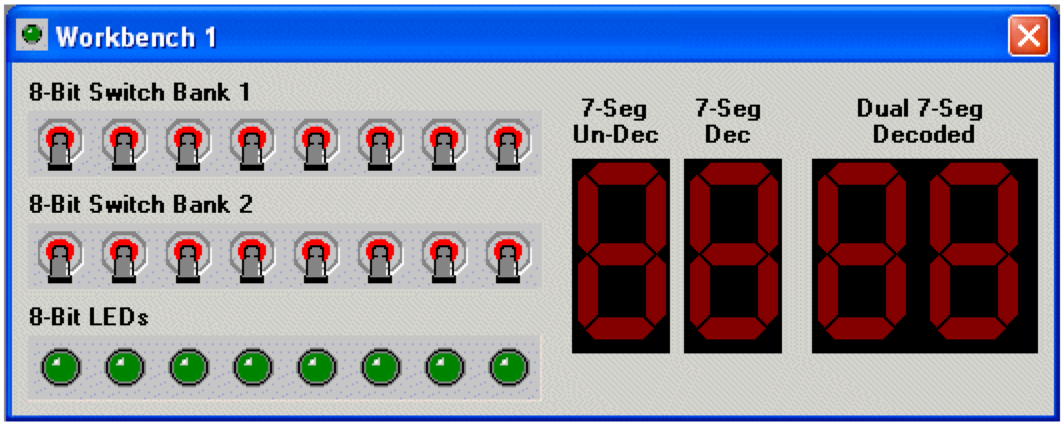
DIY Calculator: Workbench and Terminal

Introduction

The purpose of this paper is to document some new utilities that have been added to the DIY Calculator. The first is a workbench that comprises a selection of simple switches and displays. The second is a text terminal that comprises a virtual QWERTY (typewriter-style) keyboard and a simple console display.

The Workbench

Using the **Tools > Workbench #1** command results in the following interface appearing in the DIY Calculator's main window:



The two banks of 8-bit switches are connected to two of the DIY Calculator’s input ports; meanwhile, the 8-bit LED display and the three different flavors of 7-segment displays are connected to four of the DIY Calculator’s output ports as follows:

Port Type	Port Address	I/O Device
Input	$F000	8-bit Switch Bank #1
Input	$F001	8-bit Switch Bank #2
Output	$F020	8-Bit LED Display
Output	$F021	Single Un-decoded 7-Segment Display
Output	$F022	Single Decoded 7-Segment Display
Output	$F023	Dual Decoded 7-Segment Display.

A simple test program (written in the DIY Calculator’s assembly language) to read from the upper bank of 8-bit switches and copy this value to the 8-bit LED display would be as follows:

```
SW8BIT1:  .EQU   $F000     # Upper bank of 8-bit switches
LED8BIT:  .EQU   $F020     # 8-bit LED Display

          .ORG   $4000     # Set program origin
LOOP:      LDA   [SW8BIT1] # Read from upper 8-bit switches
           STA   [LED8BIT] # Write to 8-bit LED Display
           JMP   [LOOP]    # Do it all again
          .END
```

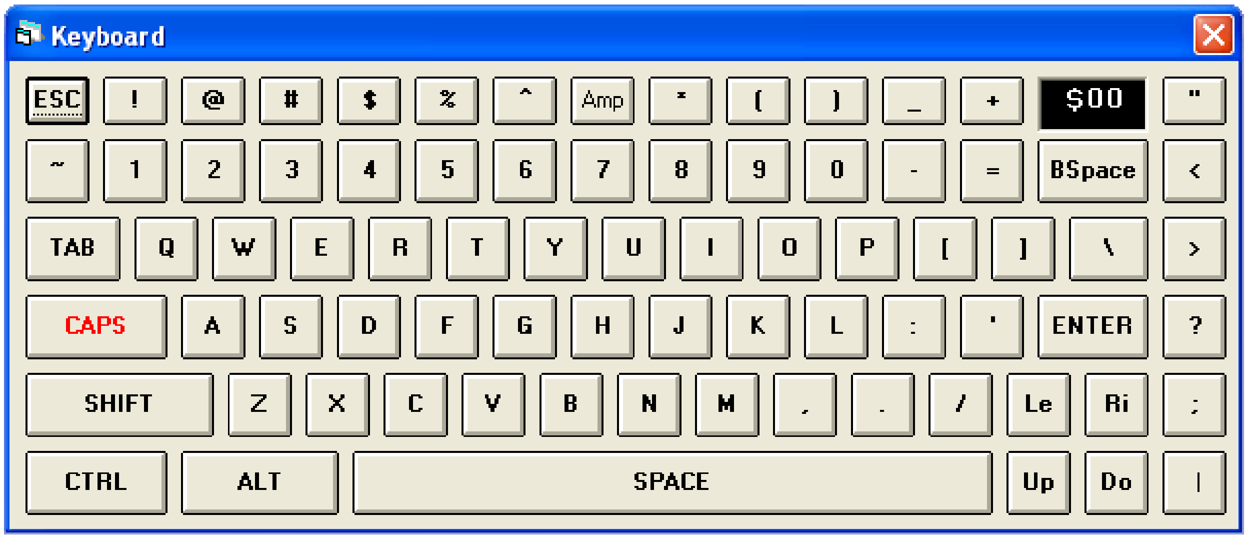
The test program above simply loops around loading the value currently on the 8-bit switches into the CPU's accumulator, and then copying this value back to the 8-bit LED display. If you power-up the DIY Calculator (using the On/Off button on the calculator's front panel) and assemble, load, and run this program, then every time you click on one of the switches the corresponding LED will change to reflect the state of that switch.

These input/output (I/O) devices can be used to create a wide variety of simple (but highly instructive) demonstration programs. Also, they can be used in conjunction with the main DIY Calculator panel; for example, you could create a program to read a button from the DIY Calculator and present its value on one of the workbench's displays. Alternatively, you could create a program to read the state of one (or both) of the 8-bit switches on the workbench and present this value to the DIY Calculator's main display.

Furthermore, the workbench devices can be used in conjunction with the virtual QWERTY (typewriter-style) keyboard and a simple console display introduced below. For example, you could create a program to read a button from the QWERTY keyboard and present its value on one of the workbench's displays. Alternatively, you could create a program to read the state of one (or both) of the 8-bit switches on the workbench and present this value to the console display.

The QWERTY Keyboard

Using the **Tools > Terminal** command results in both a QWERTY keyboard and a simple console display appearing in the DIY Calculator's main window. The QWERTY keyboard device appears as follows:


Keyboard
ESC ! @ # $ % ^ Amp * ( ) _ + $00 "
~ 1 2 3 4 5 6 7 8 9 0 - = BSpace <
TAB Q W E R T Y U I O P [ ] \ >
CAPS A S D F G H J K L : ' ENTER ?
SHIFT Z X C V B N M , . / Le Ri ;
CTRL ALT SPACE Up Do |

If you compare this device to a real keyboard, you'll observe that (in some respects) we have more keys. This is because a real keyboard "doubles" up some of the keys by means of the <Shift> key. For example, on a real keyboard, the number '4' key will represent the number '4' if the <shift> key is not pressed and the dollar character "$" if the <Shift> key is pressed. We just decided that it would be easier to split these keys out in the case of our virtual device.

This keyboard includes a virtual 8-bit latch. When the DIY Calculator is powered-up (by clicking the On/Off button on the calculator's front panel), this latch is automatically loaded with a default value of $00 (all zeros). When a key on the keyboard is clicked, an associated code is loaded into the latch. This code will remain in the latch until (a) another key is clicked, thereby overwriting the first value, or (b) the DIY Calculator's CPU accesses the value in the latch by reading from the input port (at address $F008) that is connected to the keyboard. The act of reading from this port automatically clears the latch to contain its default value of $00.

