Welcome to E-XFL.COM

Understanding Embedded - Microcontroller, Microprocessor, FPGA Modules

Embedded - Microcontroller, Microprocessor, and FPGA Modules are fundamental components in modern electronic systems, offering a wide range of functionalities and capabilities. Microcontrollers are compact integrated circuits designed to execute specific control tasks within an embedded system. They typically include a processor, memory, and input/output peripherals on a single chip. Microprocessors, on the other hand, are more powerful processing units used in complex computing tasks, often requiring external memory and peripherals. FPGAs (Field Programmable Gate Arrays) are highly flexible devices that can be configured by the user to perform specific logic functions, making them invaluable in applications requiring customization and adaptability.

Applications of Embedded - Microcontroller,

Details	
Product Status	Active
Module/Board Type	MCU Core
Core Processor	PIC16C57C
Co-Processor	-
Speed	20MHz
Flash Size	2KB EEPROM
RAM Size	32B
Connector Type	-
Size / Dimension	1.2" x 0.6" (30mm x 15mm)
Operating Temperature	0°C ~ 70°C
Purchase URL	https://www.e-xfl.com/product-detail/parallax/bs2-ic

Table of Contents

Among the changes that affect this edition of *StampWorks* is an updated PBASIC language: PBASIC 2.5. For those that come from a PC programming background, PBASIC 2.5 will make the transition to embedded programming a bit easier to deal with. And what I'm especially excited about is a new development platform: the Parallax Professional Development Board. My colleague, John Barrowman, with feedback from customers and Parallax staff alike, put about all of the features we would ever want into one beautiful product. For those of you have an NX-1000 (any of the variants), don't worry; most of the experiments will run on it without major modification.

Finally, as far as the text goes, many of the project updates are a direct result of those that have come before you, and you, my friend, have the opportunity to affect future updates. Please, if you ever have a question, comment, or suggestion, feel free to e-mail them to Editor@parallax.com.

BRANCHING – REDIRECTING PROGRAM FLOW

A branching instruction is one that causes the flow of the program to change from its linear path. In other words, when the program encounters a branching instruction, it will, in almost all cases, not be running the next [linear] line of code. The program will usually go somewhere else, often creating a program loop. There are two categories of branching instructions: *unconditional* and *conditional*. PBASIC has two instructions, **GOTO** and **GOSUB** that cause unconditional branching.

Here's an example of an unconditional branch using **GOTO**:

```
Label:
  statement 1
  statement 2
  statement 3
  GOTO Label
```

We call this an unconditional branch because it always happens. **GOTO** redirects the program to another location. The location is specified as part of the **GOTO** instruction and is called an address. Remember that addresses start a line of code and are followed by a colon (**:**). You'll frequently see **GOTO** at the end of the main body of code, forcing the program statements to run again.

Conditional branching will cause the program flow to change under a specific set of circumstances. The simplest conditional branching is done with an **IF-THEN** construct. PBASIC includes two distinct versions of **IF-THEN**; the first is used specifically to redirect program flow to another point based on a tested condition.

Take a look at this listing:

```
Start:
  statement 1
  statement 2
  statement 3
  IF (condition) THEN Start
```

In this example, statements 1- 3 will run at least once and then continue to run as long as the condition evaluates as True. When required, the condition can be tested prior to the code statements:

```
Start:
  IF (condition) THEN
    statement 1
    statement 2
    statement 3
  ENDIF
```

Note that the code statements are nested in an **IF-THEN-ENDIF** structure which does not require a branch label. If the condition evaluates as False, the program will continue at the line that follows **ENDIF**. Another use of this conditional structure is to add the **ELSE** clause:

```
Start:
  IF (condition) THEN
    statement 1
    statement 2
    statement 3
  ELSE
    statement 4
    statement 5
    statement 6
  ENDIF
```

If the condition evaluates as True then statements 1 – 3 will run, otherwise statements 4 – 6 will run.

