Linux Fu: Scripting for Binary Files

Originalartikel

Backup

<html> If you ever need to write a binary file from a traditional language like C, it isn’t all that hard to do. About the worst thing you might have to deal with is attempts to fake line endings across Windows and Linux, but there’s usually a way to turn that off if it is on by default. However, if you are using some type of scripting language, binary file support might be a bit more difficult. One answer is to use a tool like xxd or t2b (text-to-binary) to handle the details. You can find the code for t2b on GitHub including prebuilt binaries for many platforms. You should be able to install xxd from your system repository. These tools take very different approaches. You might be familiar with tools like od or hexdump for producing readable representations of binary files. The xxd tool can actually do the same thing — although it is not as flexible. What xxd can even reverse itself so that it can rebuild a binary file from a hex dump it creates (something other tools can’t do). The t2b tool takes a much different approach. You issue commands to it that causes it to write an original hex file. Both of these approaches have some merit. If you are editing a binary file in a scripting language, xxd makes perfect sense. You can convert the file to text, process it, and then roll it back to binary using one program. On the other hand, if you are creating a binary file from scratch, the t2b program has some advantages, too. I decided to write a few test scripts using bash to show how it all works. These aren’t production scripts so they won’t be as hardened as they could be, but there is no reason they couldn’t be made as robust as you were willing to make them. <h2>Cheating a Little</h2> I decided to write two shell scripts. One will generate an image file. I cheated in two ways there. First, I picked the PPM (Portable Pix Map) format which is very simple to create. And second I ignored the format that uses ASCII instead of binary. That’s not strictly cheating because it does make a larger file, as you’d expect. So there is a benefit to using the binary format. The other script takes a file in the same format and cuts the color values within it by half. This shows off both tools since the first job is generating an image file from data and the second one is processing an image file and writing out a new one. I’ll use t2b for the first job and xxd for the second. <h2>PPM File Format</h2> <img data-attachment-id=„309835“ data-permalink=„https://hackaday.com/2018/06/22/linux-fu-the-great-power-of-make/tux2/“ data-orig-file=„https://hackadaycom.files.wordpress.com/2018/05/tux2.png“ data-orig-size=„958,1245“ data-comments-opened=„1“ data-image-meta=„{"aperture":"0","credit":"","camera":"","caption":"","created_timestamp":"0","copyright":"","focal_length":"0","iso":"0","shutter_speed":"0","title":"","orientation":"0"}“ data-image-title=„tux2“ data-image-description=„“ data-medium-file=„https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=308“ data-large-file=„https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=481“ class=„alignright wp-image-309835 size-thumbnail“ src=„https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=192&h=250“ alt=„“ width=„192“ height=„250“ srcset=„https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=192&h=250 192w, https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=384&h=500 384w,

https://hackadaycom.files.wordpress.com/2018/05/tux2.png?w=308&h=400 308w“ sizes=„(max-width: 192px) 100vw, 192px“/>The PPM format is part of a family of graphics formats from the 1980s. They are very simple to construct and deconstruct, although they aren’t known for being small. However, if you needed to create a graphic from a Raspberry Pi program, it is sometimes handy to create them using this simple file format and then use ImageMagick or some other tool to convert to a nicer format like PNG. There are actually three variants of the format. One for black and white, one for grayscale, and another for color. In addition, each of them can contain ASCII data or binary data. There is a very simple header which is always in ASCII. We’ll only worry about the color format. The header will start with the string “P6.” That usually ends with a newline, although defensively, you ought to allow for any whitespace character to end the header fields. Then the X and Y limits — in decimal and still in ASCII — appear separated by whitespace. This is usually really a space and a newline at the end. The next part of the header is another ASCII decimal value indicating the maximum value for the color components in the image. After that, the data is binary RGB (red/green/blue) triplets. By the way, if the P6 had been a P3, everything would remain the same, but the RGB triplets would be in ASCII, not binary. This could be handy in some cases but — as I mentioned — will result in a larger file. Here’s a sample header with a little bit of binary data following it: The green text represents hex numbers and the other boxes contain ASCII characters. You can see the first 15 bytes are header and after that, it is all image data. <h2>T2B</h2> The t2b program takes a variety of commands to generate output. You can write a string or various sizes of integers. You can also do things like repeat output a given number of times and even choose what to output based on conditions. There’s a way to handle variables and even macros. As an example, my script will write out an image with three color bars in it. The background will be black with a white border. The color bars will automatically space to fit the box. I won’t use too many of the t2b features, but I did like using the macros to make the resulting output easier to read. Here’s the code for creating the header (with comments added): <pre class=„brush: bash; title: ; notranslate“ title=„“> strl P6 # Write P6 followed by a newline (no quotes needed because no whitespace in the string) str $X # Write the X coordinate (no newline) u8 32 # a space strl $Y # The Y coordinate (with newline) strl 255 #