Observe the small two-digit hexadecimal display toward the upper-right of the keyboard. This displays the code associated with the last key to be pressed. Assuming that the <Caps> key is in its active state as discussed below, then clicking one of the alpha keys 'A', 'B', C', and so forth will cause this display to show values of $41, $42, $43, respectively, where these values are the ASCII codes for uppercase 'A', 'B', and 'C' characters. Just as a reminder, the ASCII table is shown below:

Not Standard ASCII				Standard ASCII											
$00		$10		$20	SP	$30	0	$40	@	$50	P	$60	`	$70	p
$01		$11		$21	!	$31	1	$41	A	$51	Q	$61	a	$71	q
$02		$12		$22	"	$32	2	$42	B	$52	R	$62	b	$72	r
$03		$13		$23	#	$33	3	$43	C	$53	S	$63	c	$73	s
$04		$14		$24	$	$34	4	$44	D	$54	T	$64	d	$74	t
$05		$15		$25	%	$35	5	$45	E	$55	U	$65	e	$75	u
$06		$16		$26	&	$36	6	$46	F	$56	V	$66	f	$76	v
$07		$17		$27	'	$37	7	$47	G	$57	W	$67	g	$77	w
$08		$18		$28	(	$38	8	$48	H	$58	X	$68	h	$78	x
$09		$19		$29	)	$39	9	$49	I	$59	Y	$69	i	$79	y
$0A		$1A		$2A	*	$3A	:	$4A	J	$5A	Z	$6A	j	$7A	z
$0B		$1B		$2B	+	$3B	;	$4B	K	$5B	[	$6B	k	$7B	{
$0C		$1C		$2C	,	$3C	<	$4C	L	$5C	\	$6C	l	$7C	|
$0D		$1D		$2D	-	$3D	=	$4D	M	$5D	]	$6D	m	$7D	}
$0E		$1E		$2E	.	$3E	>	$4E	N	$5E	^	$6E	n	$7E	~
$0F		$1F		$2F	/	$3F	?	$4F	O	$5F	_	$6F	o	$7F	

Now, observe the <Caps> key on the keyboard. We may refer to this as a "modifier" key. When this is in its active state (which is indicated by red annotation text as shown in the illustration on the previous page), clicking one of the alpha keys ('A', 'B', 'C', …) will cause the keyboard's latch to be loaded with the appropriate ASCII code for the uppercase version of these letters ('A', 'B', 'C',…).

Clicking the <Caps> key will cause it to toggle into its inactive state (which is indicated by black annotation text). When the <Caps> key is in its inactive state, clicking one of the alpha keys ('A', 'B', 'C', …) will cause the keyboard's latch to be loaded with the appropriate ASCII code for the lowercase version of these letters ('a', 'b', 'c',…).

On a real keyboard, the <Caps> (or <Caps Lock>) key, which stands for “Capitals”, is a “sticky” key, which means that once it’s been pressed (or clicked in our virtual world), it will remain in that state until it’s pressed (or clicked) again.

The <Shift>, <Ctrl> (“Control”), and <Alt> (“Alternate”) keys also act as modifier keys. On a real keyboard these would not be sticky, which means that you would have to continue to hold them while you pressed another key. However, we can’t emulate this on our virtual keyboard, so we’ve caused these keys to act like sticky keys.

In the case of our virtual system, the <Shift> key performs the same actions as the <Caps> key; that is, it only affects the case of the alpha characters (in the real world it would affect other characters/keys as well).

Compared to the <Shift> and <Caps> keys, the actions of the <Ctrl> and <Alt> keys are not quite so well defined, in that they tend to behave differently on different systems. In the case of our virtual QWERTY keyboard, we’ve chosen to follow a reasonably intuitive scheme. First, we’ve decided that the <Ctrl> key will only modify the codes associated with the alpha keys. For example, if we were to click the <Ctrl> key to make it active and then click the ‘A’ key, the keyboard’s latch and display would both be loaded with a value of $01; clicking the ‘B’ key would load a value of $02, and so forth. (Note that identical codes are generated for both uppercase and lowercase versions of each letter.)

Last but not least, in the case of our virtual keyboard, if the <Alt> key is in its active state, then it will modify the codes associated with all of the other keys (and modifier key combinations) by simply adding the value $80 (128 in decimal) to whatever codes they would have generated had the <Alt> key been inactive. For example, assuming that the <Alt>, <Ctrl>, <Shift>, and <Caps> keys are all inactive, clicking on the ‘A’ key will cause the keyboard’s latch and display to be loaded with a value of $61, which is the ASCII code for a lowercase ‘a’. If the <Alt> key were now placed in its active state, clicking the ‘A’ key would result in the keyboard’s latch and display being loaded with a value of $E1, which is the sum of $61 and $80.

Similarly, if the <Alt> key was inactive and the <Ctrl> key was active, clicking the ‘A’ key would cause the keyboard’s latch and display to be loaded with $01. If the <Alt> key was now placed in its active state (and the <Ctrl> key remained active), clicking the ‘A’ key would result in the keyboard’s latch and display being loaded with a value of $81, which is the sum of $01 and $80.

As was noted above, the keyboard is plugged into the input port at address $F008. Also, from our discussions on the workbench earlier in this paper, we know that the workbench’s 8-bit LED display is driven by the output port at address $F020. And we also know the DIY Calculator’s main display is driven by the output port at address $F031:

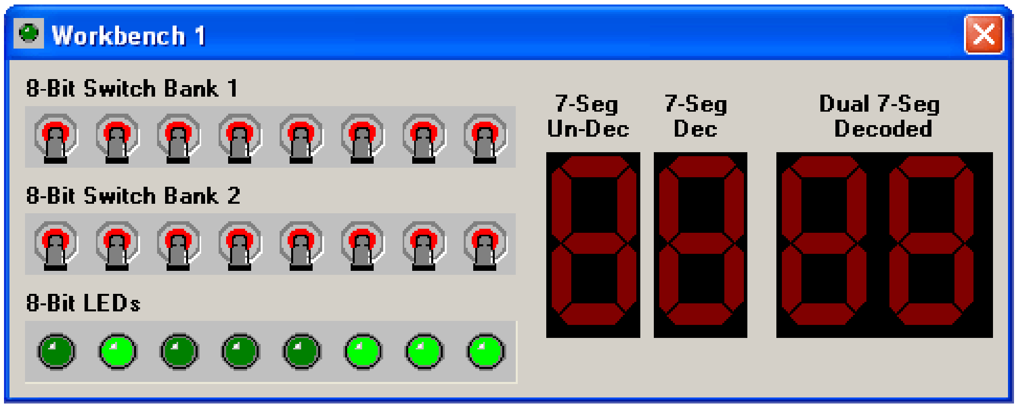
Port Type	**Port Address**	**I/O Device**
Input	$F008	Qwerty Keyboard
Output	$F020	Workbench’s 8-bit LED display
Output	$F031	Main DIY Calculator Display

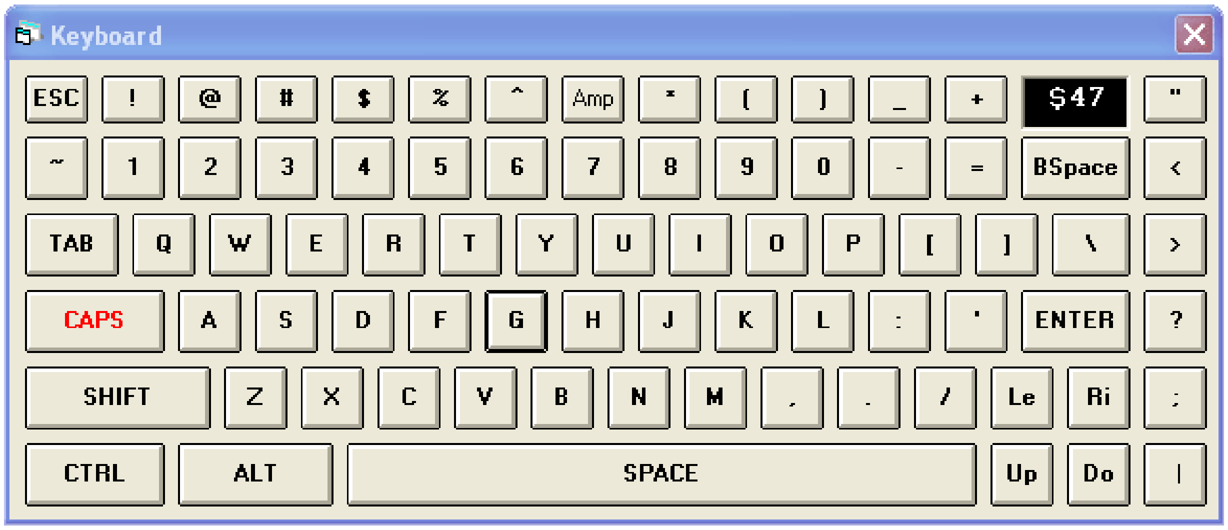
A simple test program (written in the DIY Calculator’s assembly language) to read from the QWERTY keyboard and to write to the workbench’s 8-bit LED display would be as follows:

```
QWERTY:    .EQU   $F008     # Input port connected to QWERTY keyboard
LED8BIT:   .EQU   $F020     # 8-bit LED Display on workbench

           .ORG   $4000     # Set program origin
LOOP:       LDA   [QWERTY]  # Read from QWERTY Keyboard
            JZ    [LOOP]    # Jump if the value is zero
            STA   [LED8BIT] # Write to 8-bit LED Display
            JMP   [LOOP]    # Do it all again
           .END
```

This test program simply loops around loading the value currently stored in the QWERTY keyboard's latch into the CPU's accumulator. It then uses a JZ ("jump if zero") instruction to check to see if this value is $00 (all zeros), in which case it knows that no key has been pressed, so it jumps back to read a new value from the keyboard. When a key is pressed, its non-zero value causes the JZ instruction to fail, in which case the program copies the value in the accumulator to the output port driving the 8-bit LEDs on the workbench.