As your requirements become more sophisticated, you'll find that you'll want your program to branch to any number of locations based on the value of a control variable. One approach is to use multiple **IF-THEN** constructs.

```
IF (index = 0) THEN Label_0
IF (index = 1) THEN Label_1
IF (index = 2) THEN Label_2
```

This approach is valid and does get used. Thankfully, PBASIC has a special command called **BRANCH** that allows a program to jump to any number of addresses based on the value of an index variable. **BRANCH** is a little more complicated in its setup, but very powerful in that it can replace multiple **IF-THEN** statements. **BRANCH** requires a control (index) variable and a list of addresses

The previous listing can be replaced with one line of code:

```
BRANCH index, [Label_0, Label_1, Label_2]
```

For example, this:

```
HIGH 0
```

... actually performs the same function as:

```
DIR0 = 1                                        ' make P0 an output
OUT0 = 1                                        ' set P0 high
```

but does it with just one line of code. Conservation of program space is an important aspect of microcontroller programming, and when we can save code space we should – we'll probably want or need that space later.

A very common beginner's error is this:

```
OUTPUT 0

HIGH 0
```

There is no need to manually configure the pin as an output as this function is part of the **HIGH** command. While doing this won't harm the program, it does consume valuable code space. There are very few occasions when **INPUT** and **OUTPUT** are required for proper program operation, as most PBASIC commands handle setting the pin's I/O state.

Write Code like a Pro

Note that even in this very simple program, we are following the style guidelines detailed in "The Elements of PBASIC Style". By using this professional style, the program becomes somewhat self-documenting, requiring fewer comments, and it allows the program to be modified far more easily. If, for example, we wanted to change the LED pin assignment or the flash rate, we would only have to make small changes to the declarations sections and not have to edit the entire listing. When our programs grow to several hundred lines, using cleverly-named pin definitions and constant values will save us a lot of time and frustration.

Behind the Scenes

This experiment demonstrates the ability to directly manipulate the BASIC Stamp output pins just as we could any other variable. This program also demonstrates conditional looping by adding pre- and post-loop tests to **DO-LOOP**.

The program starts by initializing the ***LEDs*** to %00000001 – this turns on the LED connected to P0. Then we drop into the first **DO-LOOP** where the value of ***LEDs*** is immediately tested. If the value of ***LEDs*** (currently %00000001) is less than %10000000 then the code within the **DO-LOOP** is allowed to run, otherwise the program continues at the line that follows **LOOP**.

Since ***LEDs*** is initially less than the test value, the program drops into the loop where it runs a small **PAUSE**, then the lit LED is moved to the left with the **<<** (shift-left) operator. Shifting left by one bit performs the same function as multiplying by two, albeit far more efficiently. After the shift the program goes back to the **DO WHILE** line where the value of ***LEDs*** (now %00000010) is tested again.

After seven passes through the upper loop, ***LEDs*** will have a value of %10000000 and the test will fail (result will be False); this will force the program to jump to the top of the second **DO-LOOP**.

The second **DO-LOOP** is nearly identical to the first except that the value of ***LEDs*** is shifted right one bit with **>>** (same as dividing by two), and the test occurs at the end of the loop. Note that when the test is placed at the end of the **DO-LOOP** structure, the loop code will run at least one time. After seven iterations of the bottom loop the test will fail and the code will drop to the **GOTO Main** line which takes us back to the top of the program.

Beginning programmers will often ask, "When should I use **WHILE** versus **UNTIL** in a loop test?"

