Displaying Data with Arduinos

David R. Brooks, *Institute for Earth Science Research and Education,* © 2020

This document shows how to use various devices to display data collected from sensors attached to Arduinos. An Arduino UNO is used for most of the examples. In general, no changes are required to UNO code other than choosing the correct board in the IDE before uploading a sketch. All the sample sketches included here were created with the Arduino IDE on a Windows 10 computer. There's no reason to believe that results will be different with other operating systems. The document is not an Arduino tutorial; it assumes some familiarity with the Arduino IDE and using Arduino-compatible sensors of various kinds to collect data.

I hope this document will make it easier to find and use appropriate data display methods and devices. Along the way I have had some occasional unexplained compile and upload glitches, but all the code presented here has run successfully through my Windows 10 computer with Arduino IDE 1.8.9. You can download all the sketches in this document here: `www.instesre.org/ArduinoDataDisplay/code.txt`.

`Serial.print...` functions

The `Serial.print()` and `Serial.println()` functions are used to display values in a serial port (COM) window created by the Arduino IDE. Once a sketch has been uploaded, access the serial port window by clicking on the small "magnifying glass" icon in the upper right-hand corner of the IDE window. Of course these functions are available only when your Arduino is connected to a computer through a USB cable. These functions will display characters, strings of characters, and real or integer numbers. They transparently convert numerical values to strings of characters to be displayed. Numbers can be displayed in a variety of formats.

Only one value can be given as an input parameter in a `Serial.print()` or `Serial.println()` function. (A character string counts as one value.) The `Serial.println()` function adds a "new line" character after displaying the parameter.

Sometimes with long sketches, you may get error or warning messages that say your sketch will not compile or that there may be problems running the sketch. A common source of these warnings is that your Arduino is short of or out of RAM (Random Access Memory). This is because strings displayed by `Serial.print()` or `Serial.println()` take a *lot* of limited RAM space (2048 bytes on an UNO). One way around this is to use the `F()` macro inside `Serial.print()` or `Serial.println()` functions when you want to display strings. This macro keeps strings in program memory and out of RAM. `F()` won't work when you're displaying numerical values. It's alleged that the `F()` macro adds more time – more microcontroller clock cycles – to the time required to display a string, but this isn't a potential problem worth worrying about for displaying data from sensors. What does `F()` actually do? Putting all the string parameters in the sketch below inside `F()` macros, for example,

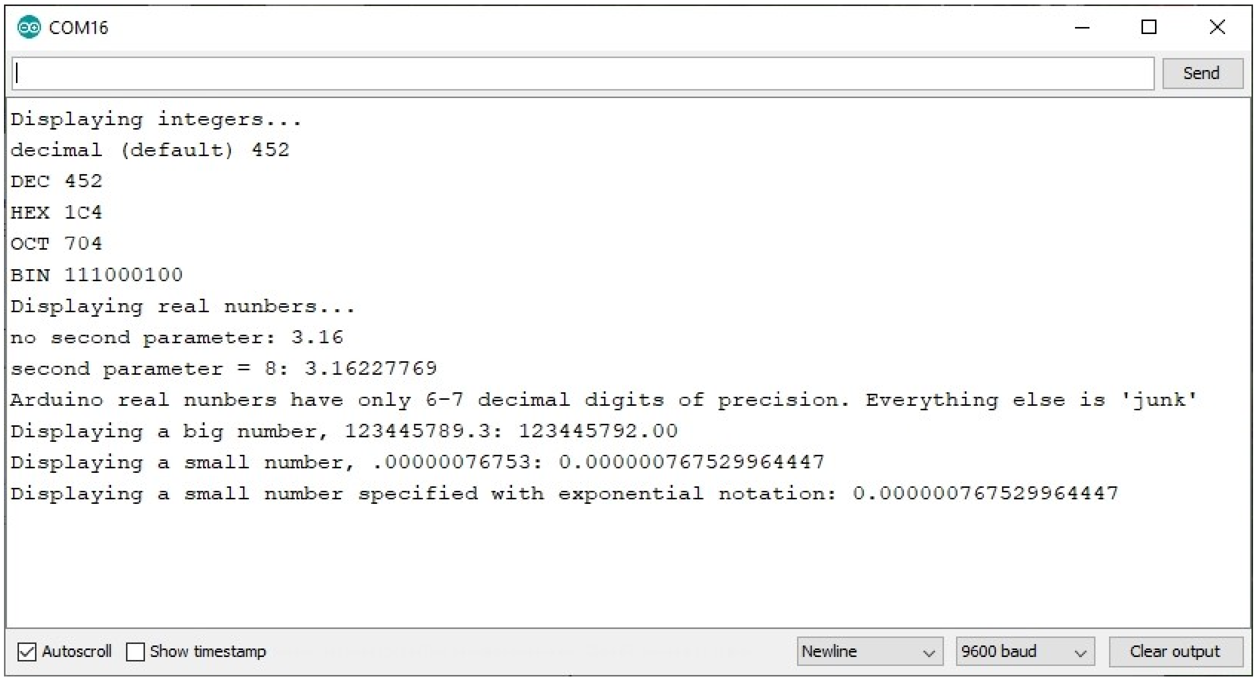
```
Serial.print(F("HEX "));
```
instead of
```
Serial.print("HEX ");
```

uses 202 bytes of RAM space instead of 426 bytes. This might not seem like much of an improvement, but this program uses only 3916 bytes (12%) of the available 32256 bytes of program space. For larger sketches, and/or sketches where you're displaying lots of strings, using `F()` can make the difference between a sketch that will work and one that won't.

For displaying real numbers, the optional second parameter is the number of digits to be displayed to the right of a decimal point. The default is 2 digits. You can specify more digits, but bear in mind that the precision for real numbers in Arduino programming is only 6-7 decimal digits (total digits, not just digits to the right of a decimal point). Specifying more digits than is available within the `float` data type – there is no "double precision" data type for real numbers in the Arduino programming language – is pointless and potentially misleading.

Arduino has problems with very large or very small real numbers. You can specify small or large numbers with exponential notation, but that doesn't remove the problems with displaying or using them. As shown below, you can add more digits with the second parameter in a print function, but that just displays junk digits and doesn't solve the problems inherent with using very small or very large real numbers in Arduino calculations. The only real solution is to be careful about what you ask your code to do. Usually, real physical values from sensor data can and should be scaled to be of a reasonable size. For Arduino data collection projects, you will almost certainly never find sensor data that has a precision of more than four or five digits. This will be even more important when you use display devices that have more limited space for representing numerical values.

```
// SerialPrintTest, D. Brooks, June 2020
int analogValue;
float realValue=sqrt(10.), big=123445789.3, small=.00000076753,
smallE=7.6753E-7;
void setup() {
  Serial.begin(9600); // Open the serial port at 9600 bps for UNO:
  analogValue = analogRead(0); // Read "junk" value from A0:
  // All these value formats are ASCII-encoded...
  Serial.println("Displaying integers...");
  Serial.print("decimal (default) "); Serial.println(analogValue);
  // default is decimal
  Serial.print("DEC "); Serial.println(analogValue, DEC);  // decimal
  Serial.print("HEX "); Serial.println(analogValue, HEX);  // hexadecimal
  Serial.print("OCT "); Serial.println(analogValue, OCT);  // octal
  Serial.print("BIN "); Serial.println(analogValue, BIN);  // binary
  Serial.println("Displaying real nunbers...");
  Serial.print("no second parameter: "); Serial.println(realValue);
  Serial.print("second parameter = 8: "); Serial.println(realValue,8);
  Serial.println("Arduino real nunbers have only 6-7 decimal digits of
precision. Everything else is 'junk'");
  Serial.print("Displaying a big number, 123445789.3: ");
Serial.println(big);
  Serial.print("Displaying a small number, .00000076753: ");
Serial.println(small,18);
  Serial.print("Displaying a small number specified with exponential
notation: ");
Serial.println(smallE,18);
}
void loop() {}
```

