In the Eye of the Beholder

Displaying Messages with the LightStick II

If you've ever seen multiple images of the current time as you sweep your eye past a digital clock, you've experienced the premise behind the LightStick II. Find out how to display your own ghostly images.

FEATURE ARTICLE

Shaun J. Greaney

You walk into the room and, tucked away in a corner, you spy a vertical column of LEDs flickering just slightly. "What's that thing supposed to do!" you ask yourself. Shrugging, you redirect your gaze to more interesting matters. As you turn away, you think you see something from the corner of your eye, but then it's gone. You look back at the LEDs to find them still flickering ever so innocently. Turning away, you again catch a glimpse of a picture. You've just discovered the LightStick.

Bill Bell's United States patent [1] calls his light stick a "Momentary Visual Image Apparatus." It is a device that creates an optical illusion that is formed by multiplexing the multiple columns of an image onto a single column of LEDs. Everyone has seen examples of m- x n-pixel arrays used to create two-dimensional images. PC graphics adapters support 1024 x 768, 800 x 600,640 x 480 to name a few. Old dot matrix printers supported 5 x 7 characters. From the instant replay screen in the football stadium to the graphics display on my HP 48SX, they all share something in common. The images all consist of static arrays of m columns of n picture elements. The specific spatial displacement in x and y required by each individual pixel is provided by the physical display.

What a light stick does differently is to display all ***m*** columns of the picture on a single column of n pixels.

Construction of the image depends on the human eye's persistence of the retina and on its natural, rapid movements from one point of fixation to another.

The persistence of the retina gives you after-images from flash bulbs and allows you to draw line images with Fourth of July sparklers. The rapid eye movements are involuntary and you are probably unaware of the fact that you are doing it. Normal visual activity consists of fixating on a point of interest, capturing the visual information, and then rapidly moving to the next point of interest. During the transition from one point to the next, normally no visual information is gathered. Everything would appear a blur and is ignored. It is during this time that the light stick works.

If you look directly at the column of LEDs, you will perceive it as a flickering or shimmering stick of light. As the individual pixels are rapidly turned on and off it appears that they are either continuously on, or off, or being slightly modulated in intensity. If instead you ignore the stick and let your eyes wander around the room, the horizontal motion of your eye will spread the columns across your retina, thus building up an image that exists as long as your persistence of the retina holds it.

My light stick was inspired by one of Bill Bell's light sticks that was on display at the Exploratorium in San Francisco back in 1986. The Exploratorium is a nonprofit science museum that everyone should visit at least once, and Bill Bell's light stick optical displays are certainly among the most striking and interesting pieces of modern art.

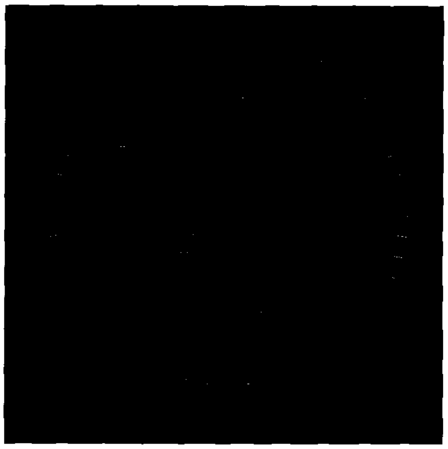


Photo **1**—*The* image produced *by the LightStick II* is painted on your retina as you *quickly look past the device. While it's easy to see the image here, it's gone in* an instant when you look at the *real* display.