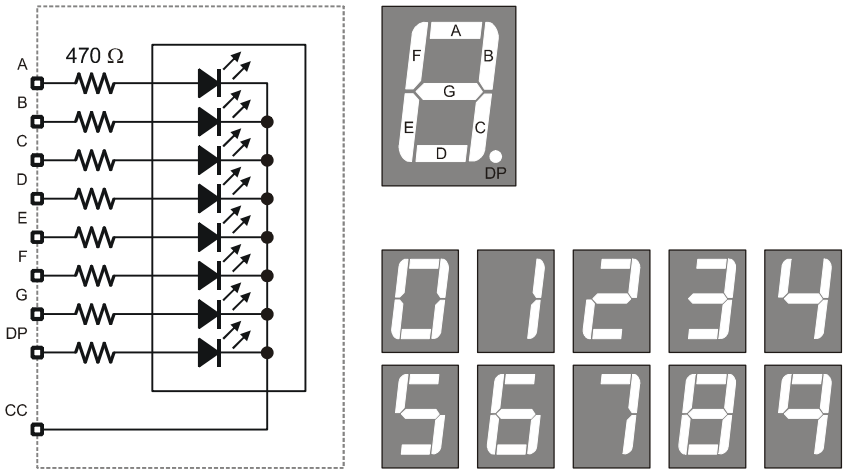
It is in fact possible to write functionally equivalent code using **WHILE** or **UNTIL**. That said, your programs will be easier to others to follow (and for you to pick up later) if the listing reads logically. To that end, it is suggested that **WHILE** is used to run the loop while a condition is true; and **UNTIL** is used to run the loop until a condition becomes true.

```
' -----[ Initialization ]--------------------------------------------------

Reset:
  LEDsDirs = %11111111                          ' make LEDs outputs


' -----[ Program Code ]-----------------------------------------------------

Main:
  DO
    GOSUB Read_Pot                              ' get raw pot value
    grafVal = (rawVal - LoScale) */ Scale       ' z-adjust, then scale
    GOSUB Show_Graph                            ' now show it
    PAUSE 50
  LOOP


' -----[ Subroutines ]------------------------------------------------------

Read_Pot:
  HIGH Pot                                      ' charge cap
  PAUSE 1                                       '   for 1 millisecond
  RCTIME Pot, 1, rawVal                         ' read the Pot
  RETURN


Show_Graph:
  hiBit = DCD (grafVal / 32)                    ' get highest bit
  IF (GraphMode = BarGraf) THEN
    newBar = 0                                  ' clear bar workspace
    IF (grafVal > 0) THEN
      DO WHILE (hiBit > 0)                      ' all bar LEDs lit?
        newBar = newBar << 1                    ' no - shift left
        newBar.BIT0 = IsOn                      ' light low end
        hiBit = hiBit >> 1                      ' mark bit lit
      LOOP
    ENDIF
    LEDs = newBar                               ' output new level
  ELSE
    LEDs = hiBit                                ' show dot value
  ENDIF
  RETURN
```

Using 7-Segment LED Displays

As you look around and notice devices that use them, you'll see that LEDs come in all manner of shape, size, and color. Early on, LED manufacturers found that they could package seven rectilinear-shaped LEDs in a Figure-8 pattern and when specific groups of LEDs were lit, the display could be any of the decimal digits and even a few alpha characters. We call these packaged groups of LEDs 7-segment displays.

In order to simplify wiring, 7-segment LED displays have a common internal connection; the LEDs used on the PDB are common-cathode, that is, the cathodes of the LEDs within the display are connected together and that connection must be made low (connected to Vss) in order to light any of the LEDs in the package. The diagram below shows the connections of a common-cathode LED display in relation to the current-limiting resistors on the PDB.

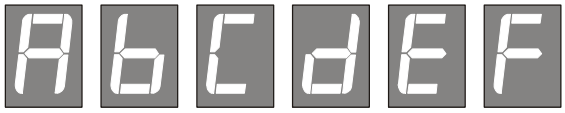


Note that the PDB has five, 7-segment, common-cathode LED modules, and the terminal marked "A" in the "SEGMENTS" section is connected to the A-segment LED in all five modules.

In the experiments that follow we will learn how to get the most out of 7-segment displays.

Take it Further

Update the program to create a single-digit hexadecimal counter. Use the patterns below for the HEX digits.



Write Code like a Pro

Note that the **DATA** table the stores the 7-segment patterns uses verbose label names and the patterns are placed in sequential order. By storing the segment information in EEPROM instead of constants, transferring these patterns to the display is greatly simplified.

Had we elected to store the patterns as constant values, we'd have to use the following bit of code to make the transfer:

```
LOOKUP idx, [Digit0, Digit1, Digit2, Digit3, Digit4,
             Digit5, Digit6, Digit7, Digit8, Digit9], Segs
```

