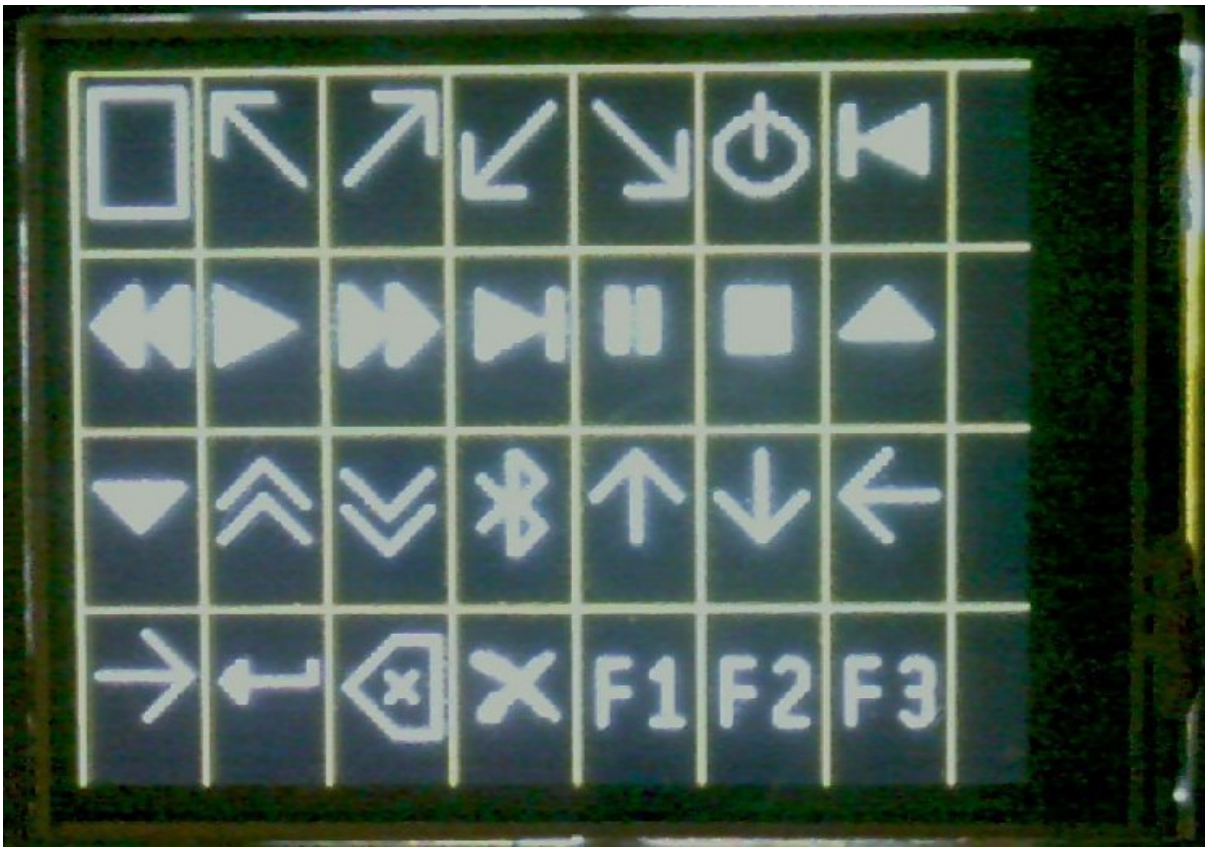
Creating Custom Symbol Fonts for Adafruit GFX Library

Created by Chris Young



https://learn.adafruit.com/creating-custom-symbol-font-for-adafruit-gfx-library

Last updated on 2024-06-03 02:50:09 PM EDT

Table of Contents

Overview

In this tutorial we will show you how to create custom symbol fonts for the Adafruit GFX library.

The Adafruit GFX library has a number of custom fonts ranging in size from 9-24 points. There is a mono space font similar to Courier, a Sans Serif similar to Arial or Helvetica, and a Serif font similar to Times. There are also bold, oblique, and bold oblique versions of each of these. But these fonts only cover the standard ASCII character set from 32-126 printable characters. There are no special symbols such as arrows, card suits, or other dingbats that you would find in your usual symbol type fonts.

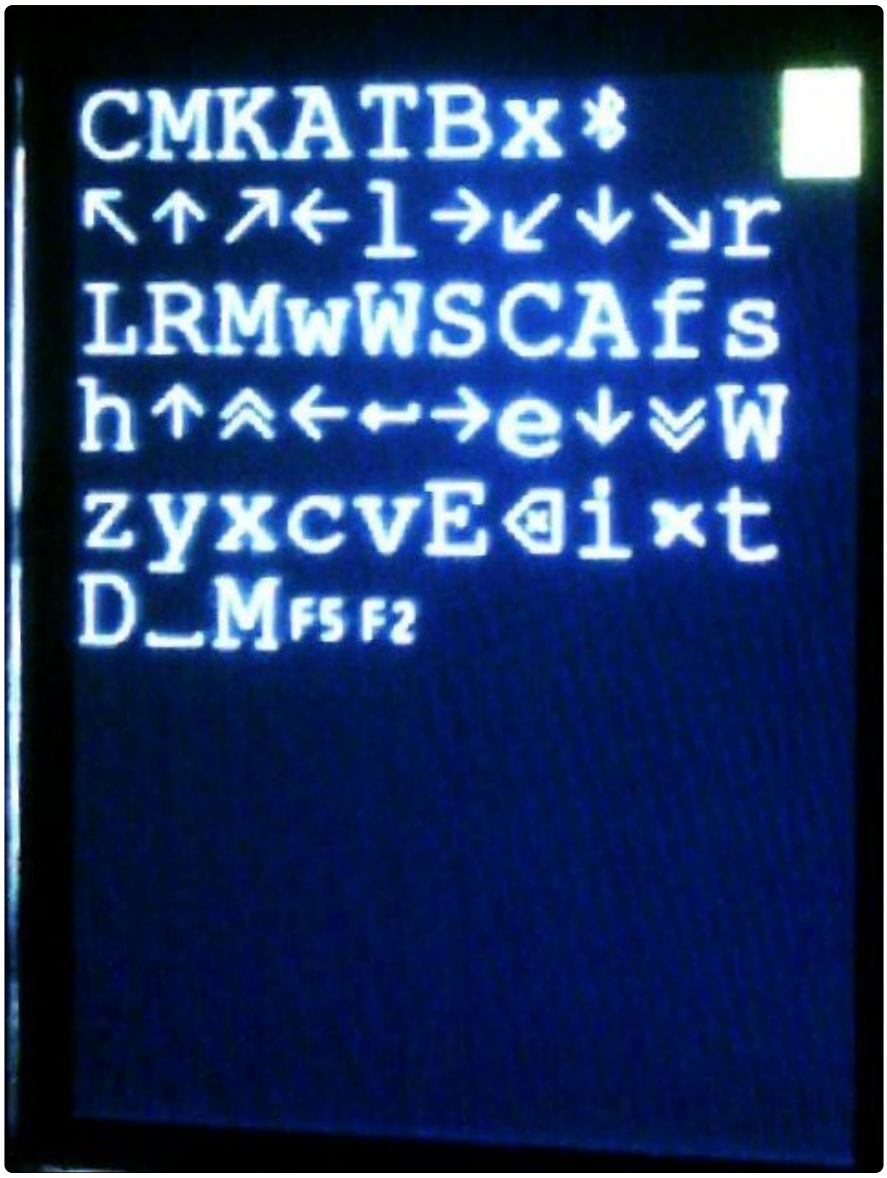
There is the ability to create a font in the proper format using a font conversion tool provided with the library. But it is a commandline utility that needs to be recompiled for your particular variety of Linux. For a Mac or PC user, compiling a utility and running it from a commandline is a hurdle many people (including myself) don't care to tackle. Even though I have several Raspberry Pi units sitting around that is still beyond my particular skill set.

