Stamp Applications no. 15 (May ’96):

Understanding the RCtime Instruction and Creating Strings in EEPROM

BS2 Programming Hints
by Scott Edwards

TWO BS2-RELATED TOPICS keep coming up lately; how to use the RCtime instruction, and how to store and manipulate text strings.

So this month I’ll explain the electronic underpinnings of RCtime; show a couple of methods of storing and retrieving EEPROM strings; and continue the BASIC-for-beginners series with systems of numbers.

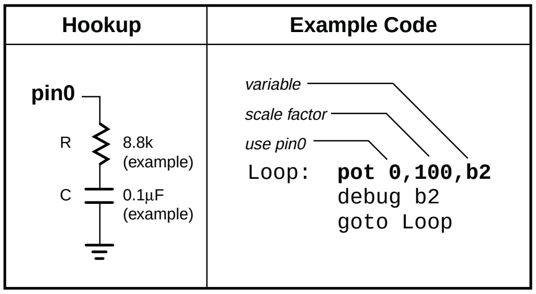
RCtime. It’s great that the Stamps have attracted so much interest from people with little background in electronics or programming. And it’s remarkable how much non-techies can accomplish by following recipes like the ones I present here.

But sometimes I think that these folks are shortchanging themselves by failing to learn the fundamentals. A case in point is the widespread misunderstanding of the new RCtime instruction—most everything you could want to know about RCtime is taught in the first five weeks of Electronics 101.

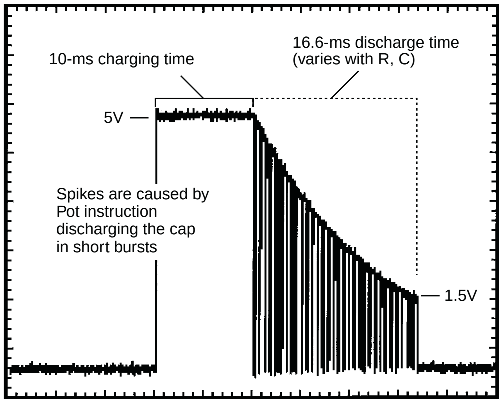
To prove it, I hauled my old textbook and applied it to RCtime with gratifying results.

Let me start with a little background: RCtime is the BS2’s counterpart to the BS1’s Pot instruction. The purpose of both instructions is to measure a resistance by seeing how long it takes to charge or discharge a capacitor through that resistance. That’s where the similarity ends.

Pot takes an active approach to this measurement. It charges the cap to +5V, then enters a loop, discharging the cap a little at a time with each loop. It does this by placing the pin into an output-low state briefly, then back into input mode to see whether the capacitor is still charged.



```
Loop:  pot 0,100,b2
       debug b2
       goto Loop
```



Figures 1 and 2. Pot hookup diagram/example code (left) and storage-oscilloscope trace (right).

Pot counts the number of loops required for the charge on the capacitor to change from 1 (> 1.5V) to 0 (< 1.5 V).

Pot's loop count can run as high as 65,535 (max value of a 16-bit number), but is scaled to fit within an 8-bit variable (0 to 255) using a factor supplied by your program. The BS1 host program STAMP.EXE includes a routine for determining this scale factor, in essence calibrating the Pot command for variations in capacitor value and resistance range.

Figure 1 shows a standard connection for the Pot instruction, while figure 2 is a storage-oscilloscope record of Pot in action. The downward spikes along the discharge curve are the brief output-low states of the Pot pin.

RCtime works in a different way, inside and out. Figure 3 shows a typical hookup. RCtime relies on your program to set the capacitor's initial state; in the case of figure 3, you would briefly output a high on the RCtime pin. It's confusing, but this would *discharge* the capacitor by placing both of its leads at the same potential, +5V.

When the RCtime instruction executes, it puts the specified pin into input mode and merely waits for the capacitor to charge through the pot or unknown resistance. This charging action is seen as the voltage at the lower, I/O-pin end of the cap getting closer to ground. All the while, RCtime is counting up at 2-microsecond (µs) intervals. When the voltage at the lower pin of the cap falls below 1.5 V, the input seen by the RCtime pin changes from a 1 to a 0 and the counting stops. RCtime returns the 16-bit count in the variable specified in the instruction.

Figure 4 is a storage-scope trace of RCtime in action. Remember, although the trace looks like a capacitor-discharge curve because the measured voltage is dropping over time, the capacitor is actually charging. We're just measuring it from underneath, so to speak.