Assume that we've used the **Tools > Terminal** and **Tools > Workbench #1** commands to launch these new utilities. Also assume that we've powered-up the DIY Calculator (using the On/Off button on the calculator's front panel) and assembled, loaded, and run the above program. In this case, clicking the 'G' button on the keyboard will cause the binary pattern 01000111 (that's $47, which is the ASCII code for 'G') to appear on the LEDs as shown below:




Keyboard
ESC ! @ # $ % ^ Amp * ( ) _ + $47 "
~ 1 2 3 4 5 6 7 8 9 0 - = BSpace <
TAB Q W E R T Y U I O P [ ] \ >
CAPS A S D F G H J K L : ' ENTER ?
SHIFT Z X C V B N M , . / Le Ri ;
CTRL ALT SPACE Up Do |

Another simple test program to read from the QWERTY keyboard and write to the DIY Calculator's main display would be as follows:

```
CLRCODE:  .EQU   $10        # Code to clear the main display
QWERTY:   .EQU   $F008      # Input port connected to QWERTY keyboard
MAINDISP: .EQU   $F031      # Output port driving the main display

          .ORG   $4000      # Set program origin
           LDA   CLRCODE    # Load accumulator with clear code
           STA   [MAINDISP] # Use this code to clear the main display
LOOP:      LDA   [QWERTY]   # Read from QWERTY Keyboard
           JZ    [LOOP]     # Jump if the value is zero
           STA   [MAINDISP] # Write this character to the main display
           JMP   [LOOP]     # Do it all again
          .END
```

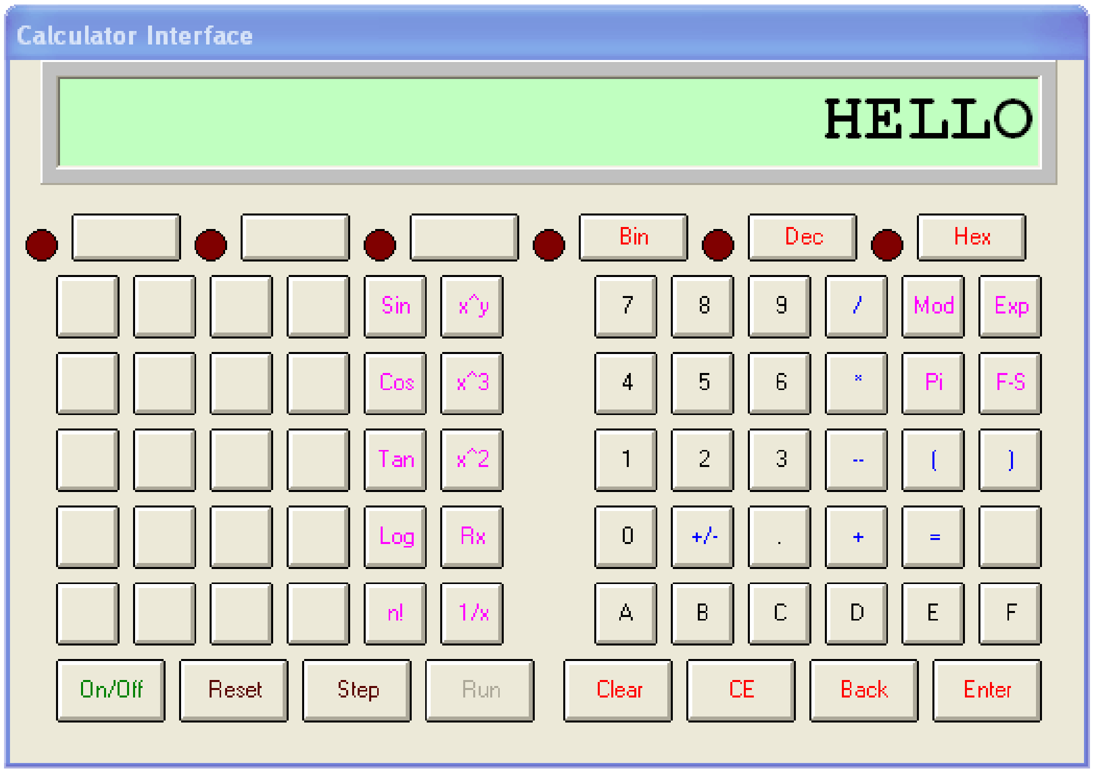
This program first clears the main display. Next, it loops around loading the value currently stored in the QWERTY keyboard's latch into the CPU's accumulator. As before, we use a `JZ` ("jump if zero") instruction to check to see if this value is $00 (all zeros), in which case we know that no key has been pressed, so we jump back to read a new value from the keyboard. When a key is pressed, its non-zero value causes the `JZ` instruction to fail, in which case the program copies the value in the accumulator to the output port driving DIY Calculator's main display.

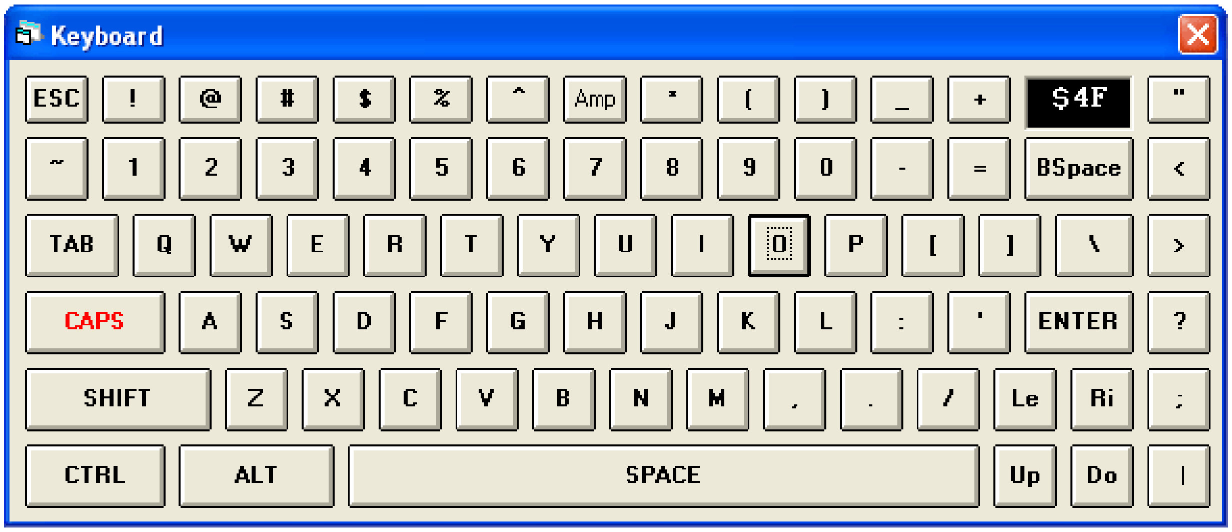
Assume that we've used the **Tools > Terminal** command to launch the QWERTY keyboard. Also assume that we've powered-up the DIY Calculator (using the On/Off button on the calculator's front panel) and assembled, loaded, and run the above program. In this case, clicking a sequence of characters such as 'H', 'E', 'L', 'L', 'O' will cause the corresponding characters to appear on the DIY Calculator's main display as shown on the following page.

Finally, observe the <Bspace> ("Backspace"), <Enter>, <Le> ("Left"), <Ri> ("Right"), <Up> ("Up"), and <Do> ("Down") keys on the QWERTY keyboard. These keys generate some non-standard codes as follows:

Key	**Code**	**Description**
<Bspace>	$04	Backspace
<Enter>	$05	Enter / New Line
<Up>	$07	Up arrow / cursor up
<Do>	$08	Down arrow / cursor down
<Ri>	$09	Right arrow / cursor right
<Le>	$0A	Left arrow / cursor left

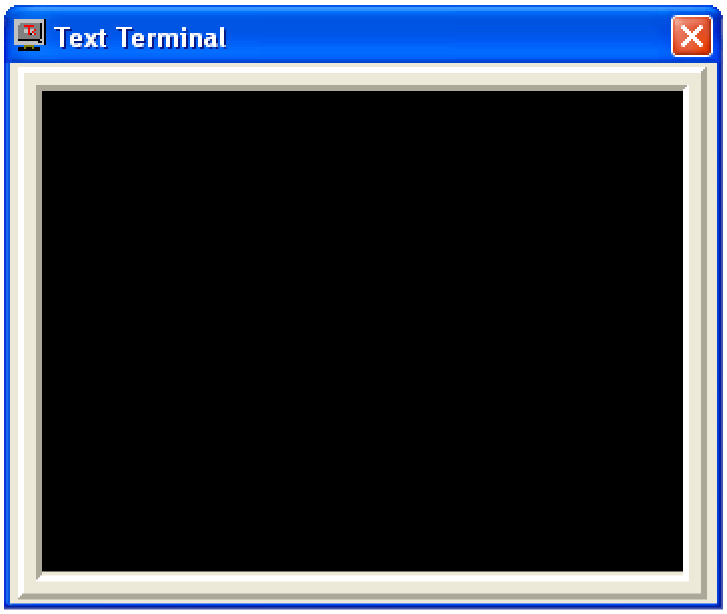
We will return to consider these keys and codes as part of the console discussions later in this document.


