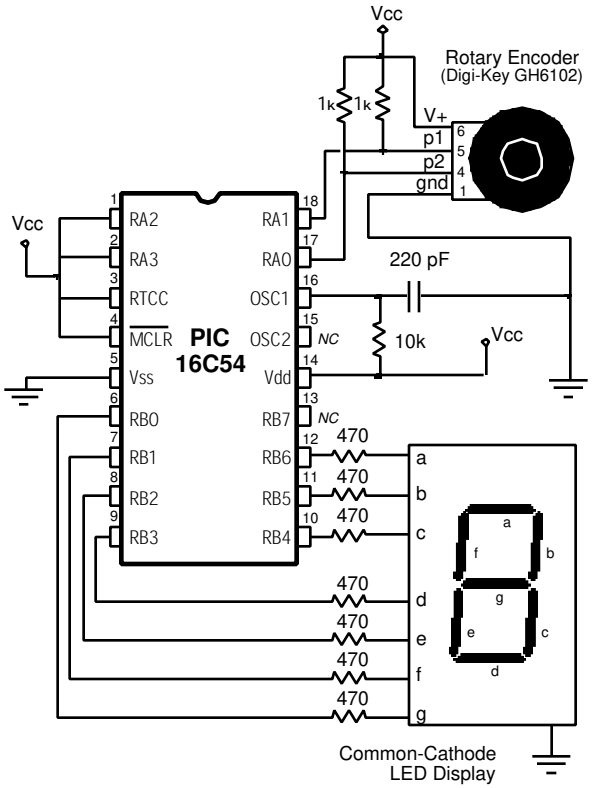


Figure 2. Sequence of two-bit numbers output by the phases of a rotary encoder.

Note 4: Reading Rotary Encoders

LED/sensor pair is arranged so that the devices face each other through the slots in the wheel. As the wheel turns, it alternately blocks and passes light, resulting in square wave outputs from the sensors.

The LED/sensor pairs are mounted offset relative to one another, so that the output square waves are offset 90 degrees in phase. This is known as quadrature, because 90 degrees amount to one-quarter of a full 360-degree cycle.



This phase relationship provides the information needed to determine the encoder's direction of rotation (see figure 1).

The dotted lines in figure 1 indicate a common method of reading

direction. For instance, if phase 1 is high and phase 2 is rising, the direction is clockwise (CW). If phase 1 is low and phase 2 is rising, the direction is counterclockwise (CCW).

For the sake of interpreting this output with a PIC or other microcontroller, it's probably more useful to look at the changing states of the phases as a series of two-bit numbers, as shown in figure 2 above.

When the encoder shaft is turning CW, you get a different sequence of numbers (01,00,10,11) than when it is turning CCW (01,11,10,00). You may recognize this sequence as Gray code. It is distinguished by the fact that only one bit changes in any transition. Gray code produces no incorrect intermediate values when the count rolls over. In normal binary counting, 11 rolls over to 00. If one bit changed slightly before the other, the intermediate number value could be incorrectly read as 01 or 10 before settling into the correct state of 00.

Interpreting this code amounts to comparing the incoming sequence to the known sequences for CW and CCW rotation. A lookup table would do the trick. However, this approach, while easy to understand, is inefficient. The shortcut method uses an interesting property of the two-bit Gray code sequence.

Pick any pair of two-bit numbers from the CW sequence shown in figure 2; for instance, the first two: 10, 11. Compute the exclusive-OR (XOR) of the righthand bit of the first number with the lefthand bit of the second. In this case, that would be 0 XOR 1 = 1. Try this for any CW pair of numbers from the table, and you'll always get 1.
Now reverse the order of the number pair: 11, 10. XOR the right bit of the first with the left of the second (1 XOR 1 = 0). Any CCW pair of numbers will produce a 0.

How it works. The schematic in figure 3 shows a typical rotary encoder connected to the lower two bits of port RA, and a seven-segment LED display to port RB. The circuit, and the program in the listing, perform a simple task. The count displayed on the LED goes up when the control is turned CW, and down when it is turned CCW. The display is in hexadecimal, using seven-segment approximations of the letters: A, b, C, d, E F.

Note 4: Reading Rotary Encoders

The program begins by setting up the I/O ports and clearing the variable *counter*. It gets an initial input from the encoder, which goes into the variable *old*, and strips off all but the two least-significant bits (LSBs).

The body of the program, starting with *:loop*, calls *check_encoder* and then displays the latest value of *counter* on the LED display.

Most of the interesting business happens in *check_encoder* itself. Here, the program gets the latest value at the encoder inputs, strips all but the two LSBs, and XORs the result into a copy of the old value. If the result is zero, the encoder hasn't moved since its last reading, and the routine returns without changing the value of *counter*.

If the value has changed, the routine moves the value in *old* one bit to the left in order to align its LSB with the high bit of the two-bit value in *new*. It XORs the variables together. It then examines the bit *old.1*. If the bit is 1, *counter* is incremented; if it's 0, *counter* is decremented.

Modifications. To avoid 'slippage' errors (where a change in encoder position does not change the counter, or results in the wrong change), *check_encoder* must be called at least once every 1/(encoder resolution * max revs per second). For instance, if the encoder might turn 300 rpm (5 revs per second) and its resolution is 32 transitions per turn, *check_encoder* must be called every 1/(32*5) seconds, or 6.25 milliseconds. For a user interface, bear in mind that generally the larger the knob, the slower the input. Substitution of a larger control knob may be all that's required to reduce the sampling rate.

In circumstances where electrical noise might be a problem, Microchip's PIC data sheet indicates that it might be wise to move the port I/O assignments to the beginning of *check_encoder*. Electrostatic discharge (ESD) from the user's fingertips, or some other electrical noise, could corrupt an I/O control register. This would prevent the routine from reading the encoder.

Program listing. This program may be downloaded from the Parallax BBS as RD_ENCDR.SRC. You can reach the BBS at (916) 624-7101.

Note 4: Reading Rotary Encoders

```
; PROGRAM: Read Rotary Encoder (RD_ENCDR.SRC)
; This program accepts input from a rotary encoder through bits RA.0 and RA.1,
; determines the direction of rotation, and increments or decrements a counter
; appropriately. It displays the hexadecimal contents of the four-bit counter on a
; seven-segment LED display connected to port RB.

encoder         =       ra
display         =       rb

; Variable storage above special-purpose registers.
                org     8

temp            ds      1
counter         ds      1
old             ds      1
new             ds      1

; Remember to change device info when programming a different PIC.
                device  pic16c54,rc_osc,wdt_off,protect_off
                reset   start

; Set starting point in program ROM to zero.
                org     0

start           mov     !rb, #0                 ; Set rb to output.
                mov     !ra, #255               ; Set ra to input.
                clr     counter
                mov     old, encoder
                and     old, #00000011b
:loop           call    chk_encoder
                mov     w, counter
                call    sevenseg
                mov     display, w
                goto    :loop

chk_encoder     mov     new, encoder            ; Get latest state of input bits.
                and     new, #00000011b         ; Strip off all but the encoder bits.
                mov     temp, new
                xor     temp, old               ; Is new = old?
                jz      :return                 ; If so, return without changing
                                                ; counter.
                clc                             ; Clear carry in preparation for
                                                ; rotate-left instruction.
                rl      old                     ; Move old to the left to align old.0
                                                ; with new.1.
                xor     old, new
                jb      old.1, :up              ; If the XOR resut is 1, increment
                                                ; counter, otherwise decrement.
:down           dec     counter
                skip
```

Note 4: Reading Rotary Encoders

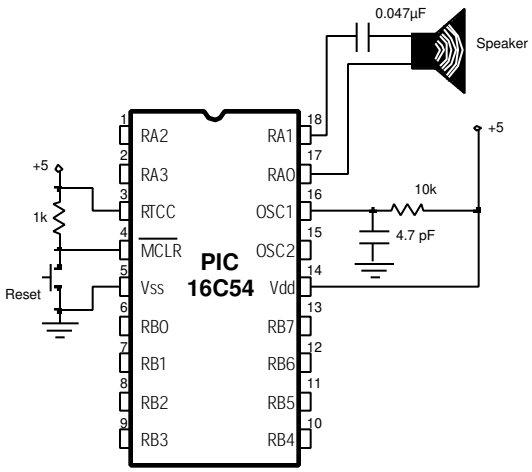
```
:up         inc     counter
            and     counter, #00001111b
            mov     old,new
:return     ret

sevenseg    jmp     pc+w                ; display lookup table
            retw    126, 48, 109, 121, 51, 91, 95, 112
            retw    127, 115, 119, 31, 78, 61, 79, 71
```

BLANK PAGE

BLANK PAGE

Note 5: Producing Sound & Music

Introduction. This application note presents a subroutine for producing tones (or musical notes) by specifying values for frequency and duration. The example program plays a brief tune through a speaker. The circuit uses two PIC outputs driven differentially to produce a 10-volt peak-to-peak signal from a single-ended power supply.



Background. Most high-level programming languages include a command for producing tones or musical notes that accepts two inputs: frequency and duration. These are the basis for programs that generate music, audible prompts, sound effects, and even Morse code. In PIC assembly language, it's easy to produce a tone by toggling an output bit. It's also easy to control the duration of the tone by counting cycles.

The problem with counting cycles is that the duration ends up varying with the frequency. A 1000-hertz (Hz) tone is made up of 1-millisecond (ms) cycles. If your program counts out 200 cycles, the resulting tone lasts 200 ms. But if you change the frequency to 5000 Hz, each cycle takes only 0.2 milliseconds, so 200 cycles last only 40 ms.

The program presented here takes a different approach. The subroutine *beep* consists of a single loop that always takes 20 microseconds (µs) to execute. A 16-bit value controls the number of loops and therefore the duration of the sound.

Note 5: Producing Sound & Music

An eight-bit counter controls the frequency of the sound. If the frequency value is 100, then the routine inverts the speaker lines every 100th execution of the main loop. The larger the frequency value, the lower the output frequency.

How it works. The circuit in figure 1 plays a short tune each time the PIC is powered up or reset (if you are using the Parallax downloader, you can omit the switch and use the built-in reset button). The speaker connects through a capacitor to two of the PIC's input/output pins. Most PIC circuits switch a load between a single pin and ground or +5 volts. As a result, the largest voltage swing they can generate is 5 volts. In this case, the program ensures that RA.0 and RA.1 are always complemented; when RA.0 is 1, RA.1 is 0 and vice versa. The speaker therefore sees a 10-volt swing. This makes a noticeably louder sound, especially with piezo devices, which are more efficient at higher voltages.
The program that controls the PIC begins by setting port RA to output and loading a pattern of alternating 0's and 1's into it. The program then looks up two values from ROM tables* *notes* and *lengths*. It stores these values in the variables *freq* and *duratn*, and calls the subroutine *beep*.

Beep accepts the values and produces a tone of the specified frequency and duration. The key to this routine is the ability of the exclusive-OR (XOR) logic function to selectively invert bits. XOR's truth table is:

A	XOR	B	Result
0		0	0
0		1	1
1		0	1
1		1	0

When B contains a 0, Result = A. When B contains a 1, Result is the inverse of A.

*You can stuff a series of bytes into the program ROM for later use by your program. Just set up a subroutine beginning with *jmp pc+w* followed by the directive *retw byte0, byte1, byte2...* When you need to access one of these numbers from your program, just move the number corresponding to the byte value you want into the *w* register, and call the table. Upon return, the byte value will be in the *w* register.

Note 5: Producing Sound & Music

Each time *beep* loops, it XOR's the speaker lines with the variable *tgl*. Most of the time, this variable contains 0's, so the XOR has no effect. The counter variable *f_temp* is decremented each time the loop executes. When *f_temp* reaches 0, *tgl* gets loaded with 1's. Now XOR'ing the speaker lines with *tgl* inverts them. The routine reloads *f_temp* with the value *freq*, and the process starts again.

After *beep* returns, the main program increments the variable *index* and checks to see whether all the notes have played. If they haven't, the program loops. If they have, it enters an endless do-nothing loop waiting to be reset or shut off.

A couple of notes about using *beep* in other applications: To calculate the value *freq* for a given frequency, divide the frequency in hertz into 50,000 (4-MHz clock) or 100,000 (8-MHz clock). If *freq* is 0, *beep* will produce a silent delay of the length set by *duratn*. To calculate *duratn*, divide the desired length of the tone by 5 ms.

Modifications. The frequency and duration values that make up the tune CHARGE were programmed by ear, but you can readily program real music by using the table below. It lists the three octaves that *beep* adequately covers. Note that the routine is not always right on the nominal frequency of the notes, but it's generally within a few hertz. The worst-case error is about 1 percent. The frequencies listed assume a 4-MHz clock. If you use an 8-MHz clock, the values for *freq* listed in the table will still accurately represent musical notes, but they'll be shifted upward by one octave.

If you use a resistor-capacitor oscillator, your notes could wind up being sharp or flat (or worse) because of variations in clock frequency. For applications in which it's important that the notes sound just right, use a ceramic resonator or crystal.

To program a tune, look up each note in the table and put the corresponding *freq* into *notes*. To calculate values for *lengths*, pick an appropriate duration for a whole note, and scale half-note, quarter-notes, eighths and so on correspondingly. The duration in milliseconds of the note will be approximately 5 x *duratn*. So, if you pick a value of 128 for a whole note, it will last 640 ms.

Note 5: Producing Sound & Music

When you're done filling in *notes* and *lengths*, count the *notes* and set the constant *end_note* equal to this number. Also make sure that the number of *notes* is equal to the number of *lengths*.

Program listing. This program may be downloaded from the Parallax BBS as CHARGE.SRC. You can reach the BBS at (916) 624-7101.

