Special Project

PhizzyB COMPUTERS

Understanding Computers

By Clive "Max" Maxfield and Alvin Brown

Bonus Article: Creating an Event Timer

This bonus article augments Part 4 of our *PhizzyB Computers* series, which appeared in the February 1999 issue of *EPE Online*. These articles are of interest to anyone who wants to know how computers perform their magic, because they use a unique mix of hardware and software to explain how computers work in a fun and interesting way.

Hello there and welcome to this special bonus article. This augments Part 4 of our *PhizzyB* series, which appeared in the February 1999 issue of *EPE Online*. This bonus article describes how to use the *PhizzyB* to create an event timer and to display the result on the *liquid crystal display (LCD)* output device described in Part 3 of this series (which appeared in the January 1999 issue of *EPE Online*). (Note that we're assuming that you've already run the LCD display and set the contrast as described in the Part 3 article).

Quite apart from anything else, this bonus article also introduces some general-purpose mathematical subroutines that are supplied with the *PhizzyB Simulator*.

DRIVING THE LCD

As we know from Part 3, the *PhizzyB's* LCD module supports two rows of sixteen characters. This module has 14 pins, but only 11 are of interest to us here (Table 1).

There are three control pins (RS, R/W, and E). The RS ("*register select*") pin is used to indicate whether wish to pass commands or data to the LCD (0 = command, 1 = data). The R/W ("*read/write*") pin indicates whether we wish to write to the LCD or read from it (0 = write, 1 = read). Meanwhile, the E ("*enable*") pin is used to initiate the actual transfer of the commands or data. For our purposes we will always commence with E = 0, and then "pulse" or "strobe" this signal by setting it to 1 and returning it to 0.

Pin	Name	Function	Signal
4	RS	Register Select	OP3
5	R/W	Read/Write	OP2
6	E	Enable	OP1
7	D0	Data bit 0	--
8	D1	Data bit 1	--
9	D2	Data bit 2	--
10	D3	Data bit 3	--
11	D4	Data bit 4	OP4
12	D5	Data bit 5	OP5
13	D6	Data bit 6	OP6
14	D7	Data bit 7	OP7

Table 1: LCD Pin connections.

The LCD module is designed to support both 8-bit and 4-bit interfaces. In the case of an 8-bit interface, the device driving this module would be capable of passing data to all eight data pins and have enough signals free to drive the control pins. But each of the *PhizzyB's* external output ports only have eight output bits called OP0 to OP7, which means that we will have to use the 4-bit interface model. In this case, four of the *PhizzyB's* output bits (OP7 through OP4) are used to pass data to four of the LCD module's data bits (D7 through D4), while the remaining output bits are used to drive the LCD's control signals.

The way this works is that instead of passing our control and data information to the LCD in 8-bit bytes, we split each byte into two 4-bit nybbles which are then passed to the LCD one after the other (most-significant followed by least-significant).

Note that the LCD's D0 to D3 pins are essentially unused, so we could theoretically leave them unconnected, but it's better practice to tie them off to 0V via pull-down resistors (as described in the Part 3 construction article). Also note that the OP0 pin on the *PhizzyB's* output port is unused.

Listing 1

```
### Constant declarations
SWITCHES: .EQU     $F011       # Switch i/p port
BARGRAPH: .EQU     $F030       # 8-bit LED o/p port
LCD:      .EQU     $F031       # LCD o/p port

### Main program initialization
          .ORG     $4000       # Program origin
           BLDSP   $4FFF       # Load stack pointer
           JSR     [INITLCD]   # Call LCD init

### Start of main program body
           JMP     [$0000]     # Soft reset

### Start of interrupt service routines

### Start of subroutines

### General-purpose temporary locations
NUMDIG:   .BYTE                # No. of digits
CNTFLAG:  .BYTE                # Counter flag
COUNTER:  .2BYTE               # 2-byte counter

          .END
```

EXPERIMENT 1

Initializing the LCD display

What we're going to do is to build a program layer by layer. So our first task will be to start the *PhizzyB Simulator*, activate the assembler, and enter the skeleton program as shown in Listing 1. Note that this listing assumes the 8-bit interrupt switch device is connected to the input port at address $F011, and that the LCD device is connected to the output port at address $F031 (plus we're going to use the 8-bit LED bargraph display at address $F030).

Also note the .BYTE and .2BYTE directives used to reserve memory locations at the end of the program. As we progress we'll be using these to store chunks of information on a temporary basis. The *PhizzyB's* assembly language is discussed in Appendix D of *The Official Beboputer Microprocessor Databook*, and this Appendix is also supplied with your PhizzyB Simulator (check the simulator's online help for more details).

Once you've entered the skeleton program shown in Listing 1, save it out as *cexp1.asm*. Note that this listing is just a skeleton program upon which we're going to hang our real program.

The STROBE routine

OK, now let's take our skeleton program and start to build something useful. One thing we're going to be doing many times is strobing the LCD's E ("*enable*") signal; that is, making it pulse from an initial value of 0 to a 1 and back again. The important thing is that we want to do this without disturbing any of the other signals driving the LCD. In order to do this, insert the STROBE ("*strobe enable signal*") subroutine shown in Listing 2 just below the *"Start of subroutines"* comment in the skeleton file. (The stack and subroutines were introduced in Part 3 of the *PhizzyB* series appearing in the January 1999 issue of *EPE Online*.)

