# Course on Internet of Things

Exercises Session 8: TFT Displays

## Introduction

A wide range of displays is available for use with the Arduino system. For the WeMos D1 mini system there are 3 display boards. Each of them is based a on different controller chip and has different resolution. Here they are:

- 64x48 pixel 0.66" TFT screen with SSD1306
- controller. The SSD1306 uses the I2C bus 128x128 pixel 1.4" display based on the ST7735 controller. The ST7735 uses the SPI bus 320x240 pixel 2.4" display based on the ILI9341 display and the XPT2046 touch controller.

| 64x48 pixels with SSD1306 | 128x128 with ST7735 | 320x240 pixel touch screen with ILI9341 |
|---|---|---|
|  |  |  |

Each of them has its advantages and disadvantages. The 0.66" display is really tiny and its resolution is very limited allowing to write just a few characters. In addition, it is monochrome only. However, the reference display driver included in MicroPython uses this controller chip. There are also higher resolution and color screens using this controller but I did not find one with the WeMos D1 mini bus interface.

The 1.4" display is already quite a bit better even though 128x128 bit resolution is still not enough for sophisticated graphics. It is enough however to write a few lines of text and produce simple graphics, e.g. creating a graph of sensor readout results.

The 2.4" display is clearly the most powerful device, adding touch functionality. It is however so big that it is difficult to plug it onto the WeMos D1 bus, even if the connector is available. The better solution is an adapter board and a connecting cable. This display board also features a touch screen and is one of the boards supported by the lvgl Graphical User Interface (GUI) library.

Since the 1.4" display is enough for most our applications, it can be nicely plugged onto the bus just like any other sensor or actuator shield and it is substantially cheaper than the 2.4" solution, this is the device we chose for the course.

## Preparing the hardware

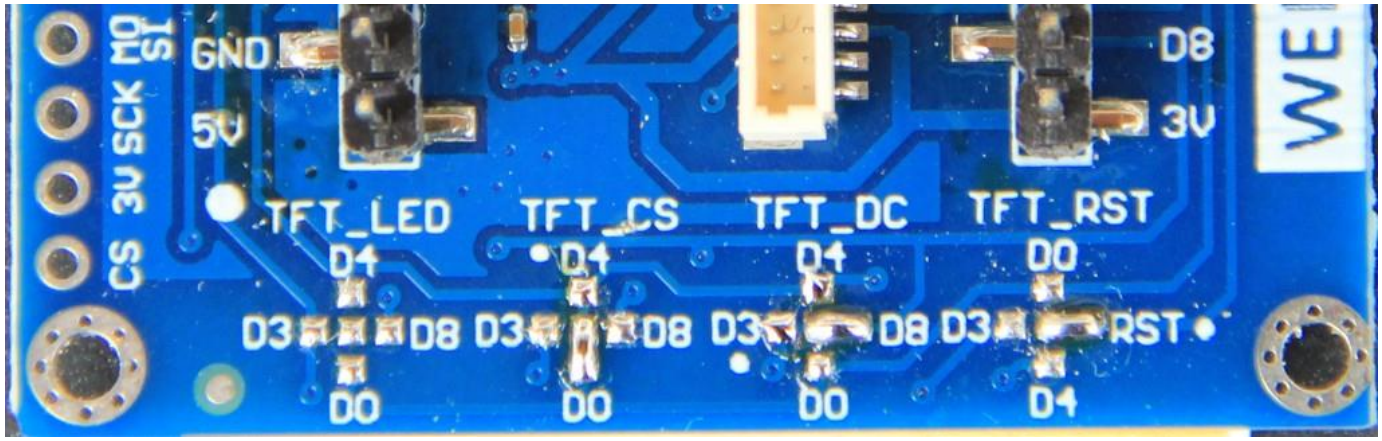
Before using the display its solder jumpers on the back must be configured. There are 4 solder fields for configuration