Even if I could compile and run the utility, I would have to search for a type font to be converted that contained the symbols I wanted. It might even require converting multiple fonts just to gather up everything I wanted.

I needed to create a custom symbol font that would display media controls such as rewind, fast-forward, play, pause, stop as well as arrow keys in eight directions and some other custom designs such as a Bluetooth or a power button. So I developed a technique described here to layout the bitmaps for these custom symbols and manually put together a custom symbol font.

I created these special symbols for an assistive technology device that I call my "Ultimate Remote". It allows me to operate TV, cable, Blu-ray and other devices via infrared remote signals as well as make a Bluetooth connection to my iPhone for switch control. I can operate these devices with just three pushbuttons. I will be documenting that device very soon, but we wanted to show how I created these custom symbols. Here is a screen grab of the cable TV controls page on my remote. It shows items like fast-forward and rewind and other typical media controls symbols.

Here are some more screen grabs showing a mouse control page and another page which implements an entire board including function keys along the bottom of the screen.


CMKATBx*
↖↑↗←l→↙↓↘r
LRMwWSCAfs
h↑⩓←↩→e↓⩔W
zyxcvE⌫ix t
D_MF5 F2


*AKxxxQ
`1234567890-=⌫
qwertyuiop[]\x
asdfghjkl;'↵
zxcvbnm,./SCAe
h↑⩓←↩→e↓⩔wW
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
MS:10

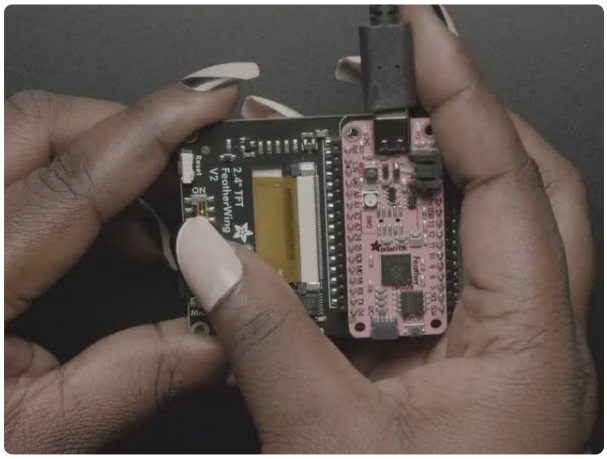
I will describe the system I used and provide you with the sample sketch so that you could use my custom symbols or adapt symbols of your own.

This learning guide presumes that you are somewhat familiar with the Adafruit GFX library and how to use custom fonts with that library. You can find more information on the library and the text capabilities at the learning guide found here (https://adafru.it/kAf). It also presumes some understanding of hexadecimal notation and how to turn a string of bits into a hex value.

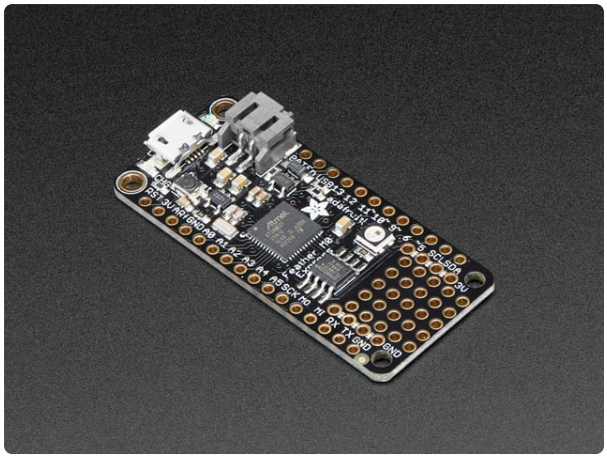
Hardware and software requirements

The sample code we are providing you is designed to be run on a variety of Adafruit boards that can use the GFX library. The screenshots shown here were using the 2.4 inch 320x240 TFT FeatherWing but we have also provided options for the 3.5 inch 480x320 FeatherWing, Adafruit HalloWing and PyGamer displays.

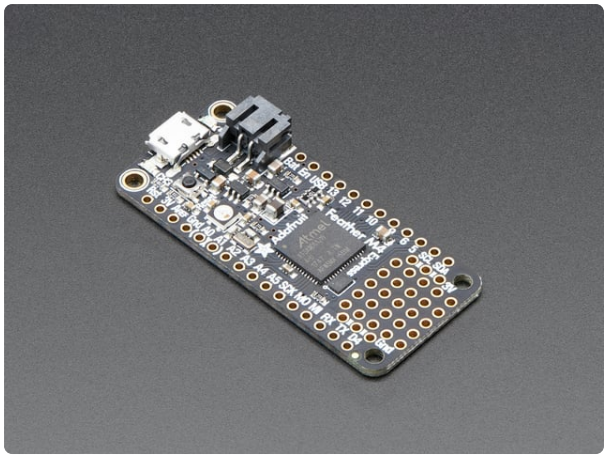
When using the TFT FeatherWings you also need a Feather board. We recommend some version of M0 or M4 or any other 32-bit Feather. The custom fonts we will be using take up a lot of memory and cannot be run on 8-bit boards such as a 32u4. We used a Feather M0 Express. The M4 boards are much faster and would be an even better choice.



