Stamp Applications no. 24 (February '97):

Modem Lets Stamps Access Global Communication Network

Dialing for Stamp Data
and Beginner's Race Timer with Display
by Scott Edwards

FOR $30 OR LESS, you can connect your Stamp project to the biggest communication network on the planet—the global phone system. All you need is a modem and a few tricks of the trade.

This month, we'll look at the powerful "AT" modem-control language with an example application that lets you access Stamp data as easily as logging onto a computer bulletin-board system (BBS) or online service. In BASIC for Beginners, we'll add a display and other features to our three-lane race timer.

Modem Fundamentals. A modem is a small, fast computer specialized for sending and receiving serial data across the phone lines in the form of audio tones. At low data rates, shifts in the frequency of the tones convey individual bits of data; at higher rates, changes in phase and/or amplitude of the signal represent multiple bit combinations.

The word *modem* is a compaction of *modulator/demodulator*. Impressing data onto tones is modulation; extracting data from modulated tones is demodulation.

The modem's computer is programmed with routines to dial the phone, detect ringing and busy signals, establish a connection with another modem, exchange data, and hang up. It has all kinds of settings for dealing with subtle variations in phone procedures and user preferences. In fact, a modem has more settings, options, instructions, and raw computing power than the BASIC Stamp II (BS2) that we'll be using to boss it around!

Almost all modems use some variation of the "Hayes" command set, named for the company that set the standard for modems in the early days of personal computing. This control language is also sometimes called the AT command set, because most commands begin with AT (for "attention," or so the story goes).

A modem has two modes of operation—command mode and data mode. When the modem is not linked to another modem, it's in command mode, awaiting instructions from the local computer. The computer can instruct the modem to dial out or answer a call from another modem. When two modems initially make contact, they investigate each other through *handshaking*—a ritual exchange of tones and preliminary data that sets the ground rules for communication.

Once handshaking is complete, the modem goes into data mode. In data mode it acts like a direct connection between local and remote computers. Once in data mode, a modem ignores commands unless it is first returned to command mode by a sort of secret knock (usually "+++" followed by a delay). Otherwise, it would be impossible for modem users to communicate about modem commands without their modems trying to carry out those commands!

When the exchange of data is complete, one of the computers issues the secret knock to return its modem to command mode, then instructs it to hang up. The other modem senses this *loss of carrier*, and hangs up too.

Meet Your Modem. The modem you'll need for Stamp applications is an *external* modem—one designed to be connected through a serial port, not installed inside a desktop computer (*internal*). If you don't have one, let me suggest Jameco part no. 132206, a 9600-baud external modem with power supply and cable for $29.95 (Sources). This is an older unit that may not be available forever, so keep your eye open for bargains on similar external-style modems.

A benny of buying an older modem is that it usually comes with a good manual on the AT command set. In the old days users generally dealt directly with the modem, manually typing commands through a terminal program. The manuals that come with newer modems assume that you're using more helpful software and therefore require less technical data.

To really understand how a modem works, you should try using it manually with your PC and terminal software. If you have Windows installed, there's a simple terminal "accessory" program. Procomm is a popular commercial package, available in both DOS and Windows flavors.

Although the modem manual lists dozens of commands, you can get by with knowing a few important ones. Table 1 lists the ones I found helpful in getting the Stamp on line. Once your terminal program is set up and talking to the modem, you may send commands by typing them at the keyboard. For example, if you type "ATDT 5551234" followed by the Enter key, the modem will go *off-hook* (connect to the phone line) and send the touch-tone digits 555-1234.

