Stamp Applications no. 19 (September '96):

Connect BS2 to Phone Lines, Give the BS1 an LED Display

Project double-header for the
BASIC Stamps I and II
by Scott Edwards

THIS MONTH we're going to blast our way through two frequently requested applications: a circuit for connecting the BS2 to the phone line to take advantage of the DTMFout instruction, and an all-in-one LED display for the BS1.

In BASIC for Beginners we'll discuss binary-coded decimal (BCD) numbers, which just happen to be useful for both DTMF and LED-display applications.

But first, a word from our sponsor regarding Internet access to this column and many of the goodies discussed here:

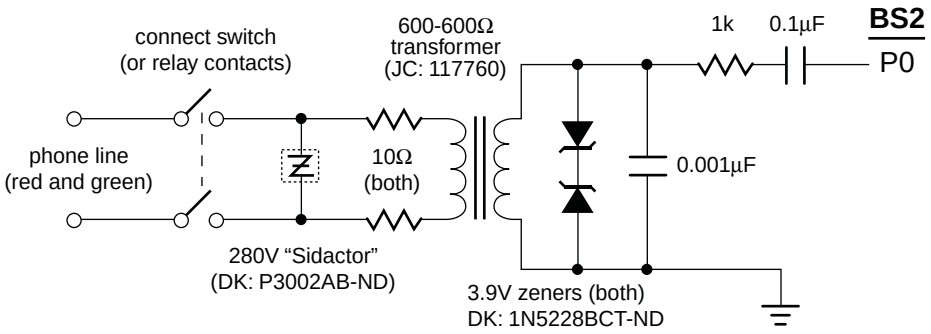
Apps on the Net. You can now retrieve past issues of *Stamp Applications* from the *Nuts & Volts* web site, www.nutsvolts.com. Click on the button for the library, and your browser will take you to ftp.nutsvolts.com and the directory /pub/nutsvolts/library. Go into the subdirectory /stampaps and you'll see a listing of Adobe Acrobat files for issues 1 through whatever of this column (this is issue 19).

If you're interested in some of the Stamp-related products sold by my business, Scott Edwards Electronics, the information is just a couple of levels up on the same ftp site. The path is /pub/nutsvolts/scott (of ftp.nutsvolts.com). Again, the files are in Acrobat format (now supported directly by many browsers) and include the current catalog, user's manuals to our most popular products, and samples of our AppKit documentation.

Now back to our regularly scheduled program:

Phone-line interface. The BS2 can generate telephone "touch" tones with its DTMFout instruction. However, you can't just hook the BS2 to the phone line and start dialing. There are several reasons:

- Audio from most sources, including the BS2, is *single-ended* (also known as ground-referenced). The signal voltage is the difference between a single wire and ground. Since phone lines travel far and wide, connecting locations with vastly different ground potentials and picking up noise along the way, they use *differential signaling*. A signal voltage is the voltage difference between the two wires that make up a phone-line pair.
- Phone systems use the presence of a load on the line to determine when to go off hook and present a dial tone to local phone or other equipment and a busy signal to others trying to dial in.
- The phone line carries more than just audio signals; a direct-current (dc) supply voltage, a high alternating-current (ac) ringing voltage, and dangerous voltage spikes all show up on the phone lines.
- Despite its rugged signals, the Federal Communications Commission (FCC) considers the phone system to be tender as a newborn babe. They insist that their baby be protected from voltages that might confuse or hurt it.



Parts Sources

Digi-Key (DK), 1-800-344-4539 or 218-681-6674	Jameco (JC), 1-800-831-4242 or 415-592-8097

Figure 1. Phoneline interface for the BS2.

For all those reasons, you need an interface to connect anything, including the BS2, to the phone line. If you're manufacturing a product, the FCC requires that you use an approved Data Access Arrangement (DAA). Because of the expense involved in getting regulatory approval for DAAs, they tend to be priced out of proportion to the cost of the components involved—$25+ in small quantities. DAAs are also generally not available from the usual electronics parts outlets.

If you just want to experiment with the BS2's DTMF capabilities, a full-blown DAA is probably overkill anyway. Using a simple circuit I found in *Encyclopedia of Electronic Circuits, Volume 5*, by Graf and Sheets (TAB/McGraw Hill, 1995; ISBN 0-07-011077-8), I had a BS2 up and dialing in just a few minutes. My adaptation of the circuit appears in figure 1, complete with part numbers and sources.