Values of Variable freq Required to Create Musical Notes (4-MHz clock)

	First Octave			Second Octave			Third Octave		
Note	Nominal (Hz)	Actual (Hz)	Value of **freq**	Nominal (Hz)	Actual (Hz)	Value of **freq**	Nominal (Hz)	Actual (Hz)	Value of **freq**
A	220	220	227	440	439	114	880	877	57
B	247	248	202	494	495	101	988	980	51
C	262	262	191	523	521	96	1047	1042	48
D	294	294	170	587	588	85	1175	1163	43
E	330	329	152	659	658	76	1319	1316	38
F	349	350	143	698	694	72	1397	1389	36
G	392	391	128	784	781	64	1568	1563	32

*To get values of **freq** for sharps (#), multiply the value listed by 0.9442 and round off the result.
For example, the **freq** value for C# in the first octave would be 0.9442 x 191 = 180.

```
; PROGRAM: CHARGE.SRC
; This program in Parallax assembly language plays a short tune, "Charge," from
; data stored in a pair of tables. It uses a general-purpose routine called beep to
; generate the notes from frequency and duration data. The tune will play when the
; PIC or Downloader is first powered up, and any time the device is reset. Note that
; although beep uses seven file registers, five of these may be  reused elsewhere in
; the program  for counting, or any use that does not require their values to be
; preserved after beep is called.

spkr        =       ra              ; Connect speaker to adjacent ra
pins.
end_note    =       6               ; number of notes in the tune

; Variable storage above special-purpose registers.
            org     8
freq        ds      1               ; passess frequency value to beep
duratn      ds      1               ; passes duration value to beep
f_temp      ds      1               ; temporary counter (frequency)
d_hi        ds      1               ; temporary counter (high byte of
                                    ; duration)
d_lo        ds      1               ; temporary counter (low byte of
                                    ; duration)
tgl         ds      1               ; temporary variable
t_pat       ds      1               ; temporary variable
index       ds      1               ; note counter used by main
                                    ; program
```

Note 5: Producing Sound & Music

```
; Device data and reset vector. Remember to change these entries if programming
; another type of PIC.
                device      pic16c54,rc_osc,wdt_off,protect_off
                reset       start

; Set ROM origin to 0 for start of program.
                org         0
start           mov         !ra, #0             ; Set port ra to output.
                mov         ra, #1010b          ; Use alternating bits as push-pull
                                                ; drivers.
                mov         index, #0           ; Start index at 0.
:loop           mov         w,index
                call        notes               ; Look up the frequency of next
                                                ; note.
                mov         freq, w             ; Put the frequency value into freq.
                mov         w,index
                call        lengths             ; Look up the length of the next
                                                ; note.
                mov         duratn, w           ; Put the length value into duratn.
                call        beep                ; Play the note.
                inc         index
                cjb         index,#end_note,:loop; All notes done? If not, loop.

:endless        goto        :endless            ; Sit in a loop until reset.


Notes           jmp         pc+w                ; Lookup table for note frequencies
                retw        90, 67, 52, 44, 52, 44 ; 4-MHz or RC operation.

; For 8-MHz operation (e.g., the downloader w/ built-in xtal) substitute the sequence
; below for Notes:
;               retw        180, 134, 104, 88, 104, 88

Lengths         jmp         pc+w                ; Lookup table for note durations.
                retw        35, 35, 35, 55, 30,127; 4-MHz or RC operation.

; For 8-MHz operation (e.g., the downloader w/ built-in xtal)  substitute the sequence
; below for Lengths:
;               retw        70, 70, 70, 110, 60, 255

beep            mov         t_pat, #255         ; All ones to invert all bits in port A.
                mov         f_temp, freq        ; If freq is zero, fill pattern with
                                                ; zeros.
                jnz         :cont               ; ..for a silent pause.
                mov         t_pat, #0
:cont           mov         d_hi, duratn        ; Variable duratn goes into the
                                                ; high byte
                clr         d_lo                ; ..of a 16-bit counter.
                mov         tgl, t_pat
:main           xor         spkr, tgl           ; XORing the speaker bits with 1s
                                                ; inverts them.
```

```
:delay      nop                     ; Zeros have no effect.
            nop                     ; Nops pad main loop to 20µs.
            nop
            djnz    f_temp, :noRoll1 ; When f_temp reaches zero,
                                    ; reload the freq.
            mov     f_temp, freq    ; ..value and put tgl_pat into tgl in
                                    ; order to
            mov     tgl, t_pat      ; ..invert the speaker bits on the
                                    ; next loop.
:dur_lo     sub     d_lo, #1        ; Decrement low byte of duration.
            jc      :noRoll2        ; If no borrow, go to noRoll2.
            sub     d_hi, #1        ; If borrow occurs, decrement the
            sc                      ; ..high byte of duration. If the high
                                    ; byte
            ret                     ; ..needs a borrow, routine is done.
            jmp     :main           ; Else loop to :main.

:noRoll1    clr     tgl             ; If f_temp is not zero, don't
            jmp     :dur_lo         ; ..pluck the speaker.

:noRoll2    nop                     ; Waste time to ensure that all
                                    ; paths through
            nop                     ; ..the routine take 20µs.
            jmp     :main
```

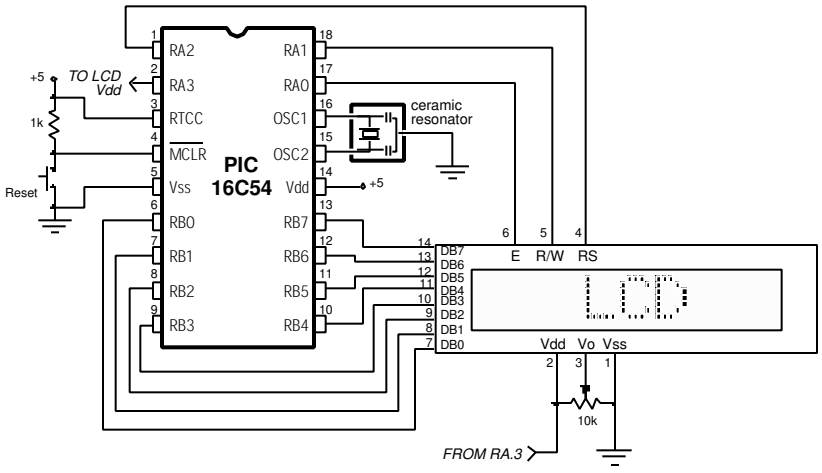
BLANK PAGE

BLANK PAGE

Note 6: Driving an LCD Display

Introduction. This application note shows how to interface PIC microcontrollers to common Hitachi liquid-crystal display (LCD) modules. The program, in Parallax assembly language, writes text to the display, reads display status, and creates custom character patterns.

Background. LCD modules based on the Hitachi 44780 controller are plentiful and inexpensive, and range in size from 8 to 80 characters. While a complete description of these LCDs' features and operation is beyond the scope of this application note, here are the highlights:



Hitachi LCD modules display the standard ASCII character set, plus selected Japanese, Greek, and math symbols. They operate from a single-ended 5-volt supply, and communicate with a bus or controller through 11 input/output (I/O) lines. The data lines are tri-state; they go into a high-impedance state when the LCD is not enabled.

The three control lines 'control' the LCD. The enable (E) line determines whether the LCD listens to the other control and data lines. When disabled, the LCD ignores all data and control signals. When enabled, the LCD checks the state of the other two control lines and responds accordingly.

Note 6: Driving an LCD Display

The read/write (R/W) line determines whether the LCD reads bits from the data lines, or writes bits to them.

Register-select (RS) determines whether the LCD treats data as instructions or characters. Here is the truth table for the control lines:

E	0	LCD disabled.
	1	LCD enabled.
R/W	0	Write to LCD.
	1	Read from LCD.
RS	0	Instructions.
	1	Characters/bytes.

Writing to the LCD requires the basic steps listed below. (Reading from the LCD follows the same sequence, but the R/W bit must be set.)

- Clear the R/W bit.
- Set or clear the RS bit as appropriate.
- Set the E bit (E=1).
- Clear the E bit (E=0).

When power is applied to the LCD, it resets itself and waits for instructions. Typically these instructions turn on the display, turn on the cursor, and set the display to print from left to right.

Once the LCD is initialized, it can receive data or instructions. If it receives a character, it prints it on the screen and moves the cursor one character to the right. The cursor marks the next location at which a character will be printed. The LCD's internal processing is similar. A memory pointer determines where the next byte will be stored. When a new byte arrives, the pointer advances. To write to sequential locations, establish the starting address and then write one byte after another.

Characters are stored in data display (DD) RAM. Regardless of the number of characters visible on the display, the LCD has 80 bytes of DD RAM. Characters in off-screen RAM can be made visible by scrolling the display.

The LCD also has 64 bytes of character-generator (CG) RAM. Data in

CG RAM determines the bit maps of the characters corresponding to codes 0 through 7 (in normal ASCII, these are control codes). To download bit maps to the LCD, first set the CG RAM address to the desired starting point (usually 0), and then write the bytes to the LCD. Because the pointer increments with each write, you don't need to keep specifying addresses. Figure 2 is an example pattern definition. The program listing shows how to define custom characters.

Address in Character Generator RAM	Bit Map	Data	"retw" Data
0000		01111	15
0001		10001	17
0010		10001	17
0011		01111	15
0100		00001	1
0101		00001	1
0110		00001	1
0111		00000	0

Figure 2. Programming a custom character pattern.

Before you can write to DD RAM after defining custom characters, the program must set a DD RAM address. The LCD reads and writes whichever RAM bank (DD or CG) was last specified in a set-address instruction. Once you have set an address in DD RAM, the next data write will display a character at the corresponding location on the screen.

Until now, we have talked about reading and writing to the LCD as though it were regular memory. It's not. The LCD's controller takes 40 to 120 microseconds (µs) to complete a read or write. Other operations can take as long as 5 milliseconds. To avoid making the PIC wait a worst-case delay between operations, the LCD has a 1µs instruction that reads the address counter and a busy flag. When the busy flag is set (1), the LCD cannot handle a read or write. The program listing includes a subroutine (blip_E) that makes sure the busy flag is cleared (0) before talking to the LCD.

The address returned along with the busy flag is either the DD or CG RAM pointer, depending on which address was last set.

Note 6: Driving an LCD Display

Figure 3 is a list of LCD instructions for reading and writing memory. Some other useful instructions appear as constants in the beginning of the program listing.

How it works. The circuit in figure 1 interfaces a PIC to an LCD module. When the power is turned on, or the circuit is reset, the PIC initializes the LCD, downloads four custom characters, and prints "Parallax" forward and backward (with custom mirrored characters).

The potentiomenter connected to the LCD's Vo pin controls contrast. If the display is hard to read, appears blank, or is filled with black pixels, adjust this control.

Power to the LCD is controlled by a PIC I/O bit. In this way, the PIC ensures that the 5-volt supply is up and stable before switching on the LCD. If the circuit used the PIC's sleep mode, it could shut down the LCD to save approximately 1.5 mA. If you use this feature, make sure that the data and control lines are cleared to 0's or set to input before putting the PIC to sleep. Otherwise, leakage through the LCD's protection diodes might continue to power it.

Set DD RAM address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

Set CG RAM address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	A	A	A	A	A	A

Read busy flag and address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	Bsy	A	A	A	A	A	A	A

Write data to RAM (CG or DD, most recently set)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D	D	D	D	D	D	D	D

Read data from RAM (CG or DD, most recently set)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D	D	D	D	D	D	D	D

Figure 3. Commonly used memory operations.
(A = address; D = data; Bsy = busy)