As you can see, using **READ** is a bit tidier. In most programs, storing table values in **DATA** statements will simplify coding and save code space if the same values are to be used in more than one place in the program.

```
' -----[ Initialization ]--------------------------------------------------

Reset:
  #IF _LcdReady #THEN
    #ERROR "Please use BS2p version: SW21-EX12-LCD_Chars.BSP"
  #ENDIF

  DIRL = %11111110                              ' setup pins for LCD
  PAUSE 100                                     ' let the LCD settle

Lcd_Setup:
  LcdBus = %0011                                ' 8-bit mode
  PULSOUT E, 3
  PAUSE 5
  PULSOUT E, 3
  PULSOUT E, 3
  LcdBus = %0010                                ' 4-bit mode
  PULSOUT E, 1
  char = %00101000                              ' multi-line mode
  GOSUB LCD_Cmd
  char = %00001100                              ' disp on, no crsr or blink
  GOSUB LCD_Cmd
  char = %00000110                              ' inc crsr, no disp shift
  GOSUB LCD_Cmd

Download_Chars:                                 ' download custom chars
  char = LcdCGRam                               ' point to CG RAM
  GOSUB LCD_Cmd                                 ' prepare to write CG data
  FOR idx1 = CC0 TO (Smiley + 7)                ' build 4 custom chars
    READ idx1, char                             ' get byte from EEPROM
    GOSUB LCD_Out                               ' put into LCD CG RAM
  NEXT


' -----[ Program Code ]-----------------------------------------------------

Main:
  char = LcdCls                                 ' clear the LCD
  GOSUB LCD_Cmd
  PAUSE 250

  FOR idx1 = 0 TO 15                            ' get message from EEPROM
    READ (Msg1 + idx1), char                    ' read a character
    GOSUB LCD_Out                               ' write it
  NEXT
  PAUSE 1000                                    ' wait 2 seconds

Animation:
  FOR idx1 = 0 TO 15                            ' cover 16 characters
    READ (Msg2 + idx1), newChar                 ' get new char from Msg2
```

This is due to **DEBUG** requiring several milliseconds to send its output to the Debug Terminal window. Removing the **DEBUG** output (simple using conditional compilation) will increase the events input rate that can be detected.

Note, too, that the subroutine expects a clean input. A noisy input could cause spurious counts, leading to early termination of the subroutine. This is easily fixed by adapting the **Get_Buttons** subroutine from the last experiment.

```
Scan_Input:                          ' use with "noisy" inputs
  nScan = 1
  FOR idx = 1 TO 5
    nScan = nScan & EventIn
    PAUSE 5
  NEXT
  xScan = nScan ^ oScan & nScan      ' look for 0 -> 1 change
  oScan = nScan                      ' save this scan
  RETURN
```

```
' -----[ Constants ]-------------------------------------------------------

TAdj            CON     $100                    ' time adjust factor
FAdj            CON     $100                    ' frequency adjust factor

Threshold       CON     200                     ' cutoff frequency to play
NoteTm          CON     40                      ' note timing


' -----[ Variables ]-------------------------------------------------------

tone            VAR     Word                    ' frequency output


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    HIGH PitchCtrl                              ' discharge cap
    PAUSE 1                                     ' for 1 ms
    RCTIME PitchCtrl, 1, tone                   ' read the light sensor
    tone = tone */ FAdj                         ' scale input
    IF (tone > Threshold) THEN                  ' play?
      FREQOUT Speaker, NoteTm */ TAdj, tone
    ENDIF
  LOOP
```

Behind the Scenes

A Theremin is an interesting musical device used to create those weird, haunting sounds often heard in old horror movies. This version uses the light falling onto a photocell to create the output tone.

Since the photocell is a resistive device, **`RCTIME`** can be used to read its value. **`FREQOUT`** is used to play the note. The constant, **`Threshold`**, is used to control the cutoff point of the Theremin. When the photocell reading falls below this value, no sound is played. This value should be adjusted to the point where the Theremin stops playing when the photocell is not covered in ambient light.

