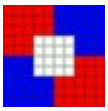
What's In A GIF



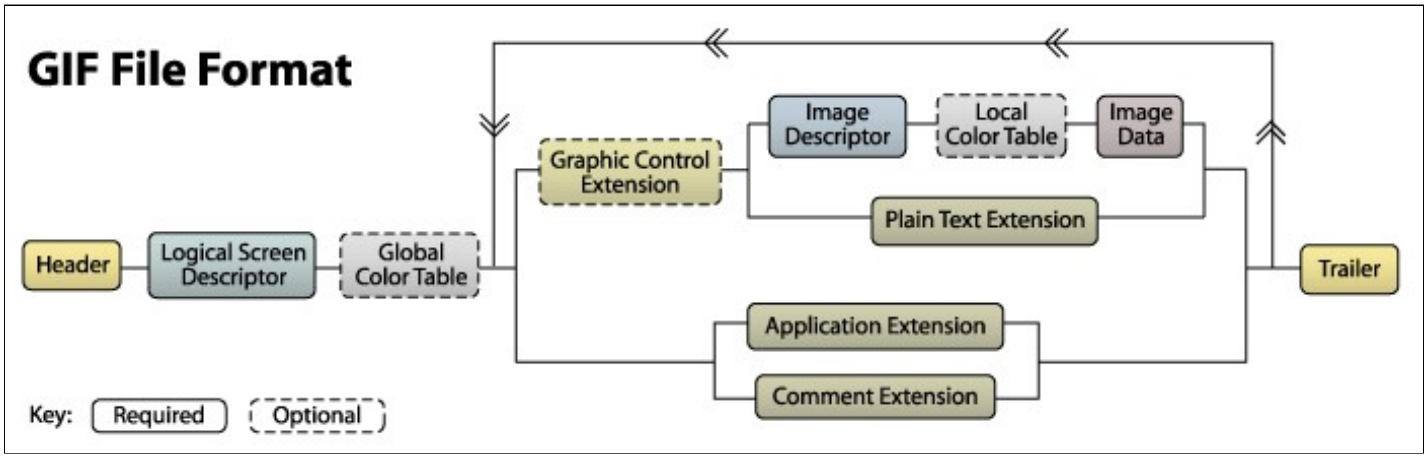
(Bits and bytes)

Back to the What's In A GIF index page.

The authority on the content of GIFs is the GIF89a specification. Originally developed at CompuServe in the late 1980s, it is now a W3C standard.

A GIF file is made up of a sequence of data blocks. The first two blocks are fixed length and fixed format. Later ones are variable length but self-describing; they consisting of a byte identifying the block type, followed by a payload length byte, followed by payload.

The following railroad diagram shows all of the different types of blocks and where they can be in the file. Every path following the arrows corresponds to a valid block sequence. The large middle secction, in particular, can be repeated any number of times.

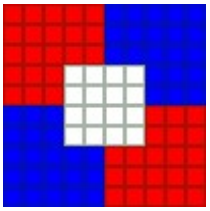


We will learn more by walking through a sample GIF file. You can see the sample file and its corresponding bytes below.

Actual Size

(10x10)

Enlarged



(100x100)

Bytes

```
47  49  46  38  39  61  0A  00  0A  00  91
00  00  FF  FF  FF  FF  00  00  00  00  FF
00  00  00  21  F9  04  00  00  00  00  00
2C  00  00  00  00  0A  00  0A  00  00  02
16  8C  2D  99  87  2A  1C  DC  33  A0  02
75  EC  95  FA  A8  DE  60  8C  04  91  4C
01  00  3B
```

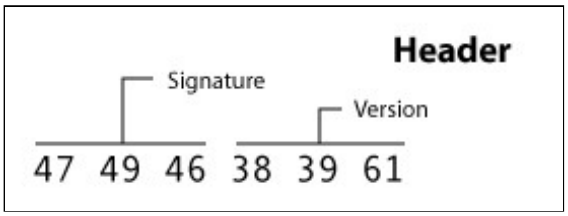
Note that not all possible block types are represented in this sample file. Later we'll provide samples of missing block types where appropriate. The different types of blocks include: header, logical screen descriptor, global color table, graphics control extension, image descriptor, local color table, image data, plain text extension, application extension, comment extension, and trailer. Let's get started with the first block!

Header Block

From the sample file: `47 49 46 38 39 61`

All GIF files must start with a header block. The header takes up the first six bytes of the file. These bytes should all correspond to ASCII character codes. The first three bytes are called the **signature**. These should always be "GIF" (ie 47="G", 49="I", 46="F"). The next three specify the **version** of the specification that was used to encode the image.

Normally the version string will be either "89a" (ie 38="8", 39="9",61="a") or "87a" (ie 38="8", 37="7",61="a"). All modern GIF-processing software recognizes both versions, For maximum compatibility, GIFLIB will normally write an 87a signature unless the file contains GIF89 features.



Logical Screen Descriptor

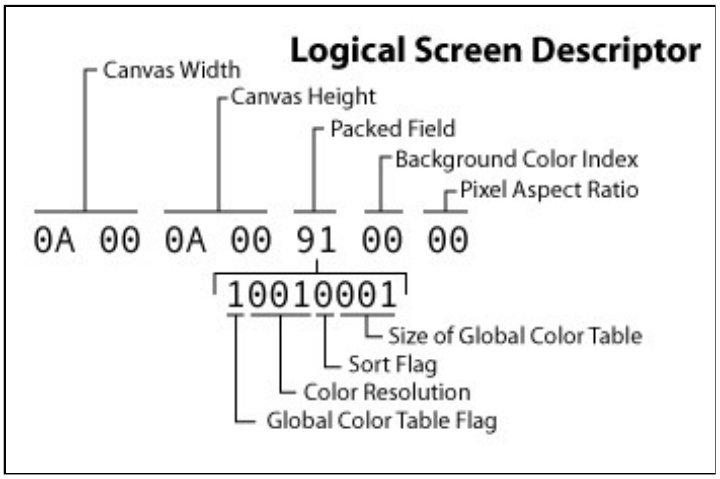
From Sample File: `0A 00 0A 00 91 00 00`

The logical screen descriptor always immediately follows the header. This block tells the decoder how much room this image will take up. It is exactly seven bytes long. It starts with the **canvas width** and **canvas height**. These value can be found in the first two pairs of two bytes each. Both are 16-bit, nonnegative integers (0-65,535).

As with all the other multi-byte values in the GIF format, the least significant byte is stored first (little-endian format). This means where we would read `0A 00` from the byte stream, we would normally write it as `000A` which is the same as 10. Thus the width of our sample image is 10 pixels. As a further example 255 would be stored as `FF 00` but 256 would be `00 01` .