Note 6: Driving an LCD Display

Program listing. This program may be downloaded from the Parallax BBS as LCD_DRVR.SRC. You can reach the BBS at (916) 624-7101.

```
; PROGRAM: Drive Liquid Crystal Display (LCD_DRVR.SRC)
; This program initializes a Hitachi LCD module, defines a set of four custom
; characters, and displays the message "Parallax" forward and backward (in custom,
; mirror-reading letters). It includes a subroutine, blip_E, that handles all the required
; handshaking to send data or instructions to the LCD.

LCD_pwr     =       ra.3            ; +5 to LCD module
RS          =       ra.2            ; 0 = write, 1 = read
RW          =       ra.1            ; 0 = instruction, 1 = data
E           =       ra.0            ; 0 = disable, 1 = enable
data        =       rb              ; Data to LCD
count       =       16              ; Number of characters in demo
                                    ; string.
char_cnt    =       32              ; Number of bytes in custom
                                    ; character definition

; Declare constants for common LCD instructions. To perform the functions listed
; below, clear bit RS and send #constant to the display. Remember to set RS again
; before sending characters to the LCD.
clear       =       1               ; Clears the display (fills it with
                                    ; blanks).
home        =       2               ; Returns display to the home
                                    ; position.
shift_l     =       24              ; Shifts display to the left.
shift_r     =       28              ; Shifts display to the right.
crsr_l      =       16              ; Moves cursor to the left.
crsr_r      =       20              ; Moves cursor to the right.
blink_c     =       11              ; Blinks whole character to indicate
                                    ; cursor position.
no_crsr     =       8               ; Turns off the cursor.

; Set RAM origin above special registers, declare variables, and set code origin.
            org     8
temp        ds      1               ; Temporary counter.
temp2       ds      1               ; Pass data or instructions to
                                    ; blip_E.
counter     ds      1               ; Index variable.
            org     0
; Device data
            device  pic16c54,xt_osc,wdt_off,protect_off
            reset   start

start       mov     ra, #0          ; Initialize ports, power LCD and
                                    ; wait for it to reset.
            mov     rb, #0
```

```
                mov     !ra,#0h             ; set control lines to output
                mov     !rb,#0h             ; set data lines to output
                setb    LCD_pwr
                call    wait

                mov     temp2,#00110000b    ; Initialize LCD: set 8-bit, 1-line
                                            ; operation.
                call    blip_E
                mov     temp2,#00001110b
                call    blip_E
                mov     temp2,#00000110b
                call    blip_E

                mov     temp2, #01000000b   ; Write to CG RAM: start at
                                            ; address 0.
                call    blip_E
                setb    RS                  ; Set RS to send data.
                mov     counter, #0

:stuff          mov     w, counter
                call    my_chars            ; Get next byte.
                mov     temp2, w            ; Write byte to CG RAM.
                call    blip_E
                inc     counter
                cjb     counter, #char_cnt,:stuff
                clrb    RS                  ; Clear RS to send instruction.
                mov     temp2, #10000000b   ; Address 0 of DD RAM.
                call    blip_E
                setb    RS

send_msg        mov     counter, #0         ; Send the message string to the
                                            ; LCD.
:loop           mov     w,counter
                call    msg
                mov     temp2,w
                call    blip_E
                inc     counter
                cjb     counter,#count,:loop
:loop2          jmp     :loop2              ; Endless loop. Reset PIC to run
                                            ; program again.

; Write data or instructions (in variable temp2) to the LCD.
blip_E          movb    pa2, RS             ; Store current state of RS in
                                            ; unused bit.
                clrb    RS                  ; Clear RS to send instruction.
                setb    RW                  ; Set RW to read.
                mov     !rb, #255           ; Port RB: all inputs.
:loop           setb    E                   ; Enable the LCD.
                nop
                mov     temp, data          ; Put LCD data (RB) into temp.
```

```
                clrb    E               ; Disable LCD.
                snb     temp.7          ; Is the LCD busy?
                jmp     :loop           ; Yes, try again later.
                movb    RS, pa2         ; No, send the data or instruction.
                clrb    RW
                mov     !rb, #0
                mov     data, temp2
                setb    E
                nop
                clrb    E
                ret

wait            mov     temp2,#200      ; Create a delay for LCD power-on
                                        ; reset.
:loop           djnz    temp,:loop
                djnz    temp2,:loop
                ret

msg             jmp     pc+w            ; Table of ASCII and custom
                                        ; characters to display.
                retw    'P','a','r','a','l','l','a','x','x',2,0,0,2,1,2,3

my_chars        jmp     pc+w            ; Table of data for custom
                                        ; 'backwards' characters.
                retw    6,4,4,4,4,4,14,0        ; backwards l
                retw    0,0,13,19,1,1,1,0       ; backwards r
                retw    0,0,14,16,30,17,30,0    ; backwards a
                retw    15,17,17,15,1,1,1,0     ; backwards P
```

BLANK PAGE

Note 7: Direct & Indirect Addressing

Introduction. This application note describes direct and indirect addressing and shows a method for avoiding the gaps in the PIC16C57's banked memory.

Direct Addressing. PIC microcontrollers have 32 or 80 bytes of RAM, which Microchip refers to as file registers. The first seven registers (eight in the case of the 28-pin PIC's) are mapped to special functions, such as the real-time clock/counter (RTCC), status bits, and input/output (I/O) ports. Figure 1 shows a simplified memory map.

The simplest way to manipulate the contents of a PIC's RAM is to specify a register address in an instruction, like so:

```
mov     010h, #100
```

This instruction, which moves the decimal number 100 into register 10 hexadecimal (h), is an example of *direct addressing*. Most of the instructions in a typical PIC program use this addressing mode.

The Parallax assembler has several helpful features for direct addressing. The first of these is labeling. Instead of referring to registers by

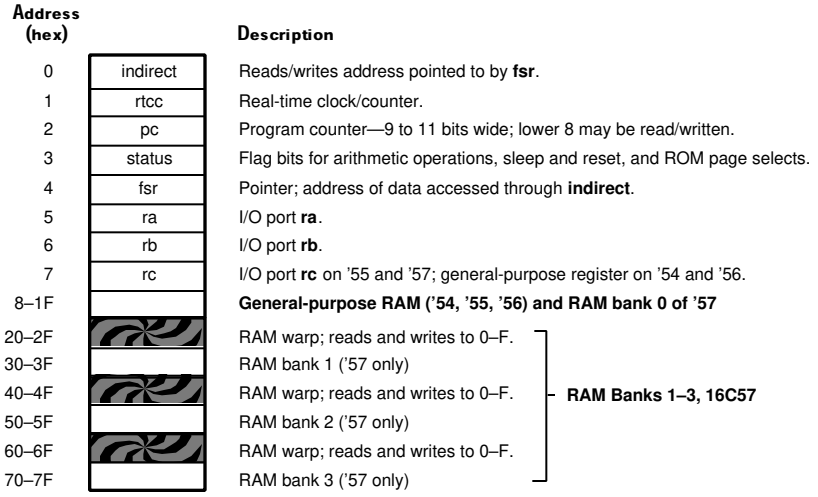


Figure 1. Simplified memory map of PIC's 16C54 through '57.

address, you may assign names to them:

```
counter     =       010h            ;Program header
            .
            .
            .
            mov     counter, #100
```

Labeled memory locations are often called variables. They make a program more understandable and easier to modify. Suppose you needed to change the location in which the counter data was stored. Without the label, you would have to rely on your text editor's search-and-replace function (which might also change other numbers containing "10"). With a label, you could change the *counter = ...* value in the program header only.

You can also define variables without specifying their address by using the *ds* (define space) directive:

```
            org     8               ;Start above special registers.
counter     ds      1               ;One byte labeled "counter."
            .
            .
            .
            mov     counter, #100
```

Using *ds* assigns the label to the next available register. This ensures that no two labels apply to the same register, making variable assignments more portable from one program to another. The only caution in using *ds* is that you must set the origin using the *org* directive twice; once for the starting point of variables in RAM, and again (usually at 0) for the starting point of your program in ROM.

Labels can be assigned to individual bits in two ways. First, if the bit belongs to a labeled byte, add *.x* to the label, where *x* is the bit number (0–7). Or assign the bit its own label:

```
LED         =       ra.3            ;Bit 3 of port ra controls LED.
```

The Parallax assembler has predefined labels for the special-purpose

registers, and the bits of the *status* register. See your manual for a list.

Indirect Addressing. The registers used in direct addressing are set forever when the program is burned into the PIC's ROM. They cannot change. However, many powerful programming techniques are based on computing storage locations. Consider a keyboard buffer. If keystrokes can't be processed immediately, they are stored in sequential bytes of memory. *Pointers*—variables containing addresses of other variables—track the locations of data entered and data processed.

The PIC's *indirect addressing* mode allows the use of pointers and the high-level data structures that go with them, such as stacks and queues. Using indirect addressing for the earlier example (writing 100 to register 10h) would look like this:

```
mov     fsr, #010h      ;Set pointer to 10h.
mov     indirect, #100  ;Store 100 to indirect.
```

The value in the file select register (*fsr;* register 04h) is used as the address in any instruction that reads/writes *indirect* (register 00h). So storing 10h in the *fsr* and then writing 100 to *indirect* is the same as writing 100 to address 10h.

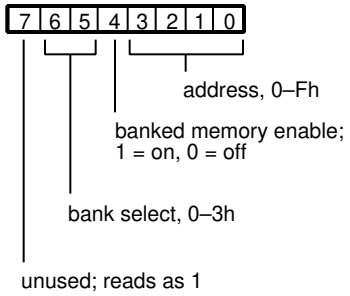


Figure 2. The 16C57 file-select register.

Memory Register banks:		
	0	10h to 1Fh
	1	30h to 3Fh
	2	50h to 5Fh
	3	70h to 7Fh

Note 7: Direct & Indirect Addressing

A more practical example would be to store a series of values from an I/O port to sequential registers in memory. All it takes is a loop like this:

```
        mov     pointer, #010h  ;Set start address.
:loop   mov     fsr, pointer    ;Put pointer into fsr.
        mov     indirect, rb    ;Move rb to indirect.
        inc     pointer         ;pointer = pointer + 1.
        cjb     pointer,#01Fh,:loop
```

This fragment assumes that a variable named *pointer* was declared previously (using =, *equ*, or *ds*), and that *rb* is set for input. The loop will rapidly fill registers 10h through 1Fh with data samples from *rb*.

PIC's with 32 bytes of RAM ('54, '55, and '56) have a five-bit-wide *fsr*. Since all registers are eight bits wide, the highest three bits of the *fsr* in these devices are fixed, and always read as 1's. Keep this in mind if you plan to perform comparisons (such as the last line of the example above) directly on the *fsr*. It will always read 224 (11100000b) higher than the actual address it points to.

The 16C57 has 80 bytes of RAM and a seven-bit-wide *fsr*. The highest bit of its *fsr* is fixed and reads as a 1. Seven bits allows for 128 addresses, but only 80 are used. The remaining 48 addresses are accounted for by three 16-byte gaps in the 57's memory map. See the RAM warps in figure 1.

Because these warps map to the lowest file registers of the PIC, they can cause real trouble by altering data in the special-purpose registers. To avoid this problem, consider using a subroutine to straighten out the memory map and avoid the warps. Below is an excerpt from a program that uses the registers from 10h on up as a storage buffer for up to 64 characters of ASCII text. For the purposes of the program, address10h is location 0 in the buffer; 7F is location 63.

When the program needs to write a value representing a position in the buffer to the *fsr*, it puts the value into the *w* register and calls *buf_ptr* (buffer pointer).

Note 7: Direct & Indirect Addressing

```
buf_ptr     mov     temp,w
            mov     fsr, temp
            cjae    temp,#030h,:bank3
            cjae    temp,#020h,:bank2
            cjae    temp,#010h,:bank1
            jmp     :bank0
:bank3      add     fsr,#010h
:bank2      add     fsr,#010h
:bank1      add     fsr,#010h
:bank0      add     fsr,#010h
            ret
```

It may be more useful in some applications to treat these memory locations as register banks, as they are described in the Microchip literature. According to this model, bit 4 of the *fsr* enables bank selection when it is a 1. The 16-byte bank in use is then selected by bits 5 and 6 of the *fsr* as shown in figure 2.

This model explains the warps in the memory map. Each of the three warp addresses (20h, 40h, and 60h) has a 0 in the bit-4 position. This disables banked memory, causing the PIC to disregard all but bits 0 through 3 of the address.

BLANK PAGE

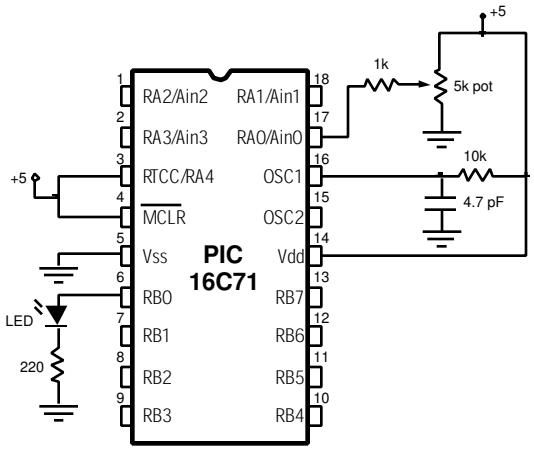
Note 8: The PIC16C71 A/D Converter

Introduction. This application note presents a program in Parallax assembly language that uses the PIC16C71's built-in analog-to-digital converter (ADC) to measure an input voltage and flash an LED at a proportional rate.

Background. One of the most popular enhancements offered by the new PIC16C71 is its eight-bit ADC, which features:

- 20-microsecond (µs) conversion time (nearly 50,000 samples per second, depending on additional processing time).
- Four multiplexed inputs.
- Built-in sample-and-hold.
- ±1 least-significant-bit accuracy (better than 20 millivolts with a 5-volt reference).
- Selectable voltage reference (Vdd or RA.3).

While using the ADC is fairly straightforward, it does require a series of decisions much like those required to select and use a separate ADC. The first consideration is hardware.



Input Characteristics. The ADC produces a digital output that is proportional to an analog input. A voltage reference determines the input voltage that will produce a full-scale (255) digital output. The voltage

reference can be the +5-volt power-supply rail, or some other voltage source between 3 volts and the power supply voltage + 0.3 volts. The ADC is most accurate with a reference voltage of 5.12 volts, according to the manufacturer's specifications.

The specifications recommend that the analog voltage source being measured have an impedance of no more than 10kΩ. Above this value, accuracy suffers. They also suggest that the source have not less than 500Ω impedance. This limits current through the PIC in the event that your program reconfigures the analog input pin as an output, or some other circuit trauma occurs.

Clock Source. The PIC's ADC, like the PIC itself, requires a clock signal. The ADC performs a conversion in 10 of its clock cycles, which must be no shorter than 2µs. Clock signals for the ADC can come from two sources, the PIC's own clock or an on-chip resistor-capacitor (RC) oscillator exclusive to the ADC.

When the PIC's clock is the source, it is divided by 2, 8 or 32, depending on the status of the ADC clock source bits (see figure 2). In order to have an ADC clock signal of 2µs or longer, the PIC clock speed must not exceed 1, 4, or 16 MHz, respectively. If you plan to run the PIC faster than 16 MHz, or you want the ADC conversion rate to be independent of the PIC clock, you must use the ADC's RC oscillator.

The tradeoff in using the RC oscillator is that its period can vary from 2 to 6µs, depending on temperature and manufacturing tolerances.

Interrupt Enable. The ADC is relatively slow—at 20 MHz the PIC can execute 100 instructions in the 20µs the ADC takes to make a conversion. In some cases, it makes sense not to force a PIC program to wait in a loop for a conversion to finish. The alternative is to configure the ADC to announce "conversion complete" through an interrupt. To keep things as simple as possible, the example program does not take advantage of interrupt capability.

Pin Configuration and Voltage Reference. Pins RA.0 through RA.3 can serve as inputs to the ADC. One of the choices you must make when setting up the ADC is which pins to configure as analog inputs, which (if any) as digital inputs, and what to use as a voltage reference. Figure 2 shows

the range of available choices.