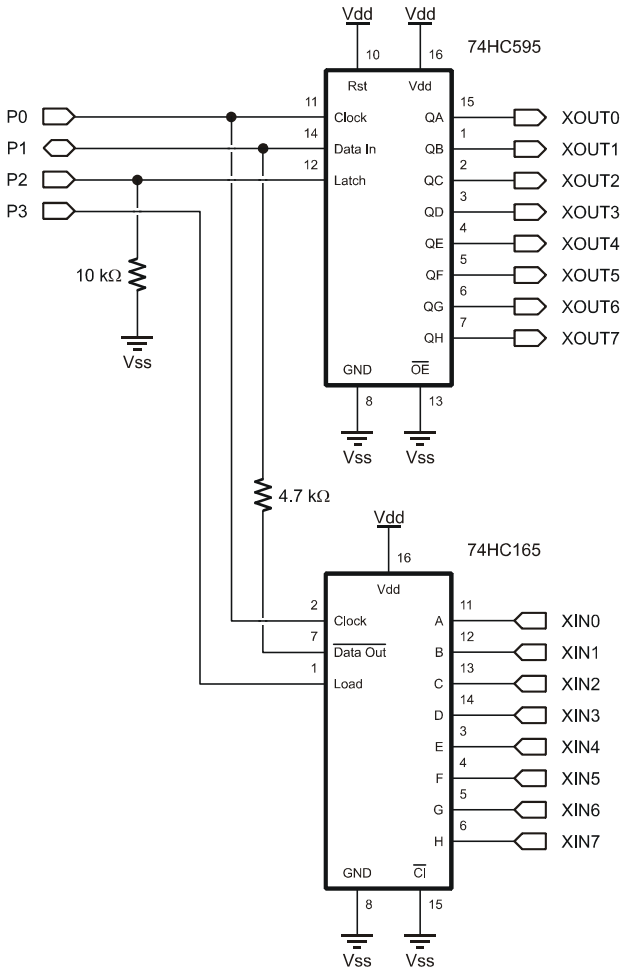
Behind the Scenes...Going Deeper

You may wonder how the BASIC Stamp is able to create a musical note using a pure digital output. The truth is that it gets a little help from the outside world. At the

EXPERIMENT #25: MIXED IO WITH SHIFT REGISTERS

This experiment demonstrates the ability to mix the 74HC595 and 74HC165 and use the fewest number of BASIC Stamp I/O pins.

Building the Circuit



Note: The 4.7 kΩ resistor is marked: yellow-violet-red.

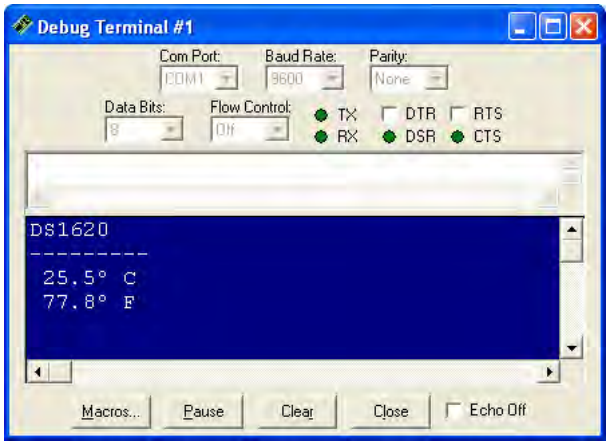
To convert from Celsius (in tenths) to Fahrenheit (also in tenths) a modification of the standard temperature equation is used:

$F_{tenths} = (C_{tenths} * 1.8) + 320$

Note that 32 degrees from the standard equation has also been converted to tenths.

For the conversion of negative temperatures the order of elements in the equation is reversed. The reason for this is that negative numbers cannot be divided in PBASIC. The **ABS** operator is used to convert the intermediate result to a positive value. When subtracted from 320 the result will be properly aligned (and signed); some negative values in the Celsius range are still positive in Fahrenheit.

The display routine uses a little trick that looks at Bit15 of the value; if Bit15 is one then the temperature is negative and a "-" will precede the temperature reading, otherwise a space will be printed.