The canvas width and height are usually ignored by modern viewers. The GIF format seems to have been designed with the idea that viewers would render multiple images in a GIF on a common canvas, giving an effect like a picture wall. But nowadays multiple-image GIFs are generally used either as animations in which each sub-image is a frame or as image libraries, with the GIF client handling compositing into some canvas about which the GIF format holds no information.Thus, the canvas width and height are mainly fossils. GIFLIB does extract them and allow you to set them, however.

The next byte contains four fields of packed data, the "logical screen descriptor". To understand these, we need to expand the byte `91` to binary as `10010001` and look at the fields inside it.



The first (most-significant) bit is the **global color table flag**. If it's 0, then there is no global color table. If it's 1, then a global color table will follow. In our sample image, we can see that we will have a global color table (as will usually be the case).

The next three bits are the **color resolution**. They are only meaningful if there is a global color table, and allow you to compute its size. If the value of this filed is N, the number of entries in the global color table will be 2 ^ (N+1) - that is, two raised to the power (N+1). Thus, the `001` in the sample image represents 2 bits/pixel; `111` would represent 8 bits/pixel.

The GIF format shows its age here. A more modern design would simply have allocated a byte or two for color table length. But GIF was designed when memory was much more expensive than it is today, and the designers felt strong pressure to economize on every bit. The consequence is that color table lengths have to be an exact power of two. Perversely, this can force a waste of memory space in images with odd color counts.

The next single bit is the **sort flag**. If the values is 1, then the colors in the global color table are sorted in order of "decreasing importance," which typically means "decreasing frequency" in the image. This can help the image decoder, but is not required. In the sample file this value has been left at 0.

The sort flag reflected the high cost of dual-port memory at the time the GIF specification was written in the late 1980s. That kind of limit disappeared in the mid-1990s, and modern GIF software ignores this flag. Until version 5.0, GIFLIB ignored it on input and zeroed it on output; 5.0 and later versions read and preserve it.

The next byte gives us the **background color index**. This byte is only meaningful if the global color table flag is 1, and if there is no global color table, this byte should be 0.. To understand it you have to remember the original "picture wall" rendering model for GIFs in which sub-images are composited onto a larger canvas. It represents which index in the global color table should be used for pixels on the virtual canvas that aren't overlayed by an image. GIFLIB supports reading and setting this byte, but modern viewers and browsers generally have no use for it.

The last byte of the logical screen descriptor is the **pixel aspect ratio**. Modern viewers don't use this. Until 5.0, GIFLIB ignored this flag on input and zeroed it on output; now it is read and preserved if present. The GIF standard doesn't give a rationale for it, but it seems likely that the designers intended it for representing image captures from the analog television of the day, which had rectangular pixel-equivalents. The GIF specification says that if there was a value specified in this byte, N, the actual ratio used would be (N + 15) / 64 for all N<>0.

Global Color Table

From the sample file: `FF FF FF FF 00 00 00 00 FF 00 00 00`

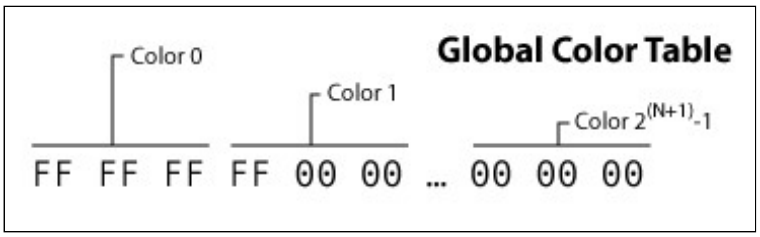
GIFs can have either a **global color table** or local color tables for each sub-image. Each color table consists of a list of RGB (Red-Green-Blue) color component intensities, three bytes for each color, with intensities ranging from 0 (least) to 255 (most). The color (0,0,0) is deepest black, the color (255,255,255) brightest white. The other extreme colors are red at (255,0,0), green at (0,255,0) and blue at (0,0,255).

As previously noted, the length of the global color table is 2^(N+1) entries where N is the value of the color depth field in the logical screen descriptor. The table will take up 3*2^(N+1) bytes in the stream.

Size In Logical Screen Desc	Number Of Colors	Byte Length
0	2	6
1	4	12
2	8	24
3	16	48
4	32	96
5	64	192
6	128	384
7	256	768

Our sample file has a global color table size of 1. This means it holds 2^(1+1)=2^2=4 colors. We can see that it takes up 12, (3*4), bytes as expected. We read the bytes three at a time to get each of the colors. The first color is #FFFFFF (white). This value is given an index of 0. The second color is #FF0000 (red). The color with an index value of 2 is #0000FF (blue). The last color is #000000 (black). The index numbers will be important when we decode the actual image data.

Note that this block is labeled as "optional." Not every GIF has to specify a global color table. However, if the global color table flag is set to 1 in the logical screen descriptor block, the color table is then required to immediately follow that block.



Graphics Control Extension

From the sample file: `21 F9 04 00 00 00 00 00`

Graphic control extension blocks are used to specify transparency settings and control animations. They are an optional GIF89 extension. The semantics of this extension will be described in detail in a later section (see Transparency and Animation); for completeness we'll describe the data fields here.

The first byte is the **extension introducer**. All *extension* blocks begin with `21`. Next is the **graphic control label**, `F9`, which is the value that flags this as a graphic control extension. Third up is the total **block size** in bytes. Next is a packed field. Bits 1-3 are reserved for future use. Bits 4-6 indicate **disposal method**. The penultimate bit is the **user input flag** and the last is the **transparent color flag**. The **delay time** value follows in the next two bytes stored in unsigned format. After that we have the **transparent color index** byte. Finally we have the **block terminator** which is always `00`.

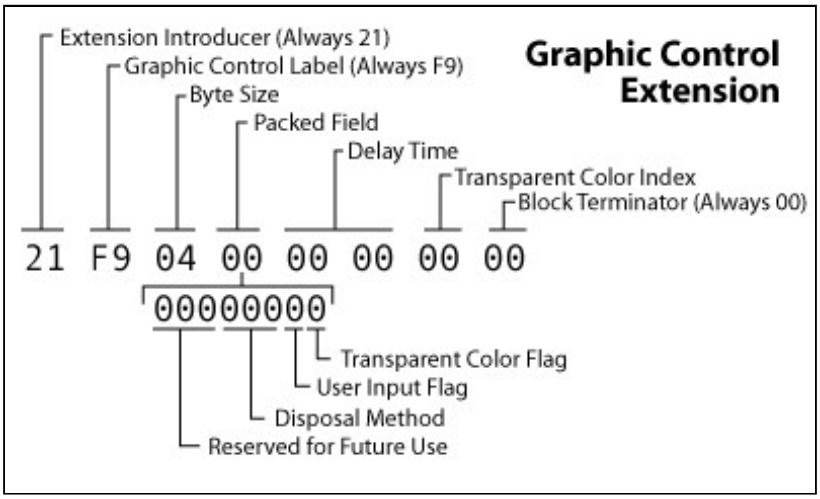


Image Descriptor

From the sample file: `2C 00 00 00 00 0A 00 0A 00 00`

A single GIF file may contain multiple images. In the original GIF rendering model these wwere meant to be composited onto a larger virtual canvas. Nowadays nultiple images are normally used for animations.