Note that the control register containing the configuration and voltage reference bits is in register page 1. To access it, you must first set bit *RP0*. The program listing shows how.

Input Selection. Only one of the pins configured for analog input can actually sample at any one time. In other words, if you want ADC input from two or more channels, your program must select a channel, wait long enough for the sample-and-hold circuit to charge up, command a conversion, get the result, and select the next channel...

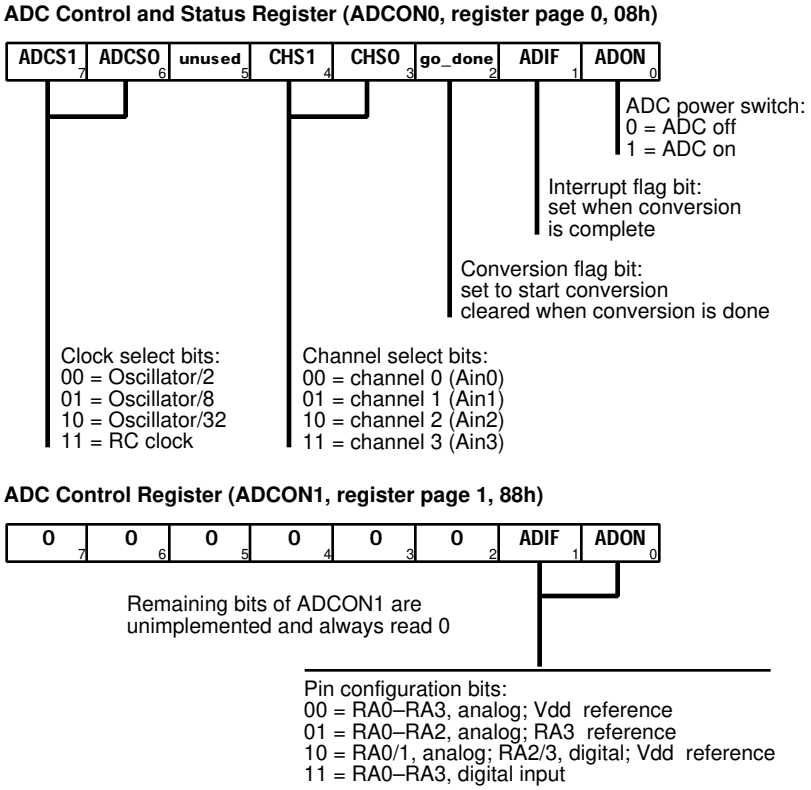


Figure 2. ADC control registers.

Note 8: The PIC16C71 A/D Converter

How long should the program wait for the sample-and-hold? Microchip suggests a worst case of 4.67µs plus two ADC clock cycles (after a conversion, the sample-and-hold takes a two-cycle break before it begins sampling again).

How it works. The PIC in figure 1 accepts a voltage from a pot wired as variable voltage divider. The PIC's ADC, which is set up to use Vdd as a reference, outputs a one-byte value that's proportional to the input voltage. This value controls a timing routine that flashes an LED. When the input voltage is near 5 volts, the LED flashes about once a second. When it's near 0, the LED flashes very rapidly.

The program listing shows how it's done. Most of the code is devoted to setting up the ADC. Constants at the beginning of the program are assigned with values that, when loaded into the appropriate ADC control registers, turn the ADC's various features on and off. If you wish to change, for instance, the pin that the circuit uses for analog input, just comment out the line containing *AD_ch = 0* and uncomment the desired channel ("commenting" and "uncommenting" are handy techniques for temporarily removing and restoring instructions in source code. Putting a semicolon (;) in front of a line causes the assembler to ignore it, as though it were a comment).

If you accidentally leave two assignements for *AD_ch* uncommented, the assembler will catch the mistake, flag the "redefinition" and tell you the line number of the error.

The assembler combines values assigned to *ADC_ch* and *ADC_clk* into a single byte by performing a logical OR (|) on the values and putting the result into another constant, *ADC_ctl*. This technique makes the program easier to understand and modify, and doesn't cost a thing in PIC program memory or processing time. The assembler does all the work.

Program listing. This program may be downloaded from the Parallax BBS as ADC71.SRC. You can reach the BBS at (916) 624-7101.

Note 8: The PIC16C71 A/D Converter

; PROGRAM: Using the 16C71's analog-to-digital converter (ADC71.SRC)

```
; This program demonstrates use of the ADC in a simple circuit that samples a
; voltage and flashes an LED at a proportional rate. The header contains a number
; of constants representing setup constants for the ADC control registers.
; Uncomment the constant corresponding to the desired ADC setting.

; The following constants set the ADC clock source and speed. Uncomment one.
;AD_clk      =       0               ; Oscillator x 2  (<= 1 MHz).
;AD_clk      =       64              ; Oscillator x 8  (<= 4 MHz).
;AD_clk      =       128             ; Oscillator x 32 (<= 16 MHz).
AD_clk      =       192             ; RC oscillator, 2–6 us.

; The following constants select a pin for ADC input. Uncomment one.
AD_ch       =       0               ; ADC channel 0 (Ain0, pin 17).
;AD_ch      =       8               ; ADC channel 1 (Ain1, pin 18).
;AD_ch      =       16              ; ADC channel 2 (Ain0, pin 1).
;AD_ch      =       24              ; ADC channel 3 (Ain0, pin 2).

AD_ctl      =       AD_clk | AD_ch  ; Logical OR.

; The following constants determine which pins will be usable by the ADC and
; whether Vdd or RA.3 will serve as the voltage reference. Uncomment one.
AD_ref      =       0               ; RA.0-3 usable, Vdd reference.
;AD_ref     =       1               ; RA.0-3 usable, RA.3 reference.
;AD_ref     =       2               ; RA.0/1 usable, Vdd reference.
;AD_ref     =       3               ; All unusable—digital inputs only.

            device  rc_osc,wdt_off,pwrt_off,protect_off
            id      'ADC1'

            org     0Ch
counter1    ds      1
counter2    ds      1

; Set starting point in program ROM to zero. Jump past interrupt vector to beginning
; of program. (This program doesn't use interrupts, so this is really not needed, but it
; will be easier to add interrupts later if required.)
            org     0
            jmp     5
            org     5

start       mov     !ra, #255       ; Set RA to input.
            mov     !rb, #0         ; Set RB to output.
            mov     intcon, #0      ; Turn interrupts off.
            mov     adcon0,#AD_ctl  ; Set AD clock and channel.
            setb    rp0             ; Enable register page 1.
            mov     adcon1,#AD_ref  ; Set usable pins, Vref.
            clrb    rp0             ; Back to register page 0.
            setb    adon            ; Apply power to ADC.
```

```
:loop       call    wait            ; Delay for time determined by
                                    ; ADC input.
            setb    rb.0            ; Turn LED on.
            call    wait            ; Delay for time determined by
                                    ; ADC input.
            clrb    rb.0            ; Turn LED off.
            goto    :loop           ; Endless loop.

wait        setb    go_done         ; Start conversion.
:not_done   snb     go_done         ; Poll for 0 (done).
            jmp     :not_done       ; If 1, poll again.
            mov     counter2,adres  ; Move ADC result into counter.

; The number of loops this delay routine makes is dependent on the result of the AD
; conversion. The higher the voltage, the longer the delay.
:loop       djnz    counter1, :loop
            djnz    counter2, :loop
            ret
```

BLANK PAGE

BLANK PAGE

Note 9: Managing Multiple Tasks

Introduction. This application note presents a program in Parallax assembly language that demonstrates a technique for organizing a program into multiple tasks.

Background. Like most computers, the PIC executes its instructions one at a time. People tend to write programs that work the same way; they perform one task at a time.

It's often useful to have the controller do more than one thing at a time, or at least seem to. The first step in this direction is often to exploit the dead time from one task—the time it would normally spend in a delay loop, for instance—to handle a second task. The PIC's speed makes this quite practical in many cases.

When several tasks must be handled at once, this approach can quickly become unworkable. What we need is a framework around which to organize the tasks. We need an operating system.

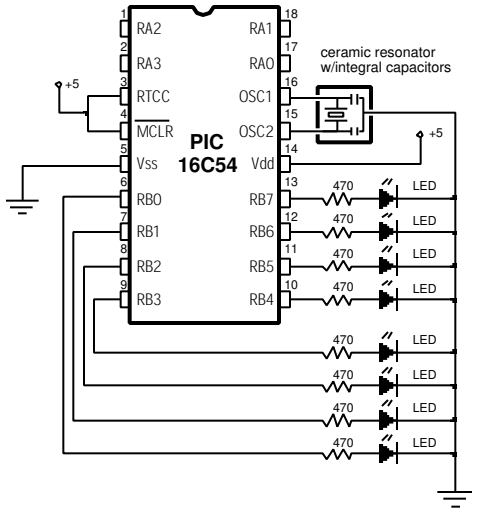
The program in the listing illustrates an extremely simple operating system that runs each of eight small subprograms in turn. When the subprograms finish their work, they jump back to the system. Notice that this method does not require the call instruction, so it leaves the two-level stack free for the use of the subprograms.

How it works. The circuit and program comprise an eight-LED flasher. Each of the LED's flashes at a different rate. While this could be accomplished differently, the program is easier to understand and maintain because the code that controls each LED is a separate task.

The "system" portion of the program acts like a spinning rotary switch. Each time it executes, it increments the task number and switches to the next task. It does this by taking advantage of the PIC's ability to modify the program counter. Once the task number is loaded into the working register, the program executes the instruction *jmp pc+w*. The destinations of these jumps contain *jmp* instructions themselves, and send the program to one of the eight tasks. Not surprisingly, a list of *jmp* instructions arranged like this is called a "jump table."

Modifications. For the sake of simplicity, this task-switching program lacks one important attribute: fixed timing. The individual tasks are

Note 9: Managing Multiple Tasks



permitted to take as much or as little time as they require. In some real applications, this wouldn't be acceptable. If the system maintains a master timer (like the variable *ticks* in this program) it should increment at a consistent rate.

With many possible paths through the code for each task, this may seem like another problem. A straightforward solution is to use the PIC's RTCC to time how long a particular task took, then use that number to set a delay to use up all of the task's remaining time. All you need to know is the worst-case timing for a given task.

Program listing. This program may be downloaded from the Parallax BBS as TASKS.SRC. The BBS phone number is (916) 624-7101.

Note 9: Managing Multiple Tasks

; PROGRAM: TASK.SRC

```
; This program demonstrates task switching using the PIC's relative addressing
; mode. It handles eight tasks, each of which flashes an LED at a different rate. The
; flashing routines base their timing on a master clock variable called ticks.

LEDs            =       rb

; Put variable storage above special-purpose registers.
                org     8

task            ds      1               ; The task number used by the
                                        ; system.
ticks           ds      1               ; Master time clock, increments
                                        ; once for each system cycle.
time0           ds      1               ; Timer for task 0.
time1           ds      1               ; Timer for task 1.
time2           ds      1               ; Timer for task 2.
time3           ds      1               ; Timer for task 3.
time4           ds      1               ; Timer for task 4.
time5           ds      1               ; Timer for task 5.
time6           ds      1               ; Timer for task 6.
time7           ds      1               ; Timer for task 7.


; Remember to change device info if using a different part.
                device  pic16c54,xt_osc,wdt_off,protect_off
                reset   start

; Set starting point in program ROM to zero.
                org     0

start           mov     !rb,#00000000b  ; Set port rb to output.
                mov     task, #7        ; Set task number.
                clr     ticks           ; Clear system clock.
                clr     LEDs            ; Clear LEDs

system          inc     task            ; Next task number.
                cjne    task, #8, :cont ; No rollover? Continue.
                clr     task            ; Rollover: reset task and
                inc     ticks           ; increment the clock.
:cont           mov     w, task         ; Prepare to jump.
                jmp     pc+w            ; Jump into table, and from there
                jmp     task0           ; to task #.
                jmp     task1
                jmp     task2
                jmp     task3
                jmp     task4
                jmp     task5
                jmp     task6
                jmp     task7
```

```
task0   cjne    ticks, #255,:cont   ; Every 255 ticks of system clock
        inc     time0               ; increment task timer. Every 3
                                    ; ticks
        cjne    time0, #3, :cont    ; of task timer, toggle LED, and
        clr     time0               ; reset task timer.
        xor     LEDs, #00000001b
:cont   jmp     system

task1   cjne    ticks, #255,:cont
        inc     time1
        cjne    time1, #8, :cont
        clr     time1
        xor     LEDs, #00000010b
:cont   jmp     system

task2   cjne    ticks, #255,:cont
        inc     time2
        cjne    time2, #6,:cont
        clr     time2
        xor     LEDs, #00000100b
:cont   jmp     system

task3   cjne    ticks, #255,:cont
        inc     time3
        cjne    time3, #11,:cont
        clr     time3
        xor     LEDs, #00001000b
:cont   jmp     system

task4   cjne    ticks, #255,:cont
        inc     time4
        cjne    time4, #12, :cont
        clr     time4
        xor     LEDs, #00010000b
:cont   jmp     system

task5   cjne    ticks, #255,:cont
        inc     time5
        cjne    time5, #4, :cont
        clr     time5
        xor     LEDs, #00100000b
:cont   jmp     system

task6   cjne    ticks, #255,:cont
        inc     time6
        cjne    time6, #23,:cont
        clr     time6
        xor     LEDs, #01000000b
:cont   jmp     system
```

```
task7       cjne    ticks, #255,:cont
            inc     time7
            cjne    time7, #9,:cont
            clr     time7
            xor     LEDs, #10000000b
:cont       jmp     system
```

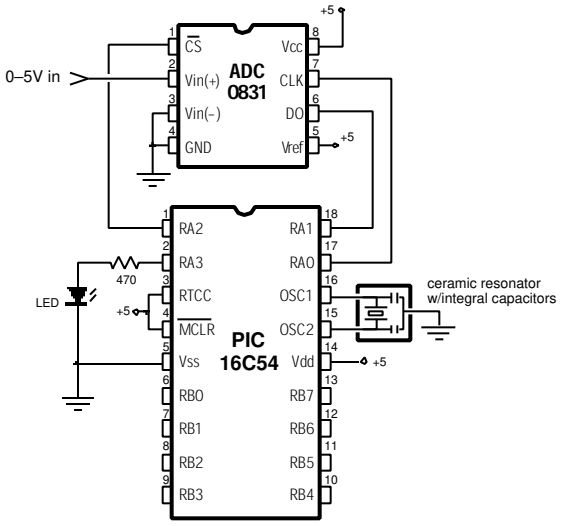
BLANK PAGE

Note 10: An External A/D Converter

Introduction. This application note shows how to interface an 8-bit serial analog-to-digital converter to 16C5x-series PICs.