The circuit meets all the basic requirements for a phone-line interface. The transformer converts single-ended audio from the BS2 into differential signals. The Sidactor and back-to-back zeners on either side of the transformer clamp voltage transients to 280V on the phone side, and approximately 4.5V on the Stamp side.

Once you're hooked up, dialing the phone is simple. The DTMFout instruction just needs to know the BS2 pin number to use and a list of digits to dial. For example, to dial 555-1234 through pin 0:

```
DTMFOUT 0,[5,5,5,1,2,3,4]
```

DTMFout sends each tone for 200 milliseconds (ms) followed by 50 ms of silence. This gives the phone-company equipment plenty of time to detect and respond to the tones. However, you can also customize the durations of the tones and silences as follows:

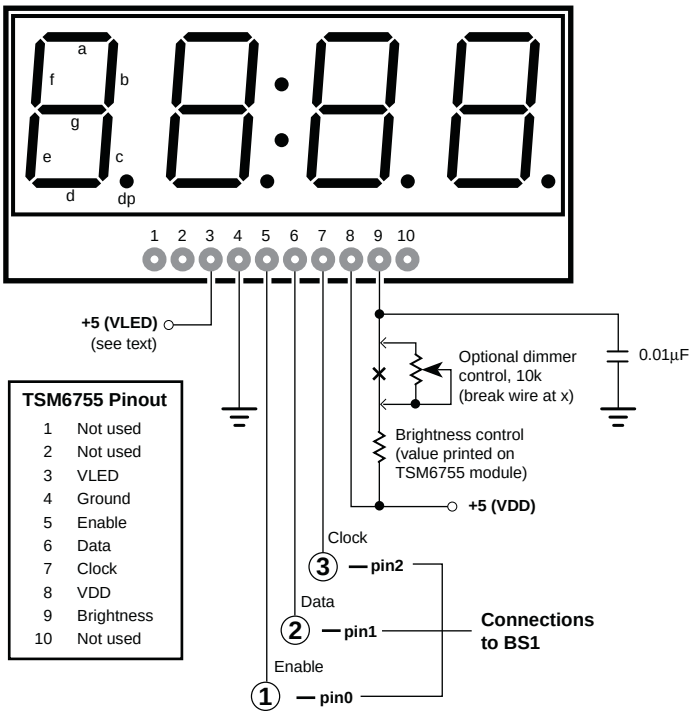
```
DTMFOUT 0,<tone>,<quiet>,[<list>]
```

The tone and quiet times are in ms. So, to dial 555-1234 with 1-second tones and 0.5-second silences, you'd program:

```
DTMFOUT 0,1000,500,[5,5,5,1,2,3,4]
```

The obvious application for DTMFout is to build a super-deluxe telephone dialer. But don't forget all of the other services you can now control from the telephone keypad, like pagers, voice mail, menu routers, automated information systems, etc. For example, you could have the BS2 dial your pager if it detected an intruder in your home, or a burst pipe in your basement, or a crashed computer at your office. If you frequently access your bank's account-information system, you could automate the menu-walkthrough with the BS2.

Integrated LED display/driver. An earlier column on the MAX7219 LED display driver (no. 10, December '95; available from the N&V web site as ST_AP10.PDF) drew a lot of interest.



Note: The two digits on the lefthand side of the display have additional vertical segments to form a + sign in the center of the digits. However, in the TSM6x55 modules, these segments are not fitted with LEDs, and will not light.

Figure 2. Connecting the TSM6755 LED display module.

NOTE:

The TSM6755 LED display module described in this article is no longer available (as of Summer 2000).

The manufacturer, Three-Five Systems, seems to have reorganized their product line around small, high-resolution displays.

There is no direct substitute that I know of.

The MAX chip makes it easy to talk to seven-segment LED displays up to eight digits long. Unfortunately, using the chip requires quite a bit of wiring to get all those LEDs connected.

Some readers asked for an integrated version of the MAX, incorporating LEDs and driver into a single module. I searched, and found the Three-Five Systems TSM6755. Its display drivers aren't as fancy as the MAX7219—you have to tell them specifically which LED segments to turn on—but the additional programming effort is more than balanced by the simplicity of the hardware. Figure 2 shows the complete hookup.

The TSM6755 module is available from my order desk as an AppKit with code on disk for the BS1, BS2 and PIC microcontrollers (Sources) and full manufacturer's docs. The module is available by itself from Farnell Components (1-800-718-1997).