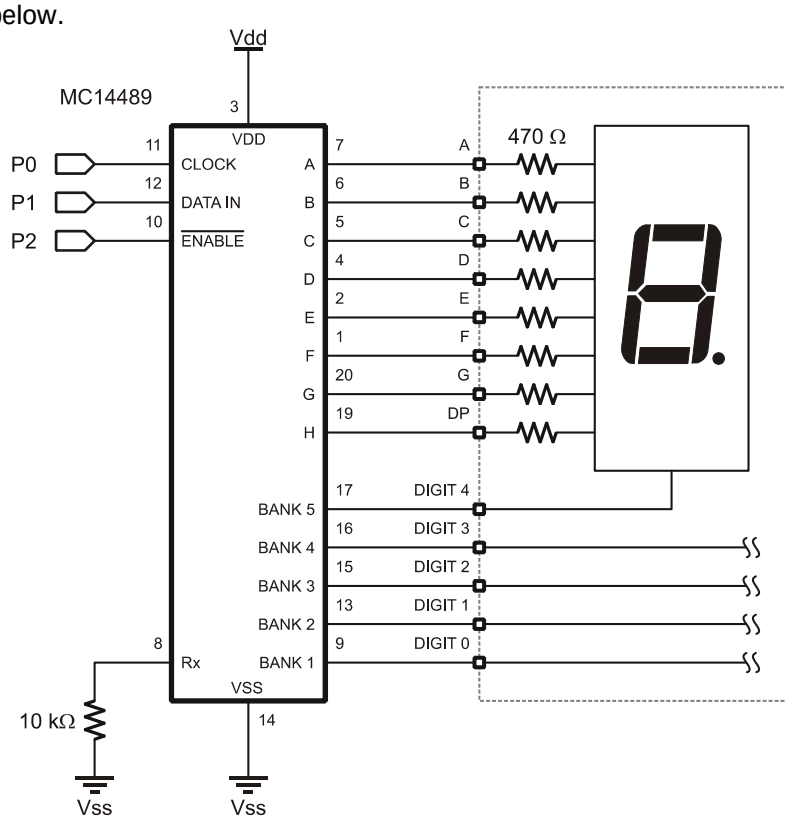


EXPERIMENT #31: ADVANCED 7-SEGMENT MULTIPLEXING

This experiment demonstrates the use of 7-segment displays with an external multiplexing controller. Multi-digit seven-segment displays are frequently used on vending machines to display the amount of money entered.

Building the Circuit

Connect four pushbuttons to P4-P7 (see Experiment #14) and add the multiplexing circuit below.



The I2C specification actually allows for multiple Masters to exist on a common bus and provides a method for arbitrating between them. That's a bit beyond the scope of what we need to do so we're going to keep things simple. In our setup, the BS2 (or BS2e or BS2sx) will be the Master and anything connected to it will be a Slave.

You'll notice in I2C schematics that the SDA (serial data) and SCL (serial clock) lines are pulled up to Vdd (usually through 4.7 kΩ). The specification calls for device bus pins to be open drain. To put a high on either line, the associated bus pin is made an input (floats) and the pull-up takes the line to Vdd. To make a line low, the bus pin pulls it to Vss (ground).

This scheme is designed to protect devices on the bus from a short to ground. Since neither line is driven high, there is no danger. We're going to cheat a bit. Instead of writing code to pull a line low or release it (certainly possible – I did it), we're going to use **SHIFTOUT** and **SHIFTIN** to move data back and forth. Using **SHIFTOUT** and **SHIFTIN** is faster and saves precious code space. If you're concerned about a bus short damaging the BASIC Stamp's SDA or SCL pins during **SHIFTOUT** and **SHIFTIN**, you can protect each of them with a 220 ohm resistor. If you're careful with your wiring and code this won't be necessary.

Low Level I2C Code

At its lowest level, the I2C Master needs to do four things:

- Generate a Start condition
- Transmit 8-bit data to the Slave
- Receive 8-bit data from Slave – with or without Acknowledge
- Generate Stop condition

A Start condition is defined as a high-to-low transition on the SDA line while the SCL line is high. All transmissions begin with a Start condition. A Stop condition is defined as a low-to-high transition of the SDA line while the clock line is high. A Stop condition terminates a transfer and can be used to abort it as well.

```
pntr            VAR     Byte                    ' ee pointer
char            VAR     Byte                    ' character for display

' -----[ EEPROM Data ]-------------------------------------------------

DayNames        DATA    "SunMonTueWedThuFriSat"

' -----[ Initialization ]-----------------------------------------------

Reset:
  #IF ($STAMP >= BS2P) #THEN
    #ERROR "Please use BS2p version: SW21-EX33-DS1307.BSP"
  #ENDIF

Setup:
  slvAddr = DS1307                              ' 1 byte in word address
  addrLen = 1

  DEBUG CLS,
        "DS1307 Demo", CR,
        "-----------"

Reset_Clock:
  GOSUB Get_Buttons                             ' scan buttons
  idx = btns & %0011                            ' isolate hrs & mins
  IF (idx = %11) THEN                           ' if both pressed, reset
    secs = $00
    mins = $00
    hrs = $06                                   ' 6:00 AM
    day = $07                                   ' Saturday
    date = $01                                  ' 1st
    month = $01                                 ' January
    year = $05                                  ' 2005
    control = 0                                 ' disable SQW output
    GOSUB Set_Clock                             ' block write clock regs
  ENDIF

' -----[ Program Code ]-------------------------------------------------

Main:
  GOSUB Get_Clock                               ' read DS1307
  hrs = hrs & $3F
  DEBUG CRSRXY, 0, 2,
        HEX2 hrs, ":", HEX2 mins, ":", HEX2 secs, CR
  GOSUB Print_Day
  PAUSE 100

  GOSUB Get_Buttons
  IF (btns > %0000) THEN                        ' button pressed?
    IF (btns <> %1000) THEN                     ' ignore back only
      hrs = hrs.NIB1 * 10 + hrs.NIB0            ' BCD to decimal
```

The CTS connection tells the PC that the BASIC Stamp is ready to receive data. Remember that the BASIC Stamp does not buffer serial data and if the PC sent a byte when the BASIC Stamp was busy processing another instruction that byte would be lost.