Each image begins with an image descriptor block. This block is exactly 10 bytes long.

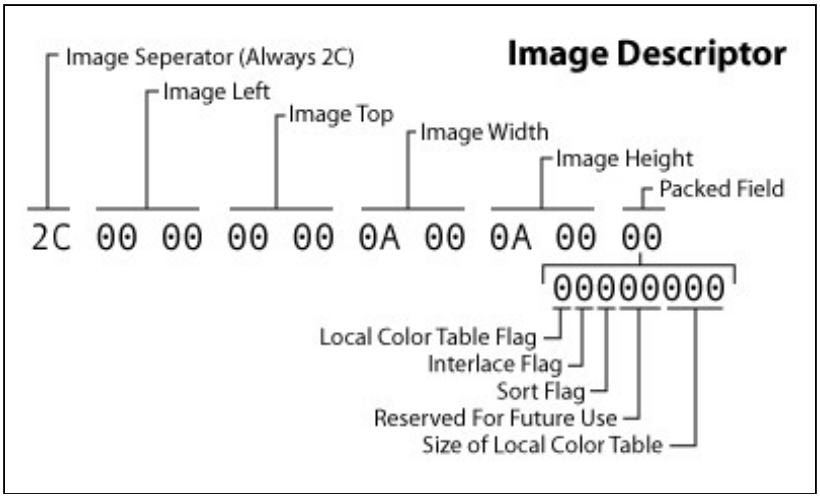
The first byte is the **image separator**. Every image descriptor begins with the value `2C`. The next 8 bytes represent the location and size of the following image.

An image in the stream may not necessarily take up the entire canvas size defined by the logical screen descriptor. Therefore, the image descriptor specifies the **image left position** and **image top position** of where the image should begin on the canvas. Both these fields are usually ignored by modern viewers and browsers.

Next, this block specifies the **image width** and **image height**. Each of these values is in the two-byte, unsigned little-endian format. Our sample image indicates that the image starts at (0,0) and is 10 pixels wide by 10 pixels tall. (This image does take up the whole canvas size.)

The last byte is another packed field. In our sample file this byte is 0 so all of the sub-values will be zero. The first (most significant) bit in the byte is the **local color table flag**. Setting this flag to 1 allows you to specify that the image data that follows uses a different color table than the global color table. (More information on the local color table follows.)

The second bit is the **interlace flag**. Interlacing changes the way images are rendered onto the screen in a way that may reduce annoying visual flicker. The effect of interlacing on a display is that the first pass of appears immediately, displaying the graphic as a first as a blur and then sharpenining it up as later passes fill in lines. That allows the human viewer to at least get an idea of what's coming up rather than waiting for the entire image to be painted, line by line. [See an example]. To support this, the scan lines of the image need to be stored in a different order than the normal top-down, separated into sections that will be rendered in four separate passes.



Local Color Table

A local color table is organized the same as a global color table. The local color table would always immediately follow an image descriptor but will only be there if the local color table flag is set to 1. It is effective only for the block of image data that immediately follows it. If no local color table is specified, the global color table is used for the following image data.

The size of the local color table can be calculated by the value given in the image descriptor. Just like with the global color table, if the image descriptor specifies a size of N, the color table will contain 2^(N+1) colors and will take up 3*2^(N+1) bytes.

Image Data

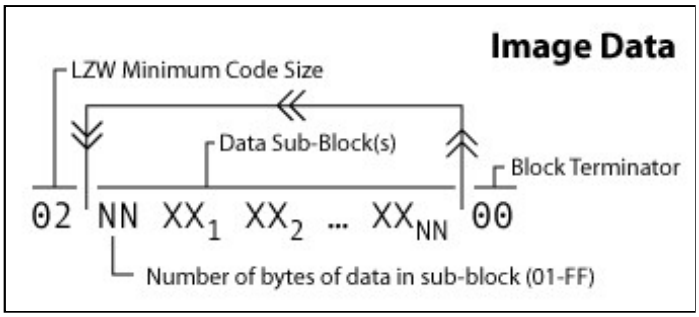
From the sample file: `02 16 8C 2D 99 87 2A 1C DC 33 A0 02 75 EC 95 FA A8 DE 60 8C 04 91 4C 01 00`

Finally we get to the actual image data. The image data is composed of a series of output codes which tell the decoder which colors to emit to the canvas. These codes are combined into the bytes that make up the block.

There's another section on decoding these output code into an image (see LZW Image Data). Here we'll just see how to determine how long the block will be.

The first byte of this block is the **LZW minimum code size**. This value is used to decode the compressed output codes. (Again, see the section on LZW compression to see how this works.) The rest of the bytes represent *data sub-blocks*. Data sub-blocks are are groups of 1 - 256 bytes. The first byte in the sub-block tells you how many bytes of actual data follow. This can be a value from 0 (`00`) it 255 (`FF`). After you've read those bytes, the next byte you read will tell you now many more bytes of data follow that one. You continue to read until you reach a sub-block that says that zero bytes follow.

You can see our sample file has a LZW minimum code size of 2. The next byte tells us that 22 bytes of data follow it (`16` hex = 22). After we've read those 22 bytes, we see the next value is 0. This means that no bytes follow and we have read all the data in this block.



Plain Text Extension

Example (not in the sample file): `21 01 0C 00 00 00 00 64 00 64 00 14 14 01 00 0B 68 65 6C 6C 6F 20 77 6F 72 6C 64 00`

The GIF89 specification allows you to specify text captions to be overlayed on the following image. This feature never took off; browsers and image-processing applications such as Photoshop ignore it, and GIFLIB doesn't try to interpret it.

The block begins with an **extension introducer** as all extension block types do. This value is always `21`. The next byte is the **plain text label**. This value of `01` is used to distinguish plain text extensions from all other extensions. The next byte is the **block size**. This tells you how many bytes there are until the actual text data begins, or in other words, how many bytes you can now skip. The byte value will probably be `0C` which means you should jump forward 12 bytes. The text that follows is encoded in data sub-blocks (see Image Data to see how these sub-blocks are formed). The block ends when you reach a sub-block of length 0.

