Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Technical: support@parallax.com

Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Tech Support: (888) 997-8267

AVR Firmware: 7SEG, Version 1
7-Segment LED Display Driver

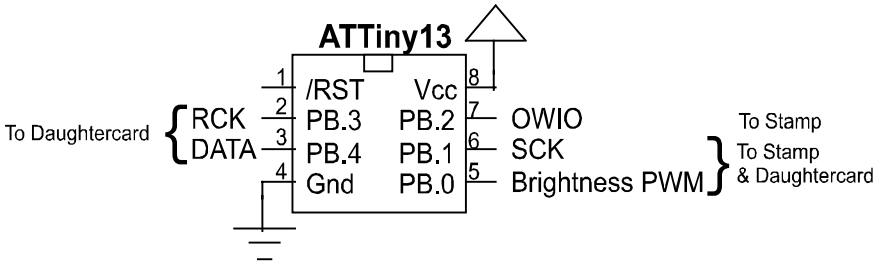
Introduction

This document describes AVR firmware that is used in conjunction with the BS2pe BASIC Stamp motherboard (the MoBoStamp-pe). This firmware can be uploaded to either or both of the motherboard's two AVR coprocessors as file **7SEG1.hex**, which can be downloaded from the 7Seg-DB product page at www.parallax.com/detail.asp?product_id=28312. Once the hex file is installed, the coprocessor is capable of communicating with the BASIC Stamp using PBASIC's **OWOUT** and **OWIN** commands. This communication takes place on the AVR's OWIO pin (see illustration below) to read data from, and write data to, the AVR. By utilizing the capabilities of the AVR coprocessor, this software driver handles all the buffering and refreshing required by Parallax's 7Seg-DB seven-segment display daughterboard, freeing the BASIC Stamp to treat it simply as an ASCII display device.

This firmware does the following:

- Accepts character sequences from the BASIC Stamp for display on the LEDs.
- Displays letters as well as numbers on the seven-segment displays.
- Handles up to eight daisy-chained four-digit displays for up to 32 digits total.
- Performs leading zero suppression in fixed-width numerical fields (e.g. **DEC4**).
- Allows full control of windowing and cursor manipulation.
- Permits left- or right-justified text within a selected window.
- Performs blinking within a selected window.
- Performs vertical and horizontal scrolling functions for special effects.
- Adjusts the display intensity from nearly off to fully on in 255 steps.

The AVR (Atmel ATTiny13) pinout is shown below:



Pin OWIO connects to the Stamp and has a pull-up resistor to Vdd. Communication is bi-directional via a protocol using open-collector signaling. Ports 2 and 3 also connect to the Stamp without external pull-ups, as well as to an attached daughtercard. Ports 0 and 1 connect to an attached daughtercard only and are the quadrature inputs from the encoder. Because ports 2 and 3 are controlled exclusively by the coprocessor, they should remain as inputs (pins 11 and 12 for socket "A"; 5 and 7, for socket "B") on the BASIC Stamp.

Getting Started

To use the seven-segment driver, you will have to upload it to the desired AVR coprocessor (usually "B") on the MoBoStamp-pe board. This is accomplished using the program **LoadAVR.exe**, available on the Parallax MoboStamp-pe product page: www.parallax.com/detail.asp?product_id=28300. Also, be sure to start your program with a **PAUSE 10** to give the AVR a chance to come out of reset before trying to access it. Finally, anytime the AVR receives a reset pulse from the BASIC Stamp's **OWOUT** or **OWIN** commands, all special effects are cleared, the active window is set to all 32 characters, and the cursor is homed. The display is not cleared, however.