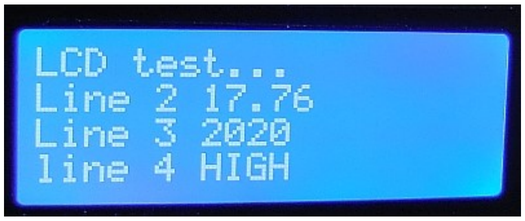


```
Displaying integers...
decimal (default) 452
DEC 452
HEX 1C4
OCT 704
BIN 111000100
Displaying real nunbers...
no second parameter: 3.16
second parameter = 8: 3.16227769
Arduino real nunbers have only 6-7 decimal digits of precision. Everything else is 'junk'
Displaying a big number, 123445789.3: 123445792.00
Displaying a small number, .00000076753: 0.000000767529964447
Displaying a small number specified with exponential notation: 0.000000767529964447
```

Liquid Crystal Displays (LCDs)

How do you display data when your Arduino project is no longer connected to your computer through a USB cable data? One solution is to use an LCD. Widely available LCDs will display 2 rows of 16 characters each or 4 rows of 20 characters each. You can get LCDs with programmable background colors and even with on-board buttons that can be read from your Arduino code to change what's being displayed.

"Bare" LCDs require several pins to interface with an Arduino. For just a few more dollars, you can get LCDs with an I2C interface which, like all I2C devices, requires only two pins for communications. You almost certainly will never have a problem with LCD I2C hardware addresses conflicting with addresses for other I2C devices. For these reasons, I never bother with LCDs that lack an I2C interface. Like all I2C devices, I2C-interfaced LCDs require software libraries (as do non-I2C LCDs). There are many such libraries available online and any reputable vendor will provide access to a library that will work with their LCDs. The left-hand image below shows 16x2 and 20x4 LCDs from sunfounder.com. (They still have the protective covering over the window.) On each of these I soldered small 4-pin screw terminal connectors for attaching to an Arduino; you could also use M/F jumper wires from the I2C interface board mounted on the back of the display.



The good news about LCDs is that their software libraries include print functions that nearly parallel the functionality of the serial window functions discussed in the previous section. Except for functions needed to set them up, you don't have to learn any new code to use LCDs. The right-hand image above shows output from the code below. White text on a blue background is standard for these devices. The backlight brightness, and hence the background/text contrast, can be adjusted with a potentiometer on the back of the board.

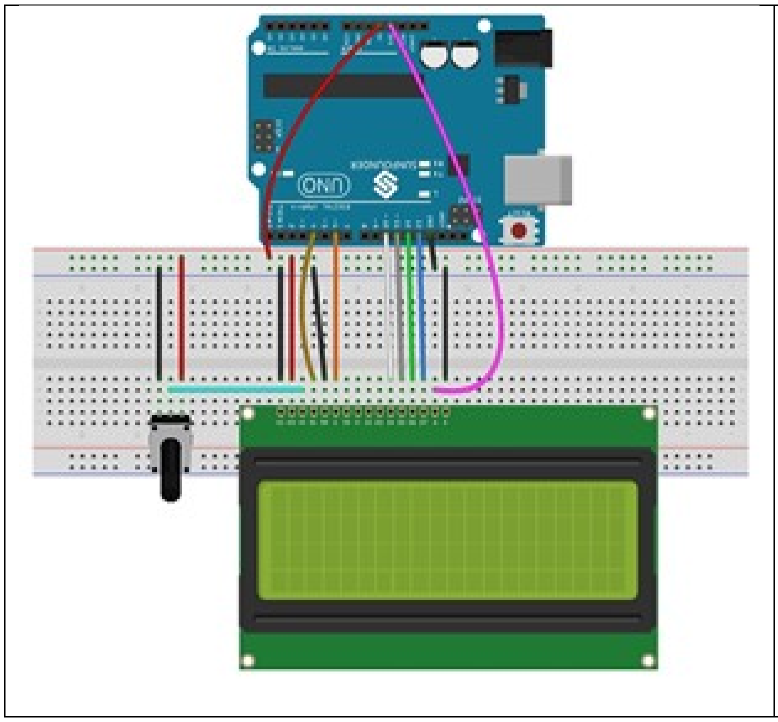
Note that the LCD print function for real numbers works like the serial print functions. In the code below, `lcd.print(17.765)` displays 17.76 – not 17.77, but `lcd.print(17.7651)` rounds to 17.77. `lcd.print(17.765,3)` displays the number as given. Writing `lcd.print(17.765,6)` displays 17.764999.

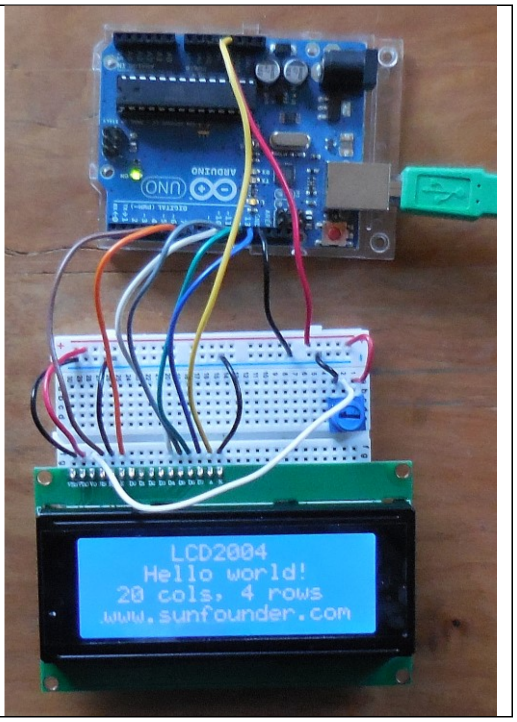
If you write data values to an LCD in a `loop()` function, you will almost certainly want to clear the LCD screen with every trip through the loop. Why? Characters displayed on the screen stay there until something else is written in that position. Suppose at one trip through the loop the value to display is 100.99 and the next time the value is 99.77. What will now appear on the screen is 99.779 because the "9" from the last print statement will still be there.

```
/* LCD1602_I2C_20x4.ino, D. Brooks, September 2017
  Email:support@sunfounder.com
  Website:www.sunfounder.com
*/
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
// set the LCD I2C address to 0x27 for a 16 chars 2 line display
LiquidCrystal_I2C lcd(0x27,20,4);
char line1[]="LCD test...";
char line2[]="Line 2";
char line3[]="Line 3";
char line4[]="line 4";
void setup()
{
  lcd.init(); // initialize the lcd
  lcd.backlight(); // turn on the backlight
  lcd.clear(); // clear the display
  lcd.setCursor(0,0);
  lcd.print(line1); lcd.print(' ');
  lcd.setCursor(0,1);
  lcd.print(line2); lcd.print(' ');lcd.print(17.765);
  lcd.setCursor(0,2);
  lcd.print(line3); lcd.print(' ');lcd.print(2020);
  lcd.setCursor(0,3);
  lcd.print(line4);  lcd.print(' '); lcd.print("HIGH");
}
void loop() {}
```