- TFT LED. his allows to control the display back light. I leave it unconfigured which results in the back-light LED being permanently on
- TFT_CS: D0. TFT_CS and TFT_DC can be selected onto D0, D3, D4 or D8 corresponding to GPIO 26, 17,16, 5. Since D3 and D4 cannot be used on the CPU with PSRAM the only selections possible are D0 and D8. I select D0 for CS and D8 for TFT_DC

  TFT_DC: D8

- TFT_RST is connected to the system reset

  Here is a photo of the jumper settings:



## The driver software

Searching the Internet I found many display drivers, none of which really satisfied me.

- The ssd1306 reference driver included in MicroPython. This driver is very basic and it relies very heavily on its framebuffer superclass .

- A pure Python version of a ST7735 driver for the ESP32: https://github.com/boochow/MicroPython-ST7735. This a port of a ST7735 driver written by Guy Carver for stm32 processors (https://github.com/GuyCarver/MicroPython/blob/master/lib/ST7735.py ). At first glance, this is exactly what I was looking for. However, Guy Carver himself was not too happy about its speed and, compared to other drivers, it has limited functionality.
  - There is a demo program putting images up onto the screen and another one exercising the driver's graphic functions like
    - drawing line
    - drawing rectangles
    - drawing filled rectangles
    - drawing text

A C-module of the above driver also written by Guy. Finding this code was not so easy. It is hidden in a fork of MicroPython, made some years back, and it was again developed for stm32 processors. Since the method of integrating C-modules into MicroPython has been made substantially easier in recent versions of MicroPython quite a bit of work was needed to adapt this driver. When trying it, it was even slower than the Python code! I figured out that the reason was the way the SPI commands are sent to the controller. If you send commands and data in large chunks then you can speed up graphics considerably. This however is true for the Python and the C code.

A driver for displays based on the SSD1351 controller: https://github.com/rdagger/micropython-ssd1351
The display module used has 128x128 resolution just like the module we have. The driver comes with a big number of demo programs, some of which are rather sophisticated. It also provides many more functions than the ST7735 driver. A very nice tutorial with Youtube video is also available. Unfortunately loading fonts is extremely slow.
I decided to try porting this library to the St7735 chip in such a way that the demo programs would run with a minimum of modification. Speeding up font loading and a port of the demo programs from the above ST7735 driver should be part of the process.

## Developing our own driver

The first goal for our own driver was to be able to run all the demo programs on it. Since the calling sequences of the drivers are different (different method names, different way to pass parameters) some modifications to the demo code was of course necessary. In addition, the method names and parameters defined in the framebuffer should be available.

Comparing the original ST7735 driver to the SSD1351 one we see the following differences

| ST7735 | SSD1351 |
|---|---|
| clips drawing elements when they exceed display boundaries | drops drawing elements if any pixel would exceed display boundaries |
| Provides 3 fonts as Python files | Provides many different fonts as C files<br>These fonts must be loaded before use, which takes a long time |
| uses .bmp files for images | uses binary rgb565 files for images |
| does not support sprites | allows use of sprites |
| Comes with 2 demo programs for graphics primitives and image display | Comes with man demo programs, some of which are rather sophisticated Even supplies simple Arcade games |

Finally, I decided to come up with a mixture of both because I prefer drawing element clamping fro just dropping the element. In addition to the original naming convention of the methods additional names were introduced following the convention in the framebuffer driver.

## Fonts

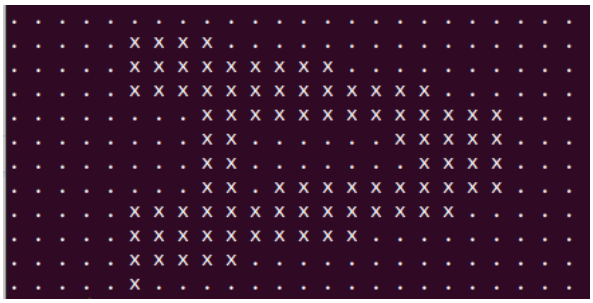
Fonts are normally loaded into RAM before use. However, this takes time and uses a big amount of RAM memory, a very scarce resource on the standard ESP32. (We are using an ESP32 with additional PSRAM where the problem is much less critical).