Remember that the E ("*enable*") signal is driven by bit 1 of the *PhizzyB's* output port. So the first thing we do is to OR whatever value is in the accumulator with a binary value of 00000010, which sets bit 1 of the accumulator to a logic 1 value (and leaves the remaining bits untouched). Next we store the contents of the accumulator to the output port driving the LCD. We then AND the current value in the accumulator with a binary value of 11111101, which forces bit 1 to a logic 0 value (leaving the remaining bits untouched), and we write *this* value to the LCD. (The AND, OR, and XOR logical instructions were introduced in Part 2 of the *PhizzyB* series appearing in the December 1998 issue of *EPE Online*.)

The WCOMMAND routine

Next we need to create the WCOMMAND ("*write command*") subroutine shown in Listing 3, which we'll use to

Listing 2

```
### Start of "strobe enable signal" subroutine
STROBE:   OR      %00000010   # Set E pin to 1
          STA     [LCD]       # Write to the LCD
          AND     %11111101   # Reset E to 0
          STA     [LCD]       # Write to the LCD
          RTS                 # Exit subroutine
### End of "strobe enable signal" subroutine
```

Listing 3

```
### Start of "write command" subroutine.
WCOMMAND: PUSHA                 # Copy ACC to stack
          AND     %11110000     # Clear LS 4 bits
          STA     [LCD]         # Present to LCD
          JSR     [STROBE]      # Call STROBE routine
          POPA                  # Get ACC from stack
          SHL                   # Shift left 1 bit
          SHL                   # Shift left 1 bit
          SHL                   # Shift left 1 bit
          SHL                   # Shift left 1 bit
          STA     [LCD]         # Present to LCD
          JSR     [STROBE]      # Call STROBE routine
          RTS                   # Exit subroutine
### End of "write command" subroutine.
```

write command codes to the LCD. As you'll see, this new subroutine will call the STROBE subroutine we just created. This idea of one subroutine calling another may be referred to as "nested subroutines." It really doesn't matter in what order you insert these subroutines in the main program, but just so we're all marching to the same drum beat, let's place this new routine just after the end of the STROBE subroutine.

As you will come to see, prior to calling this WCOMMAND subroutine we will load the accumulator with an 8-bit command code. However, as we've already discussed, we're actually going to be using the LCD's 4-bit interface mode. This means that the WCOMMAND subroutine has to split the 8-bit command code in the accumulator into two 4-bit nybbles, and to write each nybble out in turn (the most-significant (MS) nybble followed by the least-significant (LS) nybble).

The first thing the WCOMMAND subroutine does is to use a PUSHA instruction to place a copy of the accumulator onto the stack (so that we can remember what this value was later). Next we AND the accumulator with a binary value of 11110000, which keeps whatever data was in bits 7, 6, 5, and 4 of the accumulator and clears bits 3, 2, 1, and 0 to zero. Next we present this value to the LCD (remember that bits 3, 2, and 1 of the output port driving the LCD module are connected to signals RS, R/W, and E respectively), then we call our STROBE subroutine to pulse the E ("*enable*") signal.

Now we use a POPA instruction to retrieve the original copy of the accumulator back off the stack, and then we shift this value four bits to the left (each shift operation shifts a zero into the LS bit). Thus, following these four shifts we end up with the original LS nybble of our command code shifted into the MS nybble of the accumulator, at which point we present *this* value to the LCD display and again call our STROBE subroutine to pulse the E ("*enable*") signal.

The INITLCD routine

The fun stuff is coming … honest! But before we get there we still have to do a little ground work. The next thing we're going to create is an INITLCD ("*LCD initialization*") subroutine to fully initialize the LCD module. Enter this subroutine (which is shown in Listing 4) just after your WCOMMAND routine.

Listing 4

```
### Start of main "LCD initialization" subroutine.
INITLCD:  LDA     %00110000   # 8-bit mode
          STA     [LCD]       # Present MS nibble
          JSR     [STROBE]    # Call STROBE
          JSR     [STROBE]    # Call STROBE
          JSR     [STROBE]    # Call STROBE

          ## Now send 4-bit init code (MS nibble)
          LDA     %00100000   # 4-bit mode
          STA     [LCD]       # Present MS nibble
          JSR     [STROBE]    # Call STROBE

          ## Now set up rest of display
          LDA     %00101000   # 4-bit, 2-line, 5x7
          JSR     [WCOMMAND]  # Call WRITE COMMAND
          LDA     %00001000   # Disp/Cur/Blink OFF
          JSR     [WCOMMAND]  # Call WRITE COMMAND
          LDA     %00001111   # Disp/Cur/Blink ON
          JSR     [WCOMMAND]  # Call WRITE COMMAND
          LDA     %00000110   # Inc mode, d-shift off
          JSR     [WCOMMAND]  # Call WRITE COMMAND
          LDA     %00000001   # Clear disp/home cur
          JSR     [WCOMMAND]  # Call WRITE COMMAND
          RTS                 # Exit subroutine
### End of main "LCD initialization" subroutine.
```

In fact the data sheets for the LCD state that this module should self-initialize into a known good state, but this depends on certain power-up conditions being met. Also, like many engineers, we're wary of other people's promises of self-initialization delight (our hearts have been broken too many times). Thus, for our purposes we're going to assume that the LCD could power up into any random state, so our initialization is going to assume worst-case conditions and initialize the display from the ground up.