non-I2C LCD display hookup and code

As noted above, LCD displays with an I2C interface are easy to work with. But, this interface may cause some other I2C devices not to work properly. The alternative is to use an LCD *without* the I2C interface. The left-hand image shows a Fritzing breadboard layout for driving a 20×4 SunFounder LCD – see http://wiki.sunfounder.cc/index.php?title=LCD2004_Module#The_Experiment_for_Arduino. The potentiometer (I used a 10K potentiometer I had on my desk) is used to adjust the contrast of characters against the blue background. The right-hand image shows my breadboard layout. It *will* be necessary to use the potentiometer to set the contrast, so don't be discouraged it you don't see anything when you first power up the system. LCD from other sources *should* work the same.





Here's the code as supplied by SunFounder. If you haven't already installed the LiquidCrystal library, of course you will have to do so.

```
//LCD2004
//You should now see your LCD2004 display the characters
//Email:support@sunfounder.com
//Website:www.sunfounder.com
//2017.3.7
#include <LiquidCrystal.h>// include the library code
/**********************************************************/
LiquidCrystal lcd(4, 6, 10, 11, 12, 13);
/*********************************************************/
void setup()
```

```
{
  lcd.begin(20, 4);  // set up the LCD's number of columns and rows:
}
void loop()
{
    lcd.setCursor(6,0);  // set the cursor to column 19, line 0
    lcd.print("LCD2004");
    lcd.setCursor(4,1);
    lcd.print("Hello world!");
    lcd.setCursor(2,2);
    lcd.print("20 cols, 4 rows");
    lcd.setCursor(1,3);
    lcd.print("www.sunfounder.com");
}
```

TFT displays

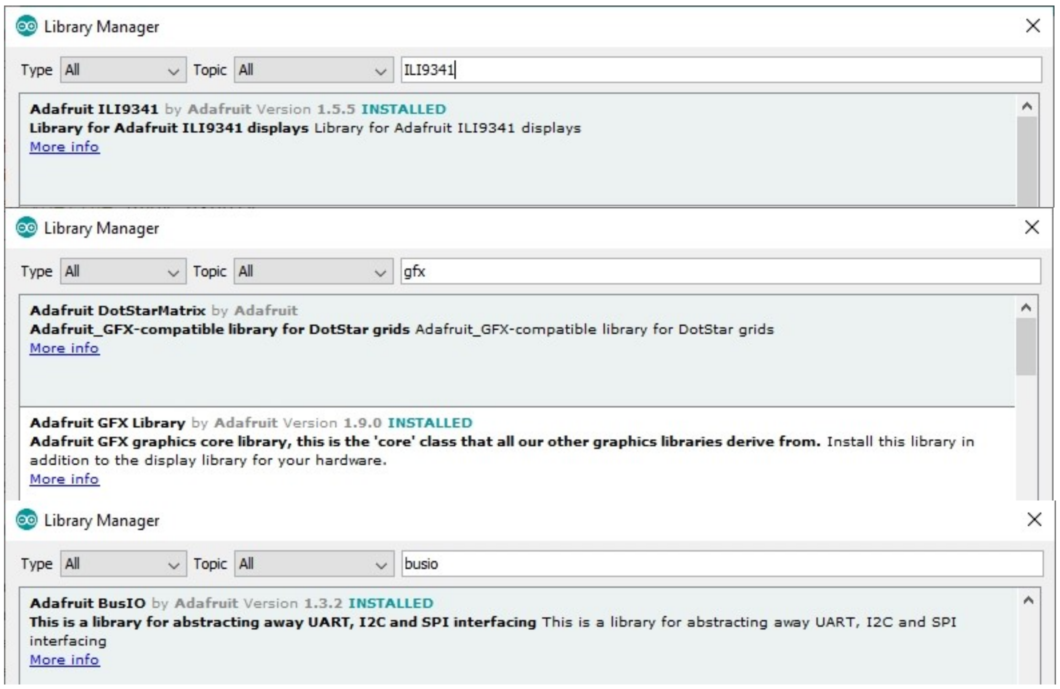
The serial port and `lcd` print functions are just for character-based data. Can you use displays to display graphical data? Yes, and the easiest way to do that is with TFT (thin-film transistor) displays. Full-color TFT displays are widely available. They can be used to print characters, but their ability to generate graphics displays is what makes them worth the coding effort.

TFT displays require software libraries to support them, and it's a good idea to get them from reliable vendors that provide the necessary software support. Adafruit.com is one such vendor and they sell several Arduino-compatible TFTs:

2.2" 240x320 pixels, ILI9341 library (ID 1480)
1.8" 128x160 pixels, ST7735R library (ID 358)
1.44" 128x128 pixels, ST7735R library (ID 2088)
1.14" 135x240 pixels, ST7789 library (ID 4383)

These all include a microSD card reader so that it's possible to fill the screen with appropriately sized bitmap (`.bmp`) files.

Each of these boards has its own software library and they all use the open-source Adafruit_GFX graphics library. All of these libraries can be found and installed using the Arduino IDE's Include Library→Manager tab. Here are examples of finding the GFX library and the appropriate libraries for the Adafruit 2.2" TFT display – they're already installed on my computer. (I'm currently using GFX version 1.9.0.)


Library Manager

Type All Topic All ILI9341

Adafruit ILI9341 by Adafruit Version 1.5.5 INSTALLED
Library for Adafruit ILI9341 displays Library for Adafruit ILI9341 displays
More info

Library Manager

Type All Topic All gfx

Adafruit DotStarMatrix by Adafruit
Adafruit_GFX-compatible library for DotStar grids Adafruit_GFX-compatible library for DotStar grids
More info

Adafruit GFX Library by Adafruit Version 1.9.0 INSTALLED
Adafruit GFX graphics core library, this is the 'core' class that all our other graphics libraries derive from. Install this library in addition to the display library for your hardware.
More info

Library Manager

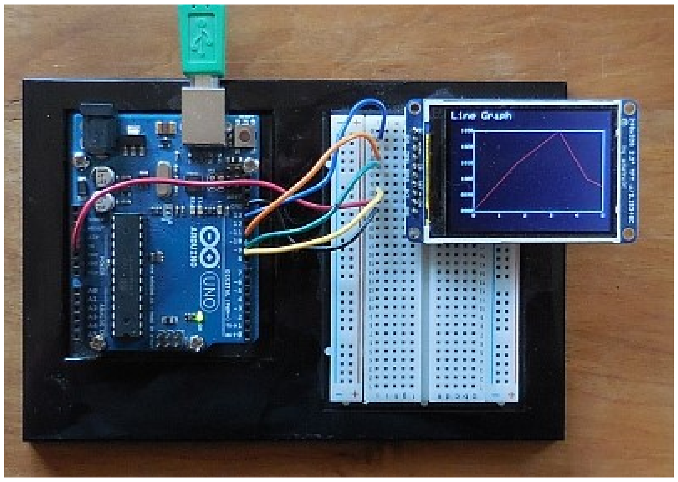
Type All Topic All busio

Adafruit BusIO by Adafruit Version 1.3.2 INSTALLED
This is a library for abstracting away UART, I2C and SPI interfacing This is a library for abstracting away UART, I2C and SPI interfacing
More info

The TFT displays themselves are very fragile. Vendors like Adafruit have mounted them on breadboard-friendly breakout boards with microSD card holders and level-shifting circuitry to allow their use with either 3.3 or 5.0 V (UNO) microcontrollers.