Table 1. Some Useful AT Modem Commands

Instruction	Operation
ATDT *n*	Dial n using touch tones. ATDT 4594802 dials 459-4802. The number to dial can include other symbols that change way it is dialed, including: , Pause dialing W Wait for second dial tone ; Stay in command mode after dialing
A/	Repeat last command entered.
ATH0	Hang up.
ATA	Answer the phone now (manual answer).
ATV0	Set modem reponses to numbers (0—10)
ATV0	Set modem reponses to words (e.g., *CONNECT 2400*).
ATZ	Return modem to default settings.
AT&W	Store configuration settings to nonvolatile memory.
ATS0=0	Turn off auto-answer function.
ATS0= *n*	Turn on auto-answer and set for n rings.
ATM0	Silence speaker.
ATM1	Turn speaker on while dialing; off during communication.
ATE0	Do not echo commands.
ATE1	Echo commands.
+++	Escape command: shifts modem from data to command mode after a preset time delay (set by register S12).
ATS12= *n*	Set escape (+++) delay to n number of 20-millisecond units. For example, ATS12=50 sets the escape delay to 1 second.

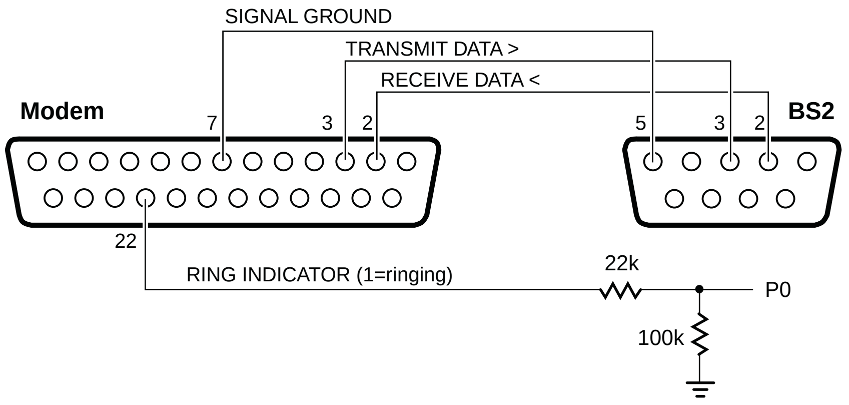

Figure 1. Modem-to-BS2 Hookup.

Configuring and Using the Modem. Once you know how to send commands to the modem, you can configure it to work in our example BS2 application. As figure 1 shows, we're using the BS2's serial programming port to talk to the modem. A peculiarity of this port is that everything sent to it on the serial-in line is sent back to the serial-out line. Modems are factory configured to do the same thing.

If you take a modem out of the box and have a BS2 send it a command, the modem will echo the command. The BS2 will echo the echo, which the modem will echo, which the BS2 will echo... *to infinity and beyond*, if I may quote Buzz Lightyear's catchy motto.

A further catch is that this loop effect prevents the BS2 from issuing the ATE0 (no-echo) command. You must configure the modem properly before it will understand anything the BS2 has to say.

Listing 1 includes a table of configuration commands to send to the modem to prepare it for use with the BS2. Once the modem is configured, you should use the AT&W command to commit the configuration to nonvolatile memory so that it is preserved after the modem is turned off and back on.

As you can see from the program listing, it is possible for even a very simple program to answer the phone, establish a link, and communicate by modem. You can use this program as a basic framework for your own applications.

Advanced Applications. If you decide to develop modem-based applications, you will almost certainly need a telephone-line simulator. I used the Digital Products Company Party Line unit, which I built from a kit (Sources). If your construction skills are at the beginner level, or you tend to disregard instruction manuals, I recommend buying the assembled version. It's not a difficult kit, but the large number of components may exceed short attention spans.

Although I demonstrated how to receive a call, you can use the same principles to have the Stamp dial a call and log onto a BBS or on-line service. Most of the work in developing this kind of application would be in carefully documenting the log-on procedures for the Stamp to follow.

Don't overlook the fact that modems allow you to access other phone services. You can dial pagers or automated touch-tone menus and send them sequences of tones. The manual I received with the Jameco modem provides an example under the heading Dialing a Voice Call/Sending Tones as Data.