RCtime leaves a lot of design details to the user—the most important being: for a given combination of resistor and capacitor, what value will the instruction return? That's where Electronics 101 comes in.

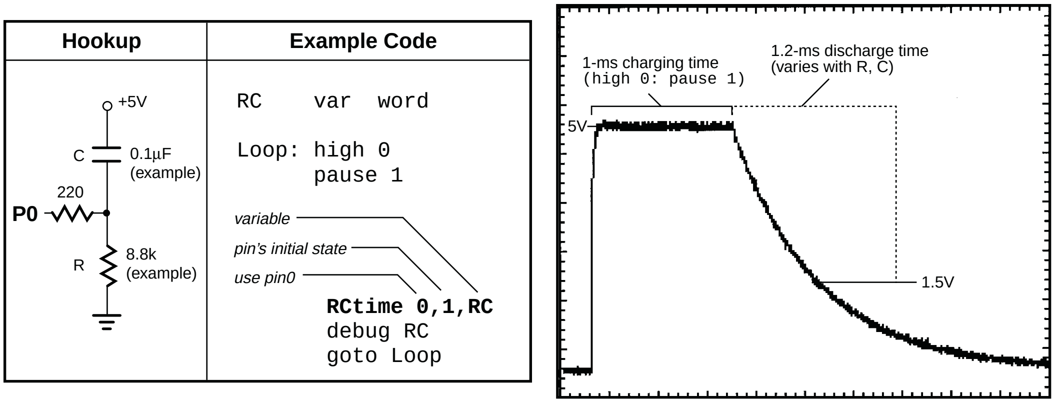
When a capacitor charges or discharges through a resistor, it follows an exponential curve, like the 'scope pictures of figures 2 and 4. Initially, the cap charges or discharges quickly, then tapers off. Although curves like this are usually accompanied by some awful math, there's a simple formula for determining how long a given resistor/capacitor combination will take to charge or discharge to the 63-percent point. This value is called the *time constant*, and it's often symbolized by the Greek letter tau (τ). The formula is:

$$\tau = R \times C$$

For example, the time constant for a 10k resistor and a 0.1µF capacitor:

$$\tau = (10 \text{ x } 10^{3}) \text{ x } (0.1 \text{ x } 10^{-6}) = 1 \text{ x } 10^{-3}$$

If you were charging a capacitor from 0V to 5V it would reach 63 percent of 5V (3.15V) in 1 millisecond. If you were discharging that same RC combination from 5V to 0V, it would hit 63-percent discharged (or 37-percent charged; 1.85V) in 1 millisecond.



```
RC     var  word

Loop:  high 0
       pause 1

       RCtime 0,1,RC
       debug RC
       goto Loop
```

Figures 3 and 4. RCtime hookup diagram/example code (left) and storage-oscilloscope trace (right).

For the RCtime instruction, what we really want to know is how long it will take for the capacitor voltage to go from 5V to 1.5V. See, 1.5V is the *logic threshold*, the point below which a BS2 pin reads 0 and above which it reads 1. It is therefore the trip point at which the RCtime instruction stops counting. Calculating how long it will take for a given RC combination to reach this point is just another textbook formula:

$$\text{time} = -\tau\left[\ln\left(\frac{V_{\text{final}}}{V_{\text{initial}}}\right)\right]$$

In the formula above, the symbol *ln* means *natural logarithm*; it's a key found on most scientific calculators. Let's see how long it would take for our 10k resistor and 0.1µF capacitor to go from 5V to the logic threshold of 1.5V:

$$\text{time} = -1\times 10^{-3}\left[\ln\left(\frac{1.5}{5.0}\right)\right] = 1.204\times 10^{-3}$$

Αβουτ 1.2 μιλλισεχονδσ αφτερ ΡΧτιμε βεγινσ, τηε ϖολταγε ατ τηε λοωερ λεγ οφ τηε χαπαχιτορ ωιλλ ηιτ 1.5ς ανδ ΡΧτιμε ωιλλ ρετυρν ωιτη α χουντ οφ 2 µs units in its variable. Since 1.204 milliseconds is 1204µs, that count would be 602 in 2µs units.

