RETROLCD.com

Bitmap Fonts

Contents

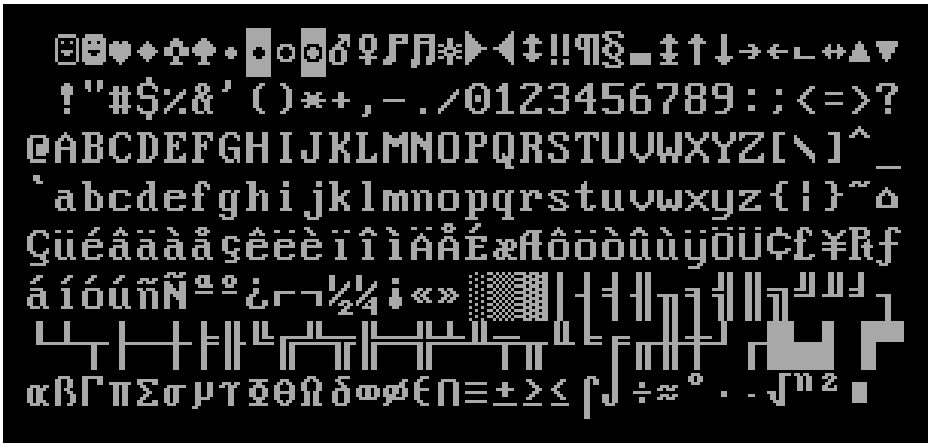
Introduction 3
Writing Code to Write Code 4
Measuring Your Grid 5
Converting an Image with PHP 6
Step 1: Load the Image 6
Step 2: Scan the Image 7
Step 3: Save the Header File 8
The 1602 Character Set 10
The 1602 Character Map 11
Converting the Image to Code 12
Conclusion 15
Rendering the Characters 16
Rendering Complete Strings 17

Introduction

Before we had True Type fonts, we had Bitmap Fonts. Bitmap Fonts are exactly what they sound like: fonts made from images. True Type fonts are made of math which is why they can scale so cleanly to any size. Bitmap fonts are made of discrete pixels and as such, don't scale well.

Early computers like the Apple][, Commodore, and IBM PC contained a special bit of memory that held the character set that was used to display information to the user. The original IBM PC had a character set called Code Page 437

https://en.wikipedia.org/wiki/Code_page_437



This set of characters includes a number that are useful for games and drawing box graphics. Before there were graphical user interfaces, interfaces were drawn entirely with text. In total there are 256 characters that are 9x16 pixels in size. However, the last pixel is always blank and simply adds space between characters. The real width of the actual images are 8 pixels which fit into a byte. Since the Arduino is equally limited, this character set makes a very good choice for a font with a lot of versatile uses.

http://www.vintagecomputing.com/index.php/archives/790/the-ibm-smiley-character-turns-30

The above link contains a conversation with one of the developers of the ASCII table used with the IBM PC and why they chose the characters they did.

256 characters times 1 byte wide times 2 bytes wide is 512 bytes of space that will be used to store this font or 4096 bits. Obviously, we do not want to have to hand code all this data in. Fortunately, we don't have to.

Writing Code to Write Code

It doesn't take long before you start getting into areas of programming that would be tedious if done by hand. The original developers of the above character set had to design each character by hand. Since the work to design the characters is already done and the work to put all the characters into an image is done, the last piece we need is to turn that image into something that the Arduino can understand. Again, we could pull out graphing paper and hand calculate all the binary values, or, we could use modern technology to do the work for us.

While this could be done in about any language, we're going to use PHP because it is very similar to C (it is basically scripted C) and includes very easy to use graphics functions.

Our project will consist of one PHP file called Codepage437.php

Codepage437.h is generated by the php file and Codepage-437.png is a slightly modified version of the file found on Wikipedia. The Wikipedia image contains a black border around the characters which I've removed in my own version.

Codepage437.h
Codepage437.php
Codepage-437.png

The most important feature of this image is that it is in a regular grid pattern. This makes the math to convert the bitmap into binary for the Arduino much easier.