Although the TSM6755 is easy to use, it shares one drawback with other LED displays: it draws a lot of current. With all 34 LEDs on at 10 milliamperes (mA) each, the display can draw 340 mA. That's more than the 50-mA rating of the BS1's voltage regulator, or the 100-mA rating of the Counterfeit (BS1 kit; Sources).

There are at least three solutions to this problem: (1) Use a separate 5-volt power supply with a rating of at least 500 mA. These show up in the electronics catalogs from time to time at bargain prices. (2) Use the circuit in figure 3 to convert any 7- to 12-volt dc supply to regulated 5 volts for VLED. (3) If you're using a Counterfeit with its 100-mA regulator, you can test the module at reduced brightness. Just use a brightness-control resistor of at least triple the value printed on the TSM6755 module.

One the software side of things, the program listing that accompanies this column tells the tale. The subroutine IIIV_write converts a 16-bit value between 0 and 9999 into decimal digits; converts the digits into patterns of LEDs corresponding to the individual numerals; and sends these patterns to the TSM6755 by a clocked (synchronous) serial method. The

subroutine allows the program to specify the location of a decimal point, and to determine whether or not to blank leading zeros. Leading zeros are the implied zeros to the left of the first non-zero digit of a fixed-length display. Huh? OK, suppose you want to display “123” on a four-digit display. Do you want to see “123” (leading zeros blanked) or “0123” (leading zeros allowed)?

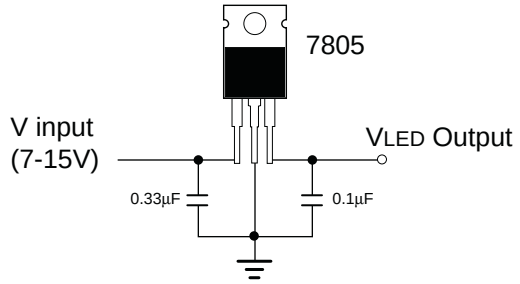


Figure 3. Voltage regulator for VLED.

Apart from the leading zeros, extracting the digits of a number is straightforward. Just take the remainder of the division by 10 (value // 10 in PBASIC) to get the ones-place digit. Then divide the value by 10 and do it again. I point this out because digit-extraction is a common requirement for driving displays. It's important enough that Parallax has added a digit-extraction function called DIG to the BS2.

BASIC for Beginners. In past columns we discussed several common numbering systems; decimal, binary, and hexadecimal. If you recall, hexadecimal (or *hex* for short) is the preferred shorthand for binary, since each digit neatly expresses a nibble (four bits). Hex digits range from 0 through F, representing decimal values of 0 through 15.

There's another important application for hex numbers: providing a reasonable way for a computer to store and manipulate decimal digits. Hex numbers used this way are referred to as *binary-coded decimal* (BCD).

The idea is this; instead of using hex numbers' full range of values (0—15), restrict them to the values that overlap with the decimal system (0—9). Each hex, excuse me *BCD*, digit fits neatly into four bits. When it's time to display those digits, no conversion is required. Just put the value of each BCD nibble on the corresponding digit of the display.

Contrast that to the approach required for converting pure binary values into decimal digits. The ones digits is extracted by taking the remainder of division by ten. The binary value is then divided by 10 and the ones digit extracted again until all digits are done.

We take built-in division capabilities for granted with the Stamps, but division is difficult for limited micros. If division can be avoided, the resulting program will be smaller and faster than one that uses division.

How do you decide when BCD might be the better way to go? This depends on your application, and which Stamp you're using. The BS1 does not have nibble variables, so its ability to deal with BCD is limited.

The BS2 does have nibble (nib) variables, and the ability to define arrays—variables with multiple cells that can be addressed by providing an index value. It has hex format modifiers for debug and serout instructions that also work for displaying BCD digits.

Listing 2 is a BS2 program that demonstrates some basics of BCD by implementing a BCD counter that counts from 0 to 999,999 on the debug screen. That points out yet another handy use of BCD; to construct counters that exceed the limits of built-in word (16-bit) variables.