Application Extension

Example (not in sample file): `21 FF 0B 4E 45 54 53 43 41 50 45 32 2E 30 03 01 05 00 00`

The GIF89 specification allows for application-specific information to be embedded in the GIF file itself. This capability is not much used. About the only known public one is the Netscape 2.0 extension (described below) which is used to loop an animated GIF file. We'll go into more detail on looping in when we talk about animation.

The Netscape 2.0 looping block must appear immediately after the global color table of the logical screen descriptor. It is 19 bytes long.

```
byte  1        : 33 (hex 0x21) GIF Extension code
byte  2        : 255 (hex 0xFF) Application Extension Label
byte  3        : 11 (hex 0x0B) Length of Application Block
                 (eleven bytes of data to follow)
bytes 4 to 11  : "NETSCAPE"
```

```
bytes 12 to 14 : "2.0"
byte  15       : 3 (hex 0x03) Length of Data Sub-Block
                 (three bytes of data to follow)
byte  16       : 1 (hex 0x01)
bytes 17 to 18 : 0 to 65535, an unsigned integer in
                 little-endian byte format. This specifies the
                 number of times the loop should
                 be executed.
byte  19       : 0 (hex 0x00) a Data Sub-Block Terminator.
```

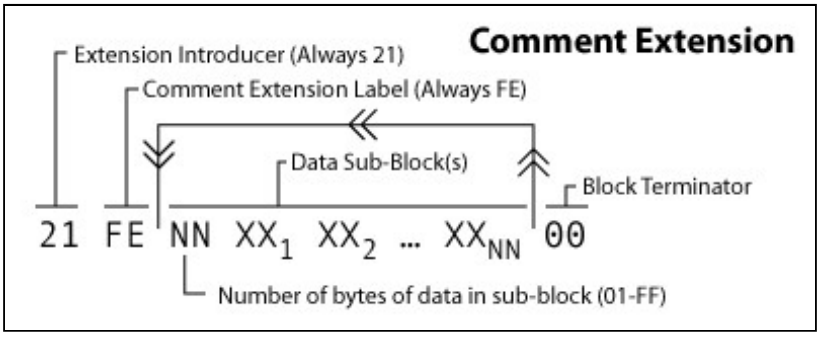
As with all extensions, we start with `21` which is the **extension introducer**. Next is the **extension label** which for application extensions is `FF`. The next value is the **block size** which tells you how many bytes there are before the actual application data begins. This byte value should be `0B` which indicates 11 bytes. These 11 bytes hold two pieces of information. First is the **application identifier** which takes up the first 8 bytes. These bytes should contain ASCII character codes that identify to which application the extension belongs. In the case of the example above, the application identifier is "NETSCAPE" which is conveniently 8 characters long. The next three bytes are the **application authentication code**. The spec says these bytes can be used to "authenticate the application identifier." With the Netscape 2.0 extension, this value is simply a version number, "2.0", hence the extensions name. What follows is the application data broken into data sub-blocks. As with the other extensions, the block terminates when you read a sub-block that has zero bytes of data.

Comment Extension

Example (not in sample file): `21 FE 09 62 6C 75 65 62 65 72 72 79 00`

One last GIF89 extension type is the comment extension. This allows you to embed ASCII text in a GIF file, and is sometimes used to include an image description, image credit, or other human-readable metadata such as the GPS location of the image capture.

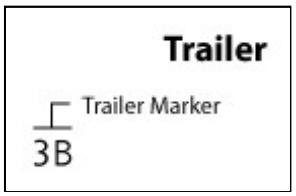
It's probably no surprise by now that the first byte is the **extension introducer** which is `21`. The next byte is always `FE` which is the **comment label**. Then we jump right to data sub-blocks containing ASCII character codes for your comment. As you can see from the example we have one data sub-block that is 9 bytes long. If you translate the character codes you see that the comment is "blueberry." The final byte, `00`, indicates a sub-block with zero bytes that follow which let's us know we have reached the end of the block.



Trailer

From sample file: `3B`

The trailer block indicates when you've reached the end of the file. It is always a byte with a value of `3B`.



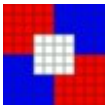
Next: LZW Image Data

Now that you know what the basic parts of a GIF file are, let's next focus our attention on how the actual image data is stored and compressed.

Continue...

Back to GIFLIB documentation

What's In A GIF



(LZW image data)

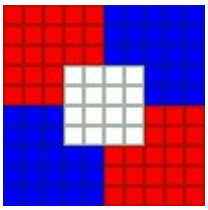
Back to the GIF index page.

Now let's look at exactly how we go about storing an image in a GIF file. The GIF format is a raster format, meaning it stores image data by remembering the color of every pixel in the image. More specifically, GIF files remember the index of the color in a color table for each pixel. To make that clearer, let's review the sample image we used in the first section.

Actual Size

(10x10)

Enlarged



(100x100)

Color Table

Index	Color
0	White
1	Red
2	Blue
3	Black

The color table came from the global color table block. The colors are listed in the order which they appear in the file. The first color is given an index of zero. When we send the codes, we always start at the top left of the image and work our way right. When we get to the end of the line, the very next code is the one that starts the next line. (The decoder will "wrap" the image based on the image dimensions.) We could encode our sample image in the following way:

> 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 0, 0, 0, 0, 2, 2, 2, 1, 1, 1, 0, 0, 0, 0, 2, 2, 2, ...

The above listing shows the sequence required to render the first five lines of the image. We could continue with this method until we've specified the color for every pixel; however, this can result in a rather large file. Luckily for us, the GIF format allows us to take advantage of repetition in our output and to compress our data.

Much of the following information came from John Barkaus's tutorial *LZW and GIF Explained*, which seems to have fallen off the web. I've tried to provide more detailed samples as well as illustrations to make the process even clearer

LZW Compression

The compression method GIF use is a variant of LZW (Lempel-Ziv-Welch) compression. To start using this method, we need a **code table**. This code table will allow us to use special codes to indicate a sequence of colors rather than just one at a time. The first thing we do is to *initialize the code table*. We start by adding a code for each of the colors in the color table. This would be a local color table if one was provided, or the global color table. (I will be starting all codes with "#" to distinguish them from color indexes.)

Code	Color(s)
#0	0
#1	1
#2	2
#3	3

#4	Clear Code
#5	End Of Information Code

I added a code for each of the colors in the global color table of our sample image. I also snuck in two special control codes. (These special codes are only used in the GIF version of LZW, not in standard LZW compression.) Our code table is now considered initialized.