Displaying Characters

Displaying characters on the seven-segment displays is as easy as sending them via PBASIC's **OWOUT** command. For example, to display the contents of the nibble variable **A**, just do the following:

```
OWOUT Owio, 0, ["A=", DEC A]
```

Owio is the pin (10 for coprocessor "A"; 6, for coprocessor "B") used to communicate with the AVR. If variable **A** had been equal to **12**, he above command would have displayed:

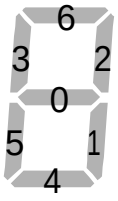
When including a period (decimal point ".") in an output string, it is automatically displayed between characters:

```
OWOUT Owio, 0, ["AB.CD"]
```

This displays as:

The displayable characters are all those with ASCII codes from **$20** to **$5F**, i.e. all the punctuation except **`{|}~**, all the numbers, and all the capital letters. Many of the capital letters are displayed in lower-case, however, in order to accommodate the limitations of a seven-segment display or to distinguish them from other characters. Several of the non-numerical characters, notably **M**, **W**, **V**, etc., may be difficult to decipher out of context. But within context, their identity is usually clear. And if you don't like a particular character, you can always change it. (See "Altering Characters" at the end of this document.)

In addition to the standard character set, it is possible to display custom characters on the fly. All ASCII characters in the range **$80** to **$FF** are custom, with each of the seven least-significant bits representing the state of one of the seven segments: **1** = "on"; **0** = "off". The correspondence between segments and bit positions is shown in the following diagram (bit **0** being the least-significant):



So executing the following would display the segments shown:

```
OWOUT Owio, 0, [%10010001] 'Display custom, segments 0 and 4.
```

To include the decimal point in a custom character, just add a period (“.”) in the string after the custom character.

Command Codes

Sending commands to the display, or activating special effects, uses control characters in the range **$00** to **$1F**, along with some of the lower-case letters **a** through **z**. We’ll consider them here, arranged in functional groups.

Window Control

When you start your program, 32 characters are available as one contiguous display workspace. This is true regardless of how many or how few physical modules you’ve daisy-chained together. The starting active workspace, or “window”, is always 32 characters long. You can restrict display actions to a smaller window by sending the window command: **w**, followed by two numbers. These numbers represent the leftmost and rightmost extents, respectively, of the active display window, beginning on the left with position **0**, and ending on the far right of eight 7Seg-DB modules with position **31**. The following code fills an eight-digit display with hyphens, then defines a window between positions **2** and **5** and prints some numbers there:

```
OWOUT Owio, 0, ["--------w", 2, 5, "123456789"]
```

Here’s what gets displayed:

Notice that only those digits able to fit in the defined window get displayed. Character positions outside the window (i.e. the hyphens) remain intact .

When defining a new window, special effects taking place in the old window (e.g. scrolling and blinking) will be cancelled, and the new window gets initiated with no special effects.

If you are using fewer than eight display modules daisy-chained together, you will probably want to define a window at the beginning of your program that encompasses only those character positions that exist physically. This will help some of the special effects features to behave more like what you expect, without actions being hidden off to the right among non-existent characters.

Cursor Control and "Window Washing"

The cursor control and window-washing (clear) commands are used to move the cursor back and forth, to position it to a fixed location, or to erase portions of the display. They are as follows:

CLS ($00)	Clear the active window.
HOME ($01)	Move cursor to the extreme left of the active window.
CRSRLF ($03)	Move cursor left by one position within the active window.
CRSRRT ($04)	Move cursor right by one position within the active window.
BKSP ($08)	Backspace. Same as **CRSRLF**. *Character to the left is not erased.*
TAB ($09)	Move to the next cursor position divisible by four *relative to the beginning of the active window*.
CLREOL ($0B)	Clear the active window from the current cursor position onward.
CR ($0D)	Carriage return. Clear the active window from the current position onward, then home the cursor.
CRSRX ($0E)	Move the cursor to the position, *relative to the beginning of the active window,* given by the following byte.