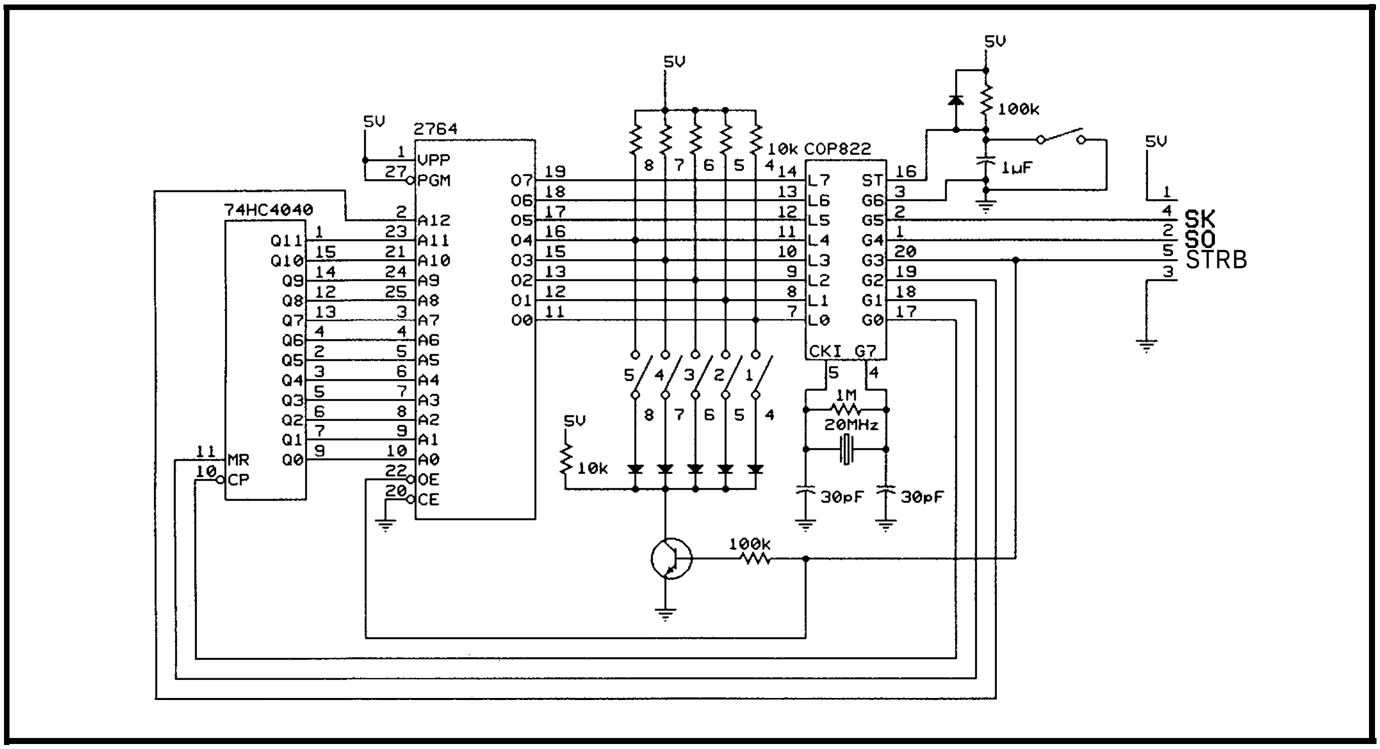


Figure la--The *LiqhtStick II controller couldn't be simpler, using a COP for the* brains, an EPROM to ho/d *image* data, *and a counter to help with the EPROM* addressing. *Data is shifted* out *serially to the stick* itself.

THE LIGHTSTICK II

The basic operation of LightStick II is very straightforward. The picture data is stored in an EPROM (see Figure 1). A microprocessor retrieves the data, one byte at a time, and shifts it into a string of 20 serial-to-parallel converters that are located adjacent to the LEDs in the stick. This continues until an entire column-20 bytes-is shifted in. It waits for a specified period of time, then strobes the data out onto the LEDs. Then it goes back and gets another column. This cycle continues and repeats after 204 columns.

The hardware consists of two subassemblies connected via a five-wire cable: the LED display and the microprocessor-based controller. The cable provides DATA, CLOCK, and STROBE logic signals for the stick as well as power and ground.

THE STICK

The stick for LightStick II consists of a single column of 160 LEDs with National Semiconductor MC74HC595N serial-to-parallel converters to drive them. While Figure lb shows just ten sets of converters and LEDs, the second group of ten is wired exactly the same.

I chose HP HDSP4830 lo-segment bargraph LED modules because of the ease of lining up 10 LED segments as opposed to individual LEDs. The 74HC595 allowed me to have a serial data bus and a 5-wire cable from the control board to the stick display board. Five-conductor DIN connectors made the whole system easy to break down and reassemble.

I chose the height of 160 pixels based on the original bitmap that I wanted to display. I obtained a bitmap of the AT&T company logo that was intended for 300-dpi laser printers. The logo is 150 pixels (or $^1/_2$ inch) high and 336 pixels wide. The LED arrays are grouped as 10 LED segments, and the microprocessor likes to work in 8-bit bytes, so 160 is a convenient height. The logo is 42 bytes (336 pixels) wide, so I divided it into two pictures and padded each out to 204 columns so that each would occupy approximately one half of the 27C64 EPROM. Splitting the logo into two parts also keeps the image simple and easy to view. More on this later.