Background. Although the PIC 16C71 offers a built-in analog-to-digital converter (ADC), some users may prefer an external device. One that's particularly suited to use with the PIC is the National Semiconductor ADC0831, available from Digi-Key, among others.

Interfacing the 831 requires only three input/output lines, and of these, two can be multiplexed with other functions (or additional 831s). Only the chip-select (CS) pin requires a dedicated line. The ADC's range of input voltages is controlled by the V_{REF} and $V_{IN}(-)$ pins. V_{REF} sets the voltage at which the ADC will return a full-scale output of 255, while $V_{IN}(-)$ sets the voltage that will return 0.



In the example application, $V_{IN}(-)$ is at ground and V_{REF} is at +5; however, these values can be as close together as 1 volt without harming accuracy or linearity. You may use diode voltage references or trim pots to set these values.

Note 10: An External A/D Converter

How it works. The sample program reads the voltage at the 831's input pin and uses the eight-bit result to control a timing routine that flashes an LED. The LED flashes slowly when the input is 5 volts, and very rapidly as it approaches ground.

The subroutine *convert* handles the details of getting data out of the ADC. It enables the ADC by pulling the CS line low, then pulses the clock (CLK) line to signal the beginning of a conversion. The program then enters a loop in which it pulses CLK, gets the bit on pin *data*, and shifts it into the received byte using the rotate left (*rl*) instruction. Remember that *rl* affects not only the byte named in the instruction, but the carry bit as well.

When all bits have been shifted into the byte, the program turns off the ADC by returning CS high. The subroutine returns with the conversion result in the variable *ADresult*. The whole process takes 74 instruction cycles.

Modifications. You can add more 831s to the circuit as follows: Connect each additional ADC to the same clock and data lines, but assign separate CS pins. Modify *convert* to take the appropriate CS pin low when it needs to acquire data from a particular ADC. That's it.

Program listing. This program may be downloaded from the Parallax BBS as AD831.SRC. You can reach the BBS at (916) 624-7101.

```
; PROGRAM: AD831.SRC
; Program demonstrates the use of an external serial ADC (National ADC0831) with
; PIC16C5x-series controllers. A variable dc voltage (0-5V) at pin 2 of the ADC
; controls the blinking rate of an LED connected to PIC pin ra.3.

clk             =       ra.0
data            =       ra.1
CS              =       ra.2
LED             =       ra.3

; Put variable storage above special-purpose registers.
                org     8

counter1        ds      1
counter2        ds      1
ADresult        ds      1
```

Note 10: An External A/D Converter

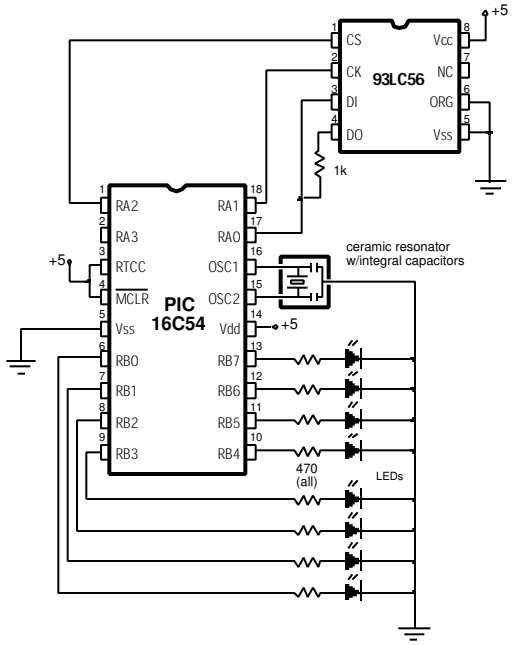
```
; Remember to change device info when programming part.
                device      pic16c54,xt_osc,wdt_off,protect_off
                reset       start

; Set starting point in program ROM to zero
                org         0
start           mov         ra, #00000100b      ; Set CS to disable ADC for now.
                mov         !ra,#00000010b      ; Make ra.1 (data) input, rest
                                                ; outputs.
                mov         !rb,#00000000b      ; Make rb output.

blink           xor         ra, #8              ; Invert bit3 of ra (LED).
                call        wait                ; Delay depends on ADC data.
                goto        blink               ; Endless loop.

wait            call        convert             ; Get AD result and put into
                                                ; counter2.
                mov         counter2,ADresult
                add         counter2,#1         ; Add 1 to avoid underflow when
                                                ; AD=0.

; Time delay produced by loop depends on value of ADresult. Higher values
; produce longer delays and slower blinking.
:loop           djnz        counter1, :loop
                djnz        counter2, :loop
                ret

convert         clrb        CS                  ; Enable the ADC.
                mov         counter2,#8         ; Set up for eight data bits.
                setb        clk                 ; Pulse the clock line (insert
                nop                             ; nop if PIC clock >4 MHz).
                clrb        clk
                clr         ADresult            ; Clear byte to make way for new
                                                ; data.
:loop           setb        clk                 ; Pulse the clock line (insert nop
                nop                             ; if PIC clock >4 MHz).
                clrb        clk
                movb        c,data              ; Move data bit into carry.
                rl          ADresult            ; Rotate carry into byte.
                djnz        counter2,:loop      ; Eight bits collected? If not, loop.
                setb        CS                  ; End the conversion.
                ret
```

BLANK PAGE

Note 11: Using Serial EEPROMs

Introduction. This application note shows how to use the Microchip 93LC56 EEPROM to provide 256 bytes of nonvolatile storage. It provides a tool kit of subroutines for reading, writing, and erasing the EEPROM. (Note that EEPROMs made by other manufacturers will not work with the PIC16Cxx.)

Background. Many designs take advantage of the PIC's ability to store tables of data in its EPROM program memory. The trouble is that the larger the tables are, the smaller the space left for code. And many applications could benefit from the ability to occasionally update data tables for calibration or other purposes. What the PIC needs is the equivalent of a tiny disk drive.



The Microchip 93C56 and 93LC56 electrically erasable PROMs (EEPROMs) are perfect for these applications. They communicate serially via a three- or four-wire bus using a simple synchronous (clocked) communication protocol at rates of up to 2 million bits per second (Mbps). It's possible to read a byte in as little as 10 microseconds

(including the time required to send the instruction opcode and address). Once a program has begun reading data from the EEPROM, it can continue reading subsequent bytes without stopping. These are clocked in at the full 2 Mbps rate; 1 byte every 4 microseconds.

Erasing and writing these serial EEPROMs happens at a more leisurely pace. While the opcode, address, and data can be clocked into the chip at high speed, the EEPROM requires about 2 milliseconds to erase or write a byte. During this time, the chip cannot process additional instructions. The PIC can poll a flag to determine when the automatic erase/write programming cycle is over. As soon as this flag goes high, the EEPROM is ready for more instructions.

Data stored in the EEPROM will be retained for 10 years or more, according to the manufacturer. The factor that determines the EEPROM's longevity in a particular application is the number of erase/write cycles. Depending on factors such as temperature and supply voltage, the EEPROM is good for 10,000 to 1 million erase/write cycles. This rules out its use as a substitute for ordinary RAM, since many PIC applications write to RAM thousands of times a second. At that rate, the EEPROM could be unusable in as little as 20 seconds! For a thorough discussion of EEPROM endurance, see the Microchip Embedded Control Handbook, publication number DS00092A, October 1992.

How it works. The circuit in the figure specifies a 93LC56 EEPROM, but a 93C56 will work as well. The difference is that the LC device has a wider Vcc range (2.5–5.5 V, versus 4–5.5 V), lower current consumption (3 mA versus 4 mA), and can be somewhat slower in completing internal erase/write operations, presumably at lower supply voltages. In general, the LC type is less expensive, and a better match for the operating characteristics of the PIC.

The schematic shows the data in and data out (DI, DO) lines of the EEPROM connected together to a single PIC I/O pin. The 1k resistor prevents the PIC and DO from fighting over the bus during a read operation. During a read, the PIC sends an opcode and an address to the EEPROM. As soon as it has received the address, the EEPROM activates DO and puts a 0 on it. If the last bit of the address is a 1, the PIC could end up sourcing current to ground through the EEPROM. The resistor limits the current to a reasonable level.

Note 11: Using Serial EEPROMs

The program listing is a collection of subroutines for reading, writing, and erasing the EEPROM. All of these rely on *Shout*, a routine that shifts bits out to the EEPROM. To perform an EEPROM operation, the software loads the number of clock cycles into *clocks* and the data to be output into *temp*. It then calls *Shout*, which does the rest.

If you don't have the EEPROM data handy (Microchip Data Book, DS00018D, 1991), you should know about a couple of subtleties. First, when the EEPROM powers up, it is write protected. You must call *EEnable* before trying to write or erase it. It's a good idea to call *EEdisbl* (disable writes) as soon as possible after you're done. Otherwise, a power glitch could alter the contents of your EEPROM. Also, you cannot write all locations (*EEWrall*) without first erasing all locations (*EEwipe*).

Modifications. The table of constants at the beginning of the listing specifies the opcodes for each of the EEPROM operations. Although the opcodes are only three bits long, they are combined with a trailing don't-care bit. This bit is required for compatibility with the 512-byte 93LC66. With the '66, this would be address bit A8. If you want to modify this code for the '66, add a line to the read, write, and byte erase routines to copy A8 into bit 4 of *temp* just before calling *Shout*. If you want to increase capacity by adding more EEPROMs, you can bus the data and clock lines together and provide separate chip selects to each device.

If you plan to run your PIC faster than 8 MHz, add one or two *nop*s where marked in the listing. The clock must be high for at least 500 nanoseconds. The low time must also be greater than 500 ns, but the move-data, rotate, and looping instructions provide enough delay.

Program listing. This program may be downloaded from the Parallax BBS as EEPROM.SRC. You can reach the BBS at (916) 624-7101.

Note 11: Using Serial EEPROMs

```
; PROGRAM: EEPROM.SRC
; This program is a collection of subroutines for reading, writing, and erasing the
; 93LC56 (or 'C56) serial EEPROM. As a demonstration, it writes a scanning pattern
; to the 256 bytes of the EEPROM, and then reads it back to eight LEDs connected
; to port rb.

D           =       ra.0            ; Pins DI and DO of the EEPROM
CLK         =       ra.1            ; Clock pin--data valid on rising
                                    ; edge
CS          =       ra.2            ; Chip select--high = active
ROP         =       192             ; Opcode for read
WROP        =       160             ; Opcode for write
EWEN        =       152             ; Opcode to enable erase and
                                    ; write
EWDS        =       128             ; Opcode to disable erase and
                                    ; write
ERAS        =       224             ; Opcode to erase a byte
ERAL        =       144             ; Opcode to erase all bytes
WRAL        =       136             ; Opcode to write all bytes with
                                    ; specified data

            org     8
temp        ds      1               ; Temporary variable for EE
                                    ; routines
EEdata      ds      1               ; Passes data to EEwrite/wrall,
                                    ; from EEread
EEaddr      ds      1               ; Passes address to EErase,
                                    ; EEwrite, EEread
clocks      ds      1               ; Number of clock cycles for
                                    ; SHiftOUT
tick1       ds      1               ; Timer for Delay--not required for
                                    ; EE routines
tick2       ds      1               ; Timer for Delay--not required for
                                    ; EE routines

; Remember to change device info when programming part
            device  pic16c54,xt_osc,wdt_off,protect_off
            reset   start
            org     0
start       mov     ra,#0           ; Clear ports
            mov     rb,#0
            mov     !ra,#0          ; Make all ra pins outputs initially
            mov     !rb,#0
            call    EEnable         ; Turn off write/erase protection
            mov     EEdata,#1
            mov     EEaddr,#0

:loop       call    EEwrite         ; Write scanning pattern to
                                    ; EEPROM
            call    Busy
```

```
                rr      EEdata
                ijnz    EEaddr,:loop
                call    EEdisbl         ; Turn write/erase protection back
                                        ; on
                mov     EEaddr,#0
:loop2          call    EEread
                mov     rb,EEdata
                inc     EEaddr
                call    delay
                goto    :loop2

; Shift out the bits of temp to the EEPROM data line.
Shout           rl      temp            ; Rotate bit7 of temp into carry
                movb    D,c             ; Move carry bit to input of
                                        ; EEPROM
                setb    CLK             ; Clock the bit into EEPROM
                nop                     ; Clock must be high > 500 ns
                clrb    CLK
                djnz    clocks,Shout
                ret

; Read the byte in EEaddr into EEdata.
EEread          mov     temp,#ROP       ; Move the read opcode into temp

                mov     clocks,#4       ; Number of bits to shift out (op+1)
                setb    CS
                call    Shout
                mov     clocks,#8
                mov     temp,EEaddr
                call    Shout
                mov     !ra,#1
                mov     clocks,#8
:read           setb    CLK
                movb    c,D
                rl      temp
                clrb    CLK
                djnz    clocks,:read
                mov     EEdata,temp
                mov     !ra,#0
                clrb    CS
                ret

; Call to unprotect after EEdisbl or power up.
EEnable         setb    CS
                mov     clocks,#12
                mov     temp,#EWEN
                call    Shout
                clrb    CS
                ret
```