The symbol names in boldface in the above table are predefined in PBASIC 2.5 and can be used just as they would be in a **DEBUG** statement. For example:

```
OWOUT Owio, 0, ["----", CRSRX, 1, "ABC", BKSP, CLREOL]
```

This displays:

First, four hyphens are sent to the display, then the cursor is positioned to digit 1, "ABC" is printed. Next, the cursor backs up one space, and finally everything after the "AB" is cleared.

Display Brightness

When your program starts, the LED is set to half brightness. This is to ensure that a large display doesn't demand more current than is available from the power supply. But you can change the brightness of the entire display using the brightness (**b**) command. The range is from **0** (minimum brightness) to **255** (maximum brightness). It is used as follows:

```
OWOUT Owio, 0, ["b", 4, "THIS IS DIM."]
```

This displays:

The visual brightness will not appear to be a linear function of the brightness value. This is partly because your eyes are sensitive to percentage (logarithmic) changes, rather than absolute (linear) changes. So to

get an even transition from “off” to “bright”, say, you would need to loop through a selected subset of the 256 available brightness levels to give the *appearance* of a linear transition.

Right Justified Text

Normal text sent to a display is left-justified. You can change this to right-justified by using the justify (**j**) command. After the **j** is sent, text will flow into the active window from the right, shifting existing characters to the left with each new character. Here’s how it works:

```
OWOUT Owio, 0, [“w“, 1,6, “j1234”]
```

This displays:

To change back to left-justification, send a **CLS** to clear the active window.

Flashing Text

To cause all the text in the active window to flash on and off, issue the flash (**f**) command:

```
OWOUT Owio, 0, [“CORE: w”, 6, 11, “f900*F”]
```

This displays:

The **900°F** in the window that goes from positions **6** through **11** will be flashing on and off about once a second.

To turn flashing off, use the clear flashing command (**c**). Defining a new window will also turn flashing off.

Leading Zero Suppression

In PBASIC you have two choices for outputting decimal numbers: free-formatted (left-justified) without leading zeroes, or fixed-width with leading zeroes. In many cases, though, you will want fixed width without leading zeroes, which you can get with the **7SEG1** firmware. The method is simple: just prepend the zero-suppression (**z**) command ahead of any number to be displayed, followed by a single-byte count value, and that many subsequent leading zeroes will be replaced by spaces. Here’s how a fixed-width number would be displayed without zero-suppression:

```
A = 23
OWOUT Owio, 0, [DEC4 A]
```

Now, try it with zero-suppression:

```
A = 23
OWOUT Owio, 0, [“z”, 3, DEC4 A]
```

The count value of **3** guarantees that up to three leading zeroes will be suppressed. This value is typically one less than the width of the field because, if the number being displayed is zero, you want the single digit **0** to display.

Bar Graph Display

Simple horizontal bar graphs can be displayed with the graph (**g**) command. It requires one single-byte argument: the length of the bar. When executed, a horizontal bar graph is displayed in the active window, replacing anything else that might have been there before. Here’s an example:

```
OWOUT Owio, 0, [“--------w”, 1, 6, “g”, 9]
```

This produces:

The vertical lines are the bar graph. There are nine of them, as specified after the **g** command. The remaining LED segments in the active window, **(1, 6)**, are turned off. But previously-defined characters *outside* the active window (the hyphens) remain untouched. By replacing the constant **9** with a variable name or expression, you can easily track it’s value with a dynamically-changing bar graph.