Let me now explain what those special codes are for. The first new code is the *clear code* (CC). Whenever you come across the clear code in the image data, it's your cue to reinitialize the code table. (I'll explain why you might need to do this in a bit.) The second new code is the *end of information code* (EOI). When you come across this code, this means you've reached the end of the image. Here I've placed the special codes right after the color codes, but actually the value of the special codes depends on the value of the LZW minimum code size from the image data block. If the LZW minimum code size is the same as the color table size, then special codes immediatly follow the colors; however it is possible to specify a larger LWZ minimum code size which may leave a gap in the codes where no colors are assigned. This can be summarizaed in the following table.

LWZ Min Code Size	Color Codes	Clear Code	EOI Code
2	#0-#3	#4	#5
3	#0-#7	#8	#9
4	#0-#15	#16	#17
5	#0-#31	#32	#33
6	#0-#63	#64	#65
7	#0-#127	#128	#129
8	#0-#255	#256	#257

Before we proceed, let me define two more terms. First the **index stream** will be the list of indexes of the color for each of the pixels. This is the input we will be compressing. The **code stream** will be the list of codes we generate as output. The **index buffer** will be the list of color indexes we care "currently looking at." The index buffer will contain a list of one or more color indexes. Now we can step though the LZW compression algorithm. First, I'll just list the steps. After that I'll walk through the steps with our specific example.

- Initialize code table
- Always start by sending a clear code to the code stream.
- Read first index from index stream. This value is now the value for the index buffer
- <LOOP POINT>
- Get the next index from the index stream to the index buffer. We will call this index, K
- Is index buffer + K in our code table?
- Yes:
    - add K to the end of the index buffer
    - if there are more indexes, return to LOOP POINT
- No:
    - Add a row for index buffer + K into our code table with the next smallest code
    - Output the code for just the index buffer to our code steam
    - Index buffer is set to K
    - K is set to nothing
    - if there are more indexes, return to LOOP POINT
- Output code for contents of index buffer
- Output end-of-information code

Seems simple enough, right? It really isn't all that bad. Let's walk though our sample image to show you how this works. (The steps I will be describing are summarized in the following table. Numbers highlighted in green are in the index buffer; numbers in purple are the current K value.) We have already initialized our

code table. We start by doing two things: we output our clear code (#4) to the code stream, and we read the first color index from the index stream, 1, into our index buffer [Step 0].

Now we enter the main loop of the algorithm. We read the next index in the index stream, 1, into K [Step 1]. Next we see if we have a record for the index buffer plus K in the code stream. In this case we looking for 1,1. Currently our code table only contains single colors so this value is not in there. Now we will actually add a new row to our code table that does contain this value. The next available code is #6, we will let #6 be 1,1. Note that we do not actually send this code to the code stream, instead we send just the code for the value(s) in the index buffer. The index buffer is just 1 and the code for 1 is #1. This is the code we output. We now reset the index buffer to just the value in K and K becomes nothing. [Step 2].

We continue by reading the next index into K. [Step 3]. Now K is 1 and the index buffer is 1. Again we look to see if there is a value in our code table for the buffer plus K (1,1) and this time there is. (In fact we just added it.) Therefore we add K to the end of the index buffer and clear out K. Now our index buffer is 1,1. [Step 4].

The next index in the index stream is yet another 1. This is our new K [Step 5]. Now the index buffer plus K is 1,1,1 which we do not have a code for in our code table. As we did before, we define a new code and add it to the code table. The next code would be #7; thus #7 = 1, 1, 1. Now we kick out the code for just the values in the index buffer (#6 = 1,1) to the code stream and set the index buffer to be K. [Step 6].

Step	Action	Index Stream	New Code Table Row	Code Stream
0	Init	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4
1	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4
2	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#6 - 1, 1	#4 #1
3	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1
4	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1
5	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1
6	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#7 - 1, 1, 1	#4 #1 #6
7	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6
8	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6
9	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6
10	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#8 - 1, 1, 2	#4 #1 #6 #6
11	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6
12	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#9 - 2, 2	#4 #1 #6 #6 #2
13	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2
14	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2

15	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2
16	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#10 - 2, 2, 2	#4 #1 #6 #6 #2 #9
17	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9
18	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9
19	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9
20	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#11 - 2, 2, 1	#4 #1 #6 #6 #2 #9 #9
21	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9 #9
22	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9 #9
23	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9 #9
24	Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9 #9
25	Read	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...		#4 #1 #6 #6 #2 #9 #9
26	Not Found	1 1 1 1 1 2 2 2 2 2 1 1 1 1 ...	#12 - 1, 1, 1, 1	#4 #1 #6 #6 #2 #9 #9 #7

I've included a few more steps to help you see the pattern. You keep going until you run out of indexes in the index stream. When there is nothing new to read, you simply write out the code for whatever values you may have in your index buffer. Finally you should send the end-of-information code to the code stream. In this example, that code is #5. (View the complete code table.)

As you can see we dynamically built many new codes for our code table as we compressed the data. For large files this can turn into a large number of codes. It turns out that the GIF format specifies a maximum code of #4095 (this happens to be the largest 12-bit number). If you want to use a new code, you have to clear out all of your old codes. You do this by sending the clear code (which for our sample was the #4). This tells the decoder that you are reinitializing your code table and it should too. Then you start building your own codes again starting just after the value for your end-of-information code (in our sample, we would start again at #6).

The final code stream would look like this:

> #4 #1 #6 #6 #2 #9 #9 #7 #8 #10 #2 #12 #1 #14 #15 #6 #0 #21 #0 #10 #7 #22 #23 #18 #26 #7 #10 #29 #13 #24 #12 #18 #16 #36 #12 #5

This is only 36 codes versus the 100 that would be required without compression.

LZW Decompression

At some point we will need to turn this code stream back into a picture. To do this, we only need to know the values in the stream and the size of the color table that was used. That's it. You remember that big code table we built during compression? We actually have enough information in the code stream itself to be able to rebuild it.

Again, i'll list the algorithm and then we will walk though an example. Let me define a few terms i will be using. CODE will be current code we're working with. CODE-1 will be the code just before CODE in the code stream. {CODE} will be the value for CODE in the code table. For example, using the code table we

created during compression, if CODE=#7 then {CODE}=1,1,1. In the same way, {CODE-1} would be the value in the code table for the code that came before CODE. Looking at step 26 from the compression, if CODE=#7, then {CODE-1} would be {#9}, not {#6}, which was 2,2.

- Initialize code table
- let CODE be the first code in the code stream
- output {CODE} to index stream
- <LOOP POINT>
- let CODE be the next code in the code stream
- is CODE in the code table?
- Yes:
    - output {CODE} to index stream
    - let K be the first index in {CODE}
    - add {CODE-1}+K to the code table
- No:
    - let K be the first index of {CODE-1}
    - output {CODE-1}+K to index stream
    - add {CODE-1}+K to code table
- return to LOOP POINT

Let's start reading though the code stream we've created to show how to turn it back into a list of color indexes. The first value in the code stream should be a clear code. This means we should initialize our code table. To do this we must know how many colors are in our color table. (This information comes from the first byte in the image data block in the file. More on this later.) Again we will set up codes #0-#3 to be each of the four colors and add in the clear code (#4) and end of information code (#5).