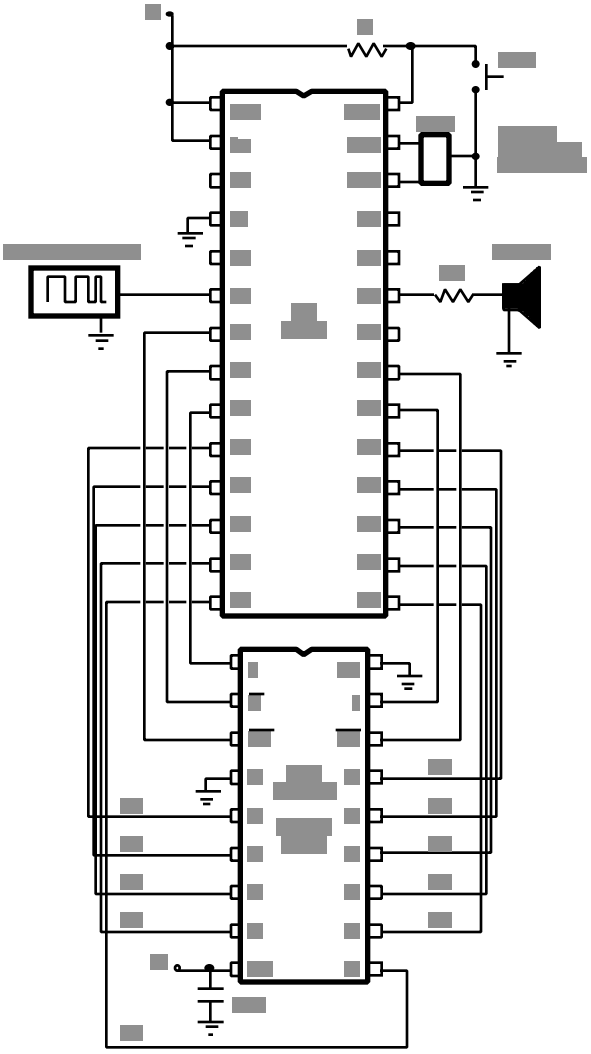
```
; Call to protect against accidental write/erasure.
EEdisbl     setb    CS
            mov     clocks,#12
            mov     temp,#EWDS
            call    Shout
            clrb    CS
            ret

; Write the byte in EEdata to EEaddr.
EEwrite mov temp,#WROP
            mov     clocks,#4
            setb    CS
            call    Shout
            mov     clocks,#8
            mov     temp,EEaddr
            call    Shout
            mov     clocks,#8
            mov     temp,EEdata
            call    Shout
            clrb    CS
            ret

; Erase the byte in EEaddr. Erasure leaves FFh (all 1s) in the byte.
EErase      mov     temp,#ERAS
            mov     clocks,#4
            setb    CS
            call    Shout
            mov     clocks,#8
            mov     temp,EEaddr
            call    Shout
            ret

; Erase the entire EEPROM--all 256 bytes. Call before using EEwrall below.
EEwipe      setb    CS
            mov     temp,#ERAL
            mov     clocks,#12
            call    Shout
            clrb    CS
            ret

; Write the byte in EEdata to every address. Call EEwipe first.
EEwrall     setb    CS
            mov     temp,#WRAL
            mov     clocks,#12
            call    Shout
            mov     clocks,#8
            mov     temp,EEdata
            call    Shout
            clrb    CS
            ret
```

```
; Check flag to determine whether the EEPROM has finished its self-timed erase/
; write. Caution: This will lock up your program until D goes high.
Busy        nop
            mov     !ra,#1
            setb    CS
:wait       jnb     D,:wait
            clrb    CS
            mov     !ra,#0
            ret

Delay       djnz    tick1,Delay         ; Delay routine for demo.
            djnz    tick2,Delay
            ret
```

BLANK PAGE

Note 12: Using Dynamic RAM

Introduction. This application note covers the use of dynamic RAM (DRAM) with PIC16C5x series PICs. It presents a circuit using a Motorola 1-Mb x 1 DRAM IC, and a program in Parallax assembly language showing how to refresh, write, and read the DRAM's 1,048,576 bits of storage.



Note 12: Using Dynamic RAM

Background. When a project requires a large amount of memory at the lowest possible cost and in the smallest available package, dynamic RAM (DRAM) is currently the only choice. Unfortunately, DRAM has a bad reputation for being difficult to work with. That reputation is built on the experiences of designers squeezing the greatest possible speed out of the large memory arrays used in general-purpose computers. In PIC applications, where speeds are more leisurely and 1 kByte is still a lot of memory, DRAMs can be downright mild-mannered.

A review of the basic differences between DRAM and static RAM (SRAM) is in order. The basic storage unit in an SRAM is a flip-flop circuit made up of several transistors. Once set or cleared, the flip-flop remains in that state until set or cleared again, or until power is removed. This is a convenient form of temporary storage, since it requires no attention from the processor except when it's being read or written.

DRAM uses even simpler storage cells, consisting of just a capacitor and a transistor. The state of the cell, set or cleared, is stored as a charge on the capacitor. However, the capacitor inevitably leaks and loses its charge over time. When this happens, the stored data is lost.

To prevent the loss of data, each cell in a DRAM must be periodically read and rewritten before its charge fades completely. This ensures that the storage capacitors always have a "fresh" charge, so the process is called *refreshing* the DRAM. The responsibility for refreshing the DRAM is shared between the external processor and circuitry inside the DRAM, as we will see later.

SRAM and DRAM also use different addressing schemes. To access a particular cell of an SRAM, you place the bits of its address on the SRAM's address pins. The number of address pins must be sufficient for each cell to have a unique binary address. A 256-cell SRAM requires 8 address lines, since 8 binary digits are required to express 256 addresses ($2^8 = 256$). By the same token, a 64k SRAM needs 16 address lines ($2^{16} = 65{,}536$) and a 1M SRAM, 20 address lines ($2^{20} = 1M$).

DRAMs use a two-dimensional addressing scheme. Each cell has two addresses, a row and a column. A 256-cell DRAM would require a 4-bit row address and a 4-bit column address ($2^4 \times 2^4 = 256$). A 64k DRAM

would need 8 rows and 8 columns (2^8 x 2^8 = 65,536) and a 1M DRAM, 10 rows and 10 columns (2^{10} x 2^{10} = 1M).

Therefore, a 1-Mb DRAM has a 10-bit wide address bus. To access a memory cell, you put the row address on the bus and activate the DRAM's row-address strobe (RAS), then put the column address onto the bus and activate the column-address strobe (CAS). At the expense of a two-step addressing processing, the number of address lines is cut in half.

The RAS and CAS lines play an important role in the refresh process. DRAM manufacturers know that refresh is a pain in the neck, so they add features to make it easier. For example, just cycling through the DRAM's row addresses will satisfy refresh requirements. The DRAM's internal circuitry takes care of the rest. This is known as RAS-only refresh, because all the user has to do is put a sequential row address on the bus, and blip the RAS line. The MCM 511000A DRAM shown in the figure supports RAS-only refresh. Although it has 10 bits worth of row addresses, the DRAM requires that only the lower 9 bits (512 addresses) be accessed in RAS-only refresh.

An even simpler refresh scheme, the one illustrated in the program listing, is known as CAS-before-RAS refresh. All the processor has to do in this case is activate CAS, activate RAS, and then deactivate both strobes. Each time it does this, it refreshes a row. In a normal access, RAS is activated before CAS, so reversing the order serves as a signal to the DRAM. When the DRAM sees this signal, it uses its own internal row-refresh counter to supply row addresses for the refresh process, ignoring the external address bus.

The MCM 511000A DRAM requires refresh every 8 milliseconds, but it doesn't care whether it is performed all at once (burst refresh), or spread out over time (distributed refresh).

Some applications don't require a separate refresh routine at all. Any application that accesses 512 row addresses every 8 milliseconds doesn't need any further refresh. If the application doesn't access all rows in the required time, but does read or write the DRAM 512 or more times every 8 milliseconds, a CAS-before-RAS refresh cycle can be tacked to the end of each read or write. This is known as hidden refresh.

Note 12: Using Dynamic RAM

Not all DRAMs support all refresh schemes, so it's important to review the manufacturer's specifications before finalizing a design. Data on the MCM 511000A for this application note came from the *Motorola Memory Data book*, DL113/D, Rev 6, 1990.

How it works. The sample application shown in the figure and program listing accepts a stream of input bits, such as the output from a square-wave generator, and records them into sequential addresses in the DRAM. At the same time, it echoes these bits to a speaker. When the DRAM is full, the program plays back the recorded bits through the speaker until the circuit is shut off or reset.

To demonstrate the circuit, reset the PIC and twiddle the tone-generator frequency knob. It will take about 52 seconds for the PIC to fill the DRAM's 1,048,576 bits of storage. When it does, you will hear the sequence of tones played back through the speaker. Because the play-back routine requires fewer instructions than recording, the pitch of the tones will be slightly higher and the playback time shorter.

The write loop takes up to 120 program cycles to execute; 60 microseconds when the PIC is operating from an 8-MHz clock. The exact number of cycles for a trip through the loop depends on how long the program spends in the subroutine *inc_xy*, which increments 10-bit row and column counters. At 60 microseconds per loop, the PIC samples the input pin approximately 17,000 times per second. If you use the circuit to record frequencies higher than half the sampling rate (approximately 8.5 kHz), you'll hear some interesting squawks. The output tone will become lower as you adjust the input tone higher.

What you're hearing is known as *aliasing noise* or *foldover*. The only way to faithfully reproduce an input frequency is to record two or more samples per cycle. You will hear this referred to as the Nyquist theorem or Nyquist limit. If you are interested in modifying this application to build a recording logic probe or tapeless audio recorder, keep Nyquist in mind.

Modifications. This application will work fine (albeit at faster rates of sampling and playback) if you remove *call refresh* from the *:record* and *:playback* loops. The reason is that with *call refresh* removed the program accesses row addresses sequentially every 30 microseconds or so. This

means that 512 rows are refreshed every 15 milliseconds. Although this exceeds the maximum refresh time allowed by the DRAM specification, DRAMs are rated conservatively. Further, you are unlikely to hear the data errors that may be occurring because of inadequate refresh!

If you want to organize data stored in the DRAM as bytes (or nibbles, 16-bit words, etc.) instead of bits, consult the DRAM specs on "fast page mode" reads and writes. Using this method of accessing the DRAM, you issue the row address just once, then read or write multiple columns of the same row. This would be faster and simpler than eight calls to the single-bit read and write routines presented here.

Program listing. This program may be downloaded from the Parallax BBS as DRAM.SRC. You can reach the BBS at (916) 624-7101.

```
; PROGRAM: Dynamic RAM Basics (DRAM.SRC)
; Upon power up or reset, this program starts sampling pin ra.0 and recording its
; state in sequential addresses of a 1Mb dynamic RAM. When the DRAM is full, the
; program plays back the recorded bits in a continuous loop.

Sin         =       ra.0            ; Signal generator input
RAS         =       ra.1            ; DRAM row-address strobe
WR          =       ra.2            ; DRAM write line (0 = write)
Dout        =       ra.3            ; DRAM data line
adrb_lo     =       rb              ; Low bits of address bus
adrb_hi     =       rc              ; High bits of address bus
Din         =       rc.2            ; DRAM Q line
CAS         =       rc.3            ; DRAM column-address strobe
Sout        =       rc.5            ; Signal out to speaker

; Put variable storage above special-purpose registers.
            org     8

r_ctr       ds      1               ; Refresh counter
row_lo      ds      1               ; Eight LSBs of row address
row_hi      ds      1               ; Two MSBs of row address
col_lo      ds      1               ; Eight LSBs of column address
col_hi      ds      1               ; Two MSBs of column address
flags       ds      1               ; Holder for bit variable flag
flag        =       flags.0         ; Overflow flag for 20-bit address

; Remember to change device info when programming part.
            device  pic16c55,xt_osc,wdt_off,protect_off
            reset   start
```