Horizontal Shifting

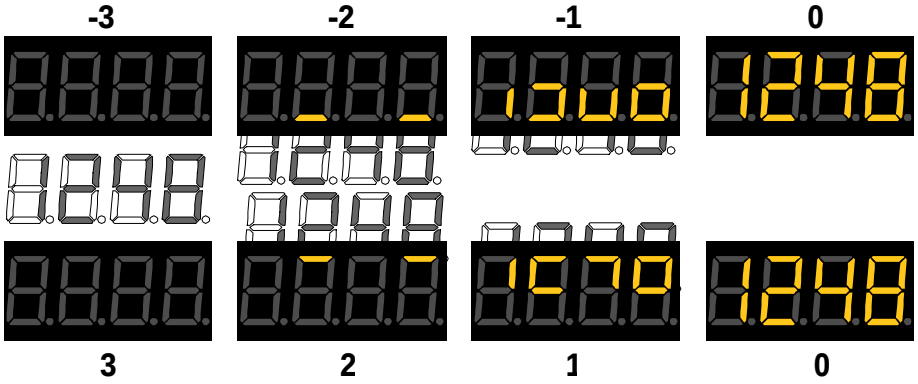
Characters displayed in an active window can be shifted left or right after the fact. This can be useful for many special effects and is simple to perform using the left (**l**) and right (**r**) shift commands. Sending an **l** shifts all the characters in the active window left by one position, shifting out the leftmost character and replacing the rightmost character with a blank. Sending an **r** does just the opposite. After a left shift, the cursor will be at the rightmost position in the active window. After a right shift, it will be at the leftmost position.

Here's a program that flows a message string onto the display from the left.

```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

Owio  PIN 6

Ptr   VAR Word              'Pointer into EEPROM DATA area.
Chr   VAR Byte              'Character read from EEPROM.

Msg   DATA "INCOMING!", 0   'The message to be displayed.

Ptr = Msg                   'Point to the beginning of the message.
DO                          'Loop through it to find the zero at the end.
  READ Ptr, Chr
  IF (Chr = 0) THEN EXIT
  Ptr = Ptr + 1
LOOP

OWOUT Owio, 0, ["w", 0, 11] 'Define a window to encompass three modules.

DO                          'Loop through the message in reverse order.
  Ptr = Ptr - 1
  READ Ptr, Chr             'Read a character.
  OWOUT Owio, 0, ["r", Chr] 'Shift the characters right, and display the new one.
  PAUSE 100                 'Pause for dramatic effect.
LOOP UNTIL Ptr = Msg
```

Vertical Scrolling

Characters displayed in the active window can be scrolled vertically for additional "wipe-on/wipe-off" effects. A character's vertical position is given by a single number, which can take on values between **–3** ($FD) and **3** ($03). The illustration below shows characters at the different vertical scroll positions:



The vertical position of the active window can be set directly with the vertical scroll (**v**) command, followed by a single byte argument between **–3** and **3** giving the position. For example, this statement would produce the results shown:

```
    OWOUT Owio, 0, [“ABv”, -1, “CD”]
```

In the example, the scroll position was set halfway through the displayed string. Yet, the entire string was affected. What this illustrates is that the setting is applied to the entire active window for characters

that are already there, as well as for those yet to be written. The scroll state of the active window persists until it's changed by another scroll command or another window is defined. In the latter case, the previously active window is cleared if its scroll state wasn't equal to zero.