Listing 1: Using the TSM6755 with BS1

```
' Program: IIIV_LED.BAS
' This program controls TSM6x55 LED displays. It demonstrates the
' basics of communicating with the TSM6x55s by displaying the
' value of a counter on the LEDs.

' Hardware interface with the TSM6x55:
SYMBOL DATA_n = 1      ' Bits are shifted out this pin # to display.
SYMBOL DATA_p = pin1   ' "    "   "       "        ".
SYMBOL CLK    = 2      ' Data valid on rising edge of this clock pin.
SYMBOL Enable = 0      ' Activates TSM6x55 to accept data.

' Variables used in the program.
SYMBOL initLED = bit0  ' Flag to trigger initialization of display.
SYMBOL colon   = bit1  ' Flag to turn on colon in the middle of display.
SYMBOL bitPat  = b1    ' Byte to be sent to the display.
SYMBOL clocks  = b2    ' Bit counter used to clock out bits.
SYMBOL digit   = b3    ' Loop counter for clocking out bytes.
SYMBOL DP      = b4    ' Position of decimal point.
SYMBOL dispVal = w3    ' Value to be displayed on the LEDs.
SYMBOL counter = w4    ' Counter for demo.

' The program begins by clearing all pins except the TSM6x55 enable
' to 0, and setting their I/O direction to output. It then writes 0s
' to the TSM6x55's internal registers to clear out any garbage that
' may be left from a previous write to the display. If this were not
' done, any 1s left in the display registers would cause the display
' to load newly arrived data before it should, garbling the display.
let port = $FF01       ' Dirs = $FF (all outputs) and Pins = 1 (low).
let initLED = 1        ' Set up to initialize the display.
gosub IIIV_write       ' Clear display registers.
let initLED = 0        ' Switch to normal operation.
let colon = 0          ' Turn off the colon.

' The decimal-point variable serves three functions: (1) If it's
' in the range of 0 to 3, it turns on the decimal point of the
' corresponding digit (numbered right to left). (2) If it's
' in the range of 1 to 3, it also turns off leading-zero blanking
' of the display, allowing numbers like "0.123" to be displayed.
' (3) If DP is 0 or greater than 3, leading-zero blanking is on,
' so that numbers like "26" don't display as "0026". When DP is 0,
' the rightmost decimal point is on _with_ leading-zero blanking,
' so that single-digit numbers display like so: "4." For our demo,
' we'll turn off the decimal points and turn on leading-zero blanking.
```

```
let DP = 255           ' No decimal point; no leading zeros.

' ====================== MAIN PROGRAM LOOP ==========================
' Now that the display is properly initialized, we're ready to send it
' data. The loop below increments a 16-bit counter and displays the
' lower 4 digits on the TSM6x55. A subroutine takes care of converting
' the value into digits, suppressing leading zeros, and sending data
' to the display.
Loop:
  let dispVal = counter         ' Copy counter into dispVal.
  gosub IIIV_write              ' Write it to the display.
  let counter = counter+1       ' Increment the counter.
  pause 200                     ' Slow things down a bit.
goto loop                       ' Do it again.

' ==================== TSM 6x55 SUBROUTINE ==========================
' This routine converts the value in dispVal into a series of bit
' patterns that light up the appropriate LEDs on the display to
' show individual digits. It sends these bytes to the display via
' clocked (synchronous) output in which a bit is placed on the data
' line, then the clock line is pulsed to shift that bit into the display.
IIIV_write:
low enable             ' Activate the display.
high DATA_n            ' Send a start bit (1) to the display.
pulsout CLK,1  ' Clock out the start bit.
for digit = 0 to 3     ' Convert and send digits 0 - 3 (1s to 1000s).
  bitPat = dispVal//10 ' Get the ones digit of dispVal.
  if DP < 4 AND DP > 0 then noZblank     ' No 0 blanking if 0 < DP < 4.

' The line below blanks 0s in the leftmost positions of the display.
' A digit is defined as a leading zero by these three rules:
' (1) The digit must be a 0 (bitPat = 0), and
' (2) It must be the leftmost digit of the number (dispVal = 0), and
' (3) It must not be the only digit of the number (digit<>0).
' If a digit meets these three rules, then it is changed from 0 to blank.
  if bitPat = 0 AND dispVal = 0 AND digit <> 0 then blank
noZblank:      ' Look up LED pattern matching digit value.
  lookup bitPat,(111,40,93,124,58,118,119,44,127,62),bitPat
```

```
' To turn on the decimal point for a particular digit, its bit pattern
' must be ORed with 128 (%10000000). So the two lines below are meant
' to work as: "If DP = digit then bitPat = bitPat OR 128." The backwards
' logic is required because PBASIC can only go to a label as the
' outcome of if/then. Below are several more examples of negative
' logic to work around this limitation.
  if DP <> digit then skip0
  bitPat = bitPat | 128         ' Turn on bit 7 of LED (decimal point).
skip0:
  if initLED = 0 then skip1     ' "If initLED = 1 then bitPat = 0"
blank:
  bitPat = 0                    ' Send all zeros to display.
skip1:
  for clocks = 1 to 8           ' Send eight bits.
    let DATA_p = 0              ' If msb of bitPat = 1, then let
    IF bitPat < $80 then skip2  '..DATA_p = 1, else DATA_p = 0.
    let DATA_p = 1
skip2:
    pulsout CLK,1               ' Pulse the clock line.
    let bitPat = bitPat * 2     ' Shift bitPat one bit to the left.
  next clocks                   ' Continue for eight bits.
  dispVal = dispVal/10          ' Divide dispVal by 10 to get next digit.
next digit
DATA_p = colon          ' Now put the colon bit on the data line.
pulsout CLK,1           ' And clock it out to the display.
pulsout CLK,1           ' This takes 3 clocks; 2 for the LEDs of
pulsout CLK,1           ' the colon, plus 1 to load the display.
high enable             ' Disable the TSM6x55.
return                  ' Return to program.
```