Note 12: Using Dynamic RAM

```
; Set starting point in program ROM to zero
                org     0

start           setb    RAS             ; Disable RAS and CAS before
                setb    CAS             ; setting ports to output.
                mov     !ra,#1          ; Make ra.0 (Sin) an input.
                mov     !rb,#0          ; Make rb (low addresses) output.
                mov     !rc,#00000100b  ; Make rc.2 (Din) an input.
                clr     flags           ; Clear the variables.
                clr     row_lo
                clr     row_hi
                clr     col_lo
                clr     col_hi
                call    refresh         ; Initialize DRAM.

:record         call    refresh         ; Refresh the DRAM.
                call    write           ; Write Sin bit to DRAM.
                call    inc_xy          ; Increment row and col
                                        ; addresses.
                jnb     flag,:record    ; Repeat until address overflows.

:play           call    refresh         ; Refresh the DRAM.
                call    read            ; Retrieve bit and write to Sout)
                call    inc_xy          ; Increment row and col
                                        ; addresses.
                goto    :play           ; Loop until reset.

write           mov     adrb_lo,row_lo  ; Put LSBs of row addr on bus (rb).
                AND     adrb_hi,#11111100b ; Clear bits adrb_hi.0 and .1.
                OR      adrb_hi,row_hi  ; Put MSBs of row adr on bus (rc).
                clrb    RAS             ; Strobe in the row address.
                movb    Dout,Sin        ; Supply the input bit to the DRAM,
                movb    Sout,Sin        ; and echo it to the speaker.
                mov     adrb_lo,col_lo  ; Put LSBs of col addr on bus (rb).
                AND     adrb_hi,#11111100b ; Clear bits adrb_hi.0 and .1.
                OR      adrb_hi,col_hi  ; Put MSBs of col addr on bus (rc).
                clrb    WR              ; Set up to write.
                clrb    CAS             ; Strobe in the column address.
                setb    WR              ; Conclude the transaction by
                setb    RAS             ; restoring WR, RAS, & CAS high
                setb    CAS             ; (inactive).
                ret

read            mov     adrb_lo,row_lo  ; Put LSBs of row addr on bus (rb).
                AND     adrb_hi,#11111100b ; Clear bits adrb_hi.0 and .1.
                OR      adrb_hi,row_hi  ; Put MSBs of row addr on bus (rc)
                clrb    RAS             ; Strobe in the row address.
                mov     adrb_lo,col_lo  ; Put LSBs of col addr on bus (rb).
                AND     adrb_hi,#11111100b ; Clear bits adrb_hi.0 and .1.
                OR      adrb_hi,col_hi  ; Put MSBs of col addr on bus (rc).
```

```
                clrb    CAS             ; Strobe in the column address.
                movb    Sout,Din        ; Copy the DRAM data to the
                                        ; speaker.
                setb    RAS             ; Conclude transaction by
                                        ; restoring
                setb    CAS             ; RAS and CAS high (inactive).
                ret

; This routine implements a CAS-before-RAS refresh. The DRAM has a counter to
; keep track of row addresses, so the status of the external address bus doesn't
; matter. The DRAM requires 512 rows be refreshed each 8 ms, so this routine must
; be called once every 125 microseconds. Changing the initial value moved into r_ctr
; will alter the refresh schedule. To refresh the entire DRAM, move #0 into r_ctr (256
; loops) and call the routine twice in a row (512).

refresh         mov     r_ctr,#8
:loop           clrb    CAS             ; Activate column strobe.
                clrb    RAS             ; Activate row strobe.
                setb    CAS             ; Deactivate column strobe.
                setb    RAS             ; Deactivate row strobe.
                djnz    r_ctr,:loop     ; Repeat.
                ret

; This routine increments a 20-bit number representing the 1,048,576 addresses of
; the DRAM. The number is broken into two 10-bit numbers, which are each stored
; in a pair of byte variables. Note that wherever the routine relies on the carry bit to
; indicate a byte overflow, it uses the syntax "add variable,#1" not "inc variable." Inc
; does not set or clear the carry flag.

inc_xy          add     row_lo,#1       ; Add 1 to eight LSBs of row addr.
                sc                      ; If carry, add 1 to MSB, else
                                        ; return.
                ret
                inc     row_hi          ; Increment MSBs of row address.
                sb      row_hi.2        ; If we've overflowed 10 bits, clear
                ret                     ; row_ hi and add 1 to column
                clr     row_hi          ; address, else return.
                add     col_lo,#1
                sc                      ; If carry, add 1 to MSBs, else
                                        ; return.
                ret
                inc     col_hi          ; Increment MSBs of col address.
                sb      col_hi.2        ; If we've overflowed 10 bits, clear
                ret                     ; col_ hi and set the flag to signal
                clr     col_hi          ; main program that we've reached
                setb    flag            ; the end of memory, then return.
                ret
```

BLANK PAGE

Note 13: Running at 32 kHz

Introduction. This application note presents a method for operating the Parallax Downloader from a watch crystal. 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.

A minor stumbling block in LP development is that you can't just plug a watch crystal into the Parallax Downloader. The Downloader's bondout chip—the device responsible for imitating a PIC in your circuit—has an XT/HS-type oscillator, which won't work with a 32-kHz crystal. However, the Downloader can run off an external clock oscillator like the ones shown in figure 1.

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

Note 13: Running at 32 kHz

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.

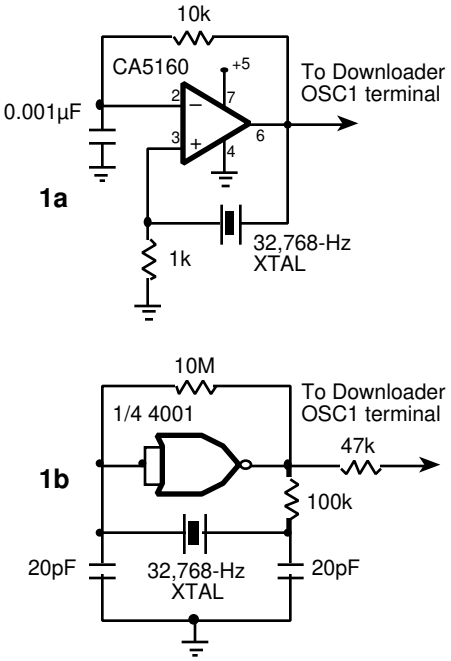


Figure 1. 32.768-kHz clocks.

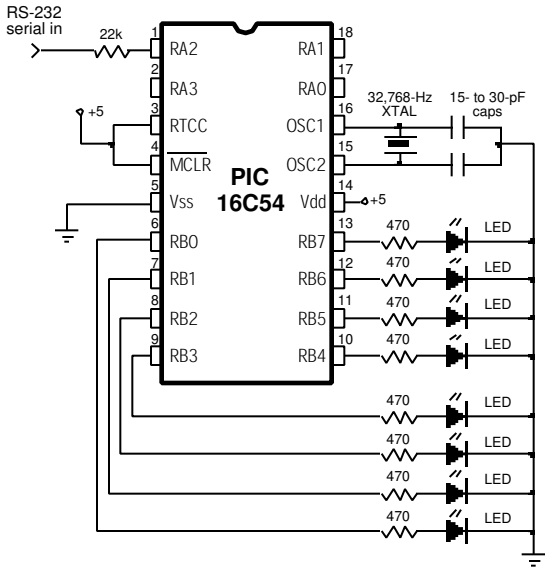
The second oscillator (figure 1b) uses a CMOS inverter as its active element. In this case, the role of the inverter is played by one of the NOR gates of a 4001 chip. Adjusting the values of the 20-pF capacitors slightly will fine-tune the frequency of oscillation. Do not omit the 47-k resistor on the 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.

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.

Program listings. These programs may be downloaded from the Parallax BBS as SLOW1.SRC and SLOW2.SRC. You can reach the BBS at (916) 624-7101.

```
; 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.

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.
```

Note 13: Running at 32 kHz

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

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

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

Note 13: Running at 32 kHz

```
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
```

BLANK PAGE

Note 14: Generating/mixing Sine Waves

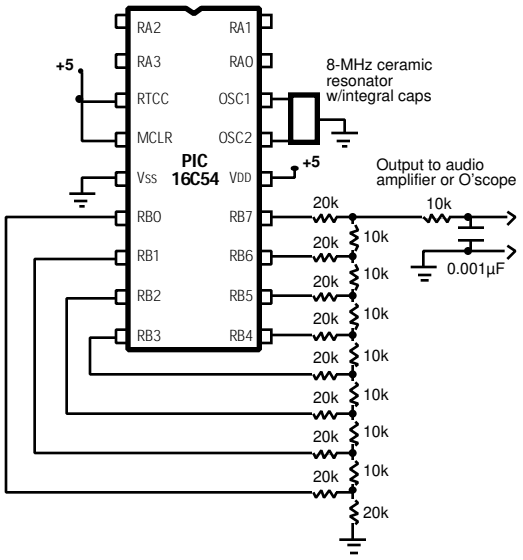
Introduction. This application note illustrates methods for generating audio-frequency waveforms using lookup tables.

Background. Digitally generating sine waves or other complex waveforms may seem like wasted effort compared to using a few discrete components or an IC to do the same job. If the goal is just to make a pure tone, that view is probably true. However, if you need precise control over the frequency, duration and phase of a waveform, digital generation becomes downright cheap and simple compared to the analog alternatives.

The basic procedure for generating a sine wave is easy enough: At precise time intervals, the program looks up a value from a table, delivers it to a D/A converter, then waits another interval to put out the next value. At the output of the D/A converter, the signal is a connect-the-dots version of a sine wave. The addition of a low-pass filter smoothes the jaggies, and we have a real sine wave.

Other waveforms can be made in a similar way, if their patterns repeat within a time interval that can be stored in a reasonably sized lookup table, or computed on-the-fly. If they cannot, then they must be synthesized by combining multiple sine waves.

The program in the listing demonstrates both techniques. When the frequency constants *freq1* and *freq2* are set to the same value, the circuit's output is a pure sine wave. That's because the two pointers *phase1* and *phase2* move through the table of sine values at the same rate. However, if the *freq* constants are different, the circuit produces a mixture of two sine waves.



Note 14: Generating/mixing Sine Waves

How it works. The resistors in figure 1 are connected in an array known as an R-2R ladder. This arrangement has an interesting property. It divides each voltage present at its inputs by increasing powers of two, and presents the sum of all these divided voltages at its output. Since the PIC is capable of driving its outputs from rail to rail (0 to +5 volts), the R-2R ladder converts the binary number present on port B into a proportional voltage from 0 to 5 volts in steps of approximately 20 millivolts.

Most commercial D/A converters work on this same principle, but have internal voltage regulators, logic, and latches. Since our demonstration doesn't require any of those things, we can use the resistor array alone. Unfortunately, it can be difficult to find a prefabricated R-2R ladder at the volume parts houses. Figure 2 shows how to construct one from 25 surface-mount 10k resistors and a tiny printed circuit pattern. Alternatively, you may adapt the program listing to drive your favorite packaged D/A converter.

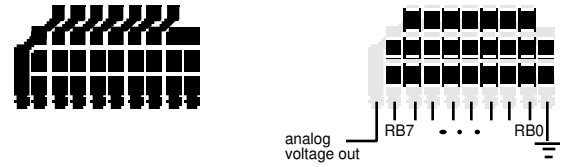


Figure 2. Constructing an R-2R digital-to-analog converter.

The program that drives the D/A converter begins by setting port rb to output and clearing its variables. Next it enters this loop:

- Add *freq1* to *acc1*
  > If *acc1* overflows (generates a carry), increment *phase1*
- Put the sine value corresponding to *phase1* into *temp*
- Add *freq2* to *acc2*
  > If *acc2* overflows (generates a carry), increment *phase2*
- Add the sine value corresponding to *phase2* into *temp*
- Copy *temp* to the output
- Do it again.

Note 14: Generating/mixing Sine Waves

The loop always takes the same amount of time to execute, so how do *freq1* and *freq2* control the frequency? If the value of *freq1* is 1 and *acc1* starts out at 0, the program will complete 255 loops before acc1 overflows, generating a carry. *Phase1* will increment once in every 256 loops. This will produce a low output frequency. If *freq1* is 100, *acc1* will overflow on the third trip through the loop, so *phase1* will increment much faster, producing a higher frequency.

This scheme, while compact, is not perfect. Take that example of *freq1* = 100. If *acc1* starts off at 0, the sequence goes as shown in the table to the right.

acc1	carry
0	0
100	0
200	0
44	1
144	0
244	0
88	1
188	0
32	1
132	0
232	0
76	1
176	0
20	1

The rate at which carries occur is, on average, proportional to *freq1*. However, the interval between carries can be 1, 2, or 3 trips through the loop. At higher frequencies, these variations become noticeable as a thickening or jittering of traces on the oscilloscope screen. If your application requires particularly pure output, keep this in mind.

Modifications. Play with the values of *freq1* and *freq2* and observe the results on the oscilloscope. Compare the results to a textbook discussion of frequency mixing.

Try removing the resistor/capacitor low-pass filter from the output of the R-2R ladder and look at the change in the output waveform. If you are listening to the output through an audio amplifier, you may find that you prefer the sound of the unfiltered D/A converter. The harmonics it produces give the tone more sparkle, according to some listeners.

Perhaps the most useful modification is this: Set *freq1* and *freq2* to the same value, say 100. Run the program and listen to the tone or observe it on the 'scope. If you use a scope, make note of the peak-to-peak amplitude of the tone. Now substitute *mov phase1, #7* for *clr phase1* at the beginning of the program. Run the program again. Do you hear (see) how the amplitude of the tone has fallen off? Try once more with *mov phase1,#8*. No tone at all. This is a graphic illustration of how out-of-phase sine waves of the same frequency progressively cancel each other. The 16 sine table entries represent amplitudes at intervals of 22.5

degrees. When *phase1* is initialized with a value of 8, while *phase2* is zero, the phase shift between the two is 22.5 x 8 = 180 degrees. Two sine waves 180 degrees out of phase cancel each other totally. Smaller phase shifts produce proportionally less attenuation.

Given the PIC's limited math capabilities, this phase shifting technique is an easy way to control the overall amplitude (volume) of the tones you generate.