The next step is to read the first color code. In the following table you will see the values of CODE highlighted in purple, and the values for CODE-1 highlighted in green. Our first CODE value is #1. We then output {#1}, or simply 1, to the index stream [Step 0].

Now we enter the main loop of the algorithm. The next code gets assigned to CODE which now makes that value #6. Next we check to see if this value is in our code table. At this time, it is not. This means we must find the first index in the value of {CODE-1} and call this K. Thus K = first index of {CODE-1} = first index of {#1} = 1. Now we output {CODE-1} + K to the index stream and add this value to our code table. The means we output 1,1 and give this value a code of #6 [Step 1].

Step	Action	Code Stream	New Code Table Row	Index Stream
0	Init	#4 #1 #6 #6 #2 #9 #9 #7 ...		1
1	Not Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#6 - 1, 1	1, 1, 1
2	Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#7 - 1, 1, 1	1, 1, 1, 1, 1
3	Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#8 - 1, 1, 2	1, 1, 1, 1, 1, 2
4	Not Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#9 - 2, 2	1, 1, 1, 1, 1, 2, 2, 2
5	Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#10 - 2, 2, 2	1, 1, 1, 1, 1, 2, 2, 2, 2, 2
6	Found	#4 #1 #6 #6 #2 #9 #9 #7 ...	#11 - 2, 2, 1	1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1

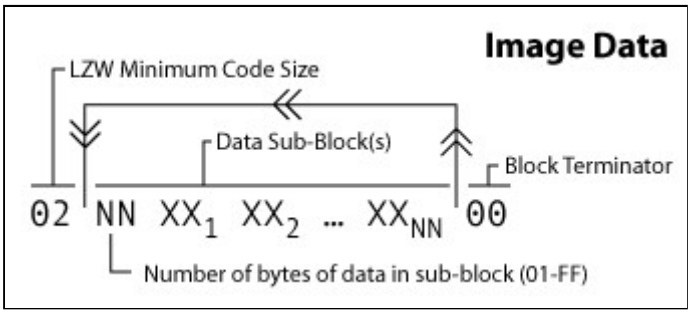
We start the loop again by reading the next code. CODE now would be #6 and this time we do have a record for this code in our code table. Thus we dump {#6} to the index stream which would be 1,1. Now we take the first index in {#6} and call that K. Here, {#6} has two indexes, the first of which is 1; thus K = 1. Before moving on, we add {CODE-1}+K to the code table. This #7 is now 1, 1, 1 [Step 2].

I've included a few more steps so you can see the algorithm in action. While the explanation may sound complicated, you can see it's actually quite simple. You'll also notice that you end up building the exact same code table as the one that was created during compression. This is the reason that LZW is so great; we can just share the codes and not the table.

Saving the Code Stream as Bytes

I've shown you how to go back and forth between index and code stream, but haven't told you what to do with them. The index stream is used to specify the color of each of the pixel of your image and really only shows up on screen. It is the code stream that is actually saved in the GIF files on your computer or transmitted over the internet. In order to save these code streams, we must turn them into bytes. The first thought might be to store each of the codes as its own byte; however this would limit the max code to just #255 and result in a lot of wasted bits for the small codes. To solve these problems, the GIF file format actually uses flexible *code sizes*.

Flexible code sizes allow for further compression by limiting the bits needed to save the code stream as bytes. The *code size* is the number of bits it takes to store the value of the code. When we talk about bits, we're referring to the 1's and 0's that make up a byte. The codes are converted to their binary values to come up with the bits. To specify the code for #4, you would look at this binary equivalent, which is 100, and see that you would need three bits to store this value. The largest code value in our sample code stream is #36 (binary: 100100) which would take 6 bits to encode. Note that the number of bits i've just given is the minimum number. You can make the number take up more bits by adding zeros to the front.



We need a way to know what size each of the codes are. Recall that the image data block begins with a single byte value called the *LZW minimum code size*. The GIF format allows sizes as small as 2 bits and as large as 12 bits. This minimum code size value is typically the number of bits/pixel of the image. So if you have 32 colors in your image, you will need 5 bits/pixel (for numbers 0-31 because 31 in binary is 11111). Thus, this will most likely be one more than the bit value for the size of the color table you are using. (Even if you only have two colors, the minimum code size most be at least 2.) Refer to the code table above to remind yourself how that works.

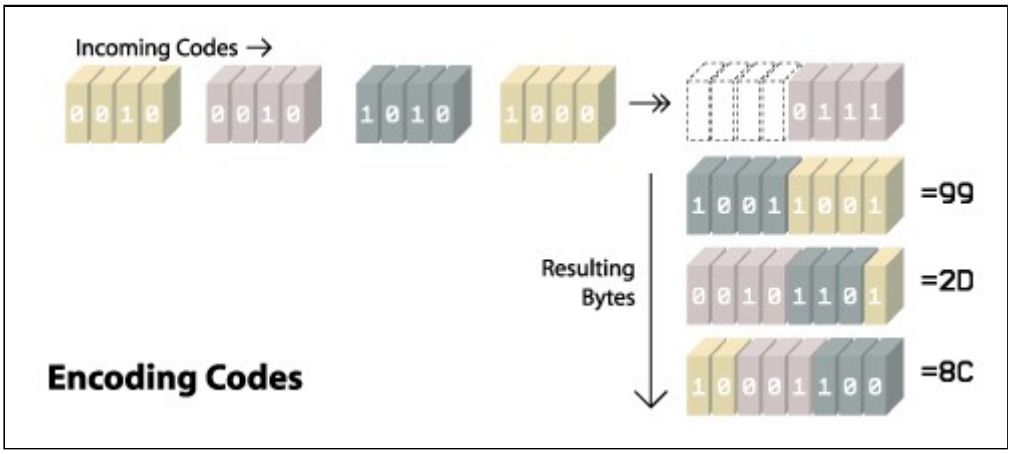
Here's the funny thing: the value for minimum code size isn't actually the smallest code size that's used in the encoding process. Because the minimum code size tells you how many bits are needed just for the different colors of the image, you still have to account for the two special codes that we always add to the code table. Therefore the actual smallest code size that will be used is one more than the value specified in the "minimum" code size byte. I'll call this new value the *first code size*.

We now know how many bytes the first code will be. This size will probably work for the next few codes as well, but recall that the GIF format allows for flexible code sizes. As larger code values get added to your code table, you will soon realize that you need larger code sizes if you were to output those values. When you are encoding the data, you increase your code size as soon as your write out the code equal to 2^(current code size)-1. If you are decoding from codes to indexes, you need to increase your code size as soon as you add the code value that is equal to 2^(current code size)-1 to your code table. That is, the next time you grab the next section of bits, you grab one more.