In your applications, make sure to take advantage of the BS2's Serin *timeout* feature. If no data is received within a set period of time (up to 65+ seconds) the BS2 can abort the Serin operation. You can see an example of this in Listing 1 when the program is listening for the connect code ("10"). If no connect occurs within 10 seconds (set by the constant tLink), the program hangs up and waits for another call.

My final hint: Use the lowest practical data rate in modem comms. The faster the rate, the more fragile the link. You can spend your nights and weekends writing clever and elaborate error-detection and correction code, but that code can grow to overwhelm the Stamp's program memory. Also, at very high data rates the Stamp may be unable to recognize longer WAIT sequences (like the password "USER" in listing 1). I've received credible information to this effect from a seasoned BS2 user, and I'll investigate the problem (and provide a solution) in a future column.

BASIC for Beginners. In the column before last (no. 22, December '96), we developed a prototype three-lane race timer for Pinewood Derby racing. (Pinewood racers are small, gravity-powered cars crafted from a block of wood and raced on an inclined track.)

In its previous state of development, the program could track the individual finish times of three cars in terms of the number of timing loops and display the loop counts on the Debug screen. Our next hurdles are to free the program of the PC by giving it a standalone display, and to convert its loop-count values into seconds.

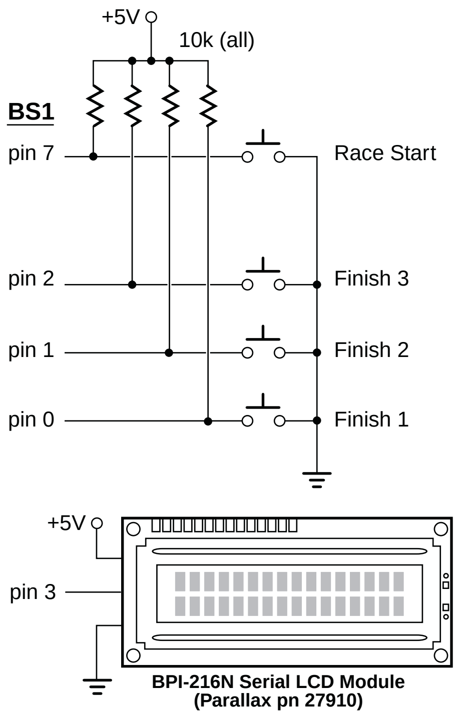

Figure 2. Race timer schematic.

For the display, I chose the 2x16 Serial LCD Module that my company makes. Aside from the fact that they're readily available, these LCDs look slick, can display mixtures of words and numbers, and are easy to drive with the Stamp's Serout instruction. Setting aside my obvious bias, I think you would be hard pressed to find a better display for the purpose.

Interfacing the display to the Stamp consists of connecting ground, +5V, and one Stamp input/output (I/O) pin. To print text to the display, output it serially through the appropriate I/O pin at 2400 baud, inverted:

```
Serout 3,N2400,("RACE IN PROGRESS")
```

To print the value of a variable, use this form:

```
Serout 3,N2400,(#time1)
```

That assumes that the variable named time1 is already defined using the Symbol directive. See listing 2 for examples.

The serial LCD also accepts instructions that control *how* it displays data. There's a list of instructions in the manual, but many applications get by with just two: clearing the screen and positioning the cursor. To tell the display that an instruction is coming, just precede it with a byte containing 254, like so:

```
Serout 3,N2400,(254,1)
```

The 254 alerts the display to expect an instruction; 1 is the instruction to clear the screen. After the instruction, the display returns to data mode.

Positioning the cursor works similarly. The 2-line by 16-character (2x16) LCD screen is mapped to a set of addresses. The first line consists of addresses 128 through 143; the second, 192 through 207. Each of these addresses is a byte of RAM in the LCD controller. The LCD has additional RAM (a total of 80 bytes), but locations outside the ranges mentioned above are not visible on a 2x16 display.