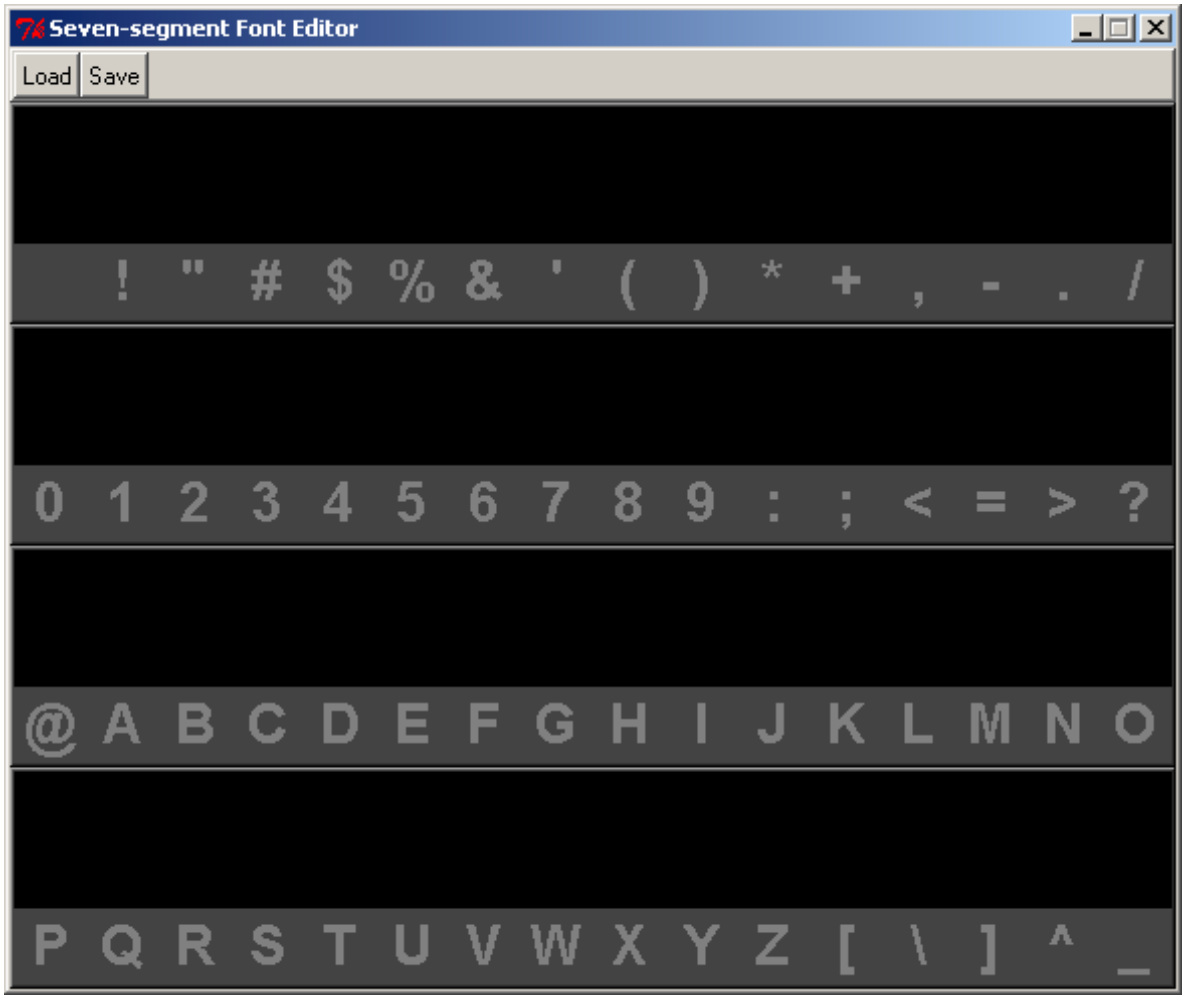
Rounding out the vertical scroll commands are scroll up (**u**) and scroll down (**d**). Each shifts the active window one position in the indicated direction.

The following program shows how to simulate the rolling number wheels on an odometer using scrolling:

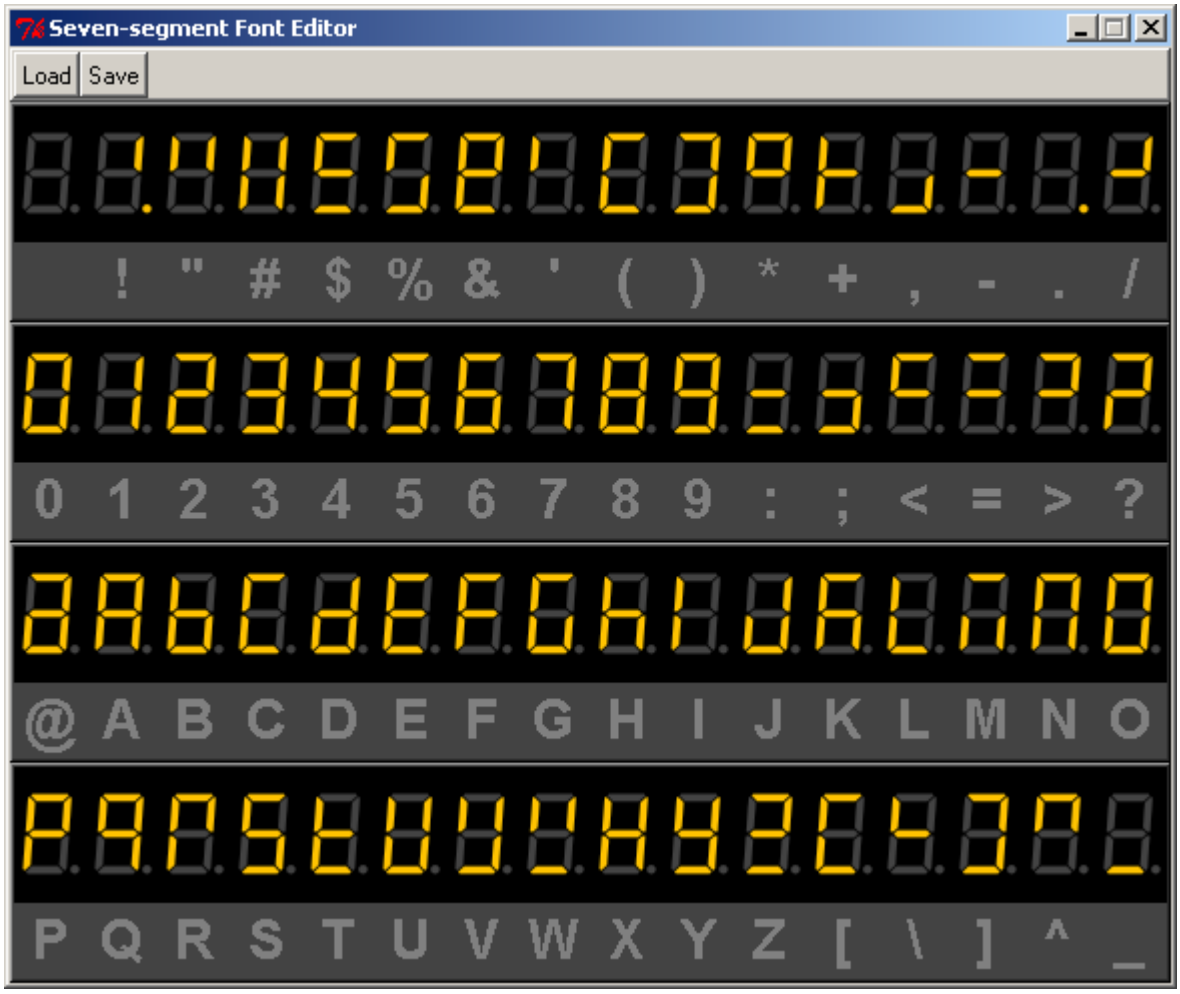
```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

'-----------------------------------------------
'Program to demonstrate scrolling on the 7Seg-DB.
'It operates like the odometer in a car,
'scrolling up the digits that change.
'This program requires a single 7Seg-DB master.
'-----------------------------------------------

Owio  CON 6

cnt   VAR Word                'The counter: 0000-9999.
val   VAR Word                'Changed part of cnt.
win   VAR Nib                 'Left side of changed window.
i     VAR Nib                 'Scroll counter.


PAUSE 10                      'Wait for AVR to come out of reset.

cnt = 9990                    'Start cnt at 9990 to demo max rollover.
OWOUT Owio, 0, [DEC4 cnt]     'Display the initial count.

DO                            'Loop forever.
  PAUSE 200                   'Leave time to view count.
  cnt = cnt + 1 // 10000      'Increment count modulo 10000.
  val = 1000                  'We'll see if it's a multiple of 1000 first.
  win = 0                     'Set the window to the left edge of display.
  DO WHILE val                'Val decreases, so loop terminates.
    IF cnt // val = 0 THEN    'Is cnt a multiple of val?
      val = cnt // (val * 10) '  Yes: Set val to the digits of cnt that changed.
      EXIT                    '       Exit LOOP.
    ENDIF
    val = val / 10            '  No:  Try a smaller value for val
    win = win + 1             '       Move the left window border to the right.
  LOOP                        '       Try again.
  OWOUT Owio, 0, ["w", win, 3]'Define the new window.
  FOR i = 1 TO 3              'Scroll current contents of window up by three (off the screen).
    OWOUT Owio, 0, ["u"]      '  Scroll up one.
    PAUSE 50                  '  Wait, for animation effect.
  NEXT
  IF val THEN                 'Is val equal to 0?
    OWOUT Owio, 0, [DEC val]  '  No:  Display it in the window (still scrolled off).
  ELSE
    OWOUT Owio, 0, [DEC4 val] '  Yes: Display "0000" in the full window (still scrolled off).
  ENDIF
  FOR i = 1 TO 3              'Scroll new value onto window. (It takes three scrolls.)
    OWOUT Owio, 0, ["u"]      '  Scroll up one.
    PAUSE 50                  '  Wait, for animation effect.
  NEXT
LOOP                          'And so ON...
```

