adafruit learning system

Hallowing Spirit Board

Created by Phillip Burgess

Last updated on 2020-05-19 12:31:46 AM EDT

Overview

A *spirit board* (or *talking board*...or best known by Hasbro's trademark *Ouija®* board) is a parlor game that lore would suggest can channel the words of the departed. The reality is more innocuous...but the spirit board's appearance in horror films such as *The Exorcist* have built up its supernatural reputation and made spirit board imagery a Halloween staple.

Participants place their hands on a *planchette* — think of it like a supernatural mouse cursor — which mysteriously moves to spell out messages.

In this project, ***HalloWing*** *(https://adafru.it/CmY) acts as the planchette,* moving around an invisible spirit board that can be seen only through the HalloWing's display. The onboard accelerometer reacts to subtle tilting to scroll around the board, or touch one of the capacitive pads to have the oracle spell out a message on its own. The powerful SAMD21 processor gives us buttery smooth animation!

Image credit: Wikimedia Commons. Public domain.

Everything needed for this project is built into the HalloWing board, no extra components are required.

Software

Easy Way

If you want to get started **quickly**, download the UF2 file linked below. Turn on Hallowing and connect a USB cable to your computer. Double-click Hallowing's reset button, wait for the HALLOWBOOT drive to appear, then drag the UF2 file to this drive. After a few seconds, the code should be finished transferring and will run.

https://adafru.it/Cpk

https://adafru.it/Cpk

This will overwrite CircuitPython if it's currently installed on your board (but your CircuitPython code and any libraries are safe).

You can **restore CircuitPython** easily by **following the directions here** (https://adafru.it/CmJ).

Using the Spirit Board

Tilt Hallowing various directions to make the board scroll, or tap any of the capacitive touch "fangs" to make it randomly read from a set of Halloween-themed messages. **Spooky!**

Build From Source

Building the project from source gives you the opportunity to **customize** the built-in messages to your liking.

This requires the **Arduino IDE** software for your computer and **Adafruit SAMD board support**, as **explained in this guide** (https://adafru.it/CpI).

Several libraries are also required, which can be installed through the **Arduino Library Manager** (Sketch→Include Library→Manage Libraries...):

- Adafruit_LIS3DH
- Adafruit_FreeTouch
- Adafruit_GFX
- Adafruit_BusIO
- Adafruit_ST7735
- Adafruit_ZeroDMA

https://adafru.it/Cxm

https://adafru.it/Cxm

Messages are in the file "**messages.h**," in the messages[] array starting around line 16. Be mindful of your syntax... quotes at each end of the string and a comma between each list item...or the code won't compile.

Other than the **space** character, which provides a **short pause** between words, there is **no punctuation** on the spirit board...only the letters **A through Z** and numbers **0 through 9** can be used. There are a few special exceptions but they require a peculiar syntax:

- Inserting "**\x1**" (backslash, x, one) in a string will make the planchette go to the word "**YES**," as when answering a

question (follow this, or any other special character, with one or more **spaces** if you want it to **pause** there).

- **\x2** goes to "**NO**."
- **\x3** goes to the **CENTER** of the "**GOOD BYE**" phrase.
- **\x4** goes to the **START** of "**GOOD BYE**."
- **\x5** goes to the **END** of "**GOOD BYE**" (you'll notice several of the example messages finish with "**\x4\x5**" to scroll across the entire "GOOD BYE" phrase).
- **\x6** goes to the center of the "**SPIRIT BOARD**" label.

The board and planchette graphics are ***not* easily customized**. The next page explains some of the program's internals which experienced programmers might be able to work from.

How it Works

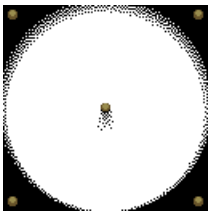
Everything that talks to the accelerometer or the capacitive touch pads is fairly vanilla Arduino code and easily figured out. The candle flicker effect of the backlight was derived from the Circuit Playground Jack-o'-Lantern guide (https://adafru.it/CpQ). Let's talk about ***graphics!***

The code that makes this work derives heavily from a prior project — the Electronic Animated Eyes (https://adafru.it/j6D) — which similarly presents full-screen smooth animation.

If you've worked with the Adafruit GFX library (https://adafru.it/doL) you've seen that it tends to be very "pixel-y"...that it takes time to draw and erase objects and there are very visible artifacts when trying to show animation. This is necessary for a couple reasons: that library has to work within the limited memory constraints of 8-bit AVR microcontrollers, and also it presents graphical objects as *persistent* — old graphics remain on the display as new graphics are overlaid.

The spirit board and eye code take a different approach that's only practical with the speed and RAM of 32-bit SAMD microcontrollers: the entire screen, *every pixel,* is computed and updated for *every frame* of animation. The normal display graphics library is still used to initialize the hardware and to specify a "window" of pixels to be updated (the full screen)...but after that the code goes in a very different direction, there are no drawPixel() or even drawBitmap() calls...