Based on this worst-case scenario, we don't know whether the LCD has powered up into its 8-bit or 4-bit modes. So the first thing our initialization routine does is to load the accumulator with a binary value of 00110000 and present this (or at least the MS nybble of it) to the LCD. Remember that the MS four output port bits are connected the LCD's MS four data bits, while the LCD's LS four data bits are tied to zero (and the *PhizzyB's* LS four output port bits drive the LCD's control signals).

From Table 2 we see that the 00110000 value we just presented to the LCD (0011 from the MS nybble of the accumulator and 0000 from the four LCD data bits ties to logic 0) instructs the display to enter its 8-bit interface mode. If the display were already in its 8-bit mode, then it would only be necessary to call our STROBE subroutine a single time. However, if the display had happened to power up in it's 4-bit mode, it would expect to see each command code presented as two nybbles, in which case we would have to call the STROBE subroutine twice.

But if the display did power up in its 4-bit mode and we called the STROBE routine twice, then we would have effectively written two 4-bit binary values of 0011 (that is 0011 followed by 0011 -- the MS byte of the accumulator twice -- which equals 00110011). This explains why we call the STROBE routine three times, because this way we know that whichever way it initially powered up, we are now absolutely sure that the display has finally received an 8-bit initialization code of 00110000.

Well we seem to have spent a lot of effort placing the display in its 8-bit mode, even though we know that we're actually going to use it in its 4-bit mode. But the key thing here is that we wanted to make sure that the display was in a very well known state. So the next step is to load the accumulator with a binary code of 00100000, present this to the LCD, and then call the STROBE routine to load the most-significant nybble into the LCD.

Just sending this nybble immediately places the display in its 4-bit mode. However, we now have a slight "gotcha," in that from Table 2 we see that we really need to send two nybbles for this command so as to also instruct the display to enter its 2-line and 5x7 dot character

Command	Binary D7	D6	D5	D4	D3	D2	D1	D0	Hex
Clear Display	0	0	0	0	0	0	0	1	01
Display & Cursor Home	0	0	0	0	0	0	1	x	02 or 03
Character Entry Mode	0	0	0	0	0	1	1 / D	S	04 to 07
Display On/Off & Cursor	0	0	0	0	1	D	U	B	08 to 0F
Display/Cursor Shift	0	0	0	1	D / C	R / L	x	x	10 to 1F
Function Set	0	0	1	8 / 4	2 / 1	10 / 7	x	x	20 to 3F
Set CGRAM Address	0	1	A	A	A	A	A	A	40 to 7F
Set Display Address	1	A	A	A	A	A	A	A	80 to FF

1 / D:	1=Increment*, 0=Decrement	R / L:	1=Right shift, 0=Left shift
S:	1=Display shift on, 0=Off*	8 / 4:	1=8-bit interface*, 0=4-bit interface
D:	1=Display on, 0=Off*	2 / 1:	1=2 line mode, 0=1 line mode*
U:	1=Cursor underline on, 0=Off*	10 / 7:	1=5x10 dot format, 0=5x7 dot format*
B:	1=Cursor blink on, 0=Off*		
D / C:	1=Display shift, 0=Cursor move	x = Don't care	* = Initialization settings

Table 2: LCD command codes.

modes. This means that we now need to load the accumulator with a binary value of 00101000, where we know (from Table 2) that the MS nybble will set the 4-bit mode (again), while the LS nybble will specify the 2 line and 5x7 dot character modes.

Don't panic! We know that wrapping your brain around this is a little convoluted. But once you've got the hang of things it's like solving a puzzle -- light dawns and you think *"so that's how it works."* Furthermore, the good news is that we've now initialized the display sufficiently that we can start to use our WCOMMAND subroutine. So the remainder of the INITLCD routine simply writes a sequence of command values to the display to set up various features like clearing the display and turning the flashing cursor on (compare each of these values to Table 2 and ensure that you fully understand what's happening).

Testing the initialization

Return to our original skeleton program in Listing 1 and consider the "*Main program initialization section*." At the end of this section we see a JSR ("*jump subroutine*") instruction, which calls our INITLCD routine. Following this in the main body of the program we see a JMP [$0000] instruction, which will cause the program to perform a "*soft reset*" (that is, to reset the *PhizzyB* under program control). However, resetting the *PhizzyB* won't affect the fact that the LCD has been initialized.

Assemble your program, download the resulting *caexp1.ram* file to the *PhizzyB*, and run the program. Observe that the *PhizzyB's* 7-segment displays flicker off while the program runs and back on again once the program has finished. Also note that the LCD module now shows the cursor flashing in the top left-hand character position, where it's poised to display a character as soon as we send it one.