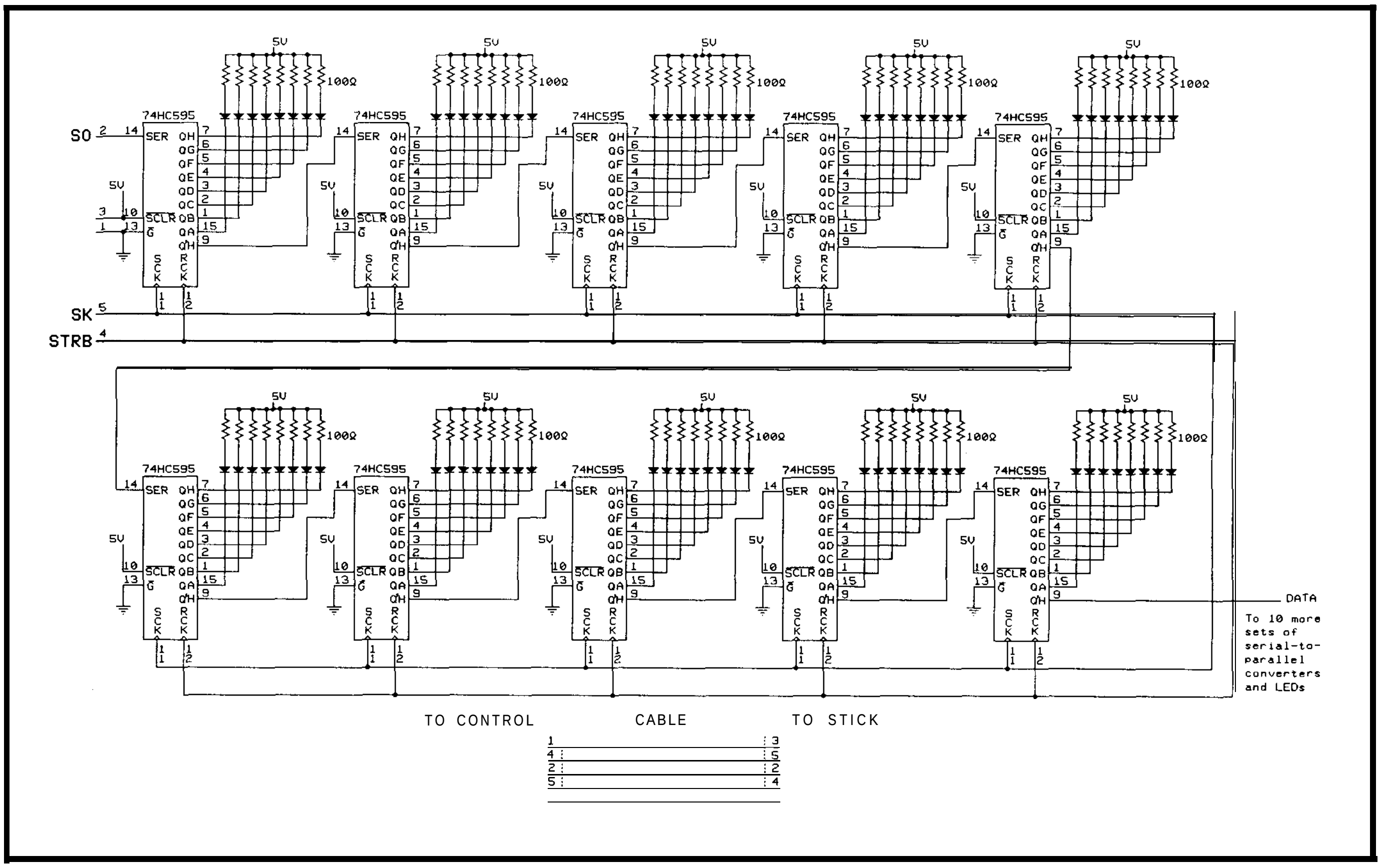
THE CONTROLLER

The controller subassembly consists of a National Semiconductor COP822 microprocessor emulator, a five-key keypad, a 27C64 EPROM, an RCA CD74HC4040E ripple counter to help out with addressing the EPROM, and a reset button.

I chose the COP822 based on its projected cost in large volumes. Because of its low cost, we were planning to use the COP822 to replace some discrete logic in one of our designs. The LightStick II project provided an excellent opportunity to learn how to program the COP822—and microprocessors in general-since I am primarily an RF engineer.

The actual processor I used was an EEPROM COP8722, which was a limited-production version of the COP822 emulators used for early development. National Semiconductor has since come out with COP8782C OTP and UV EPROM parts that would provide a more cost-effective solution for development and limited production use.

The 20-pin COP822 has two I/O ports available on it: the L port and the G port. The L port is a full 8 bits wide while the G port is only 7 bits wide. The chip also has a timer function and a serial port. To use the serial port, the upper 3 bits of the G port are config-



SO
SK
STRB
5V
100Ω
74HC595
SER
QH
QG
QF
QE
QD
QC
QB
QA
Q'H
SCLR
G
SCK
RCK
DATA
To 10 more sets of serial-to-parallel converters and LEDs
TO CONTROL CABLE TO STICK

Figure 1`b--Twenty *serial-to-pm//e/ converters (only half are shown here) are used to receive data sent by the controller for display on the LEDs.*

ured as Serial Clock, Serial Output, and Serial Input. Since I am not interested in receiving serial data, I grounded G6, the Serial Input. The lower 4 bits of the G port are used to control the addressing of the EPROM (with help from the 74HC4040) and to control the strobe signal to the 74HC595s. The strobe signal is also used to activate the 5-key keypad. I use the L port for data input from both the 8-bit-wide data from the EPROM and the 5-bit-wide keypad.

The image size of 204 x 160 (4080 bytes] allows two uncompressed images in the 64K-bit ROM. Keep in mind that the final image exists only in the persistence of the viewer's retina, so you want to keep it as narrow and as simple as possible.

The keypad is used to control the display of the image. I set a default time delay between column strobes that is a compromise between how narrow the image will appear and how much the leading edge of the image fades by the time the trailing edge is displayed. You may adjust this time delay using the "faster" and "slower" keys on the keypad. The three other keys allow you to invert the image, choose between the two images in the ROM, and toggle between Normal and Slow mode. In Slow mode the controller clicks off a column every 100 ms (adjustable) for diagnostic purposes. Also, in Slow mode, pressing both "faster" and "slower" together puts you into a Halt mode where you can single step through the ROM one column at time.

THE POWER SUPPLY

The HP HDSP4830 LEDs are rated at a maximum DC forward current of 30 mA. At 20 mA, the typical forward voltage is 1.6 V. I chose to run them at the maximum DC current to get as much light out as possible while still allowing for the Halt mode where the LEDs may be illuminated for an indeterminate period of time-a 100% duty cycle. I make the assumption that the forward voltage will approach 2.0 V at 30 mA. This leads to a current-limiting resistor of 100Ω. Fortunately, each current-limiting resistor only dissipates a maximum 116 mW, so I chose a resistor array package, making the assembly more elegant. The LEDs are conservatively rated; I plan to use them in pulsed mode most of the time, and, most importantly, I socketed them.

Assuming a worst-case LED current of 34 mA, 160 LEDs on would draw 5.44 A! This is not counting the shift registers and control circuits. I chose a supply that sources 6 A at 5 V.