These TFT displays don't use I2C interfaces. Instead they use an SPI interface, which uses several pins for communications with your Arduino rather than just two as required for I2C devices.

This image shows an Adafruit 2.2" TFT display. The microSD card is on the back of the display, but it's not being used. The display is a line graph generated with "made-up" data – see below for the code.



These boards are small and it's best not to try getting too fancy with displaying too much data or text. The default built-in 5x8 pixel font, reminiscent of very old dot matrix printer fonts, is very basic, but perfectly adequate for these kinds of data graphs. The "location" pixel for a character is in the upper left-hand corner of the pixel rectangle. It's possible to scale the font to a larger size and specifying size=2 will create blocky but perfectly usable 10x16 pixel characters. It's also possible to install "nicer" fonts (see the GFX library documentation), but this is memory-intensive and will significantly reduce the coding space needed for other things; don't bother to do it!

Data display code for TFT displays

<u>Line graphs</u>

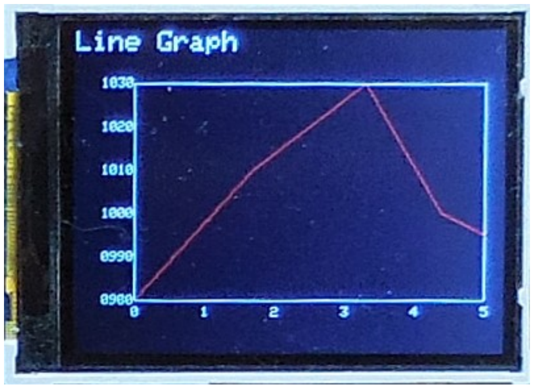
The coding challenge for creating graphs on TFTs is scaling data values to their pixel-based locations within a defined x-y graphing area. Some subset of these pixels is designated as the graph area. Pixels outside that area are used for axis labels and other text. For the 240x320 pixel 2.2" TFT display used here – see the image above – I defined a 260x200 x-y pixel space for the graph. The minimum and maximum x and y values are specified and those values are used to scale the data to their x-y pixel locations. The values being plotted can be either integers or real numbers, but in any case they must all be converted to integer values within the x-y pixel space. For these devices as addressed through the GFX graphics library, the x-y pixel coordinate (0,0) is in the upper left-hand corner of the display as oriented as shown in the above image. This makes the y-axis "upside down" from how you would usually consider drawing an x-y graph.

In the code below, 5 "made-up" barometric pressure values (in units of millibars) are graphed. The y-axis labels, written as strings rather than numbers, encompass the maximum and minimum values allowed for this graph. Because of space limitations, the x-axis labels are simply numbered 0 through 5, corresponding to (perhaps) the "time" range of the data values. It might be reasonable to double the number of x-axis labels to 0 through 10, but more labels than that would be very crowded. For all these axis labels, their location, including offsets from each axis, is determined just by trial and error.

It's possible to draw symbols at each data point, but the GFX library doesn't include primitives for symbols; you can draw small circles or cross marks, for example, but only by using the available graphics primitives. (I suppose you could use a + or some other printable character for a symbol.) For sketches where the line graph is drawn based on changing data, see the code for bar graphs, below.

```
/* LineGraph2.ino, D. Brooks, June 2020
  Draws line graphs with Adafruit 240x320 TFT display.
*/
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_ILI9341.h>
#define TFT_DC 9
#define TFT_CS 10
#define BLACK 0x0000
#define BLUE 0x001F
#define RED 0xF800
#define GREEN 0x07E0
#define CYAN 0x07FF
#define MAGENTA 0xF81F
#define YELLOW 0xFFE0
#define WHITE 0xFFFF
Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC);
const int xSize=320,ySize=240; // For 2.2" Adafruit TFT display.
// Define x/y graph space.
uint16_t x0=45,Y0=200,xLength=260,yLength=160;
uint16_t color,axisColor=WHITE,labelColor=WHITE;
```

```
uint16_t x0Label=x0-2,y0Label=Y0+2; // starting coordinates for x-axis
labels.
// Define axis values and labels.
String xLabels[]={"0","1","2","3","4","5"};
int nxChars=1,nxLabels=6,dx=xLength/(nxLabels-1);
int xLabelOffset=(int)(5.*nxChars/2.);
String yLabels[]={"0980","0990","1000","1010","1020","1030"};
int i,nyChars=4,nyLabels=6,dy=yLength/(nyLabels-1);
// Here are the data to graph...
float X[]={0,1.7,3.3,4.4,5.0};
float Y[]={980,1010,1030,1000,995};
int nX=5;
float Xmin=0,Xmax=5,Ymin=980,Ymax=1030,Xrange=Xmax-Xmin,Yrange=Ymax-Ymin;
float sumX=0,sumY=0;
void setup(void) {
  Serial.begin(9600);
  //for (i=0; i<nX; i++) {
    //Serial.print(X[i]);Serial.print(' ');Serial.println(Y[i]);
  //}
  // Scale the data to graphing space
  for (i=0; i<nX; i++) {
    X[i]=x0+(X[i]-Xmin)/Xrange*xLength;
    Y[i]=Y0-(Y[i]-Ymin)/Yrange*yLength;
    //Serial.print(i); Serial.print(' ');
    //Serial.print(X[i]);Serial.print(' ');Serial.println(Y[i]);
  }
  //Serial.print("Line Graph");
  tft.begin();
  tft.setRotation(3); // Puts x-axis along long edge.
  tft.fillScreen(BLACK);
  tft.setCursor(2,2); // Sets x and y coordinates for upper left-hand
                      // corner of a rectangular grid containing
                      // the character.
  tft.setTextColor(WHITE); tft.setTextSize(2);
  tft.println("Line Graph");
  // Outline graphing space.
  // "Fast" functions for drawing horizontal and vertical lines.
  tft.drawFastHLine(x0,Y0,xLength,axisColor);
  tft.drawFastHLine(x0,Y0-yLength,xLength,axisColor);
  tft.drawFastVLine(x0,Y0-yLength,yLength,axisColor);
  tft.drawFastVLine(x0+xLength,Y0-yLength,yLength,axisColor);
  tft.setTextSize(1); tft.setTextColor(labelColor);
  //tft.setCursor(x0Label,y0Label);
  for(int i=0; i<nxLabels; i++) {
    tft.setCursor(x0Label+i*dx,y0Label+3); tft.print(xLabels[i]);
  }
  for (int i=0; i<nyLabels; i++){
    tft.setCursor(x0Label-nyChars*5-2,Y0-i*dy-4); tft.print(yLabels[i]);
  }
  // Draw data.
  for (i=1; i<nX; i++) {
    tft.drawLine(X[i-1],Y[i-1],X[i],Y[i],RED);
  }
}
void loop() { }
```