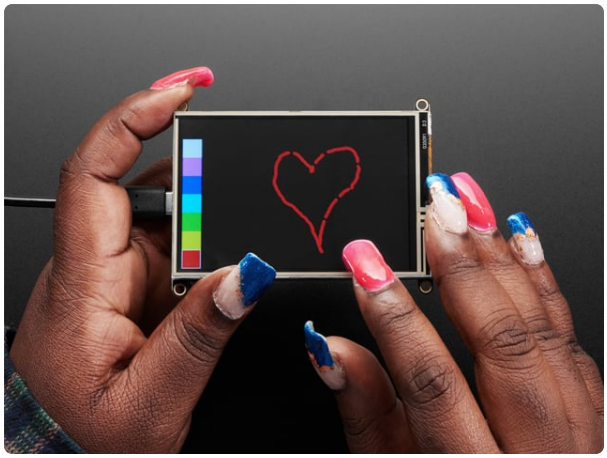
TFT FeatherWing - 2.4" 320x240 Touchscreen For All Feathers
A Feather board without ambition is a Feather board without FeatherWings! Spice up your Feather project with a beautiful 2.4" touchscreen display shield with built in microSD card...
https://www.adafruit.com/product/3315



[Adafruit Feather M0 Express]
At the Feather M0's heart is an ATSAMD21G18 ARM Cortex M0+ processor, clocked at 48 MHz and at 3.3V logic, the same one used in the new
https://www.adafruit.com/product/3403

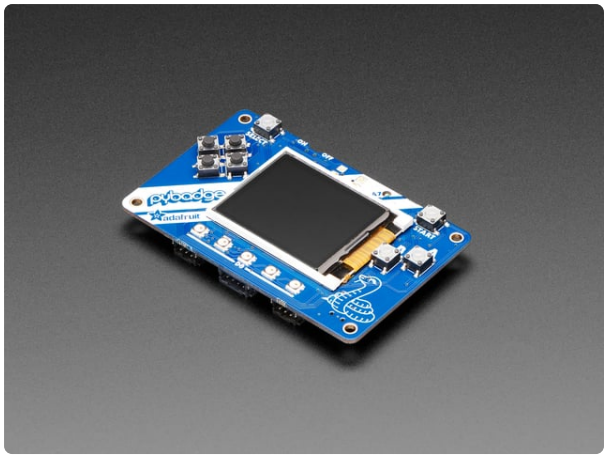


[Adafruit Feather M4 Express - Featuring ATSAMD51]
It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...
https://www.adafruit.com/product/3857



[Adafruit TFT FeatherWing - 3.5" 480x320 Touchscreen for Feathers]
Spice up your Feather project with a beautiful 3.5" touchscreen display shield with built in microSD card socket. This TFT display is 3.5" diagonal with a bright 6 white-LED...
https://www.adafruit.com/product/3651

[Adafruit HalloWing M0 Express]
This is Hallowing..this is Hallowing... Hallowing! Hallowing! Are you the kind of person who doesn't...
https://www.adafruit.com/product/3900



Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino

What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, its the Adafruit PyBadge! We wanted to see how much we...

https://www.adafruit.com/product/4200

Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino

What fits in your pocket, is fully Open Source, and can run CircuitPython, MakeCode Arcade or Arduino games you write yourself? That's right, it's the Adafruit...

https://www.adafruit.com/product/4242

You also need to download and install the Adafruit GFX library. For details on how to do that visit this tutorial (https://adafru.it/zra). You also need support for whatever display you are using. Go to the learning guide for the particular display you are using to see which libraries are needed to support it.

Finally you will need the sample sketch for this tutorial which can be found on GitHub using the button below.

font_test on GitHub

https://adafru.it/FiG

Understanding the Font Specification

After you've downloaded the sample sketch, open it up in the Arduino IDE and look at the second tab containing **SymbolMono18pt7b.h**. You might also want to go to **Arduino/libraries/Adafruit_GFX/Fonts/FreeMono18pt7b.h** and open it in your favorite text editor.

The font that we will create is going to be an 18 point monospace symbol font designed to work with the FreeMono18pt font supplied with the Adafruit GFX library.

That free monospace font defines characters from 32-126. We are going to create a font that uses 0-31 as well as 127 and upwards. We will use a custom display function to switch back-and-forth between these two fonts depending on the character value that you send.

The way that glyph information is stored in these GFX fonts is highly optimized and only uses exactly the number of bits of data necessary to define the particular bitmap. Trying to encode these bits by hand would be prohibitively difficult but we're going to cheat little bit. We're going to always use bitmaps that are exactly 16 bits wide even though some of our symbols are not fully that wide. This is going to be a tiny bit inefficient as far as storage goes. However with M0 or M4 boards, which are necessary to use these custom fonts anyway, using a few extra bytes here and there is not going to hurt too much and it's going to make this manual encoding system possible.

Let's look at the **SymbolMono18pt7b.h** file contents in the Arduino IDE second tab. The first thing we have is an array of bytes the defined the bitmaps. It looks like this.

```
const uint8_t SymbolMono18pt7bBitmaps[] PROGMEM = {
//A bunch of data goes here
};
```

Next is an array of glyph information. For example:

```
const GFXglyph SymbolMono18pt7bGlyphs[] PROGMEM = {
  //Index,  W, H,xAdv,dX, dY
  {     0, 16,21, 21, 3,-19}, // 00 test square
  {    42, 16,15, 21, 3,-18}, // 01 Upper_Left_Arrow
  {    72, 16,15, 21, 3,-18}, // 02 Upper_Right_Arrow
//Etc.
  {    42, 16,15, 21, 3,-18}};//144
```

This is an array of structures of type `GFXglyph`. The first number is the index into the bitmap array. So in this example our first glyph starts at index zero. Look at the top of the file and you can see that this glyph that we call "test square" defines 42 bytes. That means that the next glyph begins at index 42. The second glyph, a upward left pointing arrow, is 30 bytes long so that means the third glyph starts at index 72 and so on. You just keep adding the length of the previous glyph of the one prior to that.