Calculator Interface
HELLO
Bin
Dec
Hex
Sin
x^y
7
8
9
/
Mod
Exp
Cos
x^3
4
5
6
*
Pi
F-S
Tan
x^2
1
2
3
--
(
)
Log
Rx
0
+/-
.
+
=
n!
1/x
A
B
C
D
E
F
On/Off
Reset
Step
Run
Clear
CE
Back
Enter

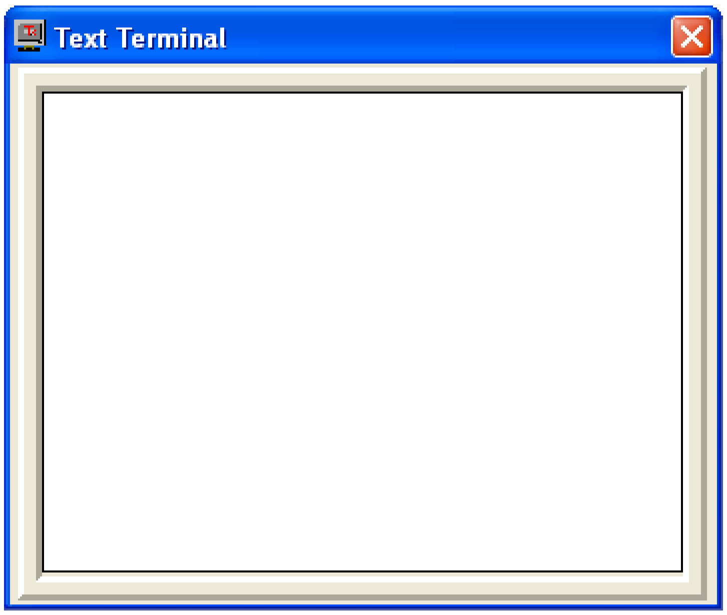

Keyboard
ESC
!
@
#
$
%
^
Amp
*
(
)
_
+
$4F
"
~
1
2
3
4
5
6
7
8
9
0
-
=
BSpace
<
TAB
Q
W
E
R
T
Y
U
I
O
P
[
]
\
>
CAPS
A
S
D
F
G
H
J
K
L
:
'
ENTER
?
SHIFT
Z
X
C
V
B
N
M
,
.
/
Le
Ri
;
CTRL
ALT
SPACE
Up
Do
|

The Console Device

As was previously notes, in addition to the QWERTY keyboard, using the **Tools > Terminal** command also results in a simple console device appearing in the DIY Calculator's main window. Assuming that the DIY Calculator is powered down, the console will initially have a black background:



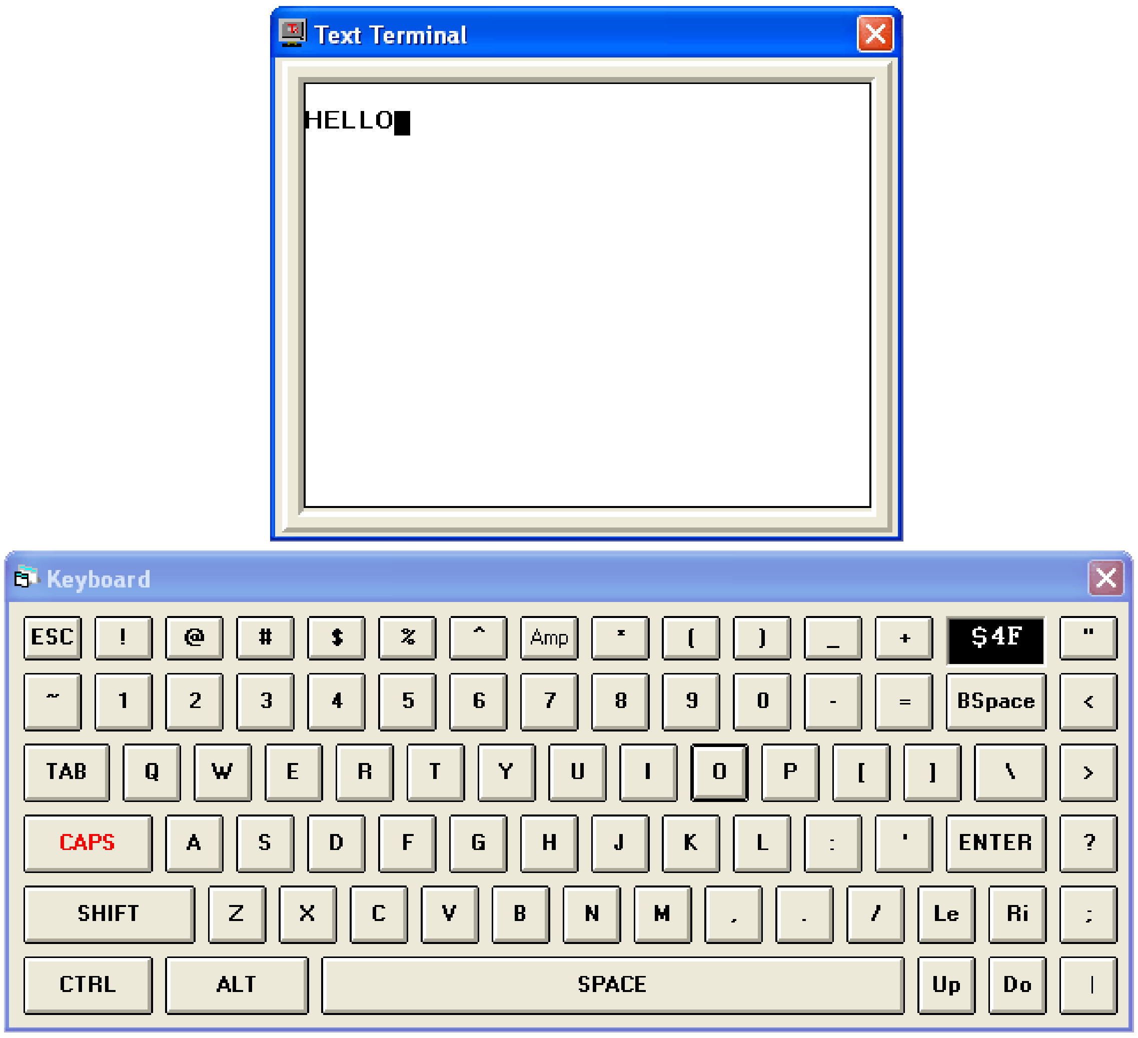
When the DIY Calculator is powered up (by means of its On/Off button), the console also powers up, which is reflected by its screen changing to white:



The console is driven by the DIY Calculator's output port at address $F028. The easiest way to understand how this works is to consider the simple test program shown below:

```
QWERTY:   .EQU   $F008     # Input port connected to QWERTY keyboard
CONSOLE:  .EQU   $F028     # Output port driving console device

          .ORG   $4000     # Set program origin
LOOP:      LDA   [QWERTY]  # Read from QWERTY Keyboard
           JZ    [LOOP]    # Jump if the value is zero
           STA   [CONSOLE] # Write character to console device
           JMP   [LOOP]    # Do it all again
          .END
```

This is exactly the same program we've used earlier in this paper, except that this time we're reading from the QWERTY keyboard and writing to the output port driving the console device. Assume that we've used the **Tools > Terminal** command to launch the QWERTY keyboard and console. Also assume that we've powered-up the DIY Calculator (using the On/Off button on the calculator's front panel) and assembled, loaded, and run the above program. In this case, clicking a sequence of characters such as 'H', 'E', 'L', 'L', 'O' will cause the corresponding characters to appear on the console:


Text Terminal
Keyboard
ESC ! @ # $ % ^ Amp * ( ) _ + $4F "
~ 1 2 3 4 5 6 7 8 9 0 - = BSpace <
TAB Q W E R T Y U I O P [ ] \ >
CAPS A S D F G H J K L : ' ENTER ?
SHIFT Z X C V B N M , . / Le Ri ;
CTRL ALT SPACE Up Do |

```
HELLO
```

In addition to the standard ASCII codes (as noted earlier in this document), the console also understands some special control codes as follows:

Code	Description
$00	Null code, doesn’t do anything
$01	Reserved for future use (doesn’t do anything at the moment)
$02	Clear the screen
$03	Place the cursor in its home position (row 0, column 0)
$04	Backspace
$05	New Line
$06	Beep the bell
$07	Up (move the cursor up one row)
$08	Down (move the cursor down one row)
$09	Right (move the cursor right one column)
$0A	Left (move the cursor left one column)

It’s very important to understand that the console and QWERTY keyboard are two completely separate devices. That is, it’s possible to drive the console using a program that in no way interacts with the QWERTY keyboard.