Since RCtime always goes from 5V to 1.5V (when configured as shown in the manual), and always works in 2µs units, we can derive an easy-to-remember version of the formula that can tell us how many units RCtime should return for any given combination of R and C:

$$\text{RCtime units} = 600 \times 10^3 \times R \times C$$

OR

$$\text{RCtime units} = 600 \times R\ (\text{in k}\Omega) \times C\ (\text{in }\mu F)$$

A few final notes about Pot and RCtime are in order. First, you may be wondering why Parallax changed this function so radically. It has to do with timing. The BS1's internal clock runs at 4 MHz, and it measures time in units of 10µs. If Pot worked like RCtime, the BS1 could only count to 120 in the 1.2 ms it takes for a 0.1µF cap to discharge through a 10k resistor. By discharging the cap in small sips, Pot extends the discharge time, allowing it to count much higher.

This also means that Pot takes much longer to execute for a given combination of R and C. The measurements shown in figures 2 and 4 used the same resistor (a pot set to 8.8k) and capacitor (0.1µF, 5-percent tolerance). The BS1 took 26.8 ms overall to make the measurement; the BS2, just 2.54 ms. Both of those figures include pre-measurement charging time.

Another difference between the instructions is that RCtime is fairly linear in its response, while Pot is not. For example, if RCtime returns a count of 100 for a 2k resistance, you can predict that it will return approximately 200 for 4k. Its response becomes less linear with very short time constants due to the increased significance of the time required for the instruction to begin its measurement loop.

The BS1's Pot instruction is slightly nonlinear. Check out application note no. 7 in either the Parallax BS1 series or in the Counterfeit Development System manual. A graph of Pot readings versus resistance shows a noticeable curve.

This difference in response is why you can't just adapt the program from that BS1 app note to work with the BS2. You have to go back to the beginning of the process described in the app note and generate a new power-series equation to match the combined characteristics of the BS2 and thermistor.

In figure 3 you may have noticed that I used a larger value resistance in series with the I/O pin than shown in the Parallax manual. They say 10 ohms; I use 220. The purpose of this resistor is to protect the BS2 I/O pin from being damaged when the value of R is near 0 ohms and the pin is output high. This amounts to a short circuit from +5V to ground through the I/O pin.

A 10-ohm resistor will limit this short-circuit current to 500 mA. (The current would actually be less, due to the on-resistance of the I/O pin itself, but you get the idea.) That current is 25 times the rated 20-mA maximum for a Stamp I/O pin! Since the time that the pin is output high is supposed to be brief (about 1 ms), the pin should not be harmed. The advantage of using such a small resistance is that it ensures that the capacitor will be fully charged very quickly.

My logic in using a 220-ohm resistor, which would hold the short-circuit current to just 23 mA, is that it's foolhardy to rely on a program being completely free of bugs in order to prevent damage to an expensive component like the BS2-IC. How does this choice affect the charging time of the capacitor? Electronics 101 says that a capacitor is 98 percent charged in four time constants: 4 x R x C. With a 220-ohm resistor and a 0.1µF capacitor, that's:

$$\text{Charge time} = 4 \times 220 \times (0.1 \times 10^{-6}) = 22 \times 10^{-6}$$

The capacitor is almost completely charged in 22µs—far, far less than the 1 ms allotted by the program.

Finally, keep in mind that RCtime is actually more of a general-purpose timing instruction than its name implies. All it really does is change a selected pin to input and count the number of 2µs intervals while that pin remains in a specified state (1 or 0). In this way, it's much like the PBASIC instruction Pulsin, except that it doesn't wait for an initial edge to start counting.

This can be very handy. Take BS1 application note no. 12, Sonar Rangefinding (also found in the Counterfeit manual). That application uses Pulsin to measure the time required for an ultrasonic pulse to travel out to some reflective object and back in order to measure the distance. To give Pulsin a complete pulse to measure, the application requires that the ultrasonic receiver be positioned so that it can hear the ultrasonic transmitter. Using RCtime would make this trick unnecessary.

Strings in EEPROM. The BS2's Serout and Debug instructions include all kinds of options, including one for outputting two different styles of strings from RAM. A string is a sequence of bytes—frequently a snippet of text.

When I saw these RAM-string options in the BS2 manual, I assumed that there were complementary ones for EEPROM strings. After all, the manual makes a big show out of the simple and elegant way you can use the DATA statement to stuff all kinds of goodies, including strings, into EEPROM.

Unfortunately, there isn't a correspondingly elegant way to get those strings back *out.*

We'll fix that. Listings 1 and 2 present a couple of BASIC subroutines that make up for the missing EEPROM string options, and show the difference between the ways that BASIC and C store strings.

