Image Formats

All GAMBY graphics are stored as a series of bits; most of these bits directly represent pixels on the screen. There are two important, slightly unusual things about all GAMBY graphics, however:

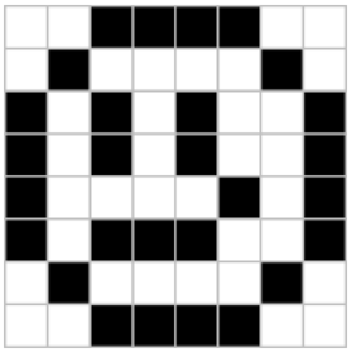
1. *Compared to most graphics, the image in 'inverted.'* A `1` represents an 'on' pixel and a `0` is an 'off' pixel, but the LCD is (by default) white-on-black, so it is the 'on' pixels that are black.
2. *Images are stored in columns, not rows.* Most image formats store data starting at the upper left pixel and moving right. GAMBY images are stored as columns, starting at the lower left and moving up. This is to take advantage of how the LCD stores and displays data.

Icons

Icons are the probably the most straight-forward type of GAMBY graphics; they are 8 pixels high and any width. They are stored in **PROGMEM** as a series of bytes, the first being the width of the icon, and those after forming each column of the image. The leftmost bit (the 'most significant bit,' or MSB) is the bottom of each column, the rightmost bit ('least significant'/LSB) is the top.

Using binary notation, it is easy to create images 'by hand': the zeroes are 'off' pixels, the ones are 'on.'

```
// 'prog_uchar' is the PROGMEM equivalent of 'byte'
PROGMEM prog_uchar smiley_icon[] = {
  8,         //  The icon's width (8 pixels).
  B00111100, //  The left column of the icon, using binary (B...)
  B01000010,
  B10101101,
  B10100001,
  B10101101,
  B10010001,
  B01000010,
  B00111100  //  The rightmost column of the icon
}
```



Because the data is stored in columns instead of rows, it may help to think of the image as having been turned 90˚ clockwise. To the right is the data for the icon with the `1` bits on a dark background; if you look closely, you can see that it's the icon image, rotated.

```
B00111100,
B01000010,
B10101101,
B10100001,
B10101101,
B10010001,
B01000010,
B00111100
```

Multi-Frame Icons

An icon can contain more than one frame, although it may be more accurate to call them 'multiple images' since they don't automatically animate -- you explicitly tell GAMBY which frame to draw. Multi-frame icons have a number of uses beyond animation: they can make it easy anywhere you want to change an icon programmatically. For example, they can be used to create an icon with an 'off' state as frame 0 and an 'on' state as frame 1.

The first frame of a multi-frame icon is the same as a standard icon: the first byte is the width, and the rest is the actual bitmap data. Each additional frame's data are added to the end of the icon's `PROGMEM` array. Each additional frame must be the same width as the first; the icon's width is only specified at the start of the data.

Blocks and Patterns

The individual blocks in GambyBlockMode and the fill patterns in GambyGraphicsMode are both 16 bit integers, with its bits forming a 4x4 pixel grid. Like icons, the leftmost bit is the bottom left pixel. The easiest way to represent a block or a pattern is using hexadecimal notation. For each column of four pixels, there are 16 possible combinations of 'on' and 'off,' which correspond to a hexadecimal digit:

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**

With this in mind, a checkerboard pattern would be **`0x33CC`**. The `0x` is the prefix for a hexadecimal number. Then, looking at the chart above, **`3`** has just the top two pixels 'on', and **`C`** has the bottom two.

1	1	0	0
1	1	0	0
0	0	1	1
0	0	1	1
3	**3**	**C**	**C**

Here's a slightly more complex example, one in which all columns are different: **`0x7DFE`**, which looks something like a square with a drop shadow.

1	1	1	0
1	0	1	1
1	1	1	1
0	1	1	1
7	**D**	**F**	**E**

Block Palettes

Patterns and individual blocks use the same format, however GambyBlockMode uses a 'palette' of 16 different blocks. These are stored as an array in the Arduino's flash memory using **`PROGMEM`** constants. Here is a code fragment with an example of a palette:

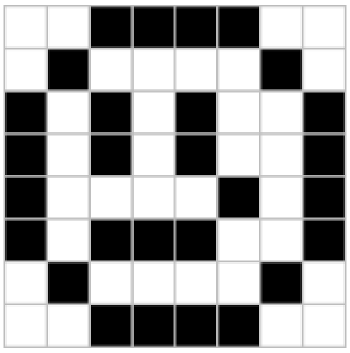
```
#include <Gamby.h>
#include <avr/pgmspace.h>

//  PROGMEM constants should use the special PROGMEM types.
//  'prog_uint16_t' is equivalent to 'unsigned int'
PROGMEM prog_uint16_t blocks[] = {
    0x0000, //  0 All 'white'
    0xffff, //  1 All 'black'
    0x5a5a, //  2 50% gray dither
    0xfaf5, //  3 75% gray dither
    0x050a, //  4 25% gray dither
    0xedb7, //  5 Light diagonal left line
    0x1248, //  6 Dark diagonal left line
    0x7bde, //  7 Light diagonal right line
    0x8421, //  8 Dark diagonal right line
    0x888f, //  9 Dark-on-light solid grid lines
    0x1110, // 10 Light-on-dark solid grid lines
    0xa080, // 11 Light-on-dark dotted grid lines
    0x5f1f, // 12 Dark-on-light dotted grid lines
    0x33cc, // 13 Checker pattern
    0xcc33, // 14 Mirrored checker pattern
    0x0001  // 15 Single pixel (upper right)
};

GambyBlockMode gamby;

void setup() {
  // Use the blocks in the 'blocks' PROGMEM constant as the palette:
  gamby.palette = blocks;
}
```

Sprites

Sprites are similar to icons, except they can be any height as well as any width. Because they do not have a fixed height, the height must be part of the data. The second byte of a sprite is the height.

You can turn an icon into a sprite by simply adding an '8' (the height) after the first number (the width). Here's the smiley icon from before, turned into a sprite.

```
// 'prog_uchar' is the PROGMEM equivalent of 'byte'
PROGMEM prog_uchar smiley_sprite[] = {
  8,         //  The sprite's width (8 pixels).
  8,         //  The sprite's height (8 pixels).
  B00111100, //  The left column of the sprite, using binary
  B01000010,
  B10101101,
  B10100001,
  B10101101,
  B10010001,
  B01000010,
  B00111100  //  The rightmost column of the sprite
}
```



Sprites do not have to be 8 pixels high, although it makes them easy to create and edit by hand; you can use a byte for each column. If the sprite has more or fewer lines, one byte is either more or less than a full column. In these cases, don't think of the bytes as individual units; think of all the bytes' ones and zeroes as a single lump of data that just happen to be grouped into sets of eight.

You can also convert a block or a pattern into a sprite fairly easily. A block/pattern is 16 bits, which is two bytes. You can simply split the hexadecimal number in half: the first two digits are one byte, the last two digits are the other byte. Wrap it in a `PROGMEM` constant definition, add the height and width (4 and 4), and you have a little sprite.

```
// The pattern from an earlier example:
const unsigned int examplePattern = 0x7DFE;

// The pattern turned into a little sprite:
PROGMEM prog_uchar sprite_from_block[] = {
  4,    //  Sprite width (4 pixels).
  4,    //  Sprite height (4 pixels).
  0x7D, //  The first and second columns
  0xFE  //  The third and fourth columns of the sprite
}
```

Sprite Masks

Masks are sprites used to make portions of another sprite transparent. They work similarly to alpha channels in more complex graphics systems, or a stencil in the real world: the 'main' sprite will only be copied to the screen where the mask has 'on' bits. Masks are identical to other sprites; it's only how they are used that makes them different. The only requirement is that the masks be the same dimensions as the main sprite. If it is any larger or smaller, the sprite won't draw correctly.

Multi-Frame Sprites

A sprite can contain more than one frame, although it may be more accurate to call them 'multiple images' since they don't automatically animate -- you explicitly tell GAMBY which frame to draw. The first frame of a multi-frame sprite is the same as a standard sprite: the first two bytes are the dimensions, and the rest is the actual bitmap data. Each additional frame's data (without dimensions) are added to the end of the sprite's `PROGMEM` array. Each additional frame must be the same dimensions as the first; the sprite's dimensions are only specified at the start of the data.

Fonts

Fonts are the most complex of GAMBY's image types. It is highly recommended that you use the font tool to create or edit them.

Fonts are stored as an array of 32 bit integers in **`PROGMEM`**, with each 32b integer being one character (letter/number/etc). The first 25 bits (0-24) form a 5x5 pixel bitmap. Bits 25 to 27 are the character's vertical offset, which allows 'descenders' (such as lowercase *g* and *y*) to extend below the baseline. Bits 28-31 are the character's width.