Since font loading in the SSD1351 driver takes ages (the demo-font program using 9 different fonts took ~ 30s before the first character could be written. After that text writing was fast) I searched for another method and ended up asking the MicroPython forum. Peter Hinch sent me a link to his micropython-font-to-py git repository explaining how the font data can be kept in flash and you can nevertheless get fast access to the them. The fonts are implemented as Python source files providing not only the font data but also access routines to the pixel data. These font files can be frozen into MicroPython and are thus stored in flash. When a character is written, only the pixel data for this character is loaded into RAM.

The problem was now how to convert the font files in C supplied with the SSD1351 driver to Python font files compatible with micropython-font-to-py. I wrote a Python script:

https://afnog.iotworkshop.africa/pub/UCC_Course_2021/TFTDisplay/convertFont.py.txt

doing exactly this. The output is actually executable (on the PC) and will print the pixel data of the letter "A" for this font.



Finally I copied the fonts into the modules/fonts directory of the

MicroPython sources and froze them into the binary. You can use the

fonts by importing the MicroPython module:

```
from ST7735 import Display, color565 import fonts/sysfont as sysfont sck = Pin(18)
miso= Pin(19)
mosi= Pin(23)
SPI_CS = 26
SPI_DC = 5
spi = SPI(2, baudrate=32000000, sck=sck, mosi=mosi, miso=miso)
display=Display(spi,SPI_CS,SPI_DC)
display.draw_text(0, 0, 'Hello World!', sysfont, color565(255, 0, 0))
```

In addition to font handling I received a link to micropython-nano-gui a lightweight and minimal MicroPython GUI library for display drivers based on the framebuf class. This library contains a few widgets to display sensor data and includes a plot library.

## The frame buffer

There are two different approaches to the display driver:

- Every graphics primitive results in a direct write to the display controller
- All graphics primitives write to frame buffer memory. When all drawing is done the frame buffer memory as a whole is copied to the hardware

The SSD1351 driver uses the first approach while the reference driver for the SSD1306 in MicroPython uses the second method. Peter Hinch's libraries depend on the underlying framebuf class.

## The goal

I now had:

a simple demo program I wrote for the SSD1306 module

2 demo programs running on the original ST7735 driver

a bunch of demo programs from the SSD1351 driver

- Peter Hinch's library for font handling, including his writer to display the fonts, and his *nanogui* library

including the *fplot* plot library

I set myself the goal to write a driver which was capable of running all of these demo programs.

I ended up writing a class with methods implementing both design principles. There is a method

- *draw_line(x0,y0,x11,y1,color) drawing* the line directly on the hardware
- and a framebuf compatible method *line(x0,y0,x11,y1,color)* which draws the line in the frame buffer memory

When using the frame buffer method you have to call the *show()* method to copy the

framebuf memory to the hardware to make the drawing visible.

Here are the methods:

These first methods write to the hardware directly

- size(): returns width and height of the display as a tuple
- clear(): clears the display
- display_on(), display_off() switches the display on or off
- cleanup(): clears the display, switches it of and deinits the SPI bus
- draw_pixel(x,y,pixel): draw a single pixel
- draw_hline(x,y,w,color): draw a horizontal line of length w, starting at the point x,y
- draw_vline(x,y,w,color): you guess!
- draw_line(x0,y0,x1,y1,color): draw a line from point x0,y0 to point x1,y1
- draw_lines(coords,color): draw lines where the point coordinates are given as a list of coordinate pairs
- draw_rectangle(x,y,w,h,color): draw a rectangle of width w and height h starting at point x,y
- draw_polygon(sides,x0,y0,r,color,rotate=0)
- draw_circle(x0,y0,r,color): draw a circle of radius r with center point x0,y0
- draw_ellipse(x0,y0,a,b.color): draw an ellipse with center point x0,y0 and half axis a,b
- draw_filledCircle(x0, y0, r, color): same as draw:circle but the circle is filled
- draw_filledEllipse(x0, y0, a, b, color)
- draw_filledPolygon(sides, x0,y0, r, color, rotate=0)
- draw_letter(self, x, y, letter, font, color, background=0,landscape=False): draw the glyph for *letter* at position x,y using the font *font.* If landscape is false the letter is drawn left to right, if landscape is true, the letter is rotated counter clock wise by 90° and is drawn from bottom to top

- draw_text(self, x, y, text, font, color, background=0,landscape=False, spacing=1, nowrap = False): I think you can guess! If nowrap is true and the text is larger than the length of a text line it is wrapped to the next line.
- draw_image(path,x0,y0,w=128, h=128): draw an image from a rgb565 raw image file to the display
- load_sprite(path, w, h): load a sprite of width w and height h from a rgb565 raw image file
- draw_sprite(buf, x, y, w, h): draw the sprite, which was previously loaded into buf

and these pass through the framebuffer:

- **fill(color): sets the entire frame buffer to color**
- **fill(0): corresponds to clear() as all pixels in the frame buffer are set to 0 (BLACK)**
- **pixel(x,y,pixel): draw a single pixel**
- **hline(x,y,w,color): draw a horizontal line of length w, starting at the point x,y**
- **vline(x,y,w,color): you guess!**
- **line(x0, y0, x1, y1, color): draw a line from point x0, y0 to point x1, y1**
- **lines(coords,color): draw lines where the point coordinates are given as a list of coordinate pairs**
- **rect(x,y,w,h,color): draw a rectangle of width w and height h starting at point x,y**
- **polygon(sides,x0,y0,r,color,rotate=0)**
- **circle(x0,y0,r,color): draw a circle of radius r with center point x0,y0**
- **ellipse(x0,y0,a,b.color): draw an ellipse with center point x0,y0 and half axis a,b**
- **fill_circle(x0, y0, r, color): same as draw:circle but the circle is filled**
- **fill_ellipse(x0, y0, a, b, color)**
- **fill_polygon(sides, x0, y0, r, color, rotate=0)**
- **letter(self, x, y, letter, font, color, background=0,landscape=False): draw the glyph for *letter* at position x,y using the font *font.* If landscape is false the letter is drawn left to right, if landscape is true, the letter is rotated counter clock wise by 90° and is drawn from bottom to top**
- **text(self, x, y, text, color, font=sysfont, background=0, landscape=False, spacing=1, nowrap = False): Please note the different parameter sequence as compared to draw_text**
- **image(path,x0,y0,w=128, h=128): draw an image from a rgb565 raw image file to the display**
- **sprite(buf, x, y, w, h): draw the sprite, which was previously loaded into buf**
- **show(): copies the frame buffer data to the hardware**