EXPERIMENT 2:

Writing characters to the display

Now that we can initialize the LCD module we can start to have some fun, but there's just one small task we have to do first ...

The WDATA routine

Before we can start displaying characters, we need to create a WDATA ("*write data*") subroutine, which we can use to write data values to the LCD. Save your existing program out as *caexp2a.asm* and then insert the subroutine shown in Listing 5 between the end of the WCOMMAND routine and the beginning of the INITLCD routine.

Note that this routine is very similar to the WCOMMAND routine we created earlier. Prior to calling the WDATA routine we will load the accumulator with an 8-bit data value. But as before, due to the fact that we're using the LCD's 4-bit interface mode, the WDATA routine has to split the 8-bit data value in the accumulator into two 4-bit nybbles, and to write each nybble out in turn (the most-significant (MS) nybble followed by the least-significant (LS) nybble).

The first thing the WDATA routine does is to use a PUSHA instruction to place a copy of the accumulator onto the stack (so that we can remember what this value was later). Next we AND the accumulator with a binary value of 11110000, which keeps whatever was in bits 7, 6, 5, and 4 of the accumulator and clears bits 3, 2, 1, and 0 to zero. Unlike the WCOMMAND routine, we now OR the value in the accumulator with 00001000, which sets the LCD's RS ("*register select*") control signal to 1. This informs the LCD that we're writing a data value. Next

Listing 5

```
### Start of "write data" subroutine.
WDATA:    PUSHA               # Copy ACC to stack
          AND     %11110000   # Clear LS 4 bits
          OR      %00001000   # Set RS bit to 1
          STA     [LCD]       # Present to LCD
          JSR     [STROBE]    # CALL STROBE

          POPA                # Get ACC from stack
          SHL                 # Shift left 1 bit
          SHL                 # Shift left 1 bit
          SHL                 # Shift left 1 bit
          SHL                 # Shift left 1 bit
          OR      %00001000   # Set RS bit to 1
          STA     [LCD]       # Present to LCD
          JSR     [STROBE]    # Call STROBE
          RTS                 # Exit subroutine
### End of "write data" subroutine.
```

we present this value to the LCD, and then we call our STROBE subroutine to pulse the E ("*enable*") signal.

Now we use a POPA instruction to retrieve the original copy of the accumulator back off the stack, and then we shift this value four bits to the left (each shift operation shifts a zero into the LS bit). Thus, following these four shifts we end up with the LS nybble shifted into the MS nybble of the accumulator. Once again we OR the accumulator with 00001000 to set the RS control to 1, present the resulting value to the LCD display, and call our STROBE subroutine to pulse the E ("*enable*") signal.

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

Table 3: ASCII codes recognized by the LCD module.

Displaying a single character

Scroll your way to the "*Start of main program body*" comment towards the beginning of your program. This section currently contains a single JMP instruction that's used to perform a soft reset. Add an LDA and a JSR before the JMP, such that this section now looks like Listing 6.

Now assemble your program, download the resulting *caexp2a.ram* file to the *PhizzyB*, and run the program. Once again observe that the *PhizzyB's* 7-segment displays flicker off while the program runs and back on again once the program has finished. Also note that the LCD module now shows an 'A' in the top left-hand character position, while the cursor has been auto-incremented to indicate the next free character position.

Displaying a string of characters

There are a number of widely used computer codes, one very common one being the *American Standard Code for Information Interchange (ASCII).* In fact our LCD module employs a subset of ASCII, and the reason we loaded the accumulator with $41 in the previous test is that we know that this value corresponds to the ASCII code for the uppercase letter 'A'. The remaining (basic) LCD codes are shown in Table 3. (Note that you can also create your own characters for use on these displays, but this is a little outside the scope of this article. However, this topic is described in the LCD articles referenced in the *Further Reading* section at the end of this article.

As a second test, save your latest program out as *caexp2b.asm*, and then delete the two instructions you added in the previous test and modify

Listing 6

```
### Start of main program body
            LDA     $41         # Load ASCII 'A'
            JSR     [WDATA]     # Write to LCD
            JMP     [$0000]     # Soft reset
```

the main body of the program to read as shown in Listing 7.

Assemble your program, download the resulting *caexp2b.ram* file to the *PhizzyB*, and run the program. Observe that the LCD now shows "JIHGFEDCB". The reason for this should be reasonably clear. First we load the accumulator with $09 (which is also 9 in decimal) and push a copy of this value onto the stack. Next we add the value of the ASCII code for the letter 'A'. This gives a result of $41 + $09 = $4A, which is the ASCII code for the letter 'J'. Next we use our WDATA subroutine to write this character to the display, then we retrieve the original value that was in the accumulator, decrement it, and (if the result is non-zero) jump back to the beginning of the loop. This also explains why this particular program doesn't display the character 'A', because as soon as the accumulator contains $00 (which would result in $41 + $00 = $41 = 'A') our program exits the loop before writing this value to the display.

Of course it would be relatively trivial to modify this program to cause it to display an 'A' at the end of the sequence. As an exercise try to think of two different ways to achieve this, and then modify your program to test these ideas (remember to save the program out under a new name).