SOFTWARE

The COP800 cross-assembler listing for the most recent version of microprocessor code is available on the Circuit Cellar BBS. The comments in the beginning include the values stored in memory locations unique to the COP8722 EEPROM emulator. Address 804 prevents modifying the ROM section once programmed. Address 805 contains configuration information relating to the mask-selectable oscillator options: divide by 10 or 20 and either RC, crystal, or

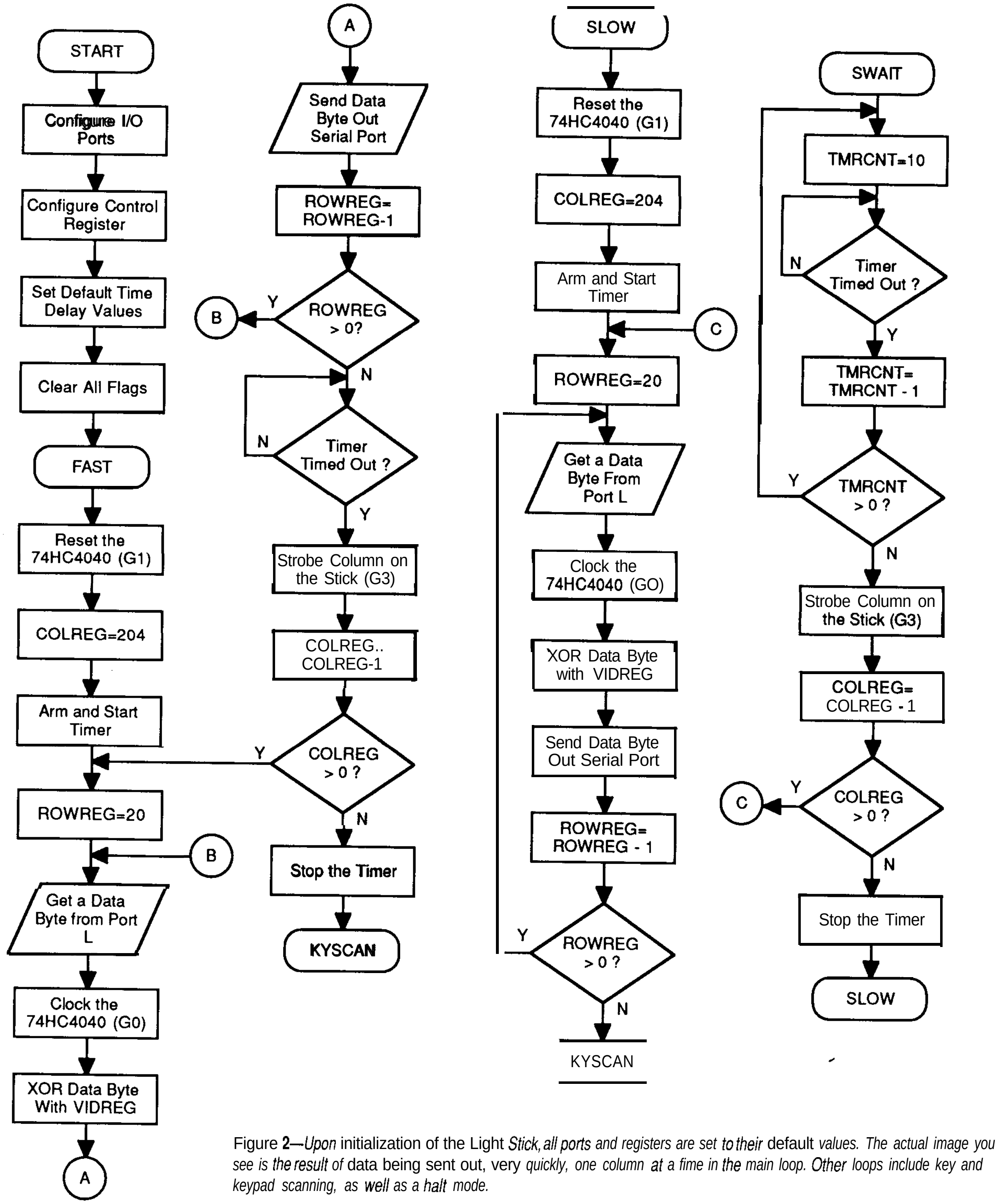


Figure 2—*Upon* initialization of the Light *Stick, all ports and registers are set to their* default *values. The actual image you see is the result of* data being sent out, very *quickly, one column at a fime in the main loop. Other loops include key and keypad scanning, as well as a halt mode.*

external clock. I chose crystal and divide by 20 to give me a 1 -µs clock with a 20-MHz crystal (the fastest instruction clock possible). Locations 800 through 803 hex are available for comments. I used them for the date and iteration number of the code.

The code may be logically broken up into five routines: an initialization routine where default values are set; two display loops, one fast and one slow; a keypad servicing routine; and an auxiliary halt mode loop. See Figure 2 for a flowchart of the whole system.

INITIALIZATION

This routine is run upon power up, hard reset, or in the event of a software interrupt. I have no provisions for using software interrupts, so if one occurs, it is probably a good idea to reinitialize everything. This routine configures ports L and G, configures the Control Register, and sets the following conditions: do not invert the outgoing data, there is no current keypress, the column time delay for

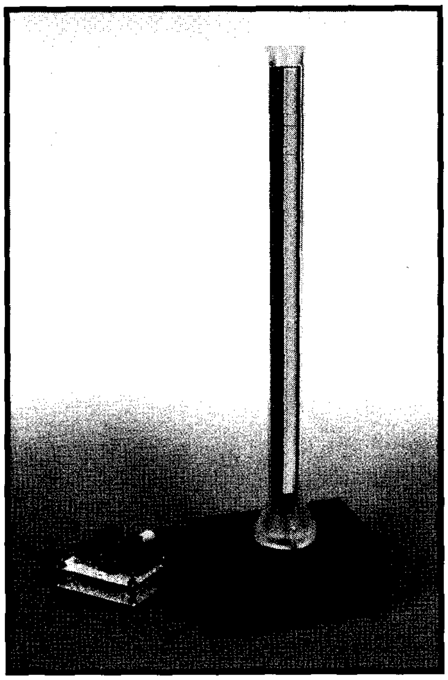


Photo 2—*The* complete *LightStick II* consists *of a controller board and the display connected by a 5-wire cable. The* display *stick stands a little under two feef tall.*

Slow mode is 100 ms, the column time delay for Fast mode is 466 us, and the serviced flag is set.