To set the printing location to a particular spot on the screen requires sending the instruction prefix followed by the screen address:

```
Serout 3,N2400,(254,200)
```

The cursor moves to address 200, which is character 8 of the screen's second line. (The manual that comes with the display includes a map of the whole screen, so it's not necessary to memorize this stuff.) The program in listing 2 uses just these clear-screen and cursor-position instructions to organize the race-time data on the display.

Now that we have a way to display data without the PC, we need to do some work on the data itself. The prototype program expresses racing time in terms of the number of program loops executed before a racer crosses the finish line. By running that program and comparing the loop values to stopwatch time, I found that 338 loops equal 1 second. A straightforward way to convert loops to seconds would be to divide by 338.

Since the Stamp deals only in integer math, race results would be only in whole numbers of seconds; 1,2,3... It's reasonable to expect that many races would be decided by a margin of a fraction of a second. We want our timer to report times with appropriate precision.

The first improvement to consider would be moving the decimal point. Instead of reporting seconds, how about tenths or hundredths of a second? The program executes 33.8 loops every 1/10th second or 3.38 loops every 1/100th. To convert loops to 1/100ths, just divide by 3.38. Oops—there's that integer math limitation again. The Stamp can divide by 3 or 4, but not by 3.38. Dead end? Not quite.

Another way of looking at a division problem is to consider it as multiplication by a fraction. Dividing a number by 3, or multiplying it by one-third have the same effect. And the Stamp *can* multiply by a fraction, which is nothing more than integer multiplication followed by integer division. The trick is to find an integer fraction that's equal to 1/3.38.

There are lots of ways to approach this problem, but I find that common sense favors this method: (1) Estimate the largest value of the numerator (the "1" in 1/3) that can be used without exceeding the maximum variable size. (2) For each integer up to the maximum, pick a likely candidate for the denominator and calculate the result.

An example will make this clear. For step 1, the largest numerator depends on the size of a Stamp word variable (65535 max), the number of loops per second (338), and the maximum number of seconds we wish to measure. If we want to measure times up to 20 seconds, then the maximum value for the numerator is 65535/(338*20) = 9. That narrows our search considerably.

In step 2, for numerators from 1 to 9, we're looking for denominators that make the fraction as close as possible to 1/3.38 (0.295858). That's pretty close to 1/3, so candidate denominators should be between three and four times the numerator. Simplified this way, we can quickly run the numbers on a hand calculator, or create a spreadsheet or PC program to do it for us. In this case, the number of trial values is small:

Numerator	Denominator	Result	Error %
1	3	0.333	+11
1	4	0.250	–15
2	7	0.286	–3.4
3	10	0.300	+1.4
4	13	0.308	+4
5	16	0.31	+5.6
5	17	0.294	–0.6
6	21	0.286	–3.4
7	23	0.304	+2.87
7	24	0.292	–1.4
8	27	0.296	+0.15
9	31	0.290	+0.29

The fraction 8/27 gives us the smallest error, just 0.15 percent. Over the course of a 20-second race, that would be 30 milliseconds, or 3/100ths of a second. Considering that the error would apply equally to all cars, and that the timing accuracy of the Stamp's internal clock is ±0.5 percent, that seems acceptable.

Now that we have a usable fraction, the rest is easy; multiply the loop count by 8/27 and display the results as 1/100ths of a second. Listing 2 shows how this is done.

For a more human-friendly display, the program inserts a decimal point so that numbers like 891 hundredths of a second show up as 8.91 seconds. And since some races may be decided by a margin of less than 1/100th of a second, the program also uses the raw loop count to declare the winner (or winners, in the case of a tie). It uses a process-of-elimination logic, reasoning that any race time that is greater than one or the other or both of the other times is a loser. Any time that's not a loser is marked on the display as a winner. If there's one winner, it will be the only time marked. If there's a tie, the tied times will be marked as winners (*co-winners*, I guess).