Displaying a string of numbers

As one final test, save your latest program out as *caexp2c.asm*, and then modify the ADD in MAINLOOP from ADD $41 to ADD $30.

Now assemble your program, download the resulting *caexp2c.ram* file to the *PhizzyB*, and run the program. From Table 3 we see that $30 is the ASCII code for '0', so it should come as no surprise that the LCD now shows "987654321". Once again it would be easy to modify this program to cause it to display a '0' at the end of the sequence as per our discussions in the previous test.

EXPERIMENT 3:

Displaying large numbers

Now this is where things start to get really interesting. In the not-so-distant future, we're going to create a program that will count the time interval between a "Start" event and a "Stop" event and display the result on our LCD module. For our purposes we've decided to use a 16-bit counter, which can be used to represent values in the range 0 through 65,535. This is the 2-byte temporary storage location called COUNTER that we reserved at the end of our program.

So the problem we now have is how to convert a 16-bit binary number into a string of ASCII-coded decimal digits. As fate would have it, the *PhizzyB Simulator* comes equipped with a set of general-purpose assembly language subroutines that can be used to perform simple arithmetic operations. These routines include ADD16 (which adds two 16-bit numbers together), SUB16 (which subtracts one 16-bit number from another), and a variety of multiplication and division subroutines.

These routines are fully documented in our book *Bebop BYTES Back (An Unconventional Guide to Computers)* — see the *Further Reading* section at the end of this article for more details. However, the routine of interest to us at this time is called UDIV10 ("*unsigned divide-by-10*") which we will incorporate into our program in a few minutes. When this subroutine is called, it first retrieves a 16-bit number from the top of the stack (it's our responsibility to ensure that this number is already there). The routine then divides this number by 10 and places a 1-byte value containing the remainder on the stack, followed by a 2-byte value containing the result from the division, followed by a 1-byte flag.

For example, if we had originally placed the number $09A3 (2467 in decimal) on the stack, then the top of the stack would look like:

```
$09  (MS byte of number)
$A3  (LS byte of number)
```

In this case the UDIV10 subroutine would return:

```
$--  (Flag - see notes)
$00  (MS byte of result)
$F6  (LS byte of result)
$07  (Remainder)
```

Listing 7

```
### Start of main program body
            LDA     $09          # Load ACC 9
MAINLOOP:   PUSHA                # Copy ACC to stack
            ADD     $41          # Add ASCII 'A'
            JSR     [WDATA]      # Write to LCD
            POPA                 # Get ACC from stack
            DECA                 # Decrement Acc
            JNZ     [MAINLOOP]   # Jump if ACC !=0
            JMP     [$0000]      # Else soft reset
```

So from the above we see that dividing $09A3 (2467 in decimal) by 10 results in $00F6 (246 in decimal) with a remainder of 7. Note that the flag byte is generated as an XOR of the MS and LS bytes of the result. If the flag is 0, it indicates that the result from the division was zero, which means that we've finished. Alternatively, a non-zero value in the flag indicates that we still have some division to do.

The cunning point about all of this is that as soon as the UDIV10 subroutine has performed its magic, we can pop the flag off the top of the stack to see if we've finished. Remember that poping the flag byte off the top of the stack leaves the stack pointer pointing at the first byte of the result. If the flag is non-zero, we simply call the UDIV10 subroutine again, which takes our 2-byte result from the previous division, divides this result by 10, places the remainder on the stack followed by the new result, and so it goes.

In order to see how this works, use the assembler to save our latest program out as *caexp3.asm*. Now scroll down to the bottom of this program and place your cursor at the end of the INITLCD subroutine (add a couple of blank lines for comfort). Next use the assembler's "*Insert -> File*" pull-down menu and select the *udiv10.asm* file from the resulting list, which will insert the UDIV10 subroutine at the current cursor position.

Note that all of the labels in this subroutine commence with an underscore character (including the name of the subroutine itself, which is actually called "_UDIV10"). This is designed to prevent our subroutine label names from conflicting with any of your label names.

Of course we now need a controlling routine to manage the data from our UDIV10 subroutine, so you also need to add the DISPLAY ("*display value*") subroutine shown in Listing 8 to your program.

Let's stroll through this routine to see what it does. First of all we load the accumulator with zero and store it to the temporary flag we called NUMDIG, which stands for "*the number of digits to be displayed*." Next we load the accumulator with the least-significant byte of our 2-BYTE counter and push this value onto the stack, then we load the accumulator with the most-significant byte of the counter and push this onto the stack. Note that the LDA [COUNTER+1] statement used to load the LS byte of our counter means "*load the accumulator with the contents of the memory location at the address of COUNTER plus 1*". This syntax and the ability to use equations in the *PhizzyB's* assembly language is discussed in