The second and third items are the width and height of the glyph. In our case, we have to have a glyph that is 16 pixels wide even if we don't really need that much space. Otherwise, we cannot manually compute the bitmap patterns. The height can vary as much as we want. Note that the test square was 21 pixels tall however the arrows are only 15. If you look further down the list you will see that glyph 134 "Space Bar" with only three pixels tall. The index will always advance by 2 times the height

value. So as we noted the test square is 21 pixels tall. That means it takes 42 bytes to define it. The arrows are 15 pixels tall and they take 30 bytes. So the index advances by that much.

The next value is the **xAdvance** value. When using these fonts with the `print()` or `println()` functions this tells the software how much space to put between each character horizontally. Because this is a fixed space font we are using a constant value of 21. Although our actual glyph is only 16 pixels wide this gives two extra pixels on one side and three on the other. You need that blank space for readability. The way we have implemented things so that it automatically switches back and forth between our symbol font and the standard mono 18 font, we will not be able to use `print` or `println`. Instead we will use a custom `drawChar` function to draw individual characters.

The final two values are called **dX** and **dY** which are used to position the glyph within the cell. As you can see the **dX** value varies between 2-4. The **dY** value is the distance from the baseline of the character to the top of the glyph. So for example the large test square starts 19 pixels up from the baseline of the character cell. However the "Space Bar" glyph is only 2 pixels above the baseline. You will have to play around with these values to get the glyph centered horizontally and vertically the way that you want it. The sample sketch we provide lets you visualize that position and you can make the appropriate adjustments.

The final part of the font definition is a structure that ties everything together. It includes a pointer to the bitmap array, a pointer to the glyph data and a few other values. In our case we have listed the next three values as "0,48, 35". The first is the ASCII value of your first character. While most fonts run from the range of 32-126 this one goes from 0-48 but we use a custom display function that actually makes the range "0-31 and 126 upwards". We will explain more when we talk about our sample sketch. The value "35" is the standard vertical spacing between rows of text when using `print` or `println` however as mentioned earlier we will be specifically placing individual characters at exact locations and not using this value.

```
const GFXfont SymbolMono18pt7b PROGMEM = {
  (uint8_t  *)SymbolMono18pt7bBitmaps,
  (GFXglyph *)SymbolMono18pt7bGlyphs,
  0,48, 35 //ASCII start, ASCII stop,y Advance
};
```

The remaining items in the file are not part of the standard Adafruit GFX font format. There are just some definitions that we need for this custom symbol font. The **DELTA_C**, **DELTA_R** and **BASE_R** values help us display these symbols in a fixed grid. The other definitions are names that we can use in our application program to reference all of the symbols that we've defined.

```
#define DELTA_C 21
#define DELTA_R 30
#define BASE_R 22

#define MY_TEST_SQUARE 0
#define MY_UPPER_LEFT_ARROW 1
#define MY_UPPER_RIGHT_ARROW 2
#define MY_LOWER_LEFT_ARROW 3
//Etc.
```

Using the font display sketch

The sample sketch called **font_test** prints out a portion of your font starting at a specified character at a specified magnification value. Starting at line 8, there are a series of `define` statements that select which kind of display board you're using. The sample images in this tutorial show a 2.4 inch 320x240 TFT FeatherWing but you can use any of the boards listed in the sketch. You should un-comment only one of the definitions. This will automatically configure for that board using a code in **board_select.h**.

```
//Uncomment only one of the following lines to select your display.
#define USE_ILI9341   //2.4 inch 320x240 TFT Feather Wing
//#define USE_HX8357    //3.5 inch 480x320 TFT Feather Wing
//#define USE_HALLOWING //Adafruit Hallowing M0 Express
//#define USE_PYGAMER   //Adafruit PyGamer
#include "board_select.h"
```

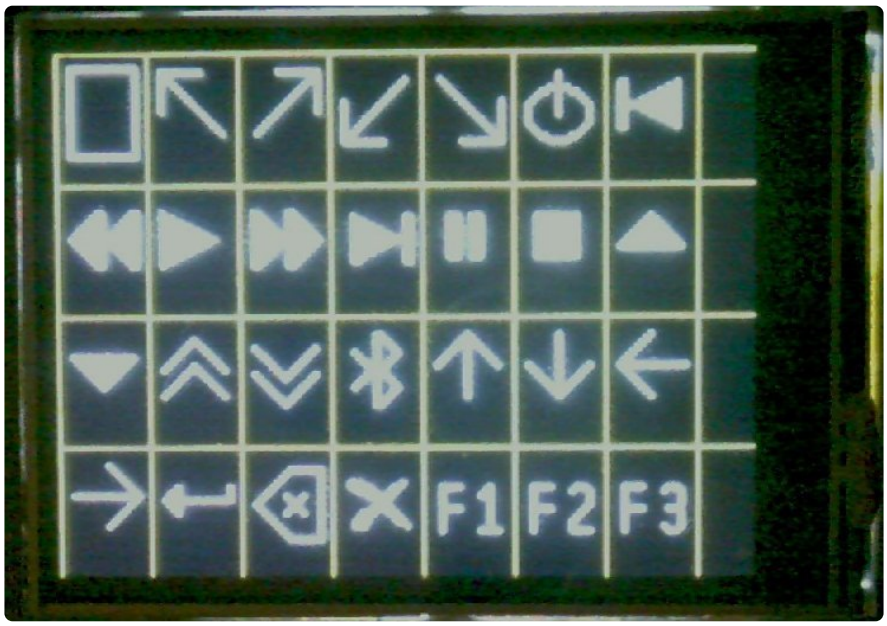
In the file **board_select.h** are all of the configurations for the various boards supported. When using the TFT FeatherWings, it presumes you're using a Feather M0 or M4. If you're using a different Feather, you may need to change some of the parameters in that file. Look for the section that look like this.

```
//Assumes Feather M0 or M4 or other compatible
#define STMPE_CS 6
#define TFT_CS   9
#define TFT_DC   10
#define SD_CS    5
```