That wraps up our race-timer project, at least for now. In the future, I'll use it as a basis to demonstrate different display techniques, hardware timing options, sensors, etc.

NOTE: This article was originally published in 1997. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.

Listing 1. BS2 Program to Demonstrate Modem Interface

```
' Program: ANSW_MDM.BS2 (Answer modem and exchange data)
' This program demonstrates how the BS2 can be used with
' an AT/Hayes-compatible modem to link up with incoming
' modem calls and exchange data. It uses the programming/
' downloading serial port plus one I/O line for the modem
' hookup. Before this program can be used, the modem must
' be configured to work with the BS2's serial port by
' disabling echo-back of commands. (Failure to do this
' will cause the modem's RD and SD lights to stay lit
' continuously and the modem to become locked up.) Prepare
' the modem by connecting it to a PC running terminal
' software at 2400 baud, N81, full duplex. Send the modem
' configuration commands, as follows.
'  Type this command  Modem Responds  Purpose/Effect
'  -----------------  --------------  ---------------------
'   ATS0=0 <Enter>      "OK" or "0"   Disable auto-answer
'   ATS12=50 <Enter>    "OK" or "0"   Set "+++" response to 1s
'   ATV0 <Enter>        "0"           Set number responses
'   ATE0 <Enter>        "0"           Disable command echo
'   AT&W <Enter>        "0"           Memorize configuration
' After the ATE0, your typing won't be visible on the screen
' unless you reconfigure the terminal for half-duplex.
' These settings are stored in nonvolatile memory, so the modem
' does _not_ have to be reprogrammed if power is turned off.
' Connect the BS2 to the modem as follows:
'           BS2                        Modem DB25
'       ------------------------------------------
'        DB9, pin 2                       pin 2
'        DB9, pin 3                       pin 3
'        DB9, pin 5                       pin 7
'        I/O pin P0 (thru 22k resistor)  pin 21
' Turn the modem on first, then the BS2. When the phone rings,
' the BS2 will instruct the modem to answer and try to establish
' a connection. If connection is successful, the remote computer
' will see a sign-on message and be prompted for a password ("USER").
' The BS2 will then transmit a simulated batch of data. When the
' BS2 hangs up, the remote computer's modem will hang up too, ending
' the exchange.

lf      con    10      ' Linefeed character to format data.
tLink   con    10000   ' # of milliseconds to wait for link up.
N2400   con    16780   ' Baudmode for 2400 bps inverted.
TxD     con    16      ' Pin to output serial data.
RxD     con    16      ' Pin to input serial data.
RI      var    IN0     ' Ring-indication output of modem.

waitForRing:                    ' When phone rings, RI goes high.
  if RI = 0 then waitForRing    ' Wait here while RI is low.

pickUpPhone:
  serout TxD,N2400,["ATA",cr]   ' Tell modem to pick up.
  serin RxD,N2400,tLink,Disconnect,[wait ("10")]         ' Wait for link.
  pause 10000                   ' 10-second pause for modem stuff.
  serout TxD,N2400,["WELCOME TO BS2 BBS!",cr,lf,"logon: "]
  serin RxD,N2400,[wait ("USER")]        ' Wait for password.
  serout TxD,N2400,100,[cr,lf,"Simulated data here...",cr,lf]
  pause 1000
  serout TxD,N2400,100,["Hanging up now.",cr,lf]

Disconnect:
  pause 2000
  serout TxD,N2400,["+++"]
  pause 2000
  serout TxD,N2400,["ATH0",cr]
goto waitForRing
```