Each character is in a 9x16 area and we only need the 8x16 image in each of those areas.

The basic logic is to read each pixel of the image and store either a "0" or a "1" in a string that will then go into the h file. We're converting numeric pixel values into the string equivalent of on and off. This image conveniently uses pure black as the background color so we will just check for black and if the pixel is black, we store a "0" otherwise we store a "1." We don't care what color the text is, just that it isn't black.

Measuring Your Grid

When converting a regular grid of images to code there are a few measurements you need to take:

- The left most X position of your tile set
- The top most Y position of your tile set

This is where your code will start looking for images in your code. For our Codepage347 image, we have cropped the image out X starts at 0 and our Y starts at 0.

- The number of pixels horizontally from the start of one tile to the start of the next
- The number of pixels vertically from the start of one tile to the start of the next

This is the boundary of each tile. Note that the boundary may not match the size of the title. We will see an example of that when we load the character set from the 1602 LCD. The Code Page 437 image has a boundary size of 9x16

- The width of each title
- The height of each tile.

This is the size of each tile. The Code Page 437 image has a tile size of 8x16.

- The height of each pixel
- The width of each pixel

As you will see in the 1602 Character set image, each pixel of the character is represented by a 5x5 square. The Code Page 437 image is actual size, so each pixel is 1x1. When the image is magnified to have bigger pixels, it is a good idea to calculate the general midpoint as well, so you are scanning the middle of each pixel. Use the floor function to round down to a whole pixel.

You may have noticed already, that this technique is also useful for sprites. There are many resources on the internet with tile sets from various video games.

While you may not use them in a commercial product or profit from them in any way, you may use them as a starting point for developing your own sprites.

Note that “profiting” from copyrighted works also includes notoriety. For example, if you have a popular independent game that uses the graphics from Mega Man and distribute it on a web-site for free but sell other products or services on that web-site, you are risking legal problems because you are using other people’s work to attract attention to your own work for profit.

Absolutely use tile sets that other people have developed to practice programing and game design with. But eventually, you will need to find your own graphics that you are legally allowed to use and distribute. A game development team generally starts with a programmer and an artist.

Converting an Image with PHP

Step 1: Load the Image

```php
<?php

$im = imagecreatefrompng( filename: 'Codepage-437.png');

$image_width = imagesx($im);
$image_height = imagesy($im);

$tile_x_offset = 0;
$tile_y_offset = 0;

$tile_boundary_width = 9;
$tile_boundary_height = 16;

$tile_columns = $image_width / $tile_boundary_width;
$tile_rows = $image_height / $tile_boundary_height;

$tile_width = 8;
$tile_height = 16;

$binary = [];
```

PHP includes a simple function called “imagecreatefrompng” which loads the file into a format in memory that PHP can work with. There are functions to load many different types of image files but we’re using a PNG.

Once the image is loaded, we get the width and height of it in pixels. “$binary” is where we will be storing our generated code. Finally, we do the calculations as described in “Measuring Your Grid.” From these measurements we can now scan the image to convert the image into binary strings.

For this particular image, we can get the number of columns from the width of the image and the width of each character. Other images may require specifying the number of rows and columns explicitly as it may be easier than trying to come up with a formula.

Step 2: Scan the Image

```php
for ($row = 0; $row < $tile_rows; $row++) {
    for ($column = 0; $column < $tile_columns; $column++) {

        for ($tile_y = 0; $tile_y < $tile_height; $tile_y++) {
            for ($tile_x = 0; $tile_x < $tile_width; $tile_x++) {

                $image_x = $tile_x_offset + $column * $tile_boundary_width + $tile_x;
                $image_y = $tile_y_offset + $row * $tile_boundary_height + $tile_y;

                if ($image_x >= $image_width) {
                    exit('Invalid Tile Width Measurements');
                }

                if ($image_y >= $image_height) {
                    exit('Invalid Tile Height Measurements');
                }

                $pixel_color = imagecolorat($im, $image_x, $image_y);

                $tile_number = $row * $tile_columns + $column;

                if (!isset($binary[$tile_number][$tile_y])) {
                    $binary[$tile_number][$tile_y] = '0b';
                }

                $binary[$tile_number][$tile_y] .= $pixel_color ? '1' : '0';
            }
        }
    }
}
```

Step 2 is to iterate over the image reading every pixel into a format that will be useful for the Arduino.