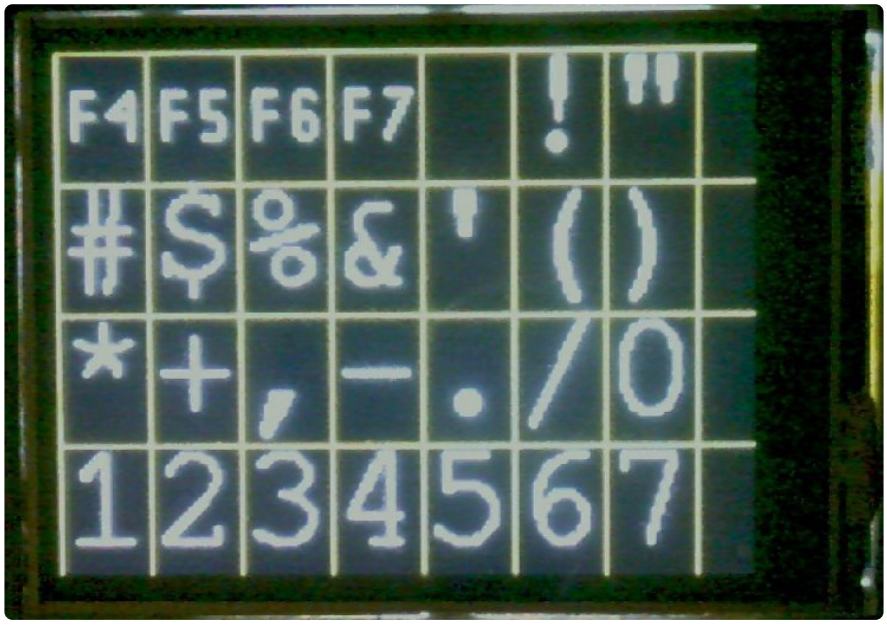
After configuring the sketch for your particular hardware, you should compile and upload the sketch. After compiling and uploading, open your serial monitor and you will see the following:

```
Type a positive number to adjust the first character displayed.
Type a negative number to change the magnifier. ie -2 multiplied by two.
Displaying 28 glyphs in 4 rows by 7 columns.
Magnifier:2
Displaying characters:0 through 27
```

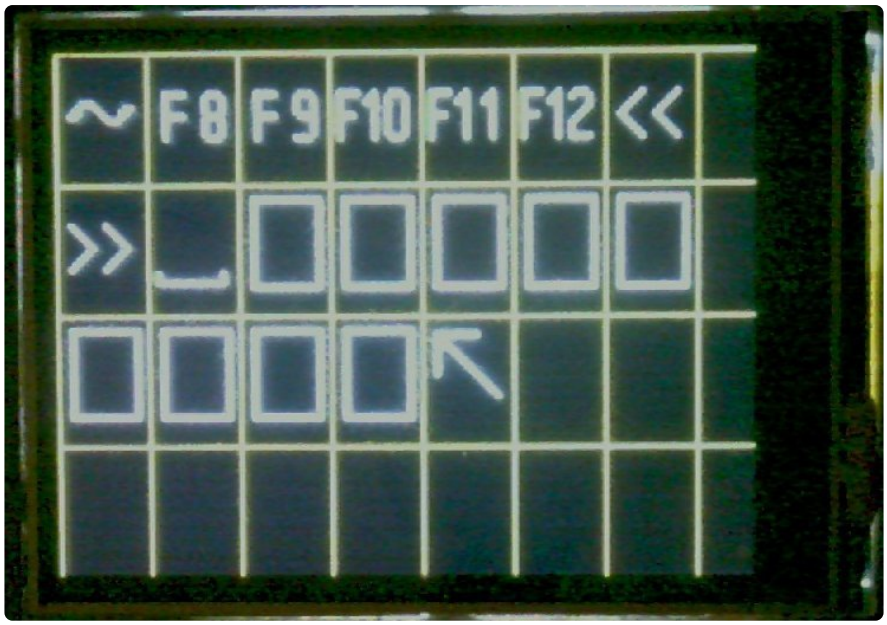
On your display you will see the first 28 glyphs defined in the font as seen below.



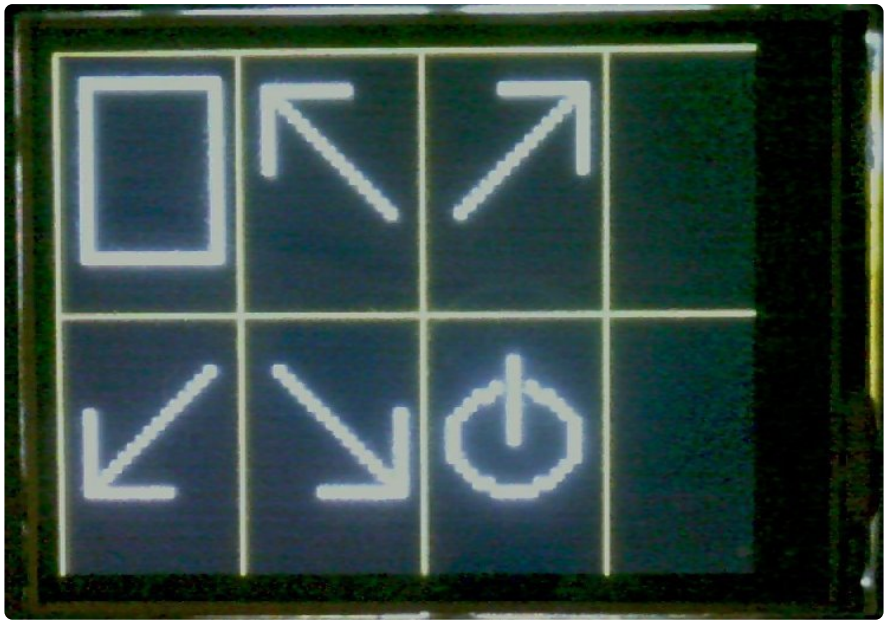
Type 28 into the serial monitor and press Enter and the display will now show glyphs 28-55. Note that the fifth one over is glyph 32 which is a blank space in the standard ASCII character set followed by 33 which is a "!" and so on. These are not symbols from our custom symbol font. These are being displayed from the **FreeMono18pt7b.h** font information.



You can continue to type in larger numbers to see the rest of the font but for our custom symbols you need to skip ahead to about 126 as shown below. Character 126 is ASCII "~" and then at 127 and beyond we continue with our other custom symbols. At the end, we put in several test squares just as placeholders for future expansion and the final arrow was just to let me know the software was displaying the proper number of glyphs.



You can also increase the magnification of the glyphs displayed. The program defaults to magnification 2 but let's try 4. You change the magnification by typing in a negative number. Let's go back to glyph zero by typing a "0" and pressing Enter. Then enter "-4" to see the following display.



Now type "-1" to see what a non-scaled version of the font looks like. This is the version that we will actually use in our program. It is displayed without gridlines.

Now that you know how to operate the display program, in the next section of the tutorial we will show you how to define some new character glyphs. This will illustrate how we created the file in the first place.