Having said this, it will very often be the case that the console and QWERTY keyboard will be used in conjunction with each other. This is why, as we previously noted, some of the keys on the QWERTY keyboard generate special codes as follows:

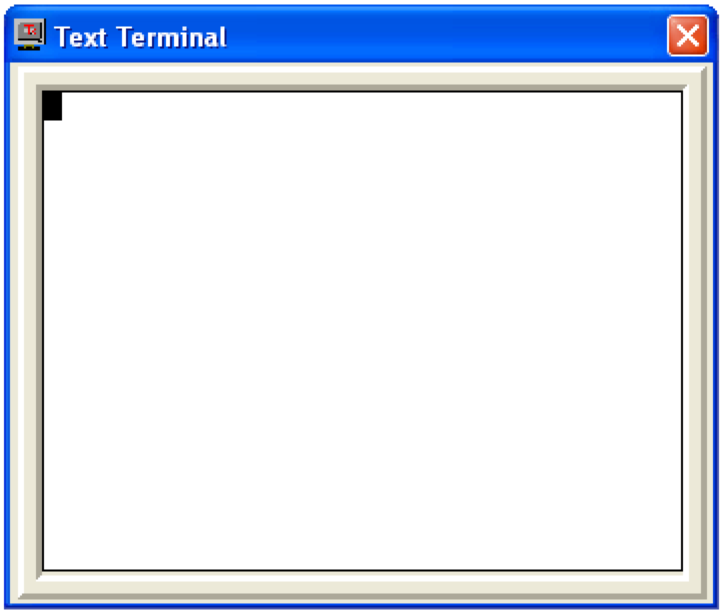
Key	Code	Description
<Bspace>	$04	Backspace
<Enter>	$05	Enter / New Line
<Up>	$07	Up arrow / cursor up
<Do>	$08	Down arrow / cursor down
<Ri>	$09	Right arrow / cursor right
<Le>	$0A	Left arrow / cursor left

Observe how these codes map onto those of the console. In order to gain a good understanding as to how all of this hangs together, we first need to note that the console comprises an array of 16 rows of characters (numbered from 0 at the top to 15 at the bottom) and 32 columns of characters (numbered from 0 at the left to 31 at the right). Now, consider the following actions and their responses (this assumes that the program described above is running):

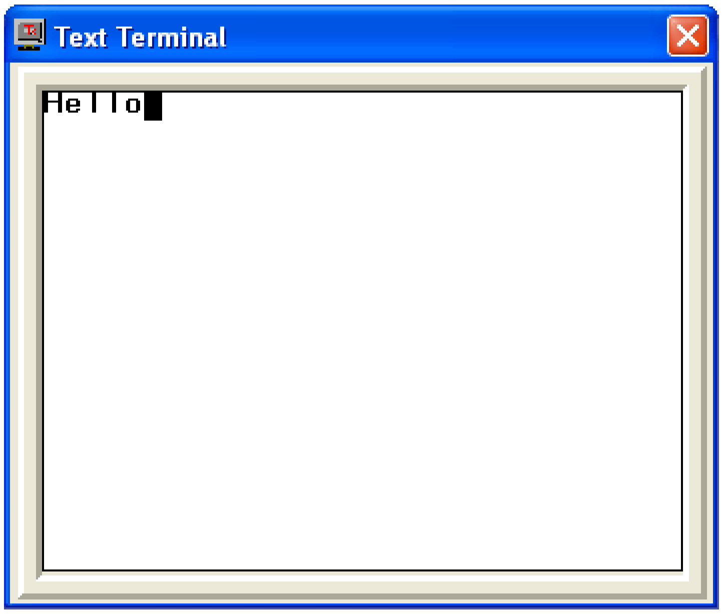
First, click the <Ctrl> key on the QWERTY keyboard to make it active (red annotation text) and then click the ‘B’ key. From our earlier discussions on the keyboard, we know that the code associated with the combination of the <Ctrl> and ‘B’ keys is $02. When our program writes this code to the console, the console will understand this as an instruction to clear the screen.

Now, click the ‘C’ key. Once again, from our earlier discussions on the keyboard, we know that the code associated with the combination of the <Ctrl> and ‘C’ keys is $03. When our program writes this code to the console, the console will understand it to be an instruction to place the cursor in its home position, which is in the top left-hand corner of the screen at row = 0 and column = 0 as shown in the image at the top of the next page.

Observe that the cursor appears to be a solid black rectangle in this picture. In fact, the pixels (picture elements) forming the cursor are each the XOR of the pixels forming the character in that position. As all of the characters currently forming the screen are blank space characters (each with an ASCII code of $20) whose pixels are all white, the XOR of these pixels results in a rectangle that is all black. However, this won’t always be the case as we shall soon discover.

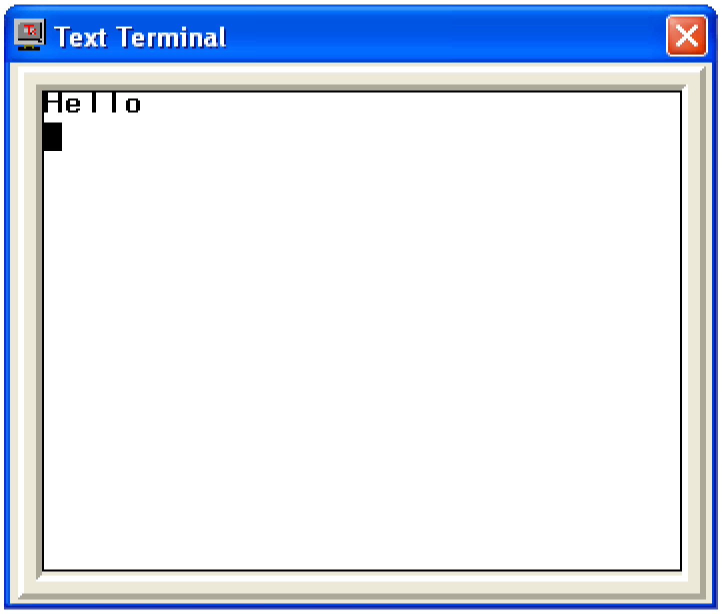


Now, click the <Ctrl> key to make it inactive (block annotation text). Next, click the ‘H’ key, then click the <Caps> key to make it inactive, then click the ‘e’, ‘l’, ‘l’, and ‘o’ keys. Observe that every time you write an ASCII character to the console, that character appears at the current cursor position and the cursor itself moves one place to the right:



```
Hello
```

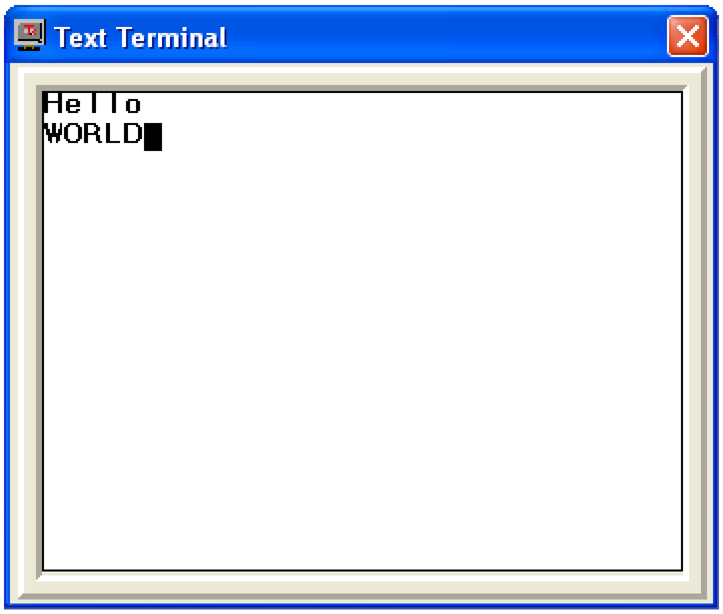
If we were to keep on entering text, then when the cursor eventually reached the right-hand-side of the screen, it would automatically wrap around to the next line. Alternatively, if we were to now click the <Enter> key on the keyboard, this has an associated code of $05, and when our program sends this to the console, it will understand this to be a “New Line” code; that is, an instruction to move the cursor to the beginning of the next line as shown in the following image:



```
Hello
```

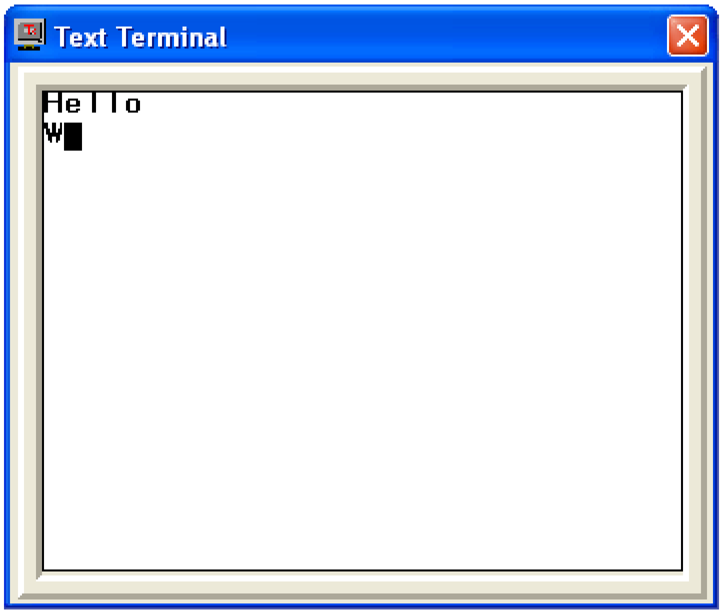
Note that, if the cursor were to be on the console's bottom row when a "New Line" code was sent to the console, this will cause the contents of the screen to move up by one row (the top row conceptually "falls off the end") and the cursor to appear in the bottom left-hand corner of the screen (row = 15, column = 0).