In BASIC, a string begins with a length byte containing a count of the number of characters (bytes) in the string. This lets BASIC's LEN() function quickly determine string length—all it does is read that first byte.

In C, a string begins with the first byte of the string itself, and ends with a byte containing 0, also known as an ASCII null. As you'll see from listing 2, this makes the code to read this kind of string faster and saves one variable used as a counter.

Both approaches can accommodate an empty string, and both would express it in the same way: a single byte containing 0.

Writing BASIC code to read out these strings is an imperfect solution. Both the BASIC and C string routines take 2 ms or more to fetch each byte of a string. This means that no matter how fast a serial data rate you use, you'll never get more *throughput* (actual number of bytes/second sent) than a 4800-baud data stream.

If this turned out to be a real problem in an application, and if all of the strings were short, you could transfer them from EEPROM to RAM, then send them. It would take the same amount of time overall, but reduce the time that the BS2 tied up the serial link.

I've lobbied Parallax to add EEPROM-string capabilities to future revisions of the PBASIC-2 firmware.

BASIC for Beginners. For several columns I've been talking about Boolean logic and its applications in IF/THEN decisions and bit manipulations. Throughout this discussion, I have deliberately skirted the issue of numbering systems.

Our experiences with the logic operators demonstrated why it's sometimes necessary to write numbers out in a way that lets you examine individual bits: It's the only way to quickly grasp the effects of the Boolean operators.

Without further ado, let's look at the

numbering systems supported by PBASIC.

Decimal. This is our everyday numbering system, also known as “base 10.” Although it hardly needs further explanation, examining the familiar decimal system in detail will give us a better handle on the other systems.

The decimal system has 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Therefore, a single decimal digit can represent a range of 10 values, 0 through 9, depending on which of the symbols it holds.

To represent numbers larger than 9 in decimal, we tack on additional digits to the left. For example, when counting upward, we say, “0, 1, 2, 3....9, 10, 11...” It’s probably been a long time since you gave that sequence any thought, but think about it now: When you count past the largest number that you can represent in a digit (9), you reset that digit to 0, and increase the next digit to the left by 1. You then continue counting, as before, in the lowest digit, until it once again exceeds the largest number symbol.

Each digit in the decimal system expresses a number multiplied by a power of 10. Yup, even the ones place is really a multiple of a power of 10—10 to the zero power (10^0). Remember that *any* number raised to the power of 0 is equal to 1. This fact will be important later.

Here’s a quick analysis of a three-digit decimal number, 172:

Digits:	1	7	2
Value:	1×10^2	7×10^1	2×10^0

A quick summary of the decimal system:

- The root of decimal, *deci-* , means 10.
- The decimal system has 10 symbols, capable of representing values from 0 to 9.
- Digits of decimal numbers have values based on their positions in the number. Each digit represents a value 0 through 9, multiplied by a power of 10.

Binary. We can figure out a lot about the binary system (aka “base 2”) by making a few quick changes to the summary of the decimal system above:

- The root of binary, *bi-* , means 2.
- The binary system has 2 symbols, 0 and 1.
- Digits of binary numbers have values based on their positions in the number. Each digit represents a value of 0 or 1, multiplied by a power of 2.

Binary digits are called *bits*, derived from *b*inary dig*it*.

Binary is the number system of digital circuits and computers. The reason is the ease with which electronic circuits can produce, sense and store two discrete voltages representing 0 and 1. Normally a 0 is a low voltage, close to 0V, and a 1 by a high voltage, close to the system power supply (often 5V as in the Stamps).

Counting in binary works just like decimal, but new digits pile up faster. For example, a four-digit decimal number can represent values to 9999, while a four-digit binary number maxes out at binary 1111, the equivalent of decimal 15. Here’s how you would count from 0 to 1111 in binary. Note that it follows the counting rules we reviewed in decimal above.

binary	decimal	binary	decimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

When writing binary numbers, it’s necessary to label them as binary. Otherwise, they can be confused with decimal numbers by people or programs. In the example above, 1111 could be mistaken for “one thousand, one hundred and eleven” if we didn’t label it in some way. PBASIC uses a percent sign in front of a binary number: %1111 is binary; 1111 is decimal.