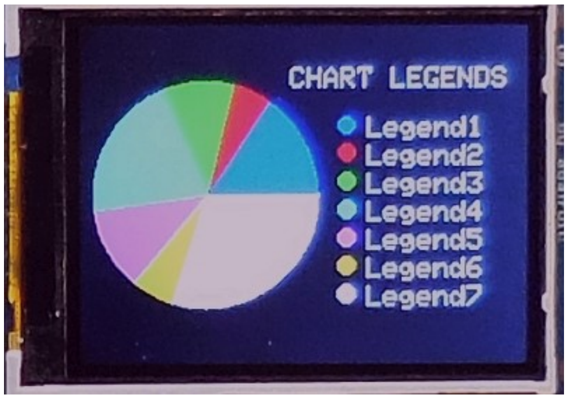
Pie charts

With graphics libraries available for some programming languages, drawing pie charts is easy because those libraries include "draw arc" and "fill arc" primitives. The GFX library has neither of these. The closest it comes is `drawTriangle()`. and `fillTriangle()`. The code below gets around this limitation by representing an entire "pie" made from 180 2° arcs as 180 filled triangles. The color used to fill the triangles changes when the "upper" angle of a pie slice crosses into the next slice as determined by scaling values to their equivalent angles. Of course this is an approximation, but for a screen this small, at 240x320 pixel resolution, the result looks like a circle, and not a series of triangles.

As a practical matter, I limited the number of pie slices to no more than 7, using predefined colors. In principle, you could increase the number of slices – you can define as many different colors as you wish – but that seemed like overkill for this small display. For drawing pie charts with changing data, see the code for bar graphs, below.

```
// pieChart.ino, D. Brooks, June 2020
#include <SPI.h>
#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#define DEG2RAD 0.0174532925
// Setup for Adafruit 2.2" TFT display.
#define TFT_DC 9
#define TFT_CS 10
Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC);
#define BLACK 0x0000 // Use black for chart background.
#define BLUE 0x001F  // Define up to 7 "standard" colors for pie slices.
#define RED 0xF800   // More than 7 slices stretches use of this small
display!
#define GREEN 0x07E0
#define CYAN 0x07FF
#define MAGENTA 0xF81F
#define YELLOW 0xFFE0
#define WHITE 0xFFFF
uint16_t colors[]={BLUE,RED,GREEN,CYAN,MAGENTA,YELLOW,WHITE};
const int xSize=320,ySize=240; // For 2.2" Adafruit TFT display.
int arraySize=8; // Define 7 data values 1-->7, with [0] set to 0.
float values[] = {0,15.4,5.2,10.9,20.4,11.5,6.6,30.1};
float angles[]={0,0,0,0,0,0,0,360}; // Will hold values coverted to angles.
// Define text for up to 7 legends.
char L1[]= "Legend1",L2[]="Legend2",L3[]="Legend3",L4[]="Legend4";
char L5[]="Legend5",L6[]="Legend6",L7[]="Legend7";
```

```
int sliceColor=0,sliceNumber=1,i;
float A,r=80,x1cos,x2cos,y1sin,y2sin,sum;
uint16_t x0=r+10,y0=ySize/2;
void setup(void) {
  //Serial.begin(9600);
  // Calculate slice boundary angles.
  sum=0.;
  for (i=0; i<arraySize; i++) {
    sum+=values[i];
    //Serial.print(values[i]); Serial.print(' ');
  }
  for (i=1; i<arraySize; i++) {
    angles[i]=angles[i-1]+values[i]/sum*360.;
    //Serial.println(angles[i]);
  }
  // Draw pie chart and its titla and legends.
  // Pixel locations determined just by trial-and-error.
  // Pie is centered in vertical middle of display.
  tft.begin();
  tft.setRotation(3);
  tft.fillScreen(0x0000);
  x1cos=r; y1sin=0; A=0; sliceColor=colors[0];
  for (i=1; i<=180; i++) {
    A+=2.;
    x2cos=r*cos(DEG2RAD*A);
    y2sin=r*sin(DEG2RAD*A);
    tft.fillTriangle(x0,y0,x0+x1cos,y0-y1sin,x0+x2cos,y0-y2sin,sliceColor);
    x1cos=x2cos; y1sin=y2sin;
    if (A>angles[sliceNumber]){
      sliceColor=colors[sliceNumber]; sliceNumber++; }
  }
  // Draw legends.
  tft.setTextSize(2);
  int y1=50,y2=45,dy=20; tft.fillCircle(190,y1+dy,6,colors[0]);
  tft.setTextColor(WHITE); tft.setCursor(205,y2+dy);
  tft.println(L1); tft.fillCircle(190,y1+2*dy,6,colors[1]);
  tft.setCursor(205,y2+2*dy); tft.println(L2);
  tft.fillCircle(190,y1+3*dy,6,colors[2]);
  tft.setCursor(205,y2+3*dy); tft.println(L3);
  tft.fillCircle(190,y1+4*dy,6,colors[3]);
  tft.setCursor(205,y2+4*dy); tft.println(L4);
  tft.fillCircle(190,y1+5*dy,6,colors[4]);
  tft.setCursor(205,y2+5*dy); tft.println(L5);
  tft.fillCircle(190,y1+6*dy,6,colors[5]);
  tft.setCursor(205,y2+6*dy); tft.println(L6);
  tft.fillCircle(190,y1+7*dy,6,colors[6]);
  tft.setCursor(205,y2+7*dy); tft.println(L7);
  tft.setCursor(150,30); tft.println("CHART LEGENDS");
}
void loop() {}
```



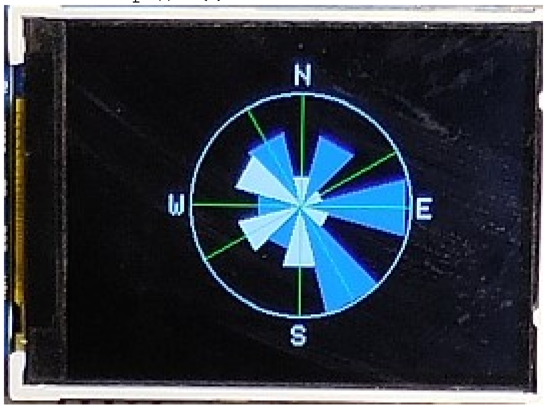
“Wind rose” graphs

This code is just a modification of the pie chart code. Wind rose graphs are often used to display meteorological data like wind speed as a function of wind direction. They can also be used to display other physical quantities whose values are direction-dependent.