Notice there are two sets of X, Y loops. The first set iterates over the characters, the second set iterates over the pixels in each character.

This logic could be implanted in a number of different ways. You may notice that the pattern

Y = X * C + B

C is the scaling factor and B is the offset. You may have first seen this function in the form

Y = mX + b

Which is the formula for a line.

At this point the $binary variable contains the font information in the following format:

```
[2] => Array
    (
        [0] => 0b00000000
        [1] => 0b00000000
        [2] => 0b01111110
        [3] => 0b11111111
        [4] => 0b11011011
        [5] => 0b11111111
        [6] => 0b11111111
        [7] => 0b11000011
        [8] => 0b11100111
        [9] => 0b11111111
        [10] => 0b11111111
        [11] => 0b01111110
        [12] => 0b00000000
        [13] => 0b00000000
        [14] => 0b00000000
        [15] => 0b00000000
    )
```

Each character is in the first dimension of the array and each line of the character is in the second dimension.

Step 3: Save the Header File

Finally, we need to convert that into code that the Arduino understands.

```
54
55  foreach ($binary as $n => $vals) {
56      $binary[$n] = implode( glue: ", //" . ($n + 1) . "\r\n  ", $vals);
57  }
58
59  $code = '
60  PROGMEM const byte CODEPAGE437[] = {
61    ' . implode( glue: ",\r\n  ", $binary) . '
62  };
63  ';
64  $fp = fopen( filename: 'Codepage437.h', mode: 'w');
65  fwrite($fp, $code);
66  fclose($fp);
```

Step 3 starts with going through the $binary array and compressing the second dimension into a single string with commas after each line. We're also putting the character number in a comment after each line so it's easy to find what you are looking for later.

After the 2nd dimension is rolled up, then we can compress the 1st dimension into a single string which is suitable to be used as source code for the Arduino.

And finally, we write that string to the .h file we will load into the Arduino.

```
PROGMEM const byte CODEPAGE437[] = {
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000,
  0b00000000, //2
  0b00000000, //2
  0b01111110  //2
```

A portion of code generated by this PHP script.

The 1602 Character Set

While the Code Page 437 character set is incredibly useful, it is on the large side relative to our intended display which is the 128x64 OLED display. At 16 pixels tall, we can only fit 4 rows of text on the screen at once. And at 8 pixels wide, we can only fit 16 characters across.

The 1602 LCD Character display is a 2-line, 16-column display with a built-in character set that is 5x8 pixels per character. This will give us 8 lines of text and 25 columns of text on our screen. The limitation of this character set is that it doesn't have as many useful special characters.