Listing 2. BASIC for Beginners Race Timer with Display

```
' Program RACE2.BAS (Three-lane race timer with display)
' This program shows how the BS1 (or Counterfeit) can
' be used to time a three-lane Pinewood Derby race.
' It converts a raw count of program loops into
' units of 1/100th of a second and presents them on
' a serial LCD display.

SYMBOL time1 = w2       ' Word variable for lane-1 time.
SYMBOL time2 = w3       ' Word variable for lane-2 time.
SYMBOL time3 = w4       ' Word variable for lane-3 time.
SYMBOL start = pin7     ' Start-switch on pin 7; 0=start.
SYMBOL status1 = bit0   ' Status of lane 1; 1=racing, 0=done.
SYMBOL status2 = bit1   ' Status of lane 2; 1=racing, 0=done.
SYMBOL status3 = bit2   ' Status of lane 3; 1=racing, 0=done.
SYMBOL win = bit3       ' Flag to indicate race winner.
SYMBOL stats = b0       ' Byte variable containing status bits.
SYMBOL pos = b11        ' Printing location.
SYMBOL digits = b10     ' Digits to display.
SYMBOL timeDat = w1     ' Timing data to convert/display.
SYMBOL iPre = 254       ' Instruction prefix for LCD.
SYMBOL clrLCD = 1       ' Clear LCD screen.
SYMBOL blank = 8        ' Blank the LCD (but retain data).
SYMBOL restore = 12     ' Restore LCD.
SYMBOL topLft = 128     ' Move to top-left of LCD screen.
SYMBOL topRt = 136      ' Move to top-right of LCD screen.
SYMBOL btmLft = 192     ' Move to bottom left of LCD screen.
SYMBOL btmRt = 200      ' Move to bottom right of LCD screen.

begin:
  stats = %111                  ' All cars in the race to begin.
  time1=0:time2=0:time3=0       ' Clear timers.
  serout 3,n2400,(iPre,clrLCD)  ' Clear the display.
  pause 5
' The line below is sneaky--it prints "Race in Progress" to the
' LCD, then blanks the LCD so that the message is hidden. That
' way, the program can display the whole 16-byte message by just
' sending a 2-byte 'unblank display' instruction.
serout 3,n2400,("Race in Progress",iPre,blank)

hold:
  if start =1 then hold         ' Wait for start signal.
  serout 3,n2400,(iPre,restore) ' Restore "Race in Progress" to LCD.
```

```
timing:                          ' Time the race.
 stats = stats & pins & %111     ' Put lowest 3 pin states into stats.
 if stats = 0 then finish        ' If all cars done, then race over.
 time1 = time1 + status1         ' If a car is in race (status=1) then
 time2 = time2 + status2         '  increment its timer. If it's done
 time3 = time3 + status3         '  (status=0) don't increment.
goto timing                      ' Loop until race over.

finish:
  serout 3,n2400,(iPre,clrLCD)                    ' Clear display.
  pause 5
  serout 3,n2400,(iPre,btmRt,"-FINAL-")           ' Print FINAL.
  timeDat=time1: pos=topLft:gosub Display         ' Display race times.
  timeDat=time2: pos=topRt:gosub Display
  timeDat=time3: pos=btmLft:gosub Display

END     ' End program--reset to time another race.

' This subroutine converts the loop count in the variable timeDat
' into a number of hundredths of a second, then prints that value
' (as seconds, with decimal point) at the screen location specified
' by the variable pos.
Display:
  serout 3,n2400,(iPre,pos)      ' Move to display location.
  if timeDat > time1 OR timeDat > time2 OR timeDat > time3 then noWin
   serout 3,n2400,("*")          ' Put * by winner (or winners, if tie)
  goto skip1
noWin:
  serout 3,n2400,(" ")           ' Put space by non-winners.
skip1:
  timeDat = timeDat*8/27         ' Convert to 100ths of a second.
  digits = timeDat/100           ' Print hundreds place followed
  serout 3,n2400,(#digits,".")   ' ..by decimal point.
  digits = timeDat//100          ' Now print remainder.
  if digits > 9 then skip0       ' If remainder is less than 10,
  serout 3,n2400,("0")           ' ..print "0" (i.e., convert
skip0:                           ' "6" to "06" for correct display.
  serout 3,n2400,(#digits)
return                           ' Return to program.
```