Listing 8

```
### Start of "display value" subroutine.
DISPLAY:  LDA     $00          # Load ACC with 0, then store this value
          STA     [NUMDIG]     # into the number of digits to be displayed
          LDA     [COUNTER+1]  # Load ACC with LS counter
          PUSHA                # Push it onto the stack
          LDA     [COUNTER]    # Load ACC with MS counter
          PUSHA                # Push it onto the stack
DIVBY10:  JSR     [_UDIV10]    # Jump to "divide-by-10" subroutine
          LDA     [NUMDIG]     # Load ACC with number of digits to display
          INCA                 # Increment ACC (i.e. number of digits)
          STA     [NUMDIG]     # Store ACC back to No. of digits to display
          POPA                 # Pop flag off top of ACC
          JNZ     [DIVBY10]    # If flag is !0 do it again (see notes)
STARTDIS: POPA                 # ... otherwise discard the next two
          POPA                 # ... bytes on top of the stack
DISDIGIT: POPA                 # Pop a digit to be displayed of the stack
          ADD     $30          # Add $30 to convert it to an ASCII character
          JSR     [WDATA]      # Write this digit to the LCD
          LDA     [NUMDIG]     # Load ACC with number of digits to display
          DECA                 # Decrement ACC (i.e. number of digits)
          STA     [NUMDIG]     # Store ACC back to No. of digits to display
          JNZ     [DISDIGIT]   # If ACC !=0 display another digit
          RTS                  # ... else return from subroutine
### End of "display value" subroutine
```

Listing 9

```
### Start of main program body
            LDA     $09          # Load ACC with $09
            STA     [COUNTER]    # Store to MS counter
            LDA     $A3          # Load ACC with $A3
            STA     [COUNTER+1] # Store to LS counter
            JSR     [DISPLAY]    # Call DISPLAY
            JMP     [$0000]      # Soft reset
```

Appendix D of *The Official Beboputer Microprocessor Databook*, and this Appendix is also supplied with your *PhizzyB Simulator* (check the simulator's online help for more details).

Now we enter a loop that starts at label DIVBY10, in which the first thing we do is to call our _UDIV10 subroutine. Next we load the value from our temporary NUMDIG location, increment it (to remind ourselves that we have one more digit to display), and save it away again. Next we pop the flag byte off the top of the stack and use a JNZ ("*jump if not zero*") instruction to test it. If the flag is non-zero we know that we have to continue dividing this number, in which case we jump back to the start of our DIVBY10 loop.

Otherwise, if the flag is zero, we know that it's time to display our number. The first thing we need to do is to throw away the two bytes of result from the top of the stack (we know that these "result bytes" are both zero now). Next we enter a new loop called DISDIGIT, which we commence by popping the first remainder off the top of the stack. We add $30 to this value to convert it into its equivalent ASCII code, and then call our WDATA subroutine to write this value out to our LCD module.

Now we load the value from our temporary NUMDIG location, decrement it (to indicate that we have one less digit to display), and save it away again. Next we use another JNZ ("*jump if not zero*") instruction to test the value in NUMDIG (a copy of which is still in the accumulator). If the value is non-zero we know that we still have some digits to display, in which case we jump back to the start of our DISDIGIT loop, otherwise we exit the DISPLAY subroutine.

All that remains is to revisit our main program body, delete everything between the comments "Start of main program body" and "Start of interrupt service routines," and replace it with the code shown in Listing 9.

All this does is to pre-load our 2-byte counter with a value of $09A3 (2467 in decimal), call our new DISPLAY subroutine, and then execute a soft reset ant exit the program. Now assemble your program, download the resulting *caexp3.ram* file to the *PhizzyB*, and run the program. After a few seconds the LCD module will display the value 2647. Experiment with pre-loading the counter with different values to see what happens, and then proceed to the final experiment in this article.

EXPERIMENT 4:

Creating an event timer

The previous experiments have provided us with most of what we need to create our event timer. However, we do need to add just a few more small routines as follows. Save your latest program out as *caexp4.asm* and then add the INITCNT ("*initialize counter*") subroutine shown in Listing 10.

As we might expect, this INITCNT routine initializes the counter by loading both its MS and LS bytes with zero. The routine also loads the temporary CNTFLAG location (that we reserved at the end of the program) with 0, and similarly for the output port driving the *PhizzyB's* 8-bit LED bargraph display. Next we need to add the counter subroutine itself as shown in Listing 11.

Listing 10

```
### Start of "initialize counter" subroutine
INITCNT:  LDA     $00          # Load ACC with 0
          STA     [COUNTER]    # Store ACC to MS byte of counter
          STA     [COUNTER+1] # Store ACC to LS byte of counter
          STA     [CNTFLAG]    # Store ACC into CNTFLAG
          STA     [BARGRAPH]   # Store ACC to bargraph display
          RTS
### End of "initialize counter" subroutine
```

Listing 11

```
### Start of "counter" subroutine.
COUNT:    LDA     [COUNTER+1] # Load ACC with LS byte of counter
          ADD     $01         # Add one to ACC
          STA     [COUNTER+1] # Store ACC to LS byte of counter
          STA     [BARGRAPH]  # Also store ACC to bargraph display
          LDA     [COUNTER]   # Load ACC with MS byte of counter
          ADDC    $00         # Add carry flag (from LS addition)
          STA     [COUNTER]   # Store ACC to MS byte of counter
          JC      [TOOBIG]    # If carry is set then we've just overflowed
                              # ... so jump to TOOBIG
          LDA     [CNTFLAG]   # ... or else load ACC with CNTFLAG and
          JZ      [COUNT]     # check to see if we continue counting
          RTS                 # If CNTFLAG !=0 then return from subroutine
TOOBIG:   LDA     %11001100   # Load ACC with some recognizable value
          STA     [BARGRAPH]  # Store ACC to bargraph display
          JMP     [$0000]     # Perform a soft reset
### End of "counter" subroutine
```

Once this COUNT routine is called, its main purpose in life is to loop around incrementing the 16-bit value stored in the 2-byte COUNTER temporary location. It also stores a copy of the least-significant byte of the count value out to the 8-bit LED bargraph display (just so that we can see that something is happening).