These two images — the spirit board and planchette — are stored in flash memory:



To do this, they had to be converted into big arrays (in the "graphics.h" file), using some Python code (shown later) to read the compressed PNG images and reorganize this into a list of unsigned 32-bit integers, uncompressed but "bit

packed" — 2 bits per pixel (4 possible colors in each image), so each integer value contains 16 pixels:

```
const uint32_t planchetteData[] = {
  0x55555555, 0x55555555, 0x51515555, 0x44414441, 0x51111414, 0x55555555,
  0x55555555, 0x55555555, 0x55555555, 0x55555555, 0x45155155, 0x11141114,
  0x04444101, 0x55544440, 0x55555555, 0x55555555, 0x55555555, 0x55555555,
... etc.
```

Stored this way, the board uses about 110K of program flash space, planchette about 4K (out of 256K available on the chip).

At the bottom of the graphics.h file are two color palettes, each containing four elements (since there are two bits…four possible values…per pixel):

```
const uint16_t boardPalette[]      = { 0x6531, 0x2F8C, 0x3BEF, 0x99DE },
               planchettePalette[] = { 0x0000, 0x0000, 0x699C, 0x8562 };
```

Though color palettes are sometimes present as a graphics hardware feature, in this case they're just *look-up tables* that our code uses to expand each 2-bit pixel value to a 16-bit color value (5 bits red + 6 bits green + 5 bits blue) that will be issued to the display. (Scrolling and overlays are also sometimes graphics hardware features…but in this case again we're doing these effects all on the CPU…the ST7735 display driver used in the Hallowing screen is fairly basic.)

The substance of the drawing code…the **drawFrame()** function, starting around line 309 in Hallowing_Spirit_Board.ino…works through **every pixel of the display**, processing **scanlines** from **top to bottom**, and on each **scanline** processing columns from **left to right**.

For each pixel of the screen the code first looks up the corresponding pixel in the planchette graphics array. This is relatively straightforward as the planchette image is exactly the same size as the screen…pixel (x, y) on the screen is pixel (x, y) in the image. Some bit-shifting and -masking is necessary as the graphics are stored in a 2-bits-per-pixel format…producing an index from 0 to 3.

If the planchette pixel value is 1, 2 or 3, this is used as an index into the planchettePalette[] array…the corresponding 16-bit value is read from the array and added to a one-scanline memory buffer which will later be issued to the display. These pixel appear opaque.

If the planchette pixel value is 0, we *disregard* the planchette palette and do a *second* 2-bit look-up in the board graphics array (some additional bit shenanigans are required since we're extracting just a section of this image, it's not pixel-aligned like the planchette), then retrieve the corresponding 16-bit value from the boardPalette[] array to add to the display buffer. This is how we get the transparent overlay effect.

The math alone would clobber lesser microcontrollers, but the *real* **SECRET SAUCE THAT MAKES THIS ALL WORK** is ***direct memory access*** or ***DMA.*** This is a capability of current devices like the SAMD21 microcontroller at the heart of Hallowing that allows peripherals (such as our LCD display on the SPI bus) to communicate directly with memory

without the CPU's intervention at every step...one sets up an operation (basically "send this buffer of data over SPI"), initiates the transfer, and it then proceeds in the background. Meanwhile, we still have *100% full use of every cycle of the CPU.* In the past we'd have to stop processing to handle the transfer, but now this is "free time." I like to say this chip has a good *walking to chewing gum* ratio — it can do a lot at once.

DMA can be really difficult to set up properly but we've taken care of a lot of the dirty work into our Adafruit_ZeroDMA library (https://adafru.it/lnb).

What we do then is **render one scanline in RAM** while the **prior scanline transfers over SPI using DMA** concurrently, then switch the render/transfer indexes between lines. This way, we don't need to buffer the entire image in RAM, only two scanlines' worth...just 512 bytes! If we can balance the render time and transfer time to operate concurrently, it's *just as fast as doing the SPI transfers alone,* like having one big image that we're simply sending out the SPI bus.

More Shenanigans

Because the SAMD21 is **little-endian** (16- and 32-bit values are stored least significant byte first) but the ST7735 SPI interface expects **big-endian** data (most significant byte first), 16-bit colors are stored "pre-swapped" in the palettes to avoid having to do this operation on every pixel.

16-bit colors destined for the ST7735 driver contain 5 bits red data, 6 bits green and 5 bits blue. For example, solid red would be represented binarily as **0b1111100000000000**. In hexadecimal notation, that's **0xF800**. But the color palette format used in this code would instead use **0x00F8**...the high and low bytes are reversed. It would be a minor operation to do that swap in code, perhaps a single cycle per pixel, but by swapping bytes when they go in the table we get this operation for free.