Now, suppose that we click the <Caps> key followed by the 'W', 'O', 'R', 'L', and 'D' keys, which would result in the console appearing as follows:



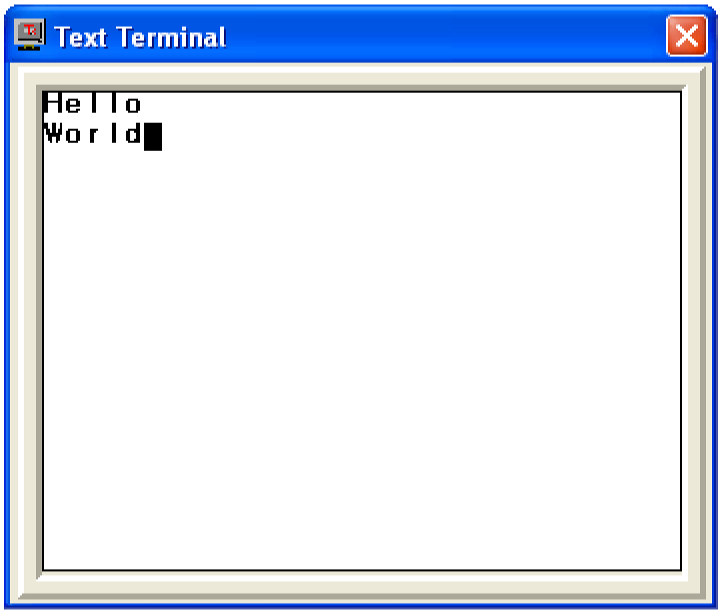
```
Hello
WORLD
```

Arrgggh! We really intended for only the 'W' to be in uppercase and for the remaining characters to be in lowercase. Well, "No Worries," as they say "Down Under" in Australia and New Zealand. Clicking the <Bspace> ("Backspace") key on the keyboard generates a code of $04, which the console understands as an instruction to move the cursor one place to the left and to then delete the character underneath the cursor and replace that character with a space. So if we click this key four times, the console will end up looking as follows:



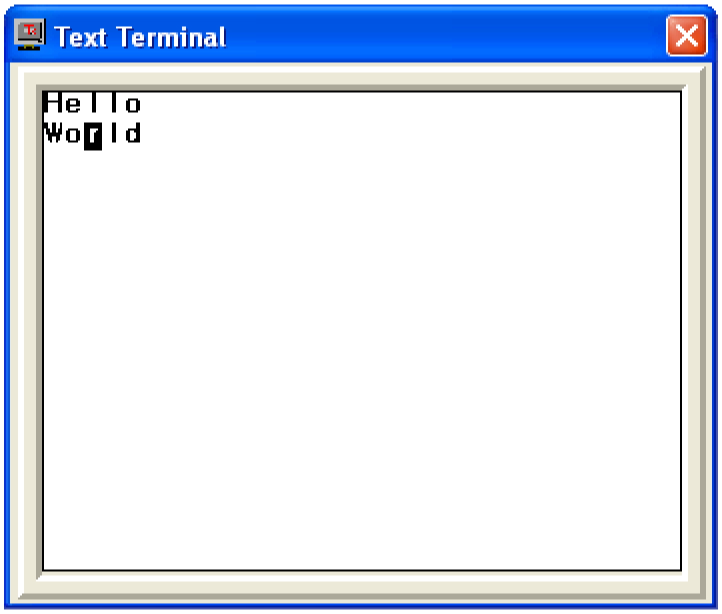
```
Hello
W█
```

Now, click the <Caps> key to make it inactive, and then click the ‘o’, ‘r’, ‘l’, and ‘d’ keys, which will leave the console looking as follows:



```
Hello
World█
```

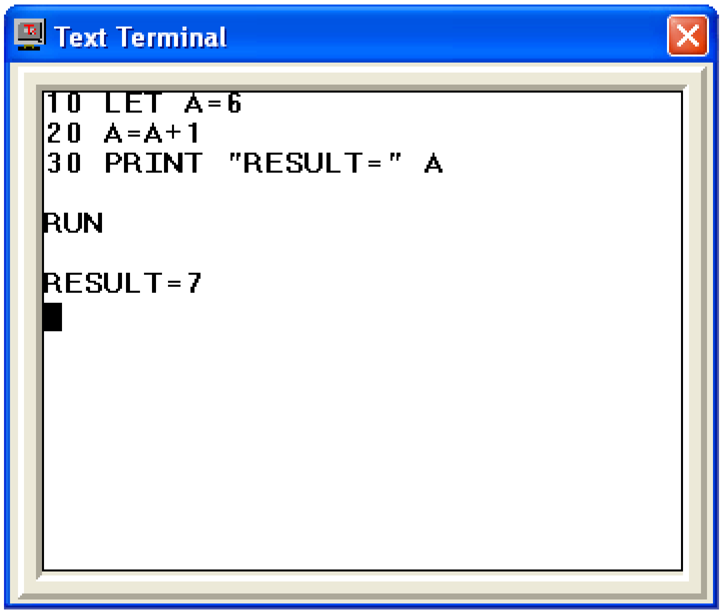
Now, just for giggles and grins, click the <Li> (“left”) button on the keyboard three times. Each click generates a code of $0A, which the console understands to be an instruction to move the cursor one place to the left. This will leave the cursor over the ‘r’ character as shown in the following image:



```
Hello
World
```

As we previously discussed, each of the pixels forming the cursor are actually the inverse (XORs) of the corresponding pixels forming the character at that position. This results in any white pixels turning black and vice versa. Use the <up> ("Up"), <Do> ("Down"), <Le> ("Left"), and <Ri> ("Right") cursor control keys on the keyboard to move the cursor around the console. Observe that the cursor will automatically stop when it reaches one of the edges.

The console can be used for a variety of tasks. For example, we could use it in conjunction with the workbench (e.g. read the 8-bit switches and write their value to the console) and the main calculator front panel (e.g. display a list of the actions being performed by a calculator program). A more advanced application would be to use the keyboard and console as the I/O devices for a BASIC interpreter. This interpreter, which would be written in the DIY Calculator's assembly language, would allow us to create and run programs looking something like the following:

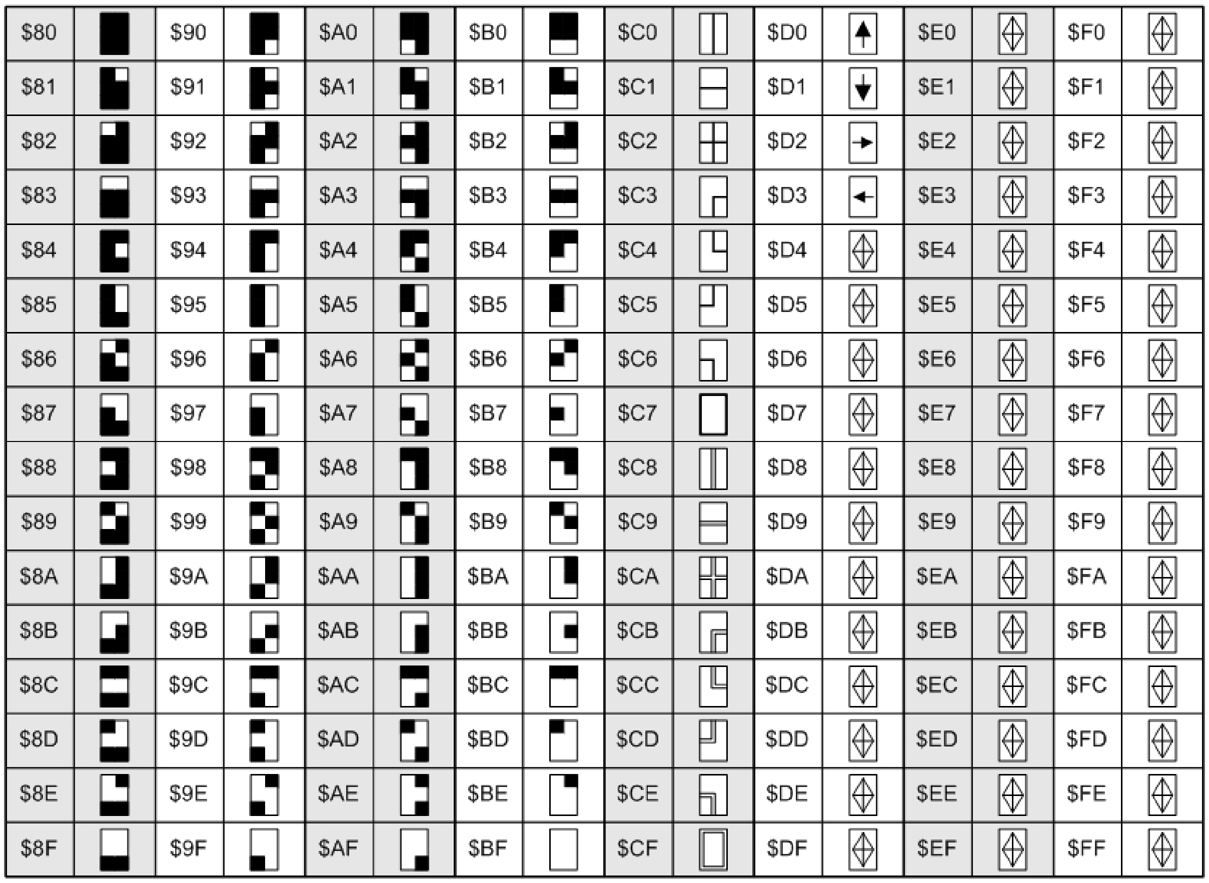


```
10 LET A=6
20 A=A+1
30 PRINT "RESULT=" A

RUN

RESULT=7
```

Some ideas pertaining to the creation of a BASIC interpreter are presented on the “More Tools” page of the DIY Calculator website (**www.DIYCalculator.com**). When such an interpreter is eventually created, it could be used to implement a wide variety of tasks, including a calculator program to run on the DIY Calculator.

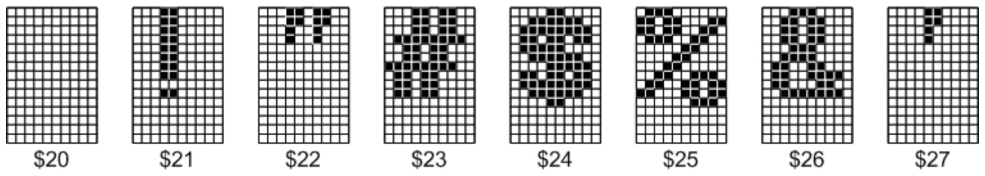
Last but certainly not least, as discussed in our book *How Computers Do Math* (ISBN: 0471732788), ASCII is a 7-bit code, which uses only the values $00 through $7F. But the DIY calculator has an 8-bit wide memory and data bus, which means that we’ve still got codes $80 through $FF to play with.

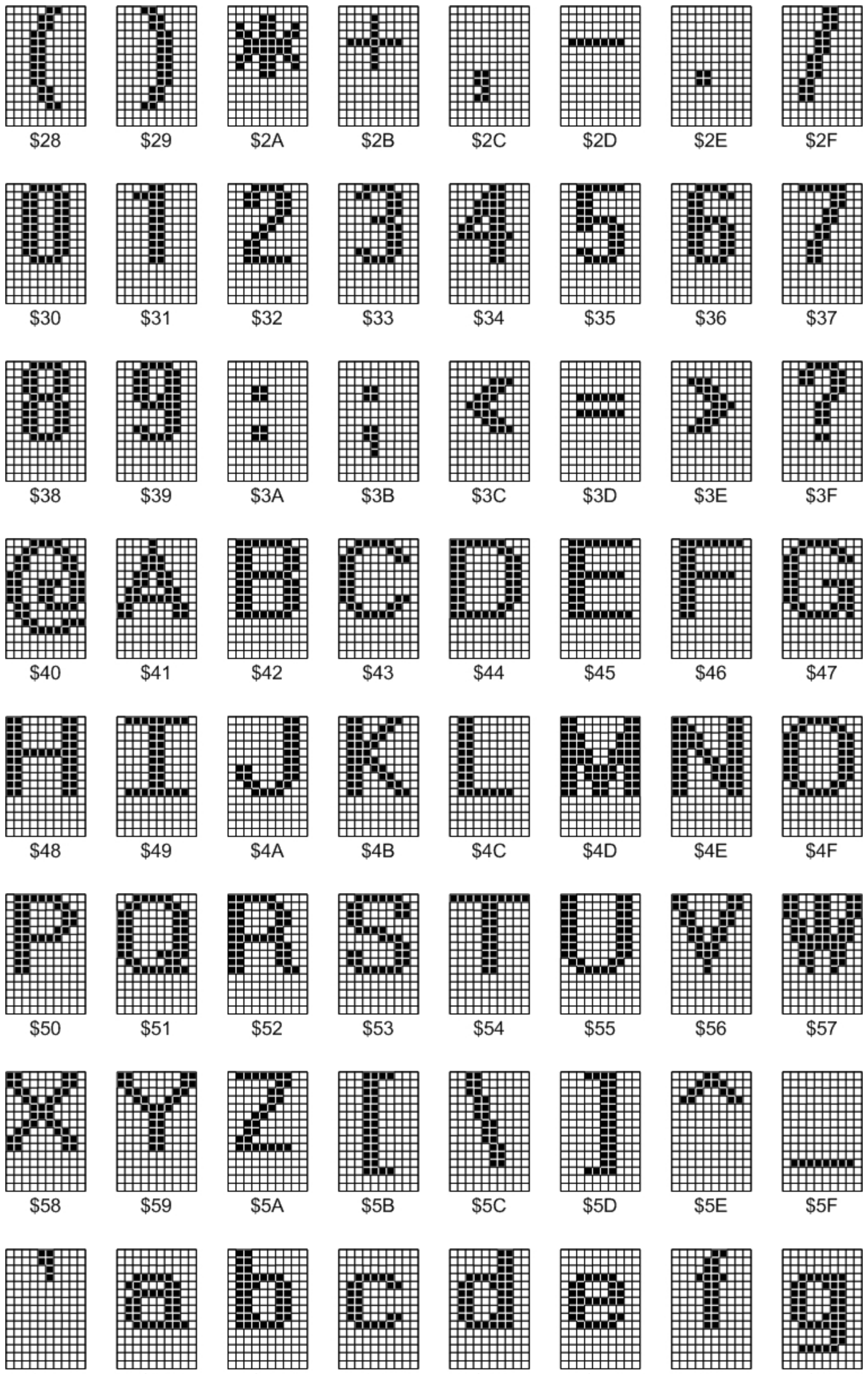
Many of the early computers employed these spare codes to implement simple “chunky graphic” characters. For your delectation and delight, the DIY Calculator’s console can accept and display our interpretation of these characters as follows:

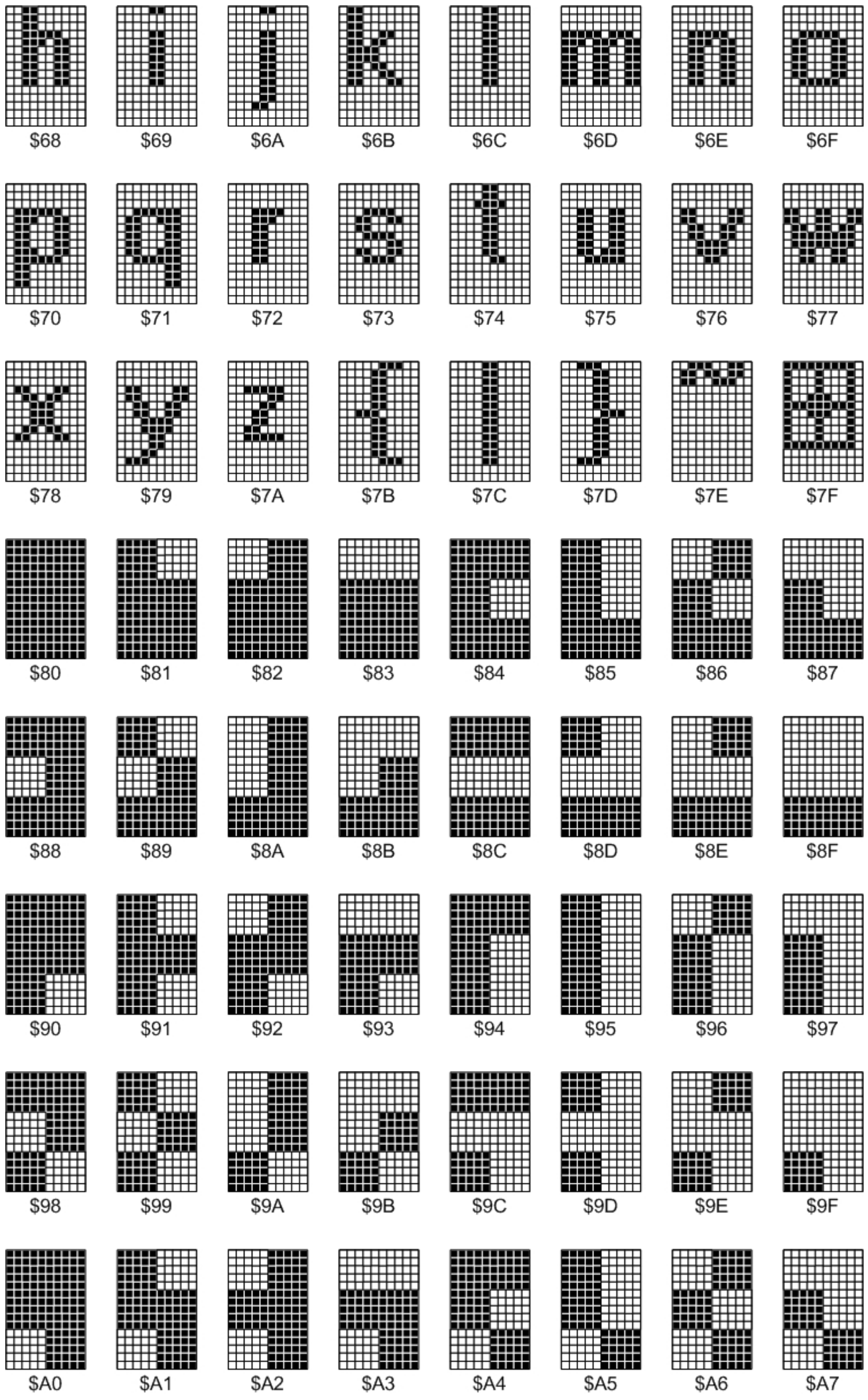


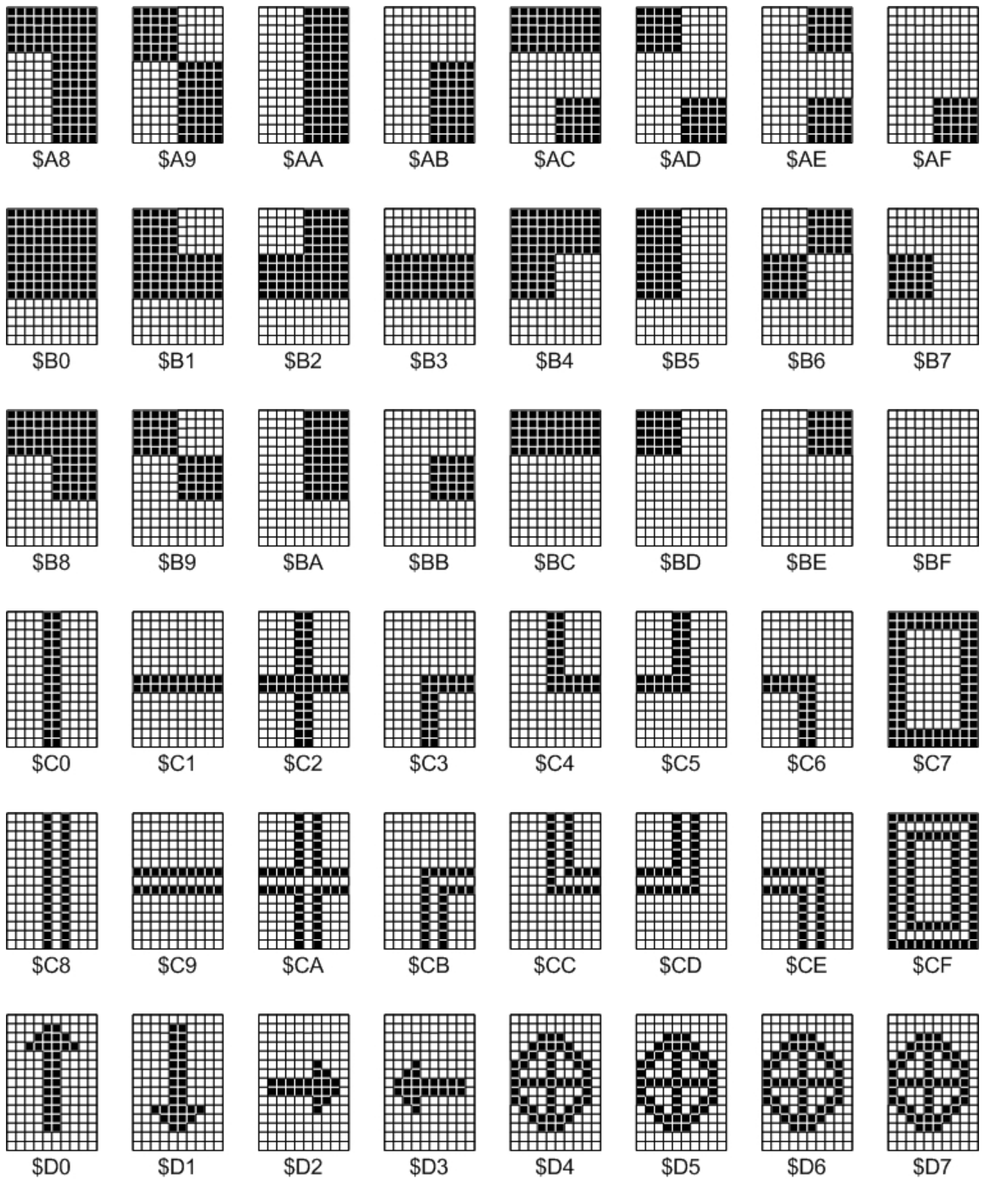
$80		$90		$A0		$B0		$C0		$D0		$E0		$F0	
$81		$91		$A1		$B1		$C1		$D1		$E1		$F1	
$82		$92		$A2		$B2		$C2		$D2		$E2		$F2	
$83		$93		$A3		$B3		$C3		$D3		$E3		$F3	
$84		$94		$A4		$B4		$C4		$D4		$E4		$F4	
$85		$95		$A5		$B5		$C5		$D5		$E5		$F5	
$86		$96		$A6		$B6		$C6		$D6		$E6		$F6	
$87		$97		$A7		$B7		$C7		$D7		$E7		$F7	
$88		$98		$A8		$B8		$C8		$D8		$E8		$F8	
$89		$99		$A9		$B9		$C9		$D9		$E9		$F9	
$8A		$9A		$AA		$BA		$CA		$DA		$EA		$FA	
$8B		$9B		$AB		$BB		$CB		$DB		$EB		$FB	
$8C		$9C		$AC		$BC		$CC		$DC		$EC		$FC	
$8D		$9D		$AD		$BD		$CD		$DD		$ED		$FD	
$8E		$9E		$AE		$BE		$CE		$DE		$EE		$FE	
$8F		$9F		$AF		$BF		$CF		$DF		$EF		$FF	

And, just in case you are interested, each of the characters displayed by the console is formed from a 10 x 15 array of pixels. The pixel patterns for both the ASCII and chunky graphics characters are as follows:




$28
$29
$2A
$2B
$2C
$2D
$2E
$2F
$30
$31
$32
$33
$34
$35
$36
$37
$38
$39
$3A
$3B
$3C
$3D
$3E
$3F
$40
$41
$42
$43
$44
$45
$46
$47
$48
$49
$4A
$4B
$4C
$4D
$4E
$4F
$50
$51
$52
$53
$54
$55
$56
$57
$58
$59
$5A
$5B
$5C
$5D
$5E
$5F
$60
$61
$62
$63
$64
$65
$66
$67


$68
$69
$6A
$6B
$6C
$6D
$6E
$6F
$70
$71
$72
$73
$74
$75
$76
$77
$78
$79
$7A
$7B
$7C
$7D
$7E
$7F
$80
$81
$82
$83
$84
$85
$86
$87
$88
$89
$8A
$8B
$8C
$8D
$8E
$8F
$90
$91
$92
$93
$94
$95
$96
$97
$98
$99
$9A
$9B
$9C
$9D
$9E
$9F
$A0
$A1
$A2
$A3
$A4
$A5
$A6
$A7



Note that the remaining codes $D8 through $FF have identical patterns to those shown for codes $D4 through $D7 above. We used this distinctive pattern to distinguish these characters from spaces in order to make them easy to spot should they appear unexpectedly (due to some programming error) on your console display.