Every time the routine increments the counter, it checks the contents of the CNTFLAG location. If this location contains zero the count continues, otherwise the subroutine exits (the contents of CNTFLAG can be modified externally as we shall see). Also note that if the count exceeds 65,535 (the maximum value that can be stored in a 2-byte location), the routine jumps to the TOBIG label, at which point it displays some recognizable value on the 8-bit LED display and exits the program. (Once you've finished this experiment, you can modify the program to display an appropriate error message on the LCD module if you wish).

We now need one last subroutine that simply clears the LCD display and "homes the cursor" (that is, returns the cursor to its home position). Our original INITLCD routine can do this, but it's a bit of an "over-kill" if all we wish to do is to re-clear the display, so let's add the CLEARLCD ("*clear LCD and home cursor*") subroutine shown in Listing 12 to our program.

Adding the interrupt service routines

In order to complete our event counter, we're going to use interrupts and interrupt service routines (these were introduced in Part 4 of the *PhizzyB* series, which appeared in the February 1999 issue of *EPE Online*). If you return to the top of your program, just before your subroutines, you'll see a comment that says "*Start of interrupt service routines*." Enter the two routines shown in Listing 13 just after this label:

The WAITIRQ routine is very simple, in that as soon as it's called it does nothing, but simply returns (we'll see how

Listing 12

```
### Start of "clear LCD and home cursor" routine
CLEARLCD: LDA     %00000001   # Clear/Home code
          JSR     [WCOMMAND]  # Call WCOMMAND
          RTS                 # Exit subroutine
### End of "clear LCD and home cursor" subroutine.
```

Listing 13

```
### Start of "wait for an interrupt" routine
WAITIRQ:  RTI                 # Just returns
### End of "wait for an interrupt" routine

### Start of "stop counter" routine
STOPCNT:  PUSHA               # Copy ACC to stack
          LDA     $01         # Load ACC with 1
          STA     [CNTFLAG]   # ACC to CNTFLAG
          POPA                # Get ACC from stack
          RTI                 # Exit routine
### End of "stop counter" routine
```

we use this in a moment). The STOPCNT routine not much more complex, but it's certainly more interesting. First of all we push a copy of the current contents of the accumulator onto the stack. Next load the accumulator with $01 and store it to our temporary CNTFLAG location. Then we retrieve our copy of the original value in the accumulator back from the stack and exit the routine.

So why is this of interest? Well if you cast your mind back to the COUNT subroutine we created a few minutes ago, you will recall that once this routine is called it will endlessly loop around incrementing our 16-bit count value.

It's easy to tell the counter when to start -- we just have to call the COUNT subroutine. Our problem is telling the counter when to stop, and that's where the STOPCNT interrupt service routine comes into play.

As soon as an interrupt occurs we'll make the CPU jump to the STOPCNT routine, which will load the CNTFLAG with $01 and then return control to the COUNT subroutine. The count subroutine will then finish the count it's currently working on, detect that the CNTFLAG is now non-zero, and exit.

Modifying the main program

So now we're really ready to rock and roll. All that remains is to revisit our main program body, delete everything between the comments "Start of main program body" and "Start of interrupt service routines," and replace it with the code shown in Listing 14.

First we use a BLDIV ("*big load interrupt vector*") instruction to load the interrupt vector with the start address of the WAITIRQ interrupt service routine. Next we use a SETIM ("*set interrupt mask*") instruction to load the interrupt mask status bit with 1, which allows the CPU to see and respond to interrupts.

Now we enter our main loop, which starts by initializing the counter followed by a HALT instruction. The HALT instruction is a little special, in that it instructs the CPU to stop doing anything until an interrupt occurs. When an interrupt does occur, the CPU will jump to the address stored in the interrupt vector, which is currently the address of the WAITIRQ routine. As we already discussed, this routine doesn't do anything except return to the main body of the program, but the combination of the HALT and the WAITIRQ routine provides us with a useful technique for specifying when we want the count to start.

As soon as we return from the WAITIRQ routine, we immediately use a BLDIV instruction to re-load the interrupt vector with the start address of the STOPCNT routine. Now we call our COUNT subroutine, which will happily loop around counting until we tell it to stop. We do this by causing a second interrupt to occur, at which point the CPU will jump to the STOPCNT interrupt service routine. As we've already discussed, the STOPCNT routine will load the CNTFLAG with $01 and return control to the COUNT subroutine, which will in turn finish its current count and return control to the main body of the program.