Maximum subpixel value (ASCII) </pre> That’s all there is to it. The RGB triples use the u8 command, although you could probably use a 24-bit command, too. I also set up some macros for the colors I used: <pre class=„brush: bash; title: ; notranslate“ title=„“> macro RED

```
begin
  u8 255
  times 2 u8 0
  endtimes
```

endmacro </pre> Once you have the t2b language down, the rest is just math. You can find the complete code on GitHub, but you’ll see it just computes 7 equal-sized regions and draws different colors as it runs through each pixel in a nested set of for loops. There’s also a one-pixel white border around the edges for no good reason. When you want to run the code you can either specify the X and Y coordinates or take the 800×600 default: <pre> ./colorbar.sh 700 700 | t2b >outputfile.ppm </pre> If you intercept the output before the t2b program, you’ll see the commands rolling out of the script. Here’s the default output to the ppm file: <h2>Shades of Gray</h2> The other script is a little different. The goal is to divide all the color values in a PPM file in half. If it were just binary data, that would be easy enough, but you need to skip the header so as not to corrupt it. That takes a little extra work. I used gawk (GNU awk) to make the work a little simpler. The code expects output from xxd, which looks like this: <pre class=„brush: bash; title: ; notranslate“ title=„“> 00000000: 5036 0a38 3030 2036 3030 0a32 3535 0aff P6.800 600.255.. 00000010: ffff ffff ffff ffff ffff ffff ffff ffff ..... 00000020: ffff ffff ffff ffff ffff ffff ffff ffff ..... 00000030: ffff ffff ffff ffff ffff ffff ffff ffff ..... 00000040: ffff ffff ffff ffff ffff ffff ffff ffff ..... </pre> The address isn’t important to us. You can ask xxd to suppress it, but it is also easy to just skip it. The character representations to the right aren’t important either. The xxd program will ignore that when it rebuilds the binary. Here’s the code in awk (which is embedded in the shell script): <pre class=„brush: bash; title: ; notranslate“ title=„“> # need to find 4 white space fields BEGIN { noheader=4 }

```
  {
  lp=1
  }
  {
  split($0, chars, "")
```

skip initial address

```
  while (chars[lp++]!=":");
  n=0;  # # of bytes read
```

get two characters

```
  while (n&lt;16 &amp;&amp; lp&lt;length(chars)) { # heuristically two space characters out of xxd ends the hex dump line (ascii follows) if (chars[lp] ~ /[ \t\n\r]/) { if (chars[++lp] ~ /[ \t\n\r]/) { break; # no need to look at rest of line } } b=chars[lp++] chars[lp++]; n++; # if header then skip white space if (noheader&gt;0) {
    if (b=="20" || b=="0a" || b=="0d" || b=="09") noheader--;
  }
  else {
  # if not header than /2
   bn=strtonum("0x" b)/2;
   bs=sprintf("%02x",bn);
   chars[lp-2]=substr(bs,1,1);
   chars[lp-1]=substr(bs,2,1);
  }
}
```

recombine array and print

```
p=""
for (i=1;i&lt;=length(chars);i++) p=p chars[i];
print p
}
```

</pre> The awk code simply skips the address and then pulls up to 16 items from a line of data. The first task is to count whitespace characters to skip over the header. I made the assumption that there would not be runs of whitespace, although a more robust program would probably consume multiple spaces (easy to fix). After that, each byte gets divided and reassembled. This task is more character oriented and awk doesn’t handle characters well without a trick. <img data-attachment-id=„312269“ data-permalink=„https://hackaday.com/2018/06/29/linux-fu-scripting-for-binary-files/fifty-2/“ data-orig-file=„https://hackadaycom.files.wordpress.com/2018/06/fifty.png“ data-orig-size=„800,600“ data-comments-opened=„1“ data-image-meta=„{"aperture":"0","credit":"","camera":"","caption":"","created_timestamp":"0","copyright":"","focal_length":"0","iso":"0

","shutter_speed":"0","title":"","orientation":"0"}“ data-image-title=„fifty“ data-image-description=„“ data-medium-file=„https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=400“ data-large-file=„https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=800“ class=„alignleft size-thumbnail wp-image-312269“ src=„https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=250&h=188“ alt=„“ width=„250“ height=„188“ srcset=„https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=250&h=188 250w, https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=500&h=376 500w, https://hackadaycom.files.wordpress.com/2018/06/fifty.png?w=400&h=300 400w“ sizes=„(max-width: 250px) 100vw, 250px“/>In particular, I used the split command to convert the current line into an array with each element containing a character. This includes any whitespace characters because I used an empty string as the split delimiter: <pre> split($0, chars, „“) </pre> After processing the array — which isn’t hard to do — you can build a new string back like this: <pre> p=„“ for (i=1;i<=length(chars);i++) p=p chars[i]; </pre> The output file will feed back to xxd with the -r option and you are done: <pre> xxd infile.ppm | ./half.sh | xxd -r >outfile.ppm </pre> <h2>Two is the Loneliest</h2> This is a great example of how the Unix philosophy makes it possible to build tools that are greater than the sum of their parts. A simple program changes a text-processing language like awk into a binary file manipulation language. Great. By the way, if your idea of manipulating binary is Intel hex or Motorola S records, be sure to check out the srec_cat and related software which can manipulate those, too. Once you have a bunch of binary files, you might appreciate an online hex editor. By the way, a couple of years ago, I mentioned using od to process binary files in awk. That’s still legitimate, of course, but xxd allows you to go both ways, which is a lot more useful. </html>