After initializing the LED outputs and the DS1620, the program enters the main loop and waits for input from the terminal program. First, **SERIN** waits for the "?" character to arrive, ignoring everything else until that happens. The question mark, then, is what signifies the start of a query. Once a question mark arrives, the **HEX** modifier causes the BASIC Stamp to look for valid hex characters (0 - 9, A - F). The arrival of any non-hex character (usually a carriage return [Enter] when using a terminal) tells the BASIC Stamp to stop accepting input (to the variable called ***cmd***) and continue on.

What actually has happened is that the BASIC Stamp has used the **SERIN** instruction to do a text-to-numeric conversion. Now that a command is available, the program uses **SELECT-CASE** to process valid commands, and sends a message to the terminal if the command entered is not used by the program.

For valid commands the BASIC Stamp responds to a request sending a text string using **SEROUT**. As with **SERIN**, flow control is used with **SEROUT** as well. The RTS (Request To Send) connection allows the PC to let the BASIC Stamp know that it is ready to receive data.

Each of the response strings consists of a label, the equal sign, the value of that particular parameter and finally, a carriage return. When using a terminal program, the output is easily readable. Something like this:

```
ID = StampWorks 2.1
```

The carriage return at the end of the output gives us a new line when using a terminal program and serves as an "end of input" when we process the input with our own program (similar to StampPlot Lite). The equal sign can be used as a delimiter when another computer program communicates with the BASIC Stamp. We'll use it to distinguish the label from its value.

Most of the queries are requests for information. Two of them, however, can modify information that is stored in the BASIC Stamp.

The first command is "?F1" which will allow us to write a string value to the BASIC Stamp's EEPROM (in a location called ID). When $F1 is received as a command value, the program jumps to the subroutine called **Set_ID**. On entry to **Set_ID**, the EE pointer called *eeAddr* is initialized, and then the BASIC Stamp waits for a character to arrive. Notice that no modifier is used here. Since terminal programs and the BASIC Stamp represent characters using ASCII codes, we don't have to do anything special. When a character does arrive, **WRITE** is used to put the character into EEPROM and the address pointer is incremented. If the last character was a carriage return (13), the program displays the new string (using the code at **Show_ID**), otherwise it loops back and waits for another character.

The second modifying query is "?B1" which allows us to set the status of four LEDs. Take a look at the subroutine called **Set_Leds**. This time, the **BIN** modifier of **SERIN** is used so that we can easily define individual bits we wish to control. By using the **BIN** modifier, our input will be a string of ones and zeros (any other character will terminate the binary input). In this program, a "1" will cause the LED to turn on and a "0" will cause the LED to turn off. Here's an example of using the B1 query.

```
?B1 0011<CR>
```

The figure below shows an actual on-line session using the BASIC Stamp's Debug Terminal window.

Striking Out on Your Own

Congratulations, you're a BASIC Stamp programmer! So what's next? Well, that's up to you. Many new programmers get stuck when it comes to developing their own projects. Don't worry, this is natural – and there are ways out of being stuck. The following workflow tips and resources will help you succeed in bringing your good ideas to fruition.

Plan Your Work, Work Your Plan

You've heard it a million times: plan, plan, and plan. Nothing gets a programmer into more trouble than bad or inadequate planning. This is particularly true with the BASIC Stamp as resources are so limited. Most of the programs we've fixed were "broken" due to bad planning and poor formatting which lead to errors.

Talk It Out

Talk yourself through the program. Don't just think it through, talk it through. Talk to yourself–out loud–as if you were explaining the operation of the program to a fellow programmer. Often, just hearing our own voice is what makes the difference. Better yet, talk it out as if the person you're talking to isn't a programmer. This will force you to explain details. Many times we take things for granted when we're talking to ourselves or others of similar ability.

Write It Out

Design the details of your program on a white (dry erase) board before you sit down at your computer. And use a lot of colors. You'll find working through a design visually will offer new insights, and the use of this medium allows you to write code snippets within your functional diagrams.

Design with "Sticky Notes"

Get out a pad of small "sticky notes". Write module names or concise code fragments on individual notes and then stick them up on the wall. Now stand back and take a