The format of the packed pixel data is also a little unconventional. For each 32-bit value (representing a span of 16 pixels, at 2 bits per pixel), it's fairly common in graphics coding to have the most significant bits represent the leftmost pixel...for example, a single 2-bit pixel with an index of "1" followed by 15 "0" pixels would be represented in hexadecimal notation as **0x40000000**. But this code stores the pixels within each value in the *reverse* order...not the *bits,* but each *2-bit value*...so the most significant two bits are the *rightmost* pixel within that 16-pixel span, making the prior example instead **0x00000001**. Formatting the data this way seemed to make the rendering code a little simpler and might avoid some shift and multiply operations. This might be wrong, I haven't actually benchmarked different approaches, just call it a hunch.

If you want to have a go at making your own (a "Luigi board" perhaps?), here's the **Python** code that generated the tables in graphics.h. This was run on my regular desktop machine...not the Hallowing board or any other CircuitPython device...just regular Python, command-line style. The output was redirected to a file and manually cleaned up a bit before merging into the Arduino sketch.

This script *does not generate the color palettes,* though I suppose it could. Instead I just entered those manually by querying the color palette entries in Photoshop and doing the 5/6/5 color reduction and byte-swap operations myself. Note also the normally-commented-out "p = pixels[x, y] ^ 1" line, which is there to reorder the color palette indices when converting the planchette image (so index 0 will be the transparent color). That's the thing with one-off disposable code like this...sometimes you just have to cobble in a solution, it's not meant to be elegant or performant.

If you change the graphics significantly, you'll need to update the coordinates of each letter and number in the coord[] array messages.h. The script doesn't do this nor do I have an automatic technique...it was just a tedious matter of typing in the cursor coordinates (from Photoshop's "Info" palette) for each character's center pixel.

```
#!/usr/bin/python

# 2-bit image converter -- generates PROGMEM arrays for Arduino sketches
```

```
# from PNG or GIF images.  Requires PIL or Pillow library.  This is NOT a
# general-purpose image-to-array converter, it's fairly specific to the
# 2-bit Spirit Board project and DOES NOT handle other image cases!

import sys
from PIL import Image
from os import path

# FORMATTED HEX OUTPUT ------------------------------------------------

class HexTable:

  # Initialize counters, etc. for write() function below
  def __init__(self, count, columns=12, digits=2):
    self.hexLimit   = count   # Total number of elements in array
    self.hexCounter = 0       # Current array element number 0 to hexLimit-1
    self.hexDigits  = digits  # Digits per array element (after 0x)
    self.hexColumns = columns # Max number of elements before line wrap
    self.hexColumn  = columns # Current column number, 0 to hexColumns-1
    # hexColumn is initialized to columns to force first-line indent

  # Write hex value (with some formatting for C array) to stdout
  def write(self, n):
    if self.hexCounter > 0:
      sys.stdout.write(",")                     # Comma-delimit prior item
      if self.hexColumn < (self.hexColumns - 1): # If not last item on line,
        sys.stdout.write(" ")                   # append space after comma
    self.hexColumn += 1                         # Increment column number
    if self.hexColumn >= self.hexColumns:       # Max column exceeded?
      sys.stdout.write("\n  ")                  # Line wrap, indent
      self.hexColumn = 0                        # Reset column number
    sys.stdout.write("{0:#0{1}X}".format(n, self.hexDigits + 2))
    self.hexCounter += 1                        # Increment item counter
    if self.hexCounter >= self.hexLimit: print(" };\n"); # Cap off table

# IMAGE CONVERSION ----------------------------------------------------

def convertImage(filename):
    prefix = path.splitext(path.split(filename)[1])[0]
    im     = Image.open(filename)

    if im.mode == 'P':

      # PALETTED IMAGE (assume 2-bit for now)
      # 16 pixels per 32 bit value
      pixels = im.load()
      hex    = HexTable(((im.size[0] + 15) / 16) * im.size[1], 6, 8)

      sys.stderr.write("Image OK\n")
      sys.stdout.write(
        "#define %s_WIDTH  %d\n"
        "#define %s_HEIGHT %d\n\n"
        "const uint32_t %sData[] = {" %
        (prefix.upper(), im.size[0], prefix.upper(), im.size[1], prefix))

      for y in range(im.size[1]):
        bits = 0
        sum  = 0
        for x in range(im.size[0]):
          p     = pixels[x, y]
```

```
          # I wanted white to be index 0 in the planchette image but
          # Photoshop had other plans on export.  Soooo I manually
          # fiddle with the palette index here in that case...
          #p     = pixels[x, y] ^ 1
          sum  += p << bits
          bits += 2
          if bits >= 32:
            hex.write(sum)
            bits = 0
            sum  = 0
        if bits > 0: # Scanline pad
          hex.write(sum)

for i, filename in enumerate(sys.argv): # Each argument...
  if i == 0: continue # Skip first argument; is program name
  convertImage(filename)
```