MAIN LOOP

The main loop's primary responsibility is to extract one column of picture data from the 27C64 and shift it into the 74HC595s as quickly as possible so as to allow for maximum variability in column timing. After shifting in an entire column, it waits for the column timer to expire before strobing the new column onto the stick and starting over again.

Optimizing the routine for speed means optimizing for the minimum number of clock cycles used by the instructions within the Fast loop. For this reason, keypad scans are made outside the loop, in between picture displays. Register Indirect instructions only take one clock cycle as opposed to three clock cycles for Memory Direct instructions. If two or more instructions are going to operate on the same location in memory (Port G, for instance), it makes sense to spend the three clock cycles it takes to load the address of the memory location into register B, thereby minimizing the overall total number of clock cycles.

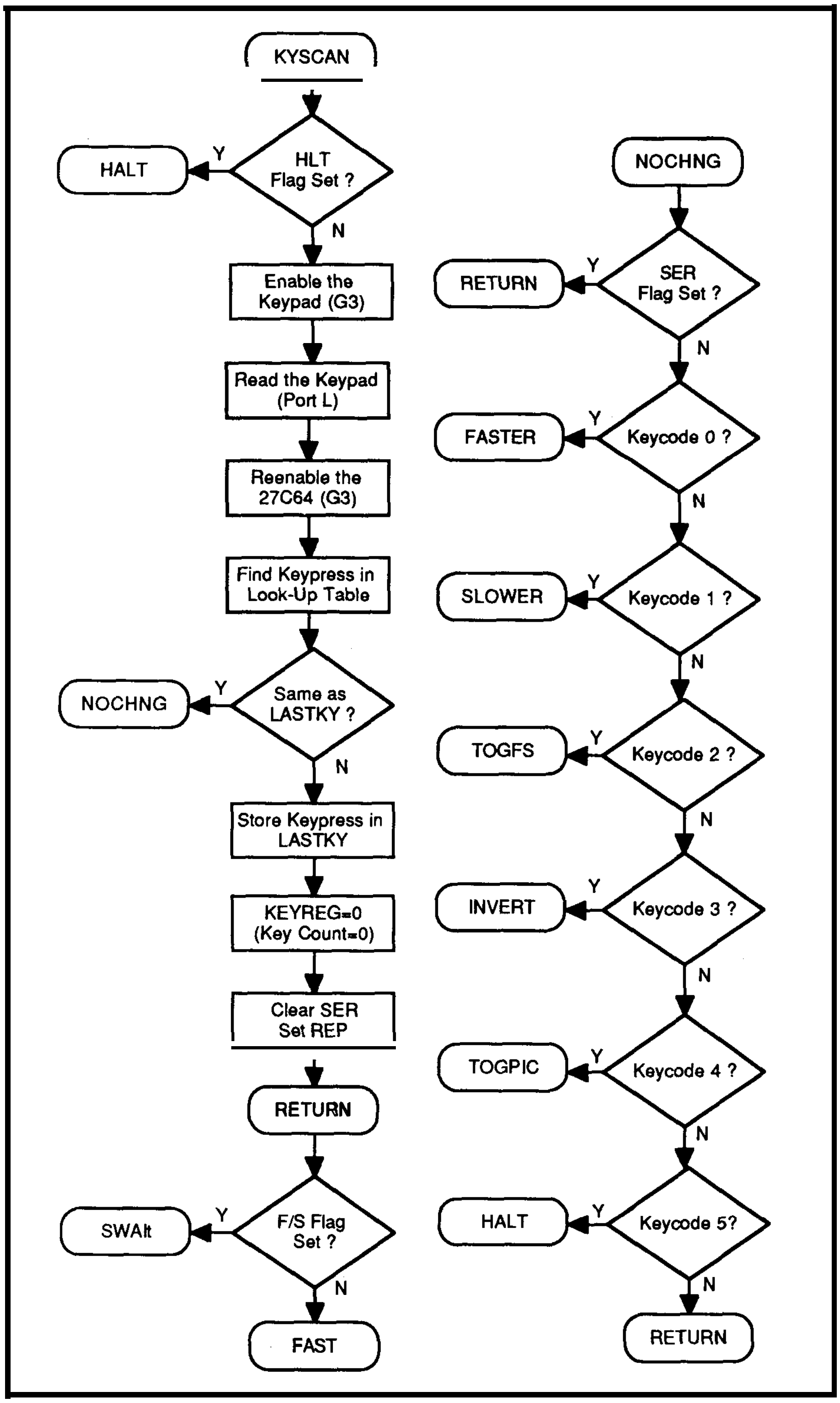


Figure 2—*continued*

I/O savings are realized by using the 74HC4040 ripple counter to sequentially step through one half of the 27C64's address space. Since both the ripple counter and the 27C64 take a little while to stabilize, I clock the 74HC4040 immediately after getting a data byte to maximize the time available for this settling.

SECONDARY LOOP

The secondary loop performs basically the same as the main loop except it does a key scan after every column instead of after each complete picture since I am not as time constrained.