As soon as we return to the main body of the program, we reload the interrupt vector with the address of the WAITIRQ subroutine and then call our DISPLAY subroutine, which displays our 2-byte count value on the LCD module. Next we use a new HALT instruction to wait for another interrupt to occur. As soon as this interrupt does occur we clear the LCD and jump back to the beginning of the loop.

Listing 14

```
### Start of main program body
          BLDIV   WAITIRQ     # Load IV with address of "wait for IRQ"
          SETIM               # Set the interrupt mask flag
MAINLOOP: JSR     [INITCNT]   # Call initialize counter subroutine
          HALT                # Wait for an interrupt to occur
          BLDIV   STOPCNT     # Load IV with address of "stop counter"
          JSR     [COUNT]     # Call the counter subroutine
          BLDIV   WAITIRQ     # Load IV with address of "wait for IRQ"
          JSR     [DISPLAY]   # Call the display value subroutine
          HALT                # Wait for an interrupt to occur
          JSR     [CLEARLCD]  # Call the "clear LCD" subroutine
          JMP     [MAINLOOP]  # Jump back and do it all again
```

Running the program

Assemble this new program, load the resulting *ca-exp4.ram* file into the *PhizzyB*, and set the program running. Nothing appears to be happening at first (after the display has cleared), because the program has reached it's first HALT instruction and is waiting for you to do something.

Click (press and quickly release) the interrupt request (IRQ) button on the *PhizzyB* and observe the 8-bit LED bargraph start a binary count. Now click the IRQ button again and (after a short delay) observe the count value appear on the LCD display.

Remember that the 8-bit LED bargraph display only reflects the least-significant byte of the 2-byte count value. This means that the value on the LEDs won't match the value shown on the LCD once the count exceeds 255 (binary 11111111, which is the maximum value that can be displayed on this 8-bit LED). This discrepancy does not matter, of course, because the LCD is the main display, and we're only using the LED display to indicate when a count is taking place.

Now you're at the second HALT instruction. Clicking the IRQ button again will clear the LCD display and return you to the beginning of the loop. Clicking IRQ once more will start a new count, and so forth.

FURTHER EXPERIMENTS

There are a number of experiments you can you use your event timer for. As an example, instead of your starting and stopping the counter by manually activating the IRQ button on the *PhizzyB*, you could use the 8-bit interrupt-driven switch device we described in Part 4 of the *PhizzyB* series. Furthermore, as opposed to using the pushbuttons on that device, you could place two micro-switches at the top and bottom of a ramp and then connect them to the 16-pin header on the input switch device. (The micro-switches should be wired in a "normally-open-active-closed" configuration. That is, in parallel to the push-switches on the input device.) This would allow you to time the interval taken for something (say a can of beans) to roll down the ramp and display the results on the LCD. Then you could determine how varying the slope of the ramp modifies the duration of the object rolling down the ramp.

Also, note that the value displayed on the LCD module simply reflects the number of times the counter cycled around its loop, and doesn't reflect "real time." So another thing you could do is to correct the final count value to display real time in seconds. As a simple example, as we were penning these words we set our counter running for approximately 100 seconds (measured roughly by wristwatch) and received a count value of 3300, which is approximately 33 cycles around the counter per second. Thus, dividing the final count value by 33 would give a result in seconds (you could use the UDIV16 general-purpose subroutine for this). You obviously want to be careful of rounding errors if you're measuring small intervals, but it you are measuring events with a duration of tens of minutes, hours, or even days, then these errors obviously become less significant.

Based on the above (rough) calculation of 33 counts per second, and knowing that our 2-byte counter can only count to 65,535 before it overflows, we now know that we can count intervals up to 65,535 / 33 seconds, which equals approximately 33 minutes. If we wished to count events with a longer duration we could increase our counter to be a 3-byte value. Three bytes equates to 2^{24} different patterns of 0s and 1s, which allows us to represent values in the range of 0 through 16,777,215. In turn, this would allow us to time events with a duration of several days (note that you would also have to modify the INITCNT, COUNT, and UDIV10 subroutines).

Finally, you could modify the output routine to display the elapsed time in Hours, Minutes, and Seconds.

FURTHER READING

If you haven't already done so, you might care to download and read the excellent two-part article on how to use LCD displays that you'll find at the *EPE Online Library* at **www.epemag.com**

The concepts of assembly language, subroutines, interrupts, and much, much more are discussed in great detail in our book *Bebop BYTES Back (An Unconventional Guide to Computers*). This modern masterpiece can be ordered from the *EPE Online Store* at **www.epemag.com**

OOPS!

For your delectation and delight, the assembly source code for all of the programs described in this article can be downloaded from the *EPE Online Library* at **www.epemag.com**

Oops, perhaps we should have mentioned this before, but then you wouldn't have had the fun of creating the programs yourself (which is much more educational in the long run). So this really means that we've done you a tremendous favor, but there's no need to write long letters to thank us :-)

Of course, if you do wish to contact us, please feel free to email us at **max@maxmon.com** and **alvin@maxmon.com** (we would be delighted to hear how you use your *PhizzyB*).