Creating new glyphs

Now we want to illustrate for you how we created the symbol font by adding some additional symbols. We are going to add symbols for the standard card suits of club, diamonds, hearts, and spades. First let's look at how some of the other symbols were created. Go into the **SymbolMono18pt7b.h** and look at line number 557 and forward that defines the Double Greater Than symbol. It looks like this.

```
//132 Double greater than
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| X . . . . . . , X . . . . . . . |*/  0x80,0x80,
/*| . X . . . . . , . X . . . . . . |*/  0x40,0x40,
/*| . . X . . . . , . . X . . . . . |*/  0x20,0x20,
/*| . . . X . . . , . . . X . . . . |*/  0x10,0x10,
/*| . . . . X . . , . . . . X . . . |*/  0x08,0x08,
/*| . . . . . X . , . . . . . X . . |*/  0x04,0x04,
/*| . . . . X . . , . . . . X . . . |*/  0x08,0x08,
/*| . . . X . . . , . . . X . . . . |*/  0x10,0x10,
/*| . . X . . . . , . . X . . . . . |*/  0x20,0x20,
/*| . X . . . . . , . X . . . . . . |*/  0x40,0x40,
/*| X . . . . . . , X . . . . . . . |*/  0x80,0x80,
```

Most of each line is simply a comment that doesn't really generate any code but it's an array of dots that allows us to visualize what the symbol will look like. We have replaced the dots with "X" wherever a pixel is supposed to be on. Note that the center column has commas rather than periods just to give us a reference point. Across the top are the bit values that we were used to generate the actual hexadecimal data. The first four columns of the first row has an X in the column

labeled 8. The next four columns are empty so the first hex value is 0x80. The pattern repeats for the right half of the symbol so we get another 0x80. As you go download row by row you can see how we generated the hex values based on where we drew the Xs. As you browse up-and-down looking at the symbols we've already created you might notice that this textbased representation using periods and Xs isn't exactly a 1-1 ratio of height to width as it appears on the actual display. This representation is going to look a little bit tall and skinny compared to how the graphic will actually look. But it gives us a pretty good idea of what we want and how to encode the data as hex values.

Now look at line 575 and forward and you will see an empty set of data that we will use to create our symbols for the four suits.

```
//   Empty array for future expansion
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . , . . . . . . . . |*/  0x00,0x00,
```

Copy and paste that section until you have 4 copies. Edit the top comment to indicate that this will be symbol "135 diamond suit". You should press the "insert" key on your keyboard. This will change the Arduino IDE editor from insert mode to overwrite mode. The cursor will change from a blinking vertical line to a blinking rectangle. Use your arrow keys to position the cursor over the dots in the comments section and then press "X" or any other distinguishing character of your choice to mark the location. When you are finished "drawing" the diamond shape it might look something like this.

```
//135 diamond suit
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . . . . . X . . . . . . . . |*/  0x00,0x00,
/*| . . . . . . X X X . . . . . . . |*/  0x00,0x00,
/*| . . . . . . X X X . . . . . . . |*/  0x00,0x00,
/*| . . . . . X X X X X . . . . . . |*/  0x00,0x00,
/*| . . . . . X X X X X . . . . . . |*/  0x00,0x00,
/*| . . . . X X X X X X X . . . . . |*/  0x00,0x00,
/*| . . . . X X X X X X X . . . . . |*/  0x00,0x00,
/*| . . . X X X X X X X X X . . . . |*/  0x00,0x00,
/*| . . . . X X X X X X X . . . . . |*/  0x00,0x00,
/*| . . . . X X X X X X X . . . . . |*/  0x00,0x00,
/*| . . . . . X X X X X . . . . . . |*/  0x00,0x00,
/*| . . . . . X X X X X . . . . . . |*/  0x00,0x00,
```

```
/*| . . . . . X X X . . . . . . . |*/  0x00,0x00,
/*| . . . . . X X X . . . . . . . |*/  0x00,0x00,
/*| . . . . . . . X . . . . . . . |*/  0x00,0x00,
```

At this point all we have done is edit the comment. We haven't really entered any actual data. The top line will be very easy because there is just a single pixel in the center column labeled "1" so this results in data "0x01,0x00". The second row has X in both column 2 and 1 so we add them together. On the right half there is a X in column 8 so we end up with "0x03,0x80". The third row is identical. The fourth row we add up 4+2+1 for the left half. On the right half we have 8+4 which is 12 but represented in hexadecimal as "c" therefore the data for that line is "0x07,0xc0". You can use the insert key overwrite feature to change the zeros to the proper hexadecimal value. When completed, the section of code looks like this.

```
//135 diamond suit
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . X X X . . . . . . . . |*/  0x03,0x80,
/*| . . . . . X X X . . . . . . . . |*/  0x03,0x80,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . X X X X X X X X X . . . . |*/  0x1f,0xf0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
```

Now scroll down into the glyph information table and edit the data for glyph 135. Because the "space bar" glyph prior to this was three rows of pixels tall, it took six bytes of data to define it. The "space bar" data started at index 976 so our data for the diamond will start at 982. As always the width is 16 pixels. We have defined 15 rows of pixels for the height. The **xAdvance** is always 21, we will try **dX** of 3 and **dY** of 16. When edited, that section of code looks like this.

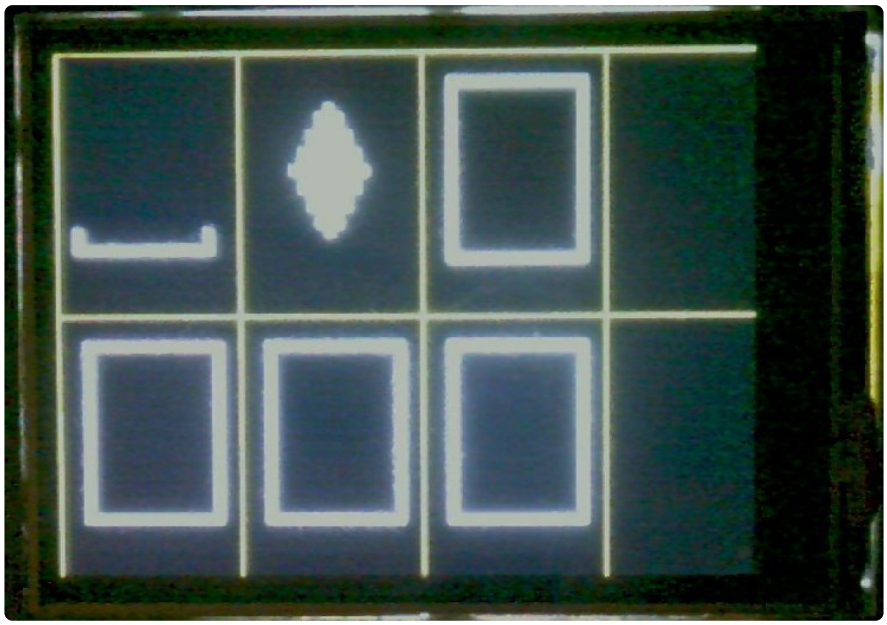
```
  {   954, 16,11, 21, 3,-13}, //133 double greater than
  {   976, 16, 3, 21, 2,- 2}, //134 space bar
  {   982, 16,15, 21, 3,-16}, //135 diamond suit
  {     0, 16,21, 21, 3,-19}, //136
  {     0, 16,21, 21, 3,-19}, //137
```