Note that the largest code size allowed is 12 bits. If you get to this limit, the next code you encounter should be the *clear code* which would tell you to reinitialize the code table. You then go back to using the first code size and grow again when necessary.

Jumping back to our sample image, we see that we have a minimum code size value of 2 which means out first code size will be 3 bits long. Out first three codes, #1 #6 and #6, would be coded as 001 110 and 110. If you see at Step 6 of the encoding, we added a code of #7 to our code table. This is our clue to increase our code size because 7 is equal to 2^3-1 (where 3 is our current code size). Thus, the next code we write out, #2, will use the new code size of 4 and therefore look like 0010. In the decoding process, we again would increase our code size when we read the code for #7 and would read the next 4, rather than the next 3 bits, to get the next code. In the sample table above this occurs in Step 2.

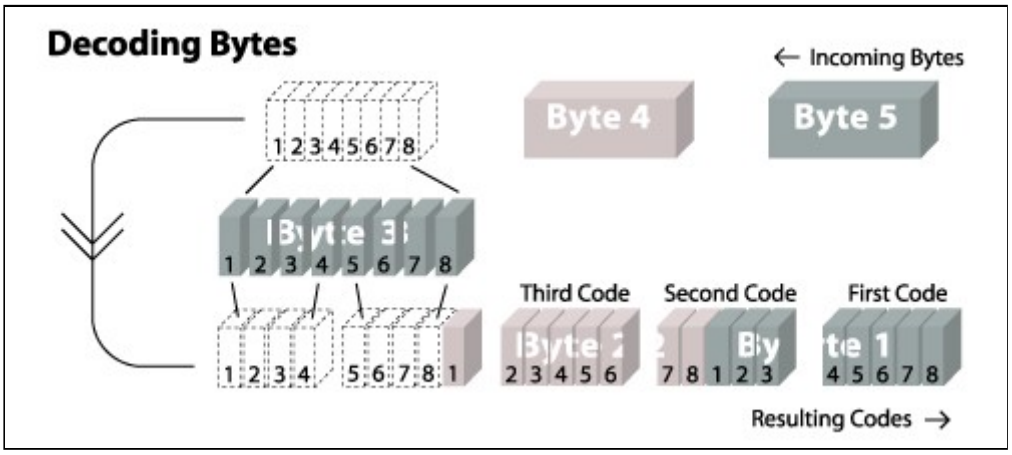
Finally we must turn all these bit values into bytes. The lowest bit of the code bit value gets placed in the lowest available bit of the byte. After you've filled up the 8 bits in the byte, you take any left over bits and start a new byte. Take a look at the following illustration to see how that works with the codes from our sample image.



You can see in the first byte that was returned (`8C`) that the lowest three bits (because that was our first code size) contain 110 which is the binary value of 4 so that would be the clear code we started with, #4. In the three bits to the left, you see 001 which out or first data code of #1. You can also see when we switched into code sizes of 4 bits in the second byte (`2D`).

When you run out of codes but have filled less than 8 bits of the byte, you should just fill the remaining bits with zeros. Recall that the image data must be broken up onto data sub-blocks. Each of the data sub-blocks begins with a byte that specifies how many bytes of data. The value will be between 1 and 255. After you read those bytes, the next byte indicates again how many bytes of data follow. You stop when you encounter a subblock that has a lenght of zero. That tells you when you've reached the end of the image data. In our sample the image the byte just after the LZW code size is `16` which indicates that 22 bytes of data follow. After we reach those, we see the next byte is `00` which means we are all done.

Return codes from bytes the basically just the same process in reverse. A sample illustration of the process follows which shows how you would extract codes if the first code size were 5 bits.



Next: Animation and Transparency

That is pretty much everything you need to know to read or generate a basic image file. One of the reasons the GIF becames such a popular format was because it also allowed for "fancier" features. These features include animation and transparency. Next we'll look at how those work.

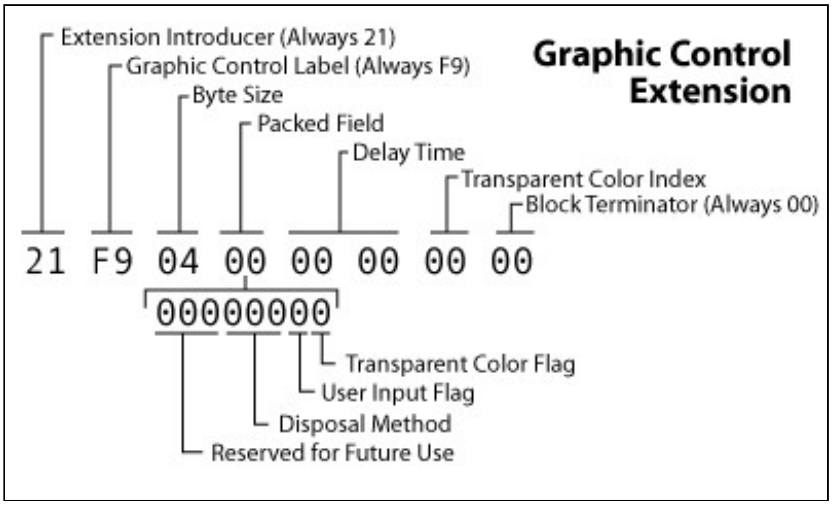
Continue...

What's In A GIF

(animation and transparency)

Back to the What's In A GIF index page.

In addition to being able to store simple image data, GIF files (specifically GIF89a files) allow for some special features. Tricks such as transparency and animation can be accomplished with the help of the Graphics Control Extension block. Here's a sample of what this block looks like:



I'll show you how to manipulate the bytes in this block to achieve special effects.