## The demos

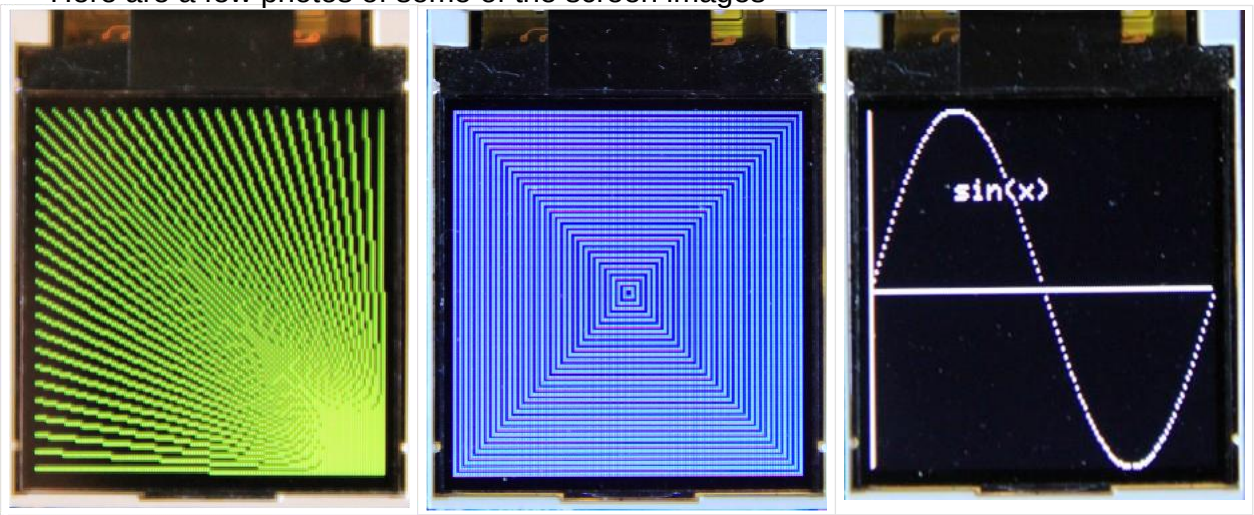
From the tests with the SSD1306 driver the demo_ssd1306_fb.py: This program demonstrates

- the drawing of lines rectangles
- filled rectangles
- circles
- the plot of a sine function with a text

- demo_ssd1306.py has the same functionality but instead of passing

through the frame buffer it writes to the hardware directly

Here are a few photos of some of the screen images



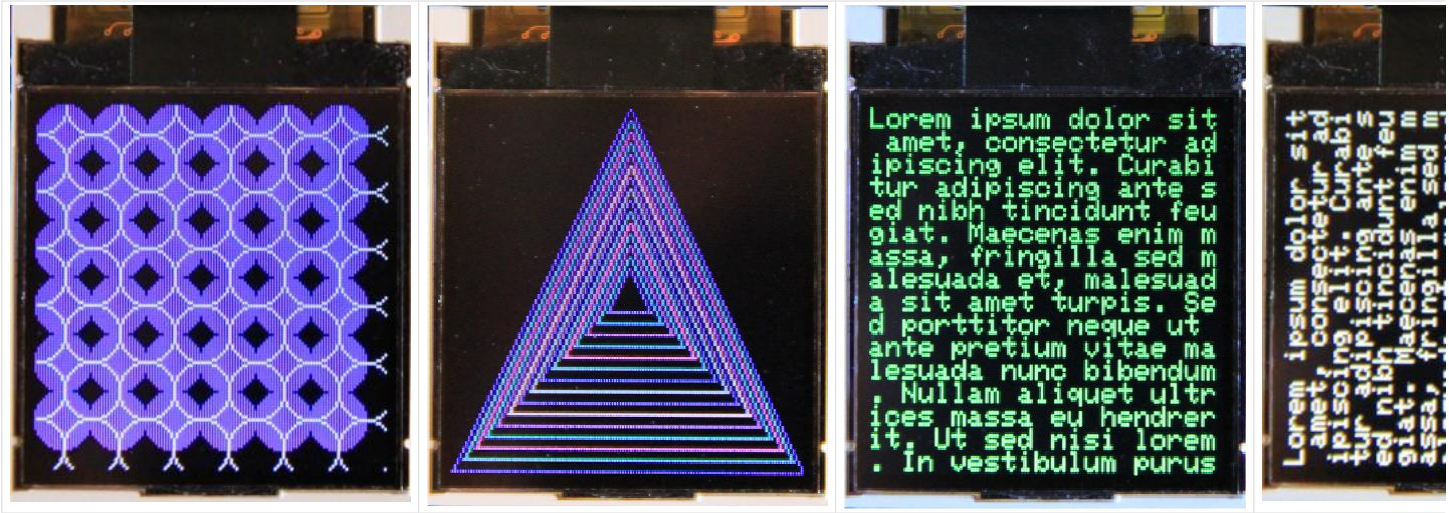

The next demo program originated from the original ST7735

- driver. It again comes in two versions: graphicstest.py and
- graphicstest_fb.py