I defined 12 30° “petals.” Similar to other graphs, the length of each petal is adjusted to the radius of the rose expressed in pixels. The lack of a “fill arc” routine in the GFX graphics library means that the petals are drawn as triangles rather than arcs. Alternating colors makes each petal easier to distinguish from adjacent ones. Here’s the code:

```
// WindRose.ino, D. Brooks, July 2020
#include "SPI.h"
#include "Adafruit_GFX.h"
#include "Adafruit_ILI9341.h"
#define DEG2RAD 0.0174532925
// Setup for Adafruit 2.2" TFT display.
#define TFT_DC 9
#define TFT_CS 10
Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC);
#define BLACK 0x0000 // Use black for chart background.
#define BLUE 0x001F  // These are the values for 8 "standard" colors.
#define RED 0xF800   // Other values can be defined if desired.
#define GREEN 0x07E0
#define CYAN 0x07FF
#define MAGENTA 0xF81F
#define YELLOW 0xFFE0
#define WHITE 0xFFFF
uint16_t colors[]={WHITE,BLUE,RED,GREEN,CYAN,MAGENTA,YELLOW};
const int xSize=320,ySize=240; // For 2.2" Adafruit TFT display.
int arraySize=8; // Define 7 data values 1-->7, with [0] set to 0.
float values[]={0,75,15,50, 20, 52,  47, 30, 44, 31, 43, 77, 19};
float angles[]={0,30,60,90,120,150,180,210,240,270,300,330,360};
float R,r=80,maxR=0,sin30,cos30,cos60,sin60;
// Define text for up to 7 legends.
char L1[]="N",L3[]="E",L5[]="S",L7[]="W"; // intermediate angle labels can
also be defined.
int i,nSlices=12;
float x1cos,x0cos,y1sin,y0sin;
uint16_t sliceColor,x0=xSize/2,y0=ySize/2;
```

```
void setup(void) {
  //Serial.begin(9600);
  tft.begin(); tft.setRotation(3); tft.fillScreen(0x0000);
  for (i=1; i<=nSlices; i++) {
    if (values[i]>maxR) maxR=values[i];
  }
  for (i=1; i<=nSlices; i++) {
    sliceColor=colors[i%2]; R=r*values[i]/maxR;
    x0cos=R*cos(DEG2RAD*(angles[i-1]-15));y0sin=R*sin(DEG2RAD*(angles[i-1]-
15));
    x1cos=R*cos(DEG2RAD*(angles[i]-15));y1sin=R*sin(DEG2RAD*(angles[i]-15));
    tft.fillTriangle(x0,y0,round(x0+x0cos),round(y0-
y0sin),round(x0+x1cos),round(y0-y1sin),sliceColor);
  }
  tft.drawCircle(x0,y0,r,WHITE);
  tft.setTextSize(2);
  tft.setCursor(x0-5,y0-r-20); tft.println(L1);
  tft.setCursor(x0+r+5,y0-6); tft.println(L3);
  tft.setCursor(x0-5,y0+r+7); tft.println(L5);
  tft.setCursor(x0-r-15,y0-6); tft.println(L7);
  tft.drawLine(x0,y0+r,x0,y0-r,GREEN); tft.drawLine(x0-r,y0,x0+r,y0,GREEN);
  sin30=sin(DEG2RAD*30); cos30=cos(DEG2RAD*30);
  sin60=sin(DEG2RAD*60); cos60=cos(DEG2RAD*60);
  tft.drawLine(x0-r*cos30,y0+r*sin30,x0+r*cos30,y0-r*sin30,GREEN);
  tft.drawLine(x0-r*cos60,y0-r*sin60,x0+r*cos60,y0+r*sin60,GREEN);
}
void loop() {}
```



Bar graphs

The `fillRect()` primitive makes bar graphs easy to code. In this code I've limited the bars to no more than 10. To make code easier to adapt, I've separated the tasks into three separate categories:

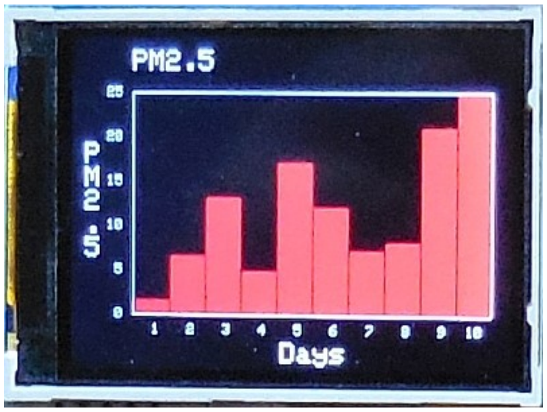
1. Outline the graphing space;
2. Draw the axis labels and titles;
3. Draw the bars.

For data that changes, you could put the first two tasks in the `setup()` function and the third data-dependent task in the `loop()` function. For this code I assumed that the value being graphed is PM2.5 airborne particulates. Perhaps these data are being read from a particulate sensor and updated at some preset interval. With an array of 10 values to be plotted, when a new value is recorded, the oldest value is removed from the array of values, the remaining values are moved

down one position, and the new value is added to the end. Only the bars need to be redrawn. However, if you do this, you first have to clear the “old” display by “erasing" each bar by overprinting it with
`tft.fillRect(X0+2+dx*i,Y0-yLength,barWidth,yLength,BLACK);`
(assuming BLACK is your background color) before writing the new bar. Finally, drawing the bars first (see the `setup()` function) allows the graph space boundaries to be drawn over the bars instead of being overwritten at the bottom and perhaps at the top by the bars. If you move the `drawBars()` function to `loop()`, then move `outLineGraphSpace()` to the end of the code in `drawBars()`.