Go back to the main sketch and in the setup function change the default magnifier to 4 and the **First_Glyph** to 134 so that we can on the symbols we are currently working on. Compile and upload the sketch.

```
  delay (1000);
  Magnifier=4;
```

```
First_Glyph=134;
Show ();
```

The results will look like this. You might want to play around with the **dX** and **dY** values to see how they affect the position of the diamond within the grid.



You can then proceed to design your own version of hearts, clubs, and spades. Here is our version.

```
//136 hearts suit
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . X X . . , . . X X . . . . |*/  0x18,0x30,
/*| . X X X X X . , . X X X X X . . |*/  0x7c,0x7c,
/*| X X X X X X X , X X X X X X X . |*/  0xfe,0xfe,
/*| X X X X X X X X X X X X X X X . |*/  0xff,0xfe,
/*| X X X X X X X X X X X X X X X . |*/  0xff,0xfe,
/*| . X X X X X X X X X X X X X . . |*/  0x7f,0xfc,
/*| . X X X X X X X X X X X X X . . |*/  0x7f,0xfc,
/*| . . X X X X X X X X X X X . . . |*/  0x3f,0xf8,
/*| . . X X X X X X X X X X X . . . |*/  0x3f,0xf8,
/*| . . . X X X X X X X X X . . . . |*/  0x1f,0xf0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
//137 clubs suit
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . X X X X X X X X X . . . . |*/  0x1f,0xf0,
/*| . . . X X X X X X X X X . . . . |*/  0x1f,0xf0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . X X . . . X . . . X X . . . |*/  0x31,0x18,
/*| . X X X X . . X . . X X X X . . |*/  0x79,0x3c,
/*| X X X X X X . X . X X X X X X . |*/  0xfd,0x7e,
/*| X X X X X X X X X X X X X X X . |*/  0xff,0xfe,
/*| X X X X X X . X . X X X X X X . |*/  0xfd,0x7e,
/*| . X X X X . . X . . X X X X . . |*/  0x79,0x3c,
/*| . . X X . . . X . . . X X . . . |*/  0x31,0x18,
```

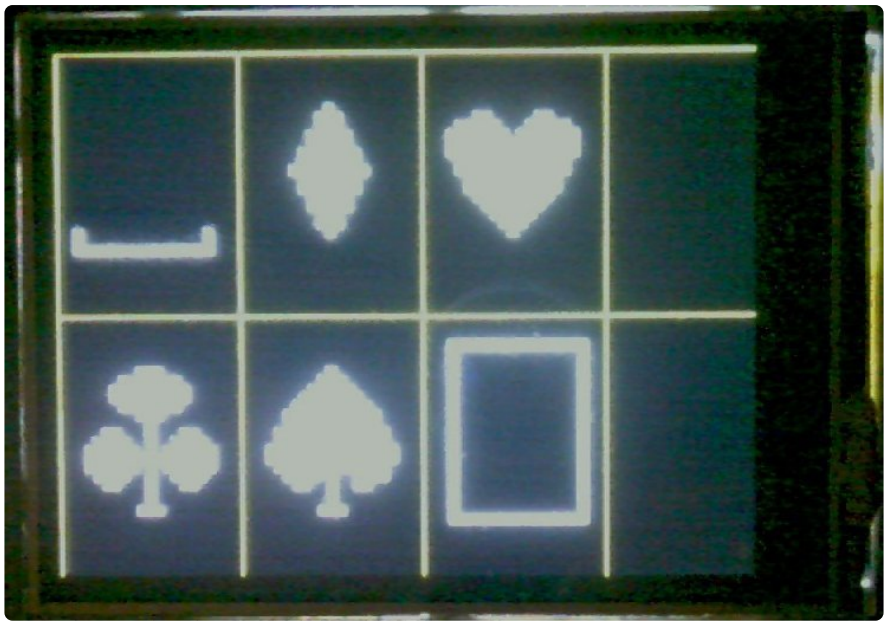
```
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
//138 spades suit
/*| 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
/*| . . . . . X X X X X . . . . . . |*/  0x07,0xc0,
/*| . . . . X X X X X X X . . . . . |*/  0x0f,0xe0,
/*| . . . X X X X X X X X X . . . . |*/  0x1f,0xf0,
/*| . . X X X X X X X X X X X . . . |*/  0x3f,0xf8,
/*| . . X X X X X X X X X X X . . . |*/  0x3f,0xf8,
/*| . X X X X X X X X X X X X X . . |*/  0x7f,0xfc,
/*| . X X X X X X X X X X X X X . . |*/  0x7f,0xfc,
/*| X X X X X X X X X X X X X X X . |*/  0xff,0xfe,
/*| X X X X X X X X X X X X X X X . |*/  0xff,0xfe,
/*| . X X X X X X X X X X X X X . . |*/  0x7f,0xfc,
/*| . . X X X X . X . X X X X . . . |*/  0x3d,0x3c,
/*| . . . X X . . X . . X X . . . . |*/  0x19,0x30,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . . . X . . . . . . . . |*/  0x01,0x00,
/*| . . . . . . X X X . . . . . . . |*/  0x03,0x80,
```

Here is the glyph data.

```
I {   976, 16, 3, 21, 2,- 2}, //134 space bar
  {   982, 16,15, 21, 3,-16}, //135 diamond suit
  {  1012, 16,14, 21, 3,-15}, //136 heart suit
  {  1040, 16,17, 21, 3,-16}, //137 clubs suit
  {  1074, 16,17, 21, 3,-16}, //138 spades suit
  {     0, 16,21, 21, 3,-19}, //139
```

Finally at the bottom of the file you should add some definitions so that we can refer to these new symbols in our application programs.

```
#define MY_SPACE_BAR 134
#define MY_DIAMOND_SUIT 135
#define MY_HEARTS_SUIT 136
#define MY_CLUB_SUIT 137
#define MY_SPADE_SUIT 138
```



In the next section we will show you how to display these characters in your application program.