This program has a text writing demo in addition, where a long text is displayed, wrapped to the next line once the end of the line is reached. The text is displayed in portrait mode in green font color and in white when showing off text drawing in landscape mode.

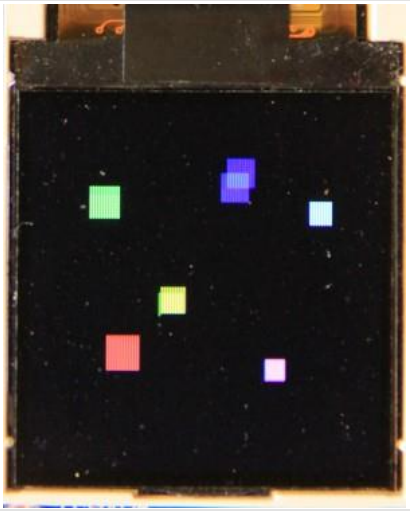
This program is quite similar in nature as demo-ssd1306.py but produces slightly different graphics and it integrates a text writing demo in addition, where a long text is displayed, wrapped to the next line once the end of the line is reached. The text is displayed in portrait mode in green font color and in white when showing off text drawing in landscape mode.

A few photos here




Then we have all the demo programs that came with the SSD1351 driver, some of which are quite sophisticated.

- demo_bouncing boxes.py: shows a number of differently colored filled rectangles that bump of the the edges of the screens
- demo_colored_squares.py: shows a number a filled rectangle in different colors
- demo_color_palette.py:

| Bouncing Boxes | Color squares | Color palette |
|---|---|---|
|  |  |  |

As explained above, text writing in different fonts is not exactly easy. While the original SSD1351 driver loaded its fonts from C files before using them, a procedure which could take several tens of seconds, the new version uses the same fonts but converted into Python code and stored in flash together with access methods to get the pixel information. Now, no font loading is necessary anymore and text drawing becomes very fast. The text drawing demo

- demo_fonts.py

only needed minor modification to reflect this change. However, a utility script to convert the original C font to a Python font is needed.

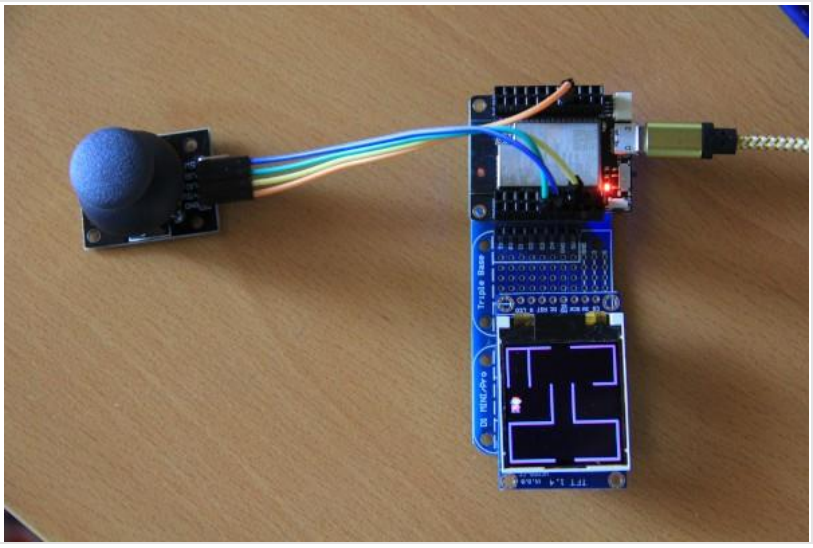

You can find the source code of the demos on the github repository for the course.