This loop is really slow, so it counts up 10 timer periods instead of a

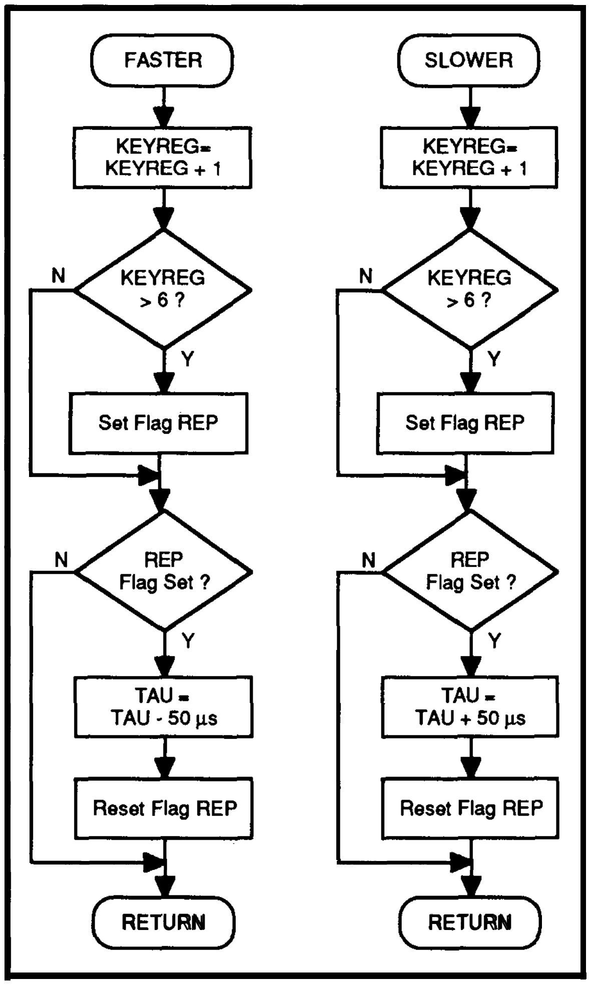


Figure *2-continued*

single timer period for the main loop. This also changes the effective timer increment/decrement time slice from 50 µs to 500 µs and allows me to use the same time delay adjustment routine for both the Fast and Slow time delays.

KEYPADSCAN

The primary responsibility of the Keypad Scan routine is to get updated configuration information in between pictures in the Main Loop or in between columns in the Secondary Loop. The types of things you want to adjust are how quickly you walk through the columns, which picture you want to display, whether or not to invert the picture ("on" for "off" and "off" for "on"), and which primary mode you want the processor to operate in: Fast, Slow, or Halt.

To help me keep track of where I am in the Keypad Scan, I defined eight flags. RE P helps me keep track of a key that is continuously held down indicating I should repeat the modification to the time delay. SE R keeps track of whether the key request has been serviced. H LT indicates Halt mode, used for branching back into the Halt routine. PI C indicates which picture is current. I NV keeps track of whether or not to invert the outgoing data. F / S indicates either Fast or Slow mode, used for branching at the end of the key scan routine. S LW is a Slower request, so I have to increase the time delay. FST is a Faster request, so I have to decrease the time delay.

HALT MODE

Halt mode may almost be considered a limited functionality subroutine of the keypad routine. When in Halt, the keypad is continuously examined for one of only two valid keypresses: Step or Continue. Just as the regular Keypad routine either jumps back to Fast or Slow depending on where you came from, when you get a Step keypress you go back to the parent process and execute only one column for Slow or one picture for Fast. When you go for another key scan you are placed back in Halt. The only ways out of Halt are Continue and Reset.

TRANSLATION UTILITIES

Since the column-at-a-time, 204 x 160, two-color picture format is not an accepted standard yet, a couple of C-code programs were written to translate data from the existing, more common standards into STK-light stick-format. Both of these programs are available on the Circuit Cellar BBS.

The company logo image data was given to me in an unformatted 336 wide by 150 high two-color binary file that was organized as a sequence of rows starting from the upper left corner of the picture. That's 6300 bytes of input picture data. I had to convert the data into a stream of ASCII represented hex bytes that were grouped one column at a time starting from the upper left hand corner of the picture.

A friend at work, Tom Recht, wrote the original translation program that took the raw binary input file and rearranged it for LightStick I (yes, there was a version before this one). I modified this program to create XLATE2.C, the LightStick II translation program.

Since the stick is 160 rows high, a 336 x 160 or 6720-pixel buffer is created and initialized. The binary input file is then read into this 6720

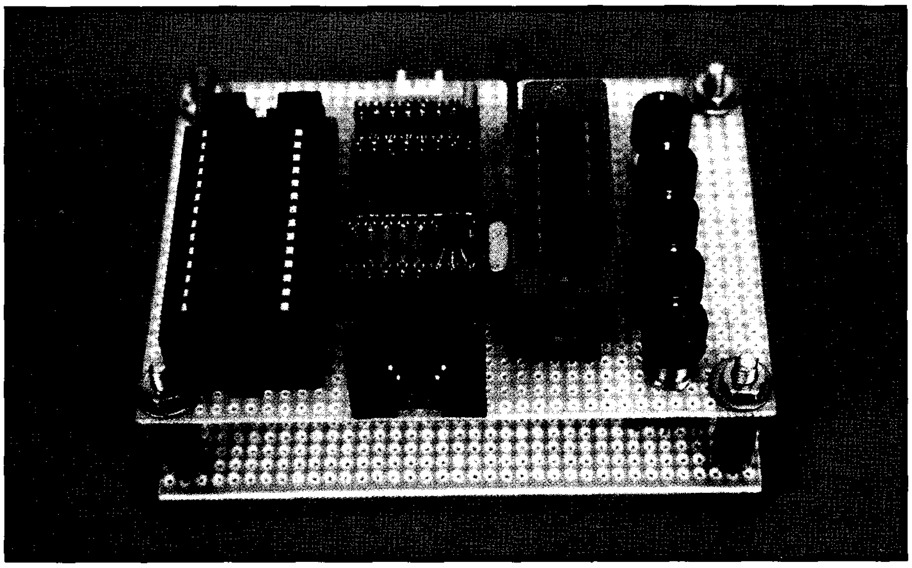


Photo 3—*The* controller *board has ZIF sockets for the the processor and EPROM to make if easier to change the control code and the image displayed.*

array of bytes where each byte represents a single pixel of the picture. The picture is stored in the original data file such that the most-significant bit of the first byte is the upper left corner of the picture, and the buffer is arranged such that the first byte is the upper left corner so that the index loops used to write the data can start at zero and increase monotonically to read from left to right and top to bottom.