Listing 2: Demonstrating BCD Counting with the BS2

```
' Program: BCD_DEMO.BS2
' This program implements a counter to demonstrate the basics
' of working with binary-coded decimal (BCD) numbers with the BS2.

' Variables used in the program.
BCDcnt  var    nib(6)  ' Six-digit BCD counter.
i       var    nib     ' Index into array of BCD digits.
nonZ    var    bit     ' Flag used in blanking leading zeros.

' You can enter a starting value for the counter by changing the
' digits stored in BCDcnt below.
BCDcnt(5) = 0: BCDcnt(4) = 0: BCDcnt(3) = 0
BCDcnt(2) = 0: BCDcnt(1) = 0: BCDcnt(0) = 0

' ====================================================================
'                  Main Loop of Demonstration Program
' ====================================================================
' All of the real action takes place in the subroutines--all this
' main program loop does is to display the BCD number, increment
' it, and repeat. Forever.
Main:
  gosub BCD_display    ' Show the BCD number.
  gosub BCD_inc        ' Add 1 to the BCD number.
goto Main              ' Do it again.

' ====================================================================
'                      BCD Subroutines
' ====================================================================

' =====BCD_display
' This routine displays the six BCD digits stored in BCDcnt() on
' the screen using the debug instruction. It incorporates logic
' to eliminate leading zeros (i.e., to display values like
' "001234" as "1234"). A zero is defined as a leading zero if
' it is to the left of the first non-zero digit. We also
' have to consider the case in which _all_ digits are zero;
' here we don't want to blank the zero in the ones place.
' In the routine, a flag (nonZ) is set to 1 by the first
' non-zero digit. The if/then instruction right below the
' label "skip1" takes care of the rest. It says, "if this
' digit is 0, AND the first non-zero digit hasn't come yet,
' AND this isn't the ones place, then don't print this digit."
```

```
BCD_display:
  nonZ = 0                      ' Turn on leading-zero blanking.
  for i = 5 to 0                ' For each digit..
    if BCDcnt(i)= 0 then skip1  ' In effect: "if digit is not 0..
    nonZ = 1                    ' ..then nonZ = 1"
  skip1:                        ' See comments above for next line.
    if BCDcnt(i) = 0 and nonZ = 0 and i <> 0 then skip2
    debug hex1 BCDcnt(i)        ' Place the digit on the screen.
  skip2:
  next                          ' Process all 6 digits.
  debug cr                      ' Send a carriage return.
return                          ' Return to program.

' =====BCD_inc
' This subroutine increments (adds 1 to) the 6-digit BCD number stored
' in BCDcnt. It works like grade school arithmetic: add 1 to the ones
' digit. If the result is 10, put zero into the ones digit and carry
' the 1 to the 10s digit. The routine uses a loop to carry the 1 all
' way up to the 6th digit when the number to be incremented is
' 999999.
BCD_inc
  i = 0                         ' Start at lowest digit.
  incLoop:
    if i > 5 then done          ' Stay within the 6 digits, 0-5.
    BCDcnt(i) = BCDcnt(i) + 1   ' Increment digit.
    if BCDcnt(i)< 10 then done  ' If no carry-the-one, we're done.
    BCDcnt(i) = 0               ' Otherwise, zero this digit..
    i = i + 1                   ' ..and increment the next digit.
  goto incLoop                  ' Keep going until out of digits..
done:                           ' ..or no more carries.
return                          ' Return to program.
```