Program listings. This program may be downloaded from the Parallax BBS as SINE.SRC. You can reach the BBS at (916) 624-7101.

```
; Program: Synthesizing sine waves (SINE.SRC)
; This program mixes two sinewave signals and outputs them to port b for
; conversion to analog by an R-2R ladder D/A converter.

output          =       rb                      ; Ladder DAC on port b.
freq1           =       100                     ; Sets frequency 1.
freq2           =       50                      ; Sets frequency 2.

; Set aside variable space above special-purpose registers.
                org     8
acc1            ds      1                       ; Accumulator for phase 1.
acc2            ds      1                       ; Accumulator for phase 2.
phase1          ds      1                       ; Position in sine table, p1.
phase2          ds      1                       ; Position in sine table, p2.
temp            ds      1                       ; Temporary variable.

; Device data and reset vector (remember to change if programming a different part).
                device pic16c54,xt_osc,wdt_off,protect_off
                reset start
                org 0

start           mov     !rb,#0                  ; Make rb an output.
                clr     acc1                    ; Clear the accumulators.
                clr     acc2
                clr     phase1                  ; Clear phase pointers.
                clr     phase2
:loop           add     acc1,#freq1             ; Add freq1 to acc1
                addb    phase1,c                ; If carry, increment phase1.
                AND     phase1,#00001111b       ; Limit to 0-15.
                add     acc2,#freq2             ; Do the same for
                addb    phase2,c                ; phase2.
                AND     phase2,#00001111b
                mov     w,phase1                ; Look up sine value in table.
                call    sine
```

```
                mov     temp,w          ; Store 1st sine value in temp.
                mov     w,phase2
                call    sine            ; Get value from table.
                add     temp,w          ; Add sines together.
                mov     output,temp     ; Output to D/A.
                goto    :loop           ; Do forever.

; Notice that the sine values in the table below are scaled so that no pair adds to
; more than 254. This prevents overflowing the variable temp when the phases are
; added together. If you add more phases, adjust the maximum values in this table
; appropriately. The values are in increments of 22.5 degrees (0.3927 radians).
sine            jmp     pc+w
                retw    64,88,109,122,127,122,109
                retw    88,64,39,19,5,0,5,19,39
```

BLANK PAGE

Note 15: Using Interrupts

Introduction. This application note shows how to use the interrupt capabilities built into the PIC 16Cxx series controllers with a simple example in Parallax assembly language.

Background. Many controller applications work like a fire department. When there's no fire, they mend hoses, polish the fire truck, and wait for something to happen. When the fire bell rings, they swing into action and handle the emergency. When it's over, they return to waiting.

If there were no fire bell, the job would be much different. The fire crew would have to go out and look for fires, and somehow still make sure that the hoses were fixed and the truck maintained. In their scurry to get everything done, they might respond late to some fires, and miss others completely.

The newer PIC 16Cxx controllers have interrupts, which work like the fire bell in the first example. When an interrupt occurs, the PIC stops what it's doing and handles the cause of the interrupt. When it's done, the PIC returns to the point in the program at which it was interrupted.

The second example, with no fire bell, is an example of polling. This is the approach used with the 16C5x PICs, which lack interrupts. For a fast PIC, polling isn't nearly as bad as in the example. The PIC can often get everything done with plenty of time to spare. But there are times when interrupts are the simplest way to do two or more things at once.

How interrupts work. For the examples here, we're going to talk about the 16C84. The same principles apply to the other interrupt-capable PICs, such as the 16C71 and 16C64, but registers and interrupt sources may vary.

First of all, the 16Cxx PICs awaken from power-up with all interrupts disabled. If you don't want to use interrupts, don't enable them. It's that simple.

The PIC 16C84 can respond to interrupts from four sources:

- A rising or falling edge (your choice) on pin RB0/INT.

- Changing inputs to pins RB4 through RB7.
- Timer (RTCC) overflow from 0FFh to 0.
- Completion of the data EEPROM programming cycle.

You can enable any combination of these sources. If you enable more than one, it will be up to your code to determine which interrupt occurred and respond appropriately. More on that later.

Let's take a simple example. We want to use the RTCC interrupt to generate a steady 1-kHz square wave on pin ra.1. Every 500 µs the RTCC will interrupt whatever the PIC is currently doing, toggle ra.1, reload the RTCC with an appropriate value and return.

The first consideration is where to put the interrupt-handler code. When the 16C84 responds to an interrupt, it jumps to location 04h in program memory. So we'll use an **org** statement to place the handler at 04h. If you look at the program listing, you'll see that we actually have two **org**s: the first positions the code **jmp start** at 0, which is where the '84 looks for its power-on startup code, and the second positions the interrupt handler at 4.

The program's startup code configures the RTCC for its initial 500-µs timing period, and enables the interrupt. To do so, it turns on the first two bit switches shown in figure 1. For any interrupt to occur, the global-interrupt enable (GIE) bit must be set. For the timer interrupt to occur, the RTCC interrupt enable (RTIE) bit must be set. Once those two switches are closed, only one switch remains before the interrupt "alarm bell" goes off—the RTCC interrupt flag (RTIF).

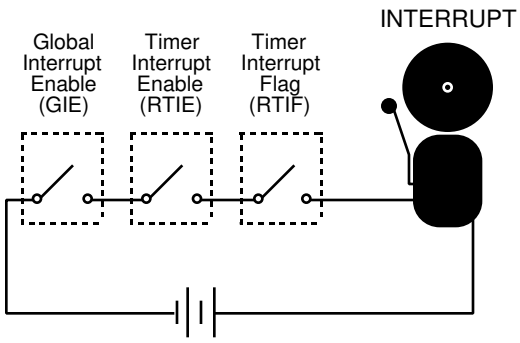


Figure 1. Logic of the interrupt-enable and flag bits for the RTCC.

When the interrupt occurs, the PIC clears GIE to disable further interrupts, pushes the program counter onto

the stack, then jumps to location 4. This process takes four instruction cycles for an internal interrupt like the RTCC, five for an external interrupt like RB0/INT.

Once at location 4, the PIC executes the code there until it encounters a **ret**, **retw**, or **reti** instruction. Any of these will pop the value from the top of the stack and cause a jump back to the point at which the program was interrupted. However, the normal way to return from an interrupt is the **reti** instruction. It automatically re-enables interrupts by setting GIE. The other two returns do not.

In the program listing, you will notice that the first thing the interrupt handler does is clear RTIF. Whenever the RTCC rolls over from 0FFh to 0, the PIC automatically sets RTIF. It does this regardless of the state of the interrupt bits. And it never turns RTIF back off. So it's the responsibility of the interrupt handler to clear RTIF. This is true of all of the interrupt flags. If the handler doesn't clear the appropriate flag, the same interrupt will occur again as soon as interrupts are re-enabled. The resulting endless loop is what the Microchip documentation calls "recursive interrupts."

With a single interrupt source, you can think of an interrupt as a hardware version of the **call** instruction. The primary difference is that this kind of **call** can occur anywhere in your program, whether or not the program is ready for it. This can pose a problem. Let's say your program is executing the Parallax instruction **add sum,delta** when the interrupt occurs. That **add** instruction actually consists of two instructions; one that loads the variable **delta** into the **w** register, and a second that adds **w** to **sum**.

Imagine that right after **delta** is loaded into **w**, the interrupt takes over. It performs, for instance, the instruction **mov rb,data**. This instruction loads the variable data into **w**, then moves **w** into **rb**. When it's done, **w** contains a copy of **data**.

When the handler returns, the PIC completes the interrupted addition. But **w** contains **data**, not the intended **delta**, causing an error.

There are two ways to prevent this. One is to disable interrupts before a two-part instruction, then enable them again afterwards. If the inter

rupt event occurs while interrupts are disabled, the PIC will set the interrupt flag, but won't jump to the handler until your program re-enables interrupts. The PIC will not miss the interrupt, it will just be slightly delayed in handling it. To use this method, the example above would be changed to:

```
clrb    GIE         ; Interrupts disabled.
add     sum,delta   ; w = delta: sum = sum+w.
setb    GIE         ; Interrupts enabled.
```

The alternative approach is to begin your interrupt handler with code that saves a copy of the **w** register. Then, just before **reti**, move the copy back into **w**.

This takes care of compound instructions that use **w**, but leaves another group of instructions vulnerable; the ones that use the status register. Conditional jumps and skips like jump-if-zero (**jz**) and compare-and-jump-if-equal (**cje**) and their many cousins are probably obvious. More subtle are the rotate instructions **rr** and **rl** (which pull the carry bit into the most- or least-significant bit of the byte being rotated). If the interrupt handler affects any of the status bits, it may alter the outcome of the interrupted instruction.

To protect **w** and **status**, you must make copies of them at the beginning of a handler, then restore those copies at the end. In order to prevent the action of restoring **w** from affecting **status**, you can use a sneaky trick. Moving a file register into **w** will set or clear the zero bit, but moving a nibble-swapped copy of the register into **w** does not. Neither does swapping the nibbles of a file register. The program listing shows how to use these loopholes to accurately copy both **w** and **status**.

Keep your interrupt handlers as short and simple as possible. Examine them carefully for their effect on other portions of the program that might be executing at the time the interrupt occurs. If necessary, protect sensitive code by bracketing it with instructions that temporarily disable interrupts. Also, keep in mind that test running your program may not catch all possible interrupt-induced bugs. Use the PSIM simulator and a sharp eye to detect potential problems.

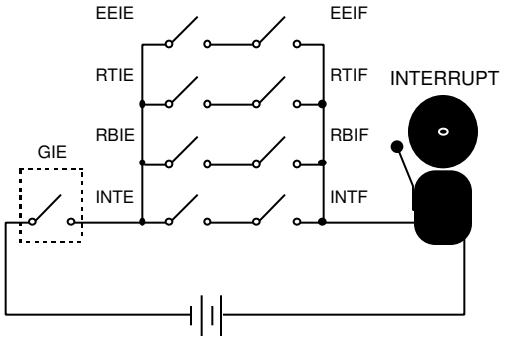
Once you understand how the mechanism works with a single inter

rupt source, you'll be relieved to know that it's not that much more difficult to handle multiple interrupt sources. Figure 2 shows the alarm-bell diagram for multiple interrupts on the 16C84. If you have all of the 16C84's interrupts enabled, your handler at 04h should begin with something like:

```
jb   INTF, rb0_edge     ; If INTF then handle rb.0 interrupt.
jb   RBIF, rb_change    ; If RBIF then handle change on rb.4-7.
jb   RTIF, timeout      ; If RTIF, then handle timer rollover
jb   EEIF, EE_wr_done ; If EEIF, then handle write complete.
```

Code at each of those labeled locations (**rb0_edge**, etc.) would then deal with that particular type of interrupt. Remember that each of the handlers must clear its corresponding flag; for example, **rb0_edge** must include the instruction **clrb INTF**.

What happens if an interrupt occurs while the PIC is already handling an interrupt? At that time, nothing. Remember that the PIC clears GIE automatically in response to an interrupt, then sets it when **rti** executes. Any interrupt event that occurs in the meantime will set the appropriate flag. That interrupt will be delayed until after the current one is finished.



INTE/F = RB0 interrupt enable/flag bits
RBIE/F = RB4–7 change interrupt enable/flag bits
RTIE/F = RTCC overflow interrupt enable/flag bits
EEIE/F = EEPROM write complete interrupt enable/flag bits

Figure 2. Logic of all four 16C84 interrupts.

Program design. Interrupts are not a cure-all for the difficulties of handling multiple tasks. In fact, you may just end up trading a difficult programming job for an even more difficult debugging job. Experts suggest using interrupts only as a last resort.

Note 15: Using Interrupts

```
; Program: INTRPT.SRC (Interrupt demonstration)
; This program illustrates the use of the RTCC interrupt
; in the PIC 16C84. The foreground program blinks an LED on
; pin ra.0, while an interrupt task outputs a 1-kHz square
; wave through pin ra.1 (assuming a 4-MHz clock).

; Device (16c84) and setup options.
          device pic16c84,xt_osc,wdt_off,pwrt_off,protect_off

; Equates for LED, tone pins. Connect the LED through a
; 220-ohm resistor. Connect a speaker or earphone through
; a 1-k resistor.
LED       =    ra.0
SPKR      =    ra.1
tone      =    6  ; Load into RTCC for 500-us delay.

; Allocate space for some variables. Notice that in the '84
; variable start at 0Ch—higher than in the 16C5x series.
          org  0Ch
w_copy    ds   1
s_copy    ds   1
countL    ds   1
countH    ds   1

; On startup, the PIC looks at address 0 for its first
; instruction. Since the interrupt handler begins at
; address 4, we'll just jump over it to get to the
; startup routine.
          org     0
          jmp     start          ; Beginning of main program.

; Next is the interrupt handler, which must begin at
; address 4. This handler copies restores w and the status
; register. Because a normal "mov w,fr" alters the z bit of
; the status register, this routine uses "mov w,<>fr," which
; does not. The routine actually swaps the byte twice,
; resulting in the correct value being written to w without
; affecting the z bit.
          org     4
handler
          clrb    RTIF           ; Clear the timer interrupt flag.
          mov     w_copy,w       ; Make a copy of w.
          mov     s_copy,status  ; Make a copy of status.
          XOR     ra,#2          ; Toggle bit ra.1.
          mov     rtcc,#tone     ; Reload rtcc for 500-us delay.
          mov     status,s_copy  ; Restore status register
          swap    w_copy         ; Prepare for swapped move.
          mov     w,<>w_copy     ; Swap/move to w, status unaffected.
          reti                   ; Return to main program.
```

Note 15: Using Interrupts

```
; Here's the startup routine and the main program loop.
; In the line that initializes "intcon," bit 7 is GIE and bit 5
; is RTIE. Writing 1s to these enables interrupts generally (GIE)
; and the RTCC interrupt specifically (RTIE).

Start
        mov     !ra,#0          ; Make ra pins outputs.
        setb    rp0             ; Switch to register page 1.
        clr     wdt             ; Assign prescaler to rtcc.
        mov     option,#0       ; Set prescaler to divide by 2.
        clrb    rp0             ; Restore to register page 1.
        mov     intcon, #10100000b    ; Set up RTCC interrupt.

; If the interrupt handler were to alter w, the LED would stop
; flashing or flash erratically, since the routine is written to
; rely on the value of w remaining 1 in order to toggle bit ra.0.
; The routine also relies on reliable operation of the status
; register because of the two skip-if-not-zero (snz) instructions.
; Although this structure is a little strange, it's an effective
; canary-in-a-coalmine demonstration that the interrupt handler's
; save/restore instructions do preserve both w and status.

        mov     w,#1            ; Bit in 0 position to toggle ra.0.
:loop   inc     countL          ; countL=countL + 1
        snz                     ; IF countL=0,
        inc     countH          ; THEN countH=countH+1
        snz                     ; IF countH=0,
        XOR     ra,w            ; THEN toggle LED.
        jmp     :loop
```