The company logo is not exactly symmetrical, so the first half is only 160 columns wide (20x8) with real data, and the file is padded with 896 null bytes to pad the 27C64 EPROM to 4096 bytes for the first half of the logo. The second half has 176 columns of real data (22x8), so it only needs 576 null bytes of padding to fill the second 4096 bytes of the EPROM.

The output file is written as ASCII represented hex bytes. Another friend at work, Bruce Haggerman, wrote a C utility that takes such a hex file and adds the appropriate addresses and checksums to create an Intel hex format file ready to download to the PROM programmer.

Eventually, the company logo gets kind of old. To give myself some practice coding C, I wrote a program that takes uncompressed, B&W, windows BMP image files, pads or truncates the image to the 204 by 160 light stick size, and generates an ASCII represented hex output file. I found the windows BMP file and image headers explained in "Graphics File Formats." [2]

In BMP2STK.C, I tried to write code that I could easily modify for other programs. To that end there is a general-purpose, configurable fileopen() routine that tries to get all the input and output file information from the command line arguments. If this fails; it will prompt the user for the information.

The routine of interest from the light stick perspective is read(). This routine defines two structures to extract size and configuration information from the file and image headers. Using information in the headers, I check for valid BMP file type, check for the two-color requirement, and get

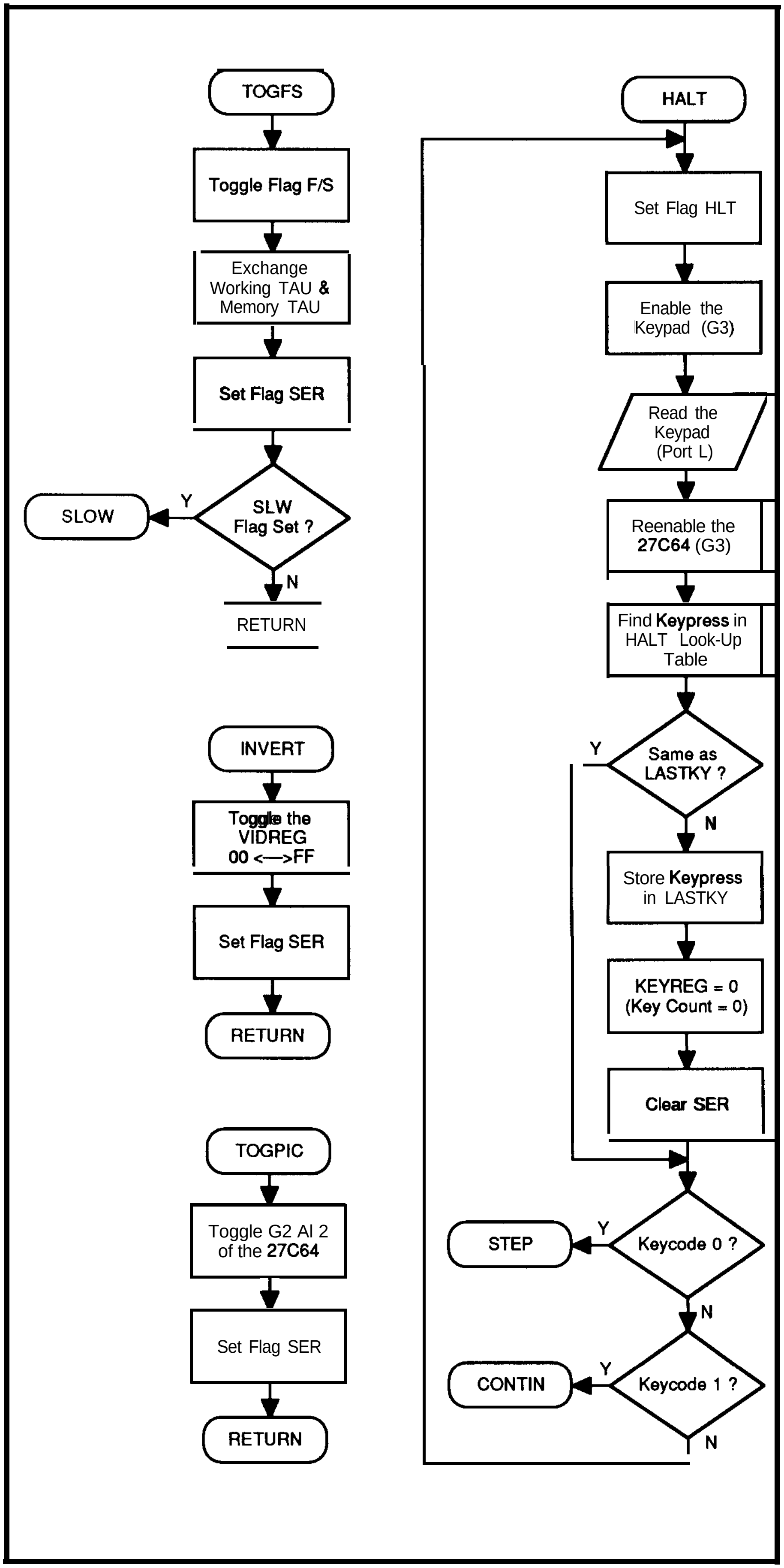


Figure *2-continued*

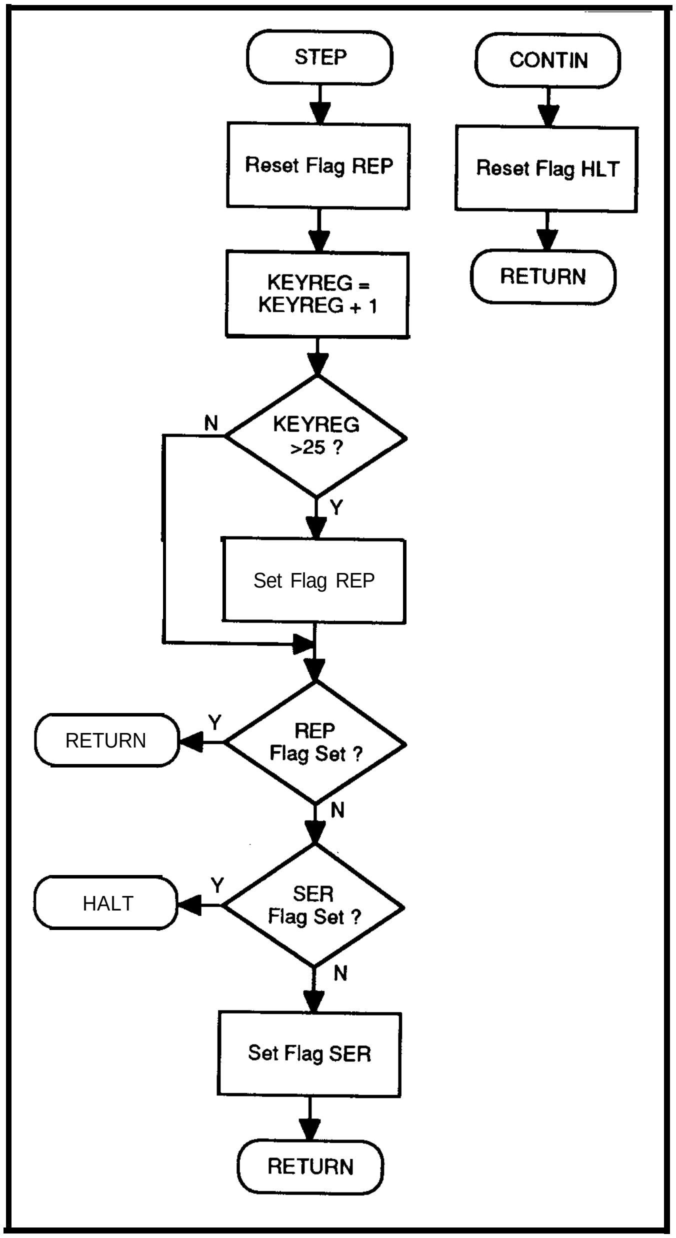


Figure 2—*continued*

loop sizes from the image size data fields.

The BMP image format requires the data stored in the BMP file to fall on 4-byte boundaries for image width. If the actual image size does not fall onto a 4-byte boundary, the image data is padded with garbage out to the next 4-byte boundary. It is the responsibility of the display program to get the actual image size from the image header and to crop this garbage data out of the displayed image.