Redefining the Standard Character Set

You can redefine the characters displayed by this firmware to suit your tastes or special application requirements. This is done using the program **7SegFont.exe**, available from the 7Seg-DB product page on Parallax's website. After starting this font editor, you will be presented with the following blank screen.



The first thing to do is to load an existing font from **7SEG1.HEX** or one of its derivatives. This is done by clicking the **Load** button and selecting the hex file you want to edit. Once this is done, you will see something like the following:



You can edit any character (except the period) by clicking segments to toggle them between “on” and “off”. Once you have the font where you want it, click **Save**, and you can save the new font to the same hex file or to a new one. After saving the font, just upload the new hex file to the AVR using **LoadAVR.exe**, and your new font will be displayed in all its glory.

Command Summary

The following chart summarizes the commands and control characters used to control your seven-segment display:

Command	Description	Parameters that Follow
CLS ($00)	Clear the active window.	None
HOME ($01)	Move cursor to the extreme left of the active window.	None
CRSRLF ($03)	Move cursor left by one position within the active window.	None
CRSRRT ($04)	Move cursor right by one position within the active window.	None
BKSP ($08)	Backspace. Same as **CRSRLF**. *Character to the left is not erased.*	None
TAB ($09)	Move to the next cursor position divisible by four, *relative to the beginning of the active window*.	None
CLREOL ($0B)	Clear the active window from the current cursor position onward.	None
CR ($0D)	Carriage return. Clear the active window from the current position onward, then home the cursor.	None
CRSRX ($0E)	Move the cursor to a new position, relative to the beginning of the active window.	One: New cursor position (**0** to **31**)
"**b**"	Set overall display brightness.	One: Brightness level (**0** to **255**)
"**c**"	Cancel flashing.	None
"**d**"	Scroll active window down.	None
"**f**"	Turn on flashing in the active window.	None
"**g**"	Draw a horizontal bar graph in the active window.	One: Length of bar (**0** to **64**)
"**j**"	Change active window to right-justification.	None
"**l**"	Shift characters in active window left by one position.	None
"**r**"	Shift characters in active window right by one position.	None
"**u**"	Scroll active window up.	None
"**v**"	Set vertical scroll position.	One: Vertical position (**-3** to **3**)
"**w**"	Set the active window	Two: Left and right extents of window (**0** to **31**)
"**z**"	Suppress leading zeroes in next numerical field.	One: The maximum number of leading zeroes to suppress (**1** to **32**)