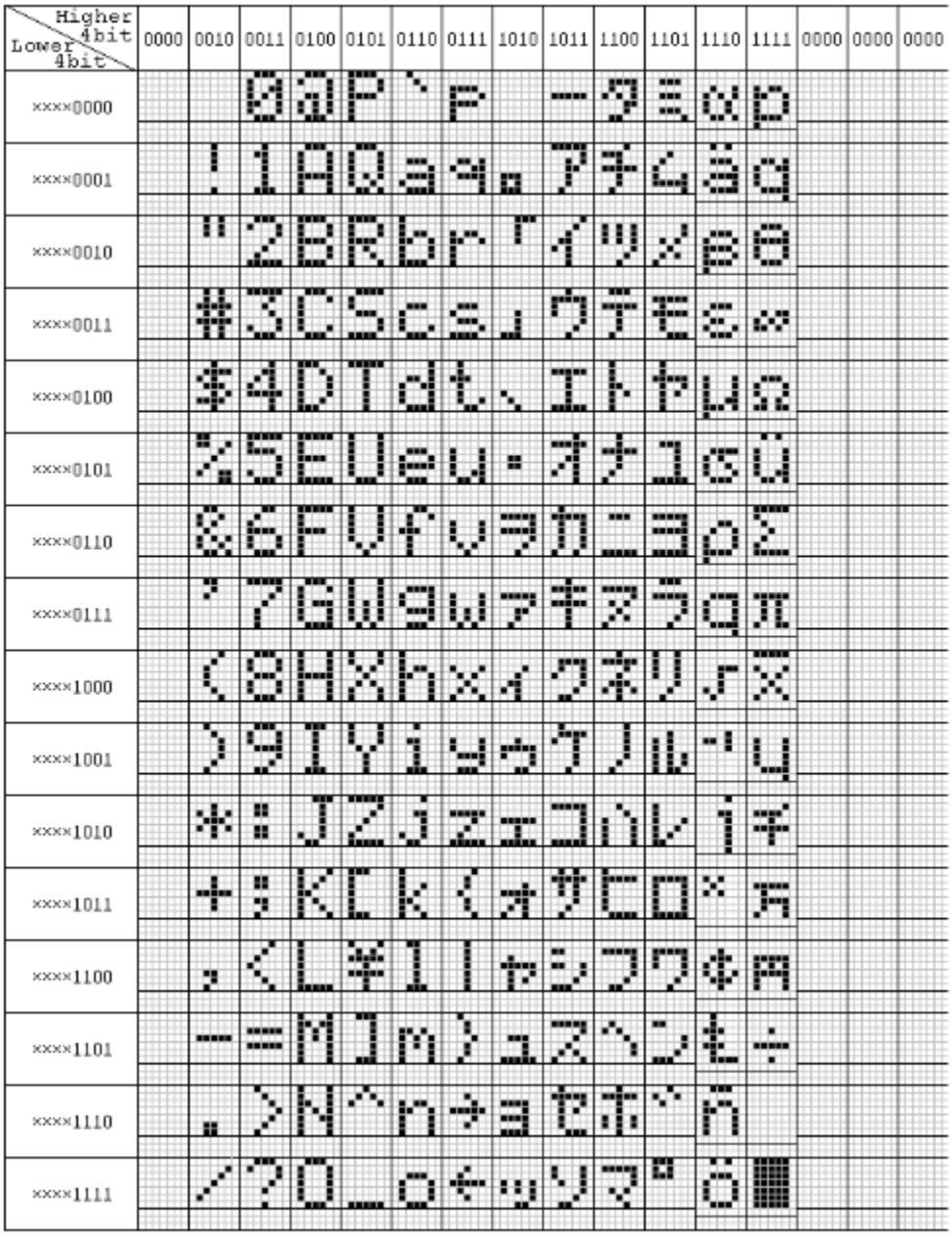
If you're looked at the Dinosaur project you will see how special characters were stored on the 1602 to create the sprites used in the game.

```
16 class DinoController {
17   private:
18     static byte dinoHitL[8] = {
19       0b00000,
20       0b00000,
21       0b00111,
22       0b00111,
23       0b00111,
24       0b00101,
25       0b00111,
26       0b00011
27     };
```

You can see that we're listing 5 bits horizontally and 8 bytes vertically. Which isn't really possible. We're actually storing 8 bits wide and 8 bytes high. The missing zeros are implied. Which is 3 bytes of wasted memory that we aren't using per sprite.

There is nothing we can do about this as this is simply how the 1602 character display works. But we're not using the 1602 LCD Character display. We just want the character map. So, we can store the characters however we want. Since the characters are 8 bits high and 5 bits tall, we're going to rotate our character map so each row in our code represents one column of the character. We can then store each character in 5 bytes instead of the 8 bytes required by the actual LCD Character display.

The 1602 Character Map



Fortunately for us, someone has already gone to the trouble of documenting all the characters in an image that is of fixed width and fixed height.