I have a 204 wide by 160 high buffer that is initialized to 00. As I read the BMP into the buffer, a row at a time, I check to see if I exceed the buffer width. If so, I keep reading bytes in and discard them. I store the data into the buffer one pixel at a time, from left to right. If the image is less than 204 pixels wide and not an integer number of 4-byte blocks wide, I stop transferring pixels at the last data bit of significance. Since the buffer is initially set to all 00, the image is effectively padded out to 204 pixels wide. The other interesting thing about BMP files is that they are arranged with the lower left corner of the picture first. This means the upper and right edge of the picture will be clipped if it is larger than 204 by 160.

The `write()` routine is very similar to that of the XLATE2.C program. It also pads 16 bytes to make up the difference between the 4080 bytes in one 204 x 160 light stick image and the 4096-byte capacity of half of the 27C64 EPROM. This program takes two BMP files as input and generates one STK file as output.

CONCLUSION

LightStick II provides a fairly flexible architecture for examining the optical effect described by Bell in his patent. This allows you to play with both the type of picture and the display parameters of the picture. Keep in mind that the COP assembly code and the C code were written by an RF engineer, so if you believe there is a more efficient or elegant way of writing the code, you are probably right. If you do rewrite any of the code to be more flexible, efficient, or elegant, I would be interested in hearing about your efforts.

The type of information that may be successfully displayed is somewhat limited by the humans that are going to observe it. When I first wrote the BMP2STK.C translation code, I feverishly converted a couple of B&W digitized pictures to the stick format. ı then spent half an hour getting a sore neck trying to view them. The best images are simple geometric shapes, words with few letters, and stick figures. Even the Circuit Cellar INK logo used in the photo for this article presents a challenging image to view in its entirety. So, don't worry if you are not the most elaborate Paintbrush artist in the world; the people who would be required to view your creation don't exist yet anyway. ❑

Shaun Greaney is an RF Design Engineer at AT&T. He holds a BSEE from Rensselaer Polytechnic Institute and an MSEE from the University of California, Berkeley. He may be reached at sjg@hocpa.att.com.

REFERENCES

1. Bell, Bill, United States Patent Number 4,470,044, "Momentary Visual Image Apparatus," 1984.
2. Kay, David C. and John R. Levine, "Graphics File Formats," Windcrest/McGraw-Hill, 1992, ISBN 0-8306-3059-7.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

404 Very Useful
405 Moderately Useful
406 Not Useful