Animation

Cartoons are created by animators who draw a sequence of pictures, each slightly different from the one before, which, when rapidly shown one after the other, give the illusion of motion. Animation in GIF images is achieved in much the same way. Multiple images may be stored in the same file and you can tell the computer how much time to wait before showing the next image. Let's walk though the parts that make up this simple traffic light animation.

sample animated traffic light

```
47 49 46 38 39 61 0B 00 1D 00 A2 05 00 FF 00 00 00 FF 00 FF
FF 00 8E 8E 8E 00 00 00 FF FF FF 00 00 00 00 00 00 21 FF 0B
4E 45 54 53 43 41 50 45 32 2E 30 03 01 00 00 00 21 F9 04 04
64 00 00 00 2C 00 00 00 00 0B 00 1D 00 00 03 30 48 BA DC DE
23 BE 48 21 AD EB 62 A5 25 D3 93 F7 8C E4 27 9A 1B D7 A1 17
9B 1E A0 F3 96 34 13 DC CF AD 37 7A 6F F7 B8 05 30 28 F4 39
76 B5 64 02 00 21 F9 04 04 32 00 00 00 2C 02 00 0B 00 07 00
10 00 00 03 19 78 27 AC CB 0D CA 49 E1 B3 0A BB CD F7 F8 CE
27 1E 62 69 9E A3 19 82 47 02 00 21 F9 04 04 64 00 00 00 2C
02 00 02 00 07 00 10 00 00 03 19 78 07 AC CB 0D CA 49 E1 B3
0A BB CD F7 F8 CE 27 1E 62 69 9E A3 19 82 45 02 00 3B
```

This file is similar to the ones we've previously encountered. The bytes start out with the GIF header. Next we have a logical screen descriptor which tells us that our image is 11px by 29 px and will have a global color table with 8 colors in it (of which we only really need 5). Immediately after, follows the global color table which tells us what those colors are (0=red, 1=green, 2=yellow, 3=light grey, 4=black, 5=white, 6=black [not used], 7=black [not used]).

Next we encounter an application extension block. This is this block that causes our animation to repeat rather than play once and stop. The first three bytes tell us we are looking at (1) an extension block (2) of type "application" which is followed by (3) 11 bytes of fixed length data. These 11 bytes contain the ASCII character codes for "NETSCAPE2.0". Then begins the actual "application data" which is contained in sub-blocks. There are two values that are stored in these sub-blocks. The first value is always the byte `01`. Then we have a value in the unsigned (lo-hi byte) format that says how many times the animation should repeat. You can see that our sample image has a value of 0; this means the animation should loop forever. These three bytes are preceded by the `03` that lets the decoder know that three bytes of data follow, and they are terminated by `00`, the block terminator.

This very basic animation is essentially made up of three different "scenes". The first is the one with the green light lit, the second with the yellow, and the last with the red. You should be able to see three separate chunks of image data in the bytes above.

1 2 3

The first chunk begins immediately after the application extension block. It is there we encounter our first graphic control extension. As with all extensions, it begins with `21`. Next, the type specific label for the graphic control type of extension is `F9`. Next we see the byte size of the data in the block; this should always be `04`. The first of these four data blocks is a packed field.

The packed field stores three values. The first three (highest) bits are "reserved for future use" so those have been left as zeros. The next three bits indicate the disposal method. The *disposal method* specifies what happens to the current image data when you move onto the next. We have three bits which means we can represent a number between 0 and 7. Our sample animated image has a value of 1 which tells the decoder to leave the image in place and draw the next image on top of it. A value of 2 would have meant that the canvas should be restored to the background color (as indicated by the logical screen descriptor). A value of 3 is defined to mean that the decoder should restore the canvas to its previous state before the current image was drawn. I don't believe that this value is widely supported but haven't had the chance to test it out. The behavior for values 4-7 are yet to be defined. If this image were not animated, these bits would have been set to 0 which indicates that do not wish to specify a disposal method. The seventh bit in they byte is the *user input flag*. When set to 1, that means that the decoder will wait for some sort of "input" from the person viewing the image before moving on to the next scene. I'm guessing it's highly unlikeley that you will encounter any other value that 0 for this bit. The final bit is the transparency flag. We will go into more detail about transparency in the next section. Since this image isn't using any transparency, we see this bit has been left at 0.

The next two bytes are the delay time. This value is in the same unsigned lformat as all the other integers in the file. This number represents the number of hundredths of a second to wait before moving on to the next scene. We see that our sample image has specified a value of 100 (`64 00`) in the first graphics control block which means we would wait 1 second before changing our green light to yellow.

Our graphics control extension block ends with the block terminator `00`. You will notice this type of block appearing two more times in this image, the second instance differing only in the delay time (the yellow light only stays up for half a second).

The next chunk is an image descriptor. The block declares that it will be drawing an image starting at the top left corner and taking up the whole canvas (11px x 29px). This block is followed by the image data that contains all the codes to draw the first scene, the one with the green light on.

Green (Difference) Yellow

If we compare the first and the second scene, we see they share many of the same pixel color values. Rather than redrawing the whole canvas, we can specify just the part that changes (that is, the smallest rectangle that covers the part that changes). You'll see that the image descriptor before the second block of image data specifies that it will start at the pixel at (2, 11) and draws a box that's 7px wide by 16px tall. This is just large enough to cover the bottom two lights. The works because we chose the "do not dispose" disposal method for out graphics control extension block. In the same way, the third and final image data block only renders the top two circles to both fill in the red and cover up the yellow.

Transparency

Normally, GIF images are rectangles that cover up what ever background may be beneath them. Transparency allows you to "see though" the image to whatever is below. This is a very simple trick to pull off in a GIF image. You can set up one color in your color table that is converted to "invisible ink." Then, as the image is drawn, whenever this special color is encountered, the background is allowed to show through.

There are only two pieces of data we have to set to pull this off. First we must set the Transparency Color Flag to 1. This is the lowest bit in the packed byte of the Graphic Control Extension. This will tell the decoder that we want our image to have a transparent component. Secondly we must tell the decoder which color we want to use as our invisible ink. The decoder will then all you to see thought every pixel that contains this color. Therefore make sure it's not a color that you are using else where in your image. The color you choose must be in the active color table and you specify its value in the Transparent Color Index byte by setting this value to the index of the color in the color table.

Let's demonstrate this by revisiting the sample image we used in Bits and Bytes. We will update this file to make the white center part transparent. Let's start creating the Graphic Control Extension block that will do this for us. Again we start with the `21 F9 04` punch. In the next byte, we need to flip the transparent color flag to 1 (we can leave the others at zero) so this whole byte is simply `01`. The next two bytes can be left at zero.

We must now specify which color to disappear. Recall that our sample image had the following global color table:

Index	Color
0	White
1	**Red**
2	**Blue**
3	**Black**

We already know what we want to make all the white sections transparent. The color white has an index of 0. Therefore we will specify a value of `00` for the transparent color index block. Had we wanted to make the red transparent we would have used `01`, or `02` for blue. Lastly we tack on the block terminator of `00` and we're done. We have created the following block:

```
21  F9  04  01  00  00  00  00
```

Now, all we have to do is plug this into our sample image right before the image descriptor. I've placed our original sample image on a black background as well as the one we just made so you can see the results. I've also included ones where red or blue are transparent. The last three differ by only the transparent color index byte.

Original	Transparent White (`00`)	Transparent Red (`01`)	Transparent Blue (`02`)

This concludes our description of the bits inside GIFs. If you have not already read the documentation of the GIFLIB API, you may wish to continue with that.

Back to GIFLIB documentation