It’s possible to draw two or even three or more bars for each x-interval, by reducing the number of intervals over the same x-axis length. You just have to supply the data and adjust the bar widths and x-offsets as appropriate. Keeping the bar-drawing code in its own function makes it easier to do this without worrying about the rest of the code.

```
/* barGraph.ino, D. Brooks, June 2020
  Draws vertical bar graphs with Adafruit 240x320 TFT.
*/
#include     // Core graphics library
#include
#define TFT_DC 9
#define TFT_CS 10
#define BLACK 0x0000
#define BLUE 0x001F
#define RED 0xF800
#define GREEN 0x07E0
#define CYAN 0x07FF
#define MAGENTA 0xF81F
#define YELLOW 0xFFE0
#define WHITE 0xFFFF
Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC);
const int xSize=320,ySize=240; // For 2.2" Adafruit TFT display.
// Define x/y graph space.
uint16_t X0=45,Y0=200,xLength=260,yLength=160;
uint16_t color,labelColor=WHITE;
float values[]={1.5,6.6,13.2,4.9,17.0,12,7.2,8.1,21,25};
float v_size=10,v_min=0, v_max=25;
void setup() {
  Serial.begin(9600);
  tft.begin(); tft.setRotation(3); tft.fillScreen(BLACK);
  drawBars(X0,Y0,values,v_size,v_min,v_max,xLength,yLength);
  outlineGraphSpace(X0,Y0,xLength,yLength); // with WHITE lines
  drawLabelsAndTitles(X0,Y0);
  // Create bars.
}
void outlineGraphSpace(uint16_t X0,uint16_t Y0,uint16_t xLength,uint16_t
yLength) {
  // Outline graphing space.
  // Use "fast" functions for drawing horizontal and vertical lines.
  // (X0,Y0) = lower left-hand corner of graph
  // xLength, yLength = length in pixels of x- and y-axes
  uint16_t axisColor=WHITE;
  tft.drawFastHLine(X0,Y0,xLength,axisColor);
  tft.drawFastHLine(X0,Y0-yLength,xLength,axisColor);
  tft.drawFastVLine(X0,Y0-yLength,yLength,axisColor);
  tft.drawFastVLine(X0+xLength,Y0-yLength,yLength,axisColor);
}
void drawLabelsAndTitles(uint16_t X0,uint16_t Y0) {
  // Define axis labels.
  String xLabels[]={" 1"," 2"," 3"," 4"," 5"," 6"," 7"," 8"," 9","10"};
  String yLabels[]={" 0"," 5","10","15","20","25"};
  // Define axis titles.
  char xTitle[]="Days",yTitle[]="PM2.5";
  int nxTitle=4, nyTitle=5; // No more than 10 characters for y title.
  int nxChars=2,nxLabels=10,dx=xLength/nxLabels;
  int nyChars=3,nyLabels=6,dy=yLength/(nyLabels-1);
  uint16_t x0Label=X0-2,y0Label=Y0+2; // starting coordinates for x-axis
labels.
  int yOffset=5; // For printing x-axis labels below x-axis.
  int xOffset=10; // Put bar label in center of its space.
  tft.setTextColor(labelColor);
  // Draw x- and y-axis labels.
  for(int i=0; i<nxLabels; i++) {
    tft.setCursor(x0Label+xOffset+i*dx,y0Label+yOffset);
tft.print(xLabels[i]);
  }
  for (int i=0; i<nyLabels; i++){
    tft.setCursor(x0Label-nyChars*5-2,Y0-i*dy-4); tft.print(yLabels[i]);
  }
  tft.setTextSize(2);
  // Center and print x-axis title.
  tft.setCursor(x0Label+xLength/2-(nxTitle*10)/2,y0Label+18);
tft.print(xTitle);
  //Center and print y-axis title vertically, one character at a time.
  int yTitleStart=35+(10-nyTitle)*17/2; // Center y title text on y-axis.
  for (int i=0; i<nyTitle; i++) {
  tft.setCursor(10,yTitleStart+i*17); tft.print(yTitle[i]);
  }
  tft.setCursor(X0,10); tft.print("PM2.5");
}
void drawBars(uint16_t X0,uint16_t Y0,float V[],int arraySize,int v_min,int
v_max,int xLength,int yLength) {
  int i,px,dx=xLength/arraySize,barWidth=25;
  for (i=0; i<arraySize; i++) {
    px=round(V[i]/v_max*yLength);
    Serial.print(V[i]);Serial.print(' ');Serial.println(px);
    //tft.fillRect(X0+2+dx*i,Y0-yLength+1,barWidth,px,BLACK);
    tft.fillRect(X0+2+dx*i,Y0-px,barWidth,px,RED);
  }
}
void loop() {}
```



E-Ink displays

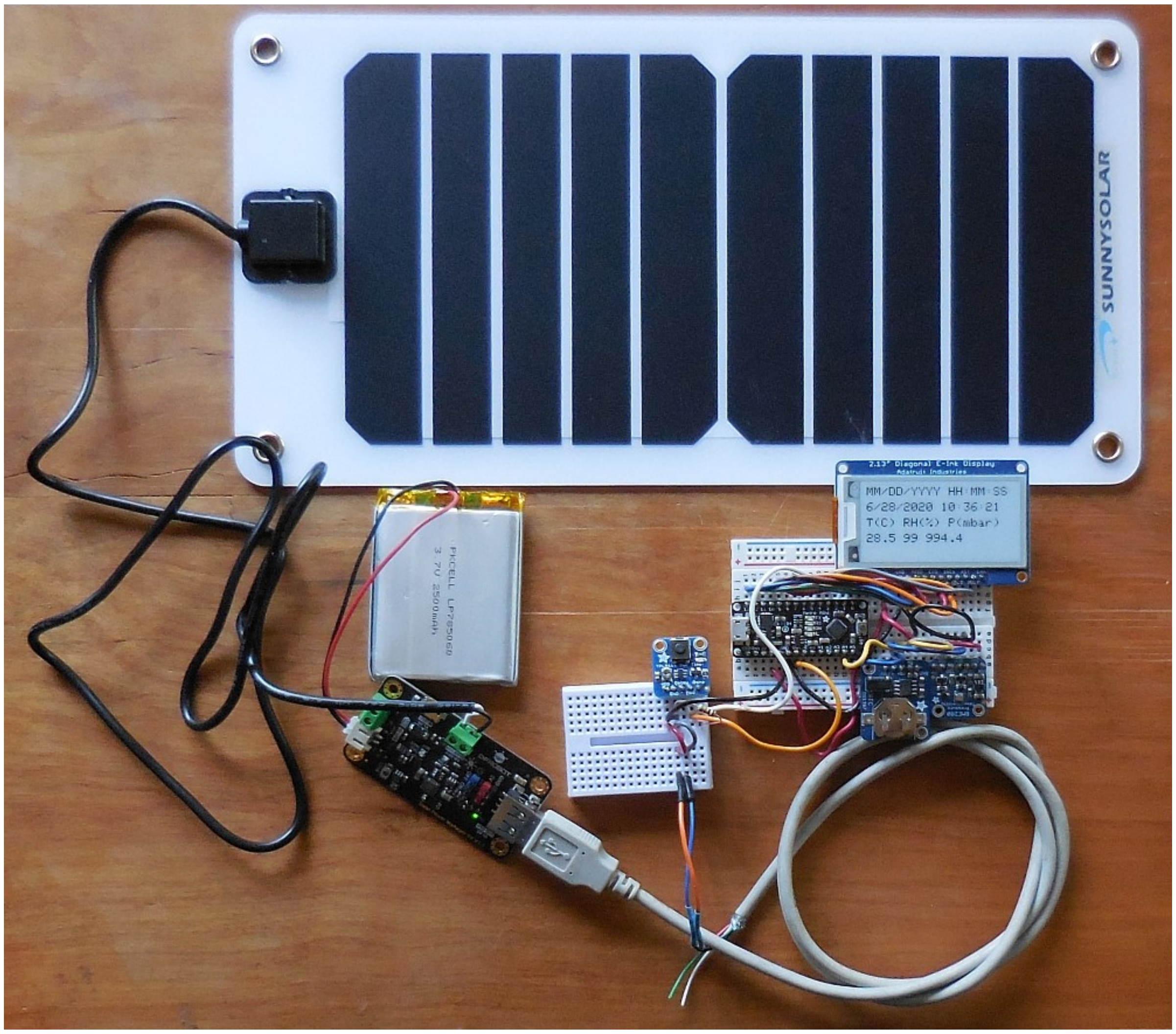
E-ink displays are familiar from e-book readers. Their huge advantage is that they are very-low power devices and once something is written to the screen, it stays there essentially indefinitely even when the power is off. Adafruit.com currently has three versions:

2.13" black on white (ID 4197)
2.13" red/black/white tri-color (ID 4086)
2.13" flexible black on white (ID4243)

In addition to displaying characters, you can also draw graphics images on this display using Adafruit's GFX library. The non-flexible displays include a microSD card for storing images. The flexible card doesn't have its own microSD card holder.