Because binary is so often associated with computers, we often refer to standard-sized groups of bits corresponding to the physical or operational capacity of the circuit or computer. The best-known of these is the *byte*—a group of eight bits. PBASIC runs on a controller that processes data in bytes, but through clever programming it also allows you to manipulate 16-bit words, or individual bits. PBASIC2 (the BS2 flavor) also supports processing of *nybbles*, the play-on-words name for groups of four bits, and the basis of the *hexadecimal* numbering system that we'll look at next. Before we do, let's analyze a typical binary number, %1011:

Digits:	1	0	1	1
Value:	1 x 2^3	0 x 2^2	1 x 2^1	1 x 2^0

In case those powers of 2 are not on the tip of your tongue, 2^0 is 1 (as any number raised to the zero power); 2^1 is 2; 2^2 is 4; and 2^3 is 8.

Hexadecimal. Our next numbering system, which is also known as *hex* or "base 16," serves as a kind of shorthand for representing binary numbers. In summary:

- The roots of hexadecimal: *hex*- means 6, and *deci*- means 10; combined, they mean 16.
- The hexadecimal system has 16 symbols, consisting of 0 through 9 (as in decimal) and A, B, C, D, E and F, representing the decimal values 10 through 15.
- Digits of hex numbers have values based on their positions in the number. Each digit represents a value of 0 through 15, multiplied by a power of 16.

Hex digits correspond to four bits, so they may be referred to as nybbles, or simply as hex digits.

Hex is handy for representing binary numbers in a compact way, since each digit corresponds neatly to exactly four bits. If you can memorize the binary equivalents of the hex counting sequence from 0 to F, you can convert any hex number to binary by merely substituting the corresponding four bits for each hex digit. Here are the first 18 hex numbers:

hex	decimal	hex	decimal
0	0	9	9
1	1	A	10
2	2	B	11
3	3	C	12
4	4	D	13
5	5	E	14
6	6	F	15
7	7	10	16
8	8	11	17

Since hex numbers can look like decimal ones, PBASIC uses the dollar-sign symbol ($) to label them. The hex equivalent of decimal 30 would be $1E in PBASIC.

Here's an analysis of a three-digit hex number, $A3F:

Digits:	A	3	F
Value:	10 x 16^2	3 x 16^1	15 x 16^0

To work out the decimal equivalent, just do the math and add the answers. I'll help by supplying the powers of 16: 16^0 = 1; 16^1 = 16; and 16^2 = 256. (Answer: 2623)

Wrapup. You'll encounter those three numbering systems often in reading PBASIC code here and elsewhere, so it pays to get comfortable with them. If you don't already have one, consider getting a calculator that has computer-math conversions built in, like the popular and inexpensive HP20S.

Next time, we'll take a look at how bytes are used to represent text in the ASCII (*asskey*) system of symbols. If you're planning to use PBASIC's serial communication capabilities, an understanding of ASCII is a must.