Unfortunately, there seems to be 3 missing columns. The first 32 characters of the LCD controller are empty. Also missing are columns 8 and 9. Those correspond to rows 1 and 5 Code Page 437. A full 64 characters just don't exist. But we must leave memory for them. This leaves a lot of space in this character set for custom characters/sprites for your project.

Converting the Image to Code

Since this character map is arranged differently than the Code Page 437 set, we will need to write another script using the same principles to generate the Arduino code.

```php
<?php

$im = imagecreatefrompng( filename: '1602.png');
$iw = imagesx($im);
$ih = imagesy($im);

$binary = [];

$fontsx = 16;
$fontsy = 16;
```

As before, we use the imagecreatefrompng function to load the PNG of the character map. Then we get the size and set the number of expected rows and columns. Note however that we don't have 16 columns. But our final code needs 16 columns.

```php
$positions = [
    0 // 0
    , 0 // 1
    , 1 // 2
    , 2 // 3
    , 3 // 4
    , 4 // 5
    , 5 // 6
    , 6 // 7
    , 0 // 8
    , 0 // 9
    , 7 // 10
    , 8 // 11
    , 9 // 12
    , 10 // 13
    , 11 // 14
    , 12 // 15
];
```

Instead of using math and logic to figure out what column of the image each column of fonts corresponds to, we're just using a precalculated look up table. The numbers correspond the column in the image that each column of characters starts at in the image.

Two things to keep in mind: the 1602 set is stored vertically and we're rotating the font, so it is stored in memory rotated 90 degrees.

While before we processed the columns and then rows, for this character set we must process rows and then columns. The image is also the inverse with black representing a 1 and anything else representing a zero.

```php
for ($column = 0; $column < $tile_columns; $column++) {
    for ($row = 0; $row < $tile_rows; $row++) {

        for ($tile_x = 0; $tile_x < $tile_width; $tile_x++) {
            for ($tile_y = 0; $tile_y < $tile_height; $tile_y++) {

                $image_x = $tile_x_offset +  $positions[$column] * $tile_boundary_width + ($tile_x + 1) * $pixel_width + $pixel_width_midpoint;
                $image_y = $tile_y_offset + $row * $tile_boundary_height + (7 - $tile_y) * $pixel_height + $pixel_height_midpoint;

                if ($image_x >= $image_width) {
                    exit('Invalid Tile Width Measurements');
                }

                if ($image_y >= $image_height) {
                    exit('Invalid Tile Height Measurements');
                }

                $pixel_color = imagecolorat($im, $image_x, $image_y);
                $tile_number = $tile_rows * $column + $row;

                if (!isset($binary[$tile_number][$tile_x])) {
                    $binary[$tile_number][$tile_x] = '0b';
                }
                $binary[$tile_number][$tile_x] .= $pixel_color ? '0' : '1';
            }
        }
    }
}
```

Again, the Y = mX + B formula is repeated many times. Each pixel of the character in the image is 5 pixels wide and hide. So we have to adjust for that. There is also a pixel of padding on the left and right side of each character, so we are excluding that. And finally, we add 2 pixels to look in the center of each square of the image to see if the cell is black or not black.

We always start at pixel 46 for the Y starting position because that is the vertical location each column starts.

If you remove the zeros of the output and replace them with spaces, you can see how the characters are stored easier.

```
[48] => Array
    (
        [ ] =>  b  11111
        [1] =>  b 1 1   1
        [2] =>  b 1  1  1
        [3] =>  b 1   1 1
        [4] =>  b  11111
    )

[49] => Array
    (
        [ ] =>  b
        [1] =>  b 1    1
        [2] =>  b 1111111
        [3] =>  b 1
        [4] =>  b
    )

[5 ] => Array
    (
        [ ] =>  b 1    1
        [1] =>  b 11    1
        [2] =>  b 1 1   1
        [3] =>  b 1  1  1
        [4] =>  b 1   11
    )
```