To demonstrate this device I built a complete system to take advantage of the properties of e-ink displays:

BME280 temperature/humidity/pressure module	adafruit.com ID 2652
DS 1307 real-time clock	adafruit.com ID 3296
TPL5110 power on-off timer	adafruit.com ID 3435
Solar power manager with 5 V solar panel	DFRobot DFR0559-1
Metro Mini 5 V	adafruit.com ID 2590
2.13" black on white e-ink display	adafruit.com ID 4197
2500 mAh 3.7 V LiPo battery	adafruit.com ID 328
Mini and half-size breadboards	various sources

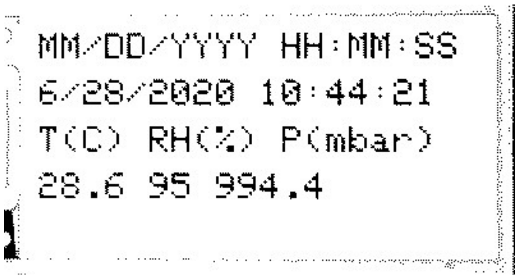


The system shown here has been running continuously for months in my office, through the winter, with the solar panel mounted inside a west-facing window in my office – not the most efficient circumstance for a solar power system but one that works perfectly well for this very low-power system. During the day the solar manager powers the system and charges the LiPo battery. At night, the battery goes through a 5 V step-up converter to keep the system running. I used a 5 V Metro Mini board from adafruit.com, but other (less expensive) boards, such as an Arduino Nano or Pro Mini would also work. The real-time clock and BME280 are I2C devices. The e-ink display is an SPI device. See the Adafruit for wiring instructions for all these devices. What's not included in the Adafruit documentation for its TPL5110 timer is a 10 kΩ resistor between the "done" pin and ground, which I and others have found necessary to ensure reliable operation of the timer. The timer delay resistor is set to produce a display update about every 3 minutes – again, see the Adafruit documentation.

The connections for this system are complicated, but the code, which uses Adafruit's GFX graphics library, is not difficult. Unlike printing characters to an LCD or TFT display screen, data to be displayed on the e-ink screen aren't written directly to the display. They are first written into a data buffer and the `display.display();` statement near the end of the code below then transfers buffer contents to the display.

```
/* BME280_eInk2.ino, D. Brooks, January 2020
  Based on Adafruit"s sample code for its 250x122 monochrome eInk display,
  ID 4197, displays date/time and T/RH/P values from a BME280 sensor.
  T in deg C, RH in %, P/100 is station pressure at site elevation, in mbar.
  Power provided by Adafruit TPL5110 low power timer board.
*/
#include "Adafruit_GFX.h"  // Adafruit Core graphics library
#include "Adafruit_EPD.h"
#include "Wire.h"
#include "RTClib.h"
RTC_DS1307 RTC;
#include "Adafruit_Sensor.h"
#include "Adafruit_BME280.h"
Adafruit_BME280 bme; // I2C
float T,P,RH;
#define EPD_CS     10
#define EPD_DC      9
#define SRAM_CS     8
#define EPD_RESET   5 // can set to -1 and share with microcontroller Reset!
#define EPD_BUSY    3 // can set to -1 to not use a pin (will wait a fixed
delay)
/* using 2.13" monochrome 250*122 EPD */
Adafruit_SSD1675 display(250, 122, EPD_DC, EPD_RESET, EPD_CS, SRAM_CS,
EPD_BUSY);
const byte donePIN=4;
void setup(void) {
  pinMode(donePIN,OUTPUT); digitalWrite(donePIN,LOW); // power on
  Serial.begin(9600);
  if (!RTC.begin()) {
    Serial.println("RTC not running."); exit(0);
  }
  else Serial.println("RTC running.");
  Serial.println(F("BME280 test"));
  if (!bme.begin()) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    exit(0);
  }
  else Serial.println("Found BME280 sensor.");
  display.begin();
  Serial.println("Initialized");
  DateTime now=RTC.now();
  display.clearBuffer(); display.fillScreen(EPD_WHITE);
  display.setTextColor(EPD_BLACK); display.setTextSize(2);
  display.setCursor(5,5);
  display.print(F("MM/DD/YYYY HH:MM:SS "));
  display.setCursor(5,30);
  display.print(now.month()); display.print('/');
  display.print(now.day());   display.print('/');
  display.print(now.year());  display.print(' ');
  display.print(now.hour());  display.print(':');
  display.print(now.minute());display.print(':');
  display.print(now.second());
  display.setCursor(5,55);
  display.print(F("T(C) RH(%) P(mbar)"));
  display.setCursor(5,80);
  display.print(bme.readTemperature(),1); display.print(' ');
  display.print(bme.readHumidity(),0); display.print(' ');
  display.print(bme.readPressure()/100.,1);
  display.display(); // writes from buffer to display
  delay(1000); digitalWrite(donePIN,HIGH); // turn off power
}
void loop() {
}
```

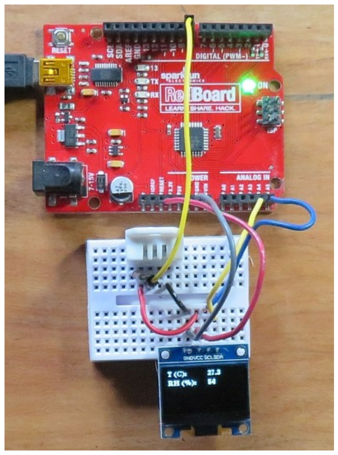
This image looked too "blue-ish" when I took it, so I saved it as a black-and-white image. In fact the text on this display really is black against a nearly "paper white" background. Unlike some other kinds of displays, this e-link display is easily readable even in bright sunlight.



```
MM/DD/YYYY HH:MM:SS
6/28/2020 10:44:21
T(C) RH(%) P(mbar)
28.6 95 994.4
```

Another display (August 2020)…

Since writing this document in June I bought some small (0.96" diagonal) 128x64 pixel OLED displays. These are low-power devices that are useful for displaying simple data from sensors in real time. You can find the document describing that project at http://www.instesre.org/ArduinoBook/OLED.htm.



Some final thoughts…

As noted earlier in this document, I have sometimes experienced unexplained compile and upload problems with sketches that use the GFX library, but eventually I was able to get all the sketches included here working. I have also noted the unfortunate absence of GFX primitives to draw and fill arcs – parts of a circle.

Many of the projects shown in this document use hardware from adafruit.com. I regularly use Adafruit products because I have found them to be reliable and extremely well documented, but other sources can also be found – sparkfun.com, for example, is another reliable US-based supplier.

Many Arduino-compatible products come from “off-shore” manufacturers, China in particular, even when they are sold by US vendors. US vendors should stand by their products regardless of their source, but buying such products online directly from off-shore sources can be a risky proposition. In some cases I have not had any problems and in other cases I have had shipping time and quality control issues. Caveat Emptor!

Comments and questions? Please feel free to contact me at brooksdr@instesre.org. To learn more about using Arduinos to collect, record, and display environmental data, see http://www.instesre.org/ArduinoBook/ArduinoBook.htm.

[illegible]

David R. Brooks, PhD
June, 2020