NOTE: This article was originally published in 1996. 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. BASIC-Style Strings for the BS2

```
' Program: BSTRING.BS2 (Demo of BASIC-style counted strings in EEPROM)
' This program shows how to use the DATA statement to store counted
' strings in EEPROM, and how to retrieve these strings for serial
' transmission.

' Define constants for serial communication with the PC through
' the built-in serial connector at 9600 baud. Note that this is
' compatible with the debug window--if the window is on the
' screen, sending data to the built-in connector at 9600 baud
' will display it. The debug window doesn't even care whether the
' serial data is inverted or not! This provides a handy check of
' serial comms without leaving the STAMP2.EXE host software.

builtIn con    16      ' Pin number for built-in serial connector.
baud    con    84      ' Baudmode constant for 9600, non-inverted.

' BASIC strings begin with a byte value (0 to 255) that specifies the
' number of characters in the string, followed by the characters
' themselves. In PBASIC-2, this translates to DATA statements that
' look like this:

'   ADDRESS            LENGTH   STRING DATA
'  CONSTANT            (BYTES)  (ASCII CHARACTERS)
'  ------------------------------------------------------------
   title       DATA    19,    ">>> HELLO DOLLY <<<"
   phrase1     DATA    12,    "Hello, Dolly"
   phrase2     DATA    17,    "Well hello, Dolly"
   phrase3     DATA    29,    "It's so nice to have you back"
   phrase4     DATA    16,    "where you belong"

' The subroutine that gets BASIC-style strings requires a variable to
' count out the bytes of the string as it fetches them. Since we're
' using a single byte to specify length, strings can't be longer than
' 256 characters. We're using a word variable for strCnt, since it
' will also be used to specify the ending address of the string,
' which can range from 0 to 2047.

strCnt  var    word    ' Counter for stringOut.
char    var    byte    ' Character (byte) to send via Serout.
strAddr var    word    ' Base address of the string.
i       var    nib     ' Small (0-15) counter for part 2 of demo.
```

```
' Our demonstration program will retrieve the strings one at a time
' and print them to the debug screen on the PC. A subroutine handles the
' details of getting the bytes and feeding them to serout.
demo:
  debug cls            ' Open a clear debug window.
  strAddr = title      ' Specify which string
  gosub stringOut      '   and display it in debug window.

  pause 2000           ' Take a 2-second intermission.

' Now we'll run through the strings in order, courtesy of a lookup
' table containing their addresses.
  for i = 0 to 3                ' For each of 4 phrases.
    serout builtIn,baud,[cls]   ' Clear the screen.
    lookup i,[phrase1,phrase2,phrase3,phrase4],strAddr  ' Get a phrase.
    gosub stringOut             ' Send to screen.
    pause 1000                  ' Time to read phrase.
  next                          ' Next phrase until done.

STOP                            ' End of program.

' =================================================================
' Here's the subroutine that reads strings. To use it, place the
' address of the string, assigned by its DATA statement, into
' the variable strAddr.  Note that the first line of this routine,
' "read strAddr,strCnt", is equivalent to the BASIC LEN function,
' which returns the length of a string. This is the primary advantage
' of BASIC strings--the ease with which you can retrieve length info.
stringOut:
  read strAddr,strCnt           ' Get the number of bytes in the string.
  strCnt = strCnt + strAddr     ' Set the endpoint of the string.
  for strAddr=strAddr to strCnt ' For each byte of the string:
    read strAddr,char           '   put the byte into char
    serout builtIn,baud,[char]  '   and send it out the port.
  next                          ' Until all bytes are sent.
return
```

Listing 2. C-Style Strings for the BS2

```
' Program: CSTRING.BS2 (C-style null-terminated strings in EEPROM)
' This program shows how to use the DATA statement to store C-style
' strings in EEPROM, and how to retrieve these strings for serial
' transmission.

' Define constants for serial communication with the PC through
' the built-in serial connector at 9600 baud.

builtIn con    16      ' Pin number for built-in serial connector.
baud    con    84      ' Baudmode constant for 9600, non-inverted.

' C strings end with an end-of string marker: a byte containing 0,
' also called a "null" or "ASCII null." You can store such strings
' in EEPROM like so:

'   ADDRESS             STRING DATA
'  CONSTANT             (ASCII CHARACTERS)             NULL
'  -----------------------------------------------------------
   title       DATA    ">>> HELLO DOLLY <<<",           0
   phrase1     DATA    "Hello, Dolly",                  0
   phrase2     DATA    "Well hello, Dolly",             0
   phrase3     DATA    "It's so nice to have you back", 0
   phrase4     DATA    "where you belong",              0

' The subroutine that gets C-style strings requires one less word
' variable than its BASIC counterpart.
char    var    byte    ' Character (byte) to send via Serout.
strAddr var    word    ' Base address of the string.
i       var    nib     ' Small (0-15) counter for part 2 of demo.

' Our demonstration program will retrieve the strings one at a time
' and print them to the debug screen on the PC. A subroutine handles the
' details of getting the bytes and feeding them to serout.
demo:
  debug cls            ' Open a clear debug window.
  strAddr = title      ' Specify which string
  gosub stringOut      '   and display it in debug window.

  pause 2000           ' Take a 2-second intermission.

' Now we'll run through the strings in order, courtesy of a lookup
' table containing their addresses.
  for i = 0 to 3                ' For each of 4 phrases.
    serout builtIn,baud,[cls]   ' Clear the screen.
    lookup i,[phrase1,phrase2,phrase3,phrase4],strAddr  ' Get a phrase.
    gosub stringOut             ' Send to screen.
    pause 1000                  ' Time to read phrase.
  next                          ' Next phrase until done.

STOP                            ' End of program.

' =================================================================
' Here's the subroutine that reads C strings. To use it, place the
' address of the string, assigned by its DATA statement, into
' the variable strAddr.
stringOut:
  read strAddr,char             ' Get the character.
  if char <> 0 then continue    ' If char is 0, then return
return
continue:                       '   else continue
  serout builtIn,baud,[char]    '   and send char out the port.
  strAddr = strAddr+1           ' Point to next character in string.
goto stringOut                  ' Repeat until char = 0.
```