The LCD1602.h file contains the following

```
PROGMEM const byte LCD1602[] = {
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000, //1
  0b00000000,
  0b00000000, //2
  0b00000000, //2
  0b00000000, //2
  0b00000000, //2
  0b00000000,
  0b00000000, //3
  0b00000000, //3
```

Notice that we preface the name of the variable with LCD instead of just calling it 1602. This is because variables in C cannot start with a number.

```
78      foreach ($binary as $n => $vals) {
79          $binary[$n] = implode( glue: ", //" . ($n + 1) . "\r\n  ", $vals);
80      }
81
82      $code = '
83      PROGMEM const byte LCD1602[] = {
84        ' . implode( glue: ",\r\n  ", $binary) . '
85      };
86      ';
87      $fp = fopen( filename: 'LCD1602.h', mode: 'w');
88      fwrite($fp, $code);
89      fclose($fp);
```

The logic for generating the source code from our array is the same as before.

Conclusion

Both character sets are converted from images to source code in much the same fashion. The difference is the math and how we store the data in strings representing binary values. This technique can be used in a lot of different contexts. While in the Dinosaur game we used graphing paper and hand coded our sprites, we can now use PHP or some other scripting language to read sprite data we can create in any drawing program.

Rendering the Characters

Display128x64.h

```
116 // 8x16 sprites
117 void Blit816(int x, int y, byte * img)
118 {
119   for (iy = 0; iy < 16; iy++) {
120     b = pgm_read_byte(img + iy);
121     for (ix = 0; ix < 8; ix++) {
122       PlotPixel(x + ix, y + iy, (byte)((b >> (7 - (ix % 8))) & 1));
123     }
124   }
125 }
126 
127 // 5x8 sprites
128 void Blit58(int x, int y, byte * img)
129 {
130   for (ix = 0; ix < 5; ix++) {
131     b = pgm_read_byte(img + ix);
132     for (iy = 0; iy < 8; iy++) {
133       PlotPixel(x + ix, y + iy, (byte)((b >> ((iy % 8))) & 1));
134     }
135   }
136 }
```

We now need a couple more function in our Display128x64.h file so that we can render the 8x16 characters and the 5x8 characters. The 8x16 character is stored in the normal fashion where the order of bits matches the visual display. We grab one byte per vertical slice of the character and then render the 8 bits.

For the 5x8 sprites, we must account for the rotation. Rather than reading the front top to bottom, we are reading from left to right. We also remove the "7 –" because the bits are in the correct order relative to the display. Normally, while the text of the binary strings looks visually correct, it is actually mirrored relative to how it's read since the lest significant bit is on the right side and we start rendering on the left side.

For the 5x8 we are rendering columns where for the 8x16 we are rendering rows.

Rendering Bitmap fonts is a great exercise for encoding and decoding data.

Rendering Complete Strings

```
138 void BlitText58(int x, int y, byte * font, byte * str)
139 {
140   while (str[0] && x < 128) {
141     Blit58(x, y, font + 5 * str[0]);
142     str++;
143     x += 5;
144   }
145 }
146
147 void BlitText816(int x, int y, byte * font, byte * str)
148 {
149   while (str[0] && x < 128) {
150     Blit816(x, y, font + 16 * str[0]);
151     str++;
152     x += 8;
153   }
154 }
```

And finally, we create a couple functions so we can easily render complete strings to the display.

We named the function to correspond to the size of the sprites / characters.

A string is simply an array of bytes and a zero-value byte indicates the end of the string. For safety, we are exiting the loop if we get to the edge of the screen.

***** ALWAYS TERMINATE YOUR STRINGS WITH A NULL CHARACTER *****

I'll leave it as an exercise for the reader at this point to implement line breaks and word wrapping.

```
144   BlitText58(0,0,LCD1602,"ABC 123");
145   BlitText816(0,10,CODEPAGE437,"ABC 123");
```

When using strings, the compiler automatically adds the null character. Just because you can't see it, doesn't mean it's not there.

Where you run into trouble is when you dynamically create strings using your own custom functions and forget the add it.