How to display your symbols

For our particular application, we needed to be able to easily switch between the standard **FreeMono18pt** font and our new **SymbolMono18pt** font. We decided at first that we would use characters 0-31 and write a special draw function that would automatically switch between the symbol font for these first 32 characters and the regular mono font for anything 32 and up. However eventually we had to expand the font beyond the 32 available slots so we picked up again at character 127 and upwards. Our special draw routine properly handles all of this. You could just as easily design a font that uses the usual 32-127 printable ASCII characters if you have some other mechanism for deciding which font to use. Our application also was going to display these characters in a grid for a virtual infrared remote to operate TV, cable, Blu-ray etc. devices. We weren't really planning on printing normal strings using `print()` or `println()`. You might be able to adapt these methods to use regular `print` routines if you need to.

Look at the sample sketch **font_test** and you will see the following function.

```
/*
 * Use this function instead of display.drawChar to draw the symbol or to use
 * the default font if it's not in the symbol range.
 */
void drawSymbol(uint16_t x, uint16_t y, uint8_t c, uint16_t color, uint16_t bg,
uint8_t Size){
  if( (c&gt;=32) &amp;&amp; (c&lt;=126) ){ //If it's 33-126 then use standard mono
18 font
      display.setFont(&amp;FreeMono18pt7b);
  } else {
```

```
    display.setFont(&amp;SymbolMono18pt7b);//Otherwise use special symbol font
    if (c&gt;126) {      //Remap anything above 126 to be in the range 32 and
upwards
      c-=(127-32);
    }
  }
  display.drawChar(x,y,c,color,bg,Size);
}
```

You can use this function instead of `display.drawChar(x,y,c,color,bg,Size)` to display the characters. It will handle switching back and forth between the symbol font and the **FreeMono** font. Note that when using these style fonts, the background color parameter is ignored as explained in the Adafruit GFX library documentation. For example if you wanted to draw our "heart suit" at location 0, 20 in red at double size you would do...

```
drawSymbol(0,20, MY_HEART_SUIT,ILI9341_RED,0,2);
```

Advanced topics

The techniques described here take advantage of the fact that having a fixed character width of 16 pixels. That makes it easy to define 2 bytes of data and to do it by hand. You want to make a smaller font it would also be extremely easy to define characters that were exactly 8 pixels wide. Such a definition might look like this.

```
//01 Upper left arrow
/*| 8 4 2 1 8 4 2 1 |*/
/*| X X X X . . . . |*/  0xf0,
/*| X X . . . . . , |*/  0xc0,
/*| X . X . . . . , |*/  0xa0,
/*| X . . X . . . , |*/  0x90,
/*| . . . . X . . , |*/  0x08,
/*| . . . . . X . , |*/  0x04,
/*| . . . . . . X , |*/  0x02,
/*| . . . . . . . X |*/  0x01,
```

Technically, it is possible to make characters 12 pixels wide but it's a little more difficult to encode the data by hand. The first two bytes of data in code 12 bits from the first row of pixels and 4 bits from the next row. This is followed by a single byte that encodes the remaining 8 bits of the second row. Then the pattern repeats with 12 bits from the third row in the first 4 bits from the fifth row. I've encoded lots of 16 bit symbols and I can tell you that just doing 3 or 4 examples here using the 12 bit method was about twice as hard. I made way more mistakes along the way. Here are some samples.

```
//00 Test square
/*| 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| X X X X X X X X X X X X |*/  0xff,0xf8, //12 bits from 1st row and 4 bits from
2nd row
/*| X . . . . . . , . . . X |*/  0x01,      //remaining 8 bits of the 2nd row
```

```
/*| X . . . . . . , . . . X |*/  0x80,0x18, //12 bits from 3rd row and 4 bits from
4th row
/*| X . . . . . . , . . . X |*/  0x01,      //remaining 8 bits of 4th row
/*| X . . . . . . , . . . X |*/  0x80,0x18, //12 bits from 5th row and 4 bits from
6th row
/*| X . . . . . . , . . . X |*/  0x01,      //remaining 8 bits of 6th row
/*| X . . . . . . , . . . X |*/  0x80,0x18, //12 bits from 7th row and 4 bits from
8th row
/*| X . . . . . . , . . . X |*/  0x01,
/*| X . . . . . . , . . . X |*/  0x80,0x18,
/*| X . . . . . . , . . . X |*/  0x01,
/*| X . . . . . . , . . . X |*/  0x80,0x18,
/*| X . . . . . . , . . . X |*/  0x01,
/*| X . . . . . . , . . . X |*/  0x80,0x18,
/*| X . . . . . . , . . . X |*/  0x01,
/*| X X X X X X X X X X X X |*/  0xff,0xf0, //12 bits from final row. Last 4 bits
unused.
//Total of 23 bits of data

//01 upper left arrow
/*| 8 4 2 1 8 4 2 1 8 4 2 1 |*/
/*| X X X X X X . . . . . . |*/  0xfe,0x08, //12 bits from 1st row and 4 bits from
2nd row
/*| X X . . . . . , . . . . |*/  0x00,      //remaining 8 bits of the 2nd row
/*| X . X . . . . , . . . . |*/  0xa0,0x09, //12 bits from 3rd row and 4 bits from
4th row
/*| X . . X . . . , . . . . |*/  0x00,      //remaining 8 bits of 4th row
/*| X . . . X . . , . . . . |*/  0x88,0x08, //12 bits from 5th row and 4 bits from
6th row
/*| X . . . . X . , . . . . |*/  0x40,      //remaining 8 bits of 6th row
/*| X . . . . . X , . . . . |*/  0x82,0x08, //12 bits from 7th row and 4 bits from
8th row
/*| . . . . . . . X . . . . |*/  0x10,
/*| . . . . . . . , X . . . |*/  0x00,0x80,
/*| . . . . . . . , . X . . |*/  0x04,
/*| . . . . . . . , . . X . |*/  0x00,0x20,
/*| . . . . . . . , . . . X |*/  0x01,
//18 bites
```

We hope that this tutorial sparks your imagination and you will come up with other creative uses of this method. Perhaps a set of chess pieces or other game tokens might be useful.