https://github.com/uraich/IoT-Course/tree/master/modules/display/demos

## Video games

In the list of demos for the SSD1351 driver you even find 2 simple video games:

Both games need additional hardware: a joystick in case of Super Mario and a linear potentiometer in the case of Arkanoid. demo-mario orriginally used a Bluetooth based joystick, which we don't have. I therefore replaced it with the analogue joystick from our sensor kit. Of course, the software needed to be adapted correspondingly.

| demo-mario | arkanoid |
| --- | --- |
|  |   |

## GUI elements in nanogui

Peter Hinch's nanogui library contains a few widgets that can be used to graphically display sensor results. Their number is fairly limited (that is why Peter calls it **nano**gui) but extensions are possible if needed. nanogui contains:

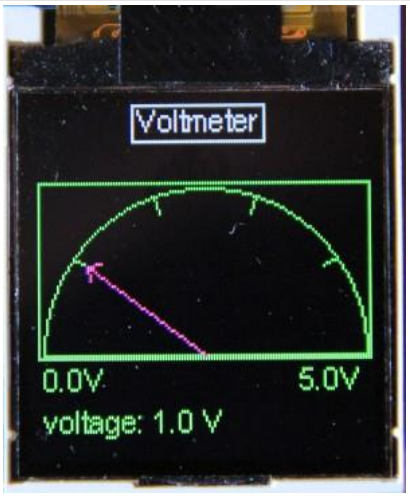
- A label
- A meter
- An Led
- A dial

Peter’s demo programs:

- color15.py
- color95.py
- alclock.py

to work with the ST7735 driver. I did not try alevel.py, written for the pyboard and making use of its accelerometer, which is not available on our ESP32 board. The color95.py program showing the meters, was adapted to present temperature and humidity values from SHT30 measurements and I created a "Scale" widget to be used as a voltmeter in conjunction with the ESP32 ADC.Particularly interesting is also the fplot.py library allowing to plot data in cartesian or polar coordinate and which has the facility to take data in real time as they are produced by the sensor. For more details please refer to https://github.com/peterhinch/micropython-nano-gui

Here again some pictures

| Clock | SHT30 meters | Voltmeter |
| --- | --- | --- |
|  |  |  |



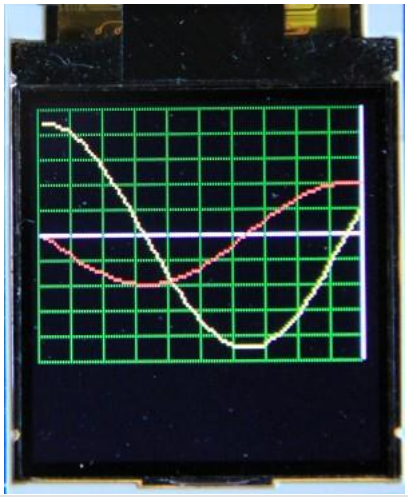
The clock is Peter's original clock program adapted to the ST7735. The SHT30 meters are meter widgets from nanogui connected to real measurements from the SHT30. Yes, also in Europe we can have Ghanaian temperatures! even though relative humidity is substantially lower.

The voltmeter meter application uses a Scale widget, not part of nanogui. Peter has supplied a superclass DObject (displayable object) from which the Scale widget is derived. The demo connects to the ESP32 ADC to display measured signal levels.

## Plotting

Peter's nanogui repository also contains a plotting package called fplot. This package allows to produce cartesian and polar plots to be generated. Particularly interesting is a feature that allows to add real time measurements to plots as they come in.

Again a few examples:

| Cartesian Plot | Polar Plot | Damped Oscillator |
|---|---|---|
|  |  |   |

-- Uli Raich - 2020-07-04