BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

INSTITUTE OF MATHEMATICS
ÚSTAV MATEMATIKY

DEMONSTRATION OF CONTROLLING TFT DISPLAY USING WIFI WITH ESP32

DEMONSTRACE OVLÁDÁNÍ TFT DISPLEJE PŘES WIFI POMOCÍ ESP32

AUTHOR **DENIS FEKETE**
AUTOR PRÁCE

BRNO 2025

Contents

Chapter 1

Introduction and theory

1.1 About

In this project I (as an author) will demonstrate connecting the ESP32 development kit board with the TFT display using SPI serial communication and also setting up the ESP32 board for it to communicate through WiFi using the UDP transport protocol. For managing ESP32 board libraries framework PlatformIO was used and for demonstrating a WiFi functionality demo Qt application and unix command netcat will be used.

1.1.1 ESP32

ESP32 is a low-cost and low-power SOC (system on chip) with 32-bit architecture that is commonly used in low-power projects. It has built-in functionalities such as WiFi, Bluetooth, SPI and a lot more features. It could be considered a competitor to other boards such as Arduino or Raspberry Pi Pico. In this project a model from Wemos was used, to be more precise, Wemos D1 R32 a ESP32 development kit board.

1.1.2 TFT Display ILI9163C

ILI9163C is a single-chip driver with integrated TFT LCD display. It is a low-power and low-cost part that was used in this project to demonstrate serial communication with ESP32 board.
laskakit.cz, Retrieved from: *https://www.laskakit.cz/128x128-barevny-lcd-tft-displej-1-44--v1-1--spi/*

1.1.3 SPI

Serial Peripheral Interface is a standard used for communication between devices over short-distance and wired communication primarily between embedded systems. It four pins (physical wired/solid metal pins that can be connected) to communicate:

- CS (Chip select) - for selecting chips, in case we were using more than one display this pin would be essential
- SCLK (Serial Clock) - transmits clock signal between communication participants
- MOSI (master out, slave in) - transmits data from master (controlling board) to slave (board being controlled, in this case TFT display)

- MISO (master in, slave out) - transmits data from slave to master, not used in this project

1.1.4 WiFi

WiFi is an standard for network communication over wireless media. In this project, it is mainly used for simple communication between user and ESP32 board. Thanks to the big community behind ESP32 frameworks and libraries for working with this technology makes it simple and straightforward.

1.1.5 UDP

User Datagram Protocol is the transport layer protocol for communicating in networks. It works by simply sending data to another station that can be identified by its IP address and port (IP is not a transport layer protocol). UDP is connectionless, meaning that no „handshaking“ for establishing communication is needed, which saves bandwidth and makes process slightly more straightforward, however, it also means that if some packets do not arrive or gets damaged, it is our job as programmers to deal with this situation.

1.1.6 PlatformIO

PlatformIO is a cross-platform, cross-architecture, multiple framework, professional tool for embedded systems engineers and for software developers who write applications for embedded products.
Platformio.org, All information about TFT display and its functions retrieved from: https://docs.platformio.org/en/latest/what-is-platformio.html

1.1.7 Qt

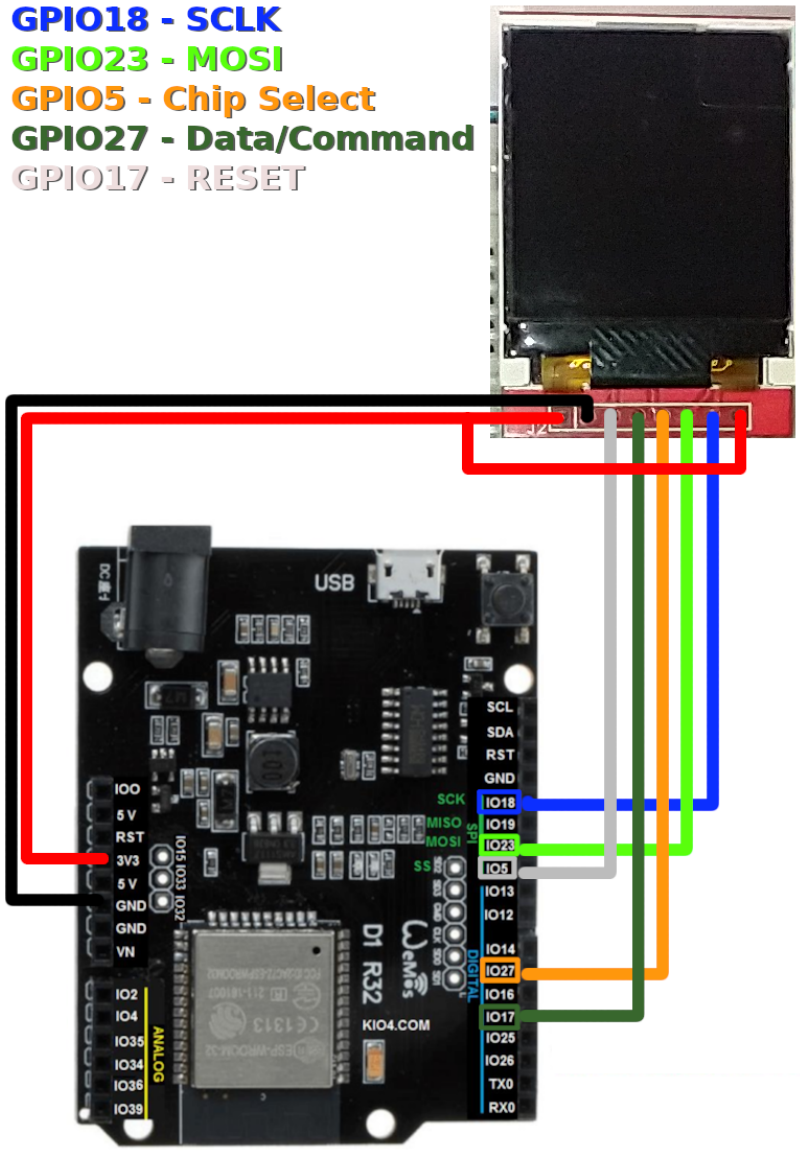
Qt is an open-source and cross-platform framework of C++ libraries that make creating graphical user interfaces easier than doing it using more barebones frameworks like OpenGl etc. Instead of Qt any framework of choice could be used; however, as an author, I have a past experience with Qt and I decided that this framework will be suitable for creating a simple demo application.

However, it should be noted that this project is not about Qt and code for implementing an application that can communicate ESP32 board will be on higher abstract level.

Chapter 2

Physical layout

2.1 Scheme



- DC (Data/Command) - used for selecting whenever a command or data will be transmitted over SPI
- CS (Chip select) - used for selecting chips, in case we were using more than one display this pin would be essential
- SCLK (Serial Clock) - transmits clock signal between communication participants
- MOSI (master out, slave in) - transmits data from master (controlling board) to slave (board being controlled, in this case TFT display)

fit.vut.cz, Image retrieved from: https://www.fit.vut.cz/person/simekv/public/IMP_projekt_board_ESP32_Wemos_D1_R32.pdf

Chapter 3

Program

3.1 Main application

Files: *main.c project_setup.h*

The main application is mostly just for initializing tasks that will handle communication with WiFi and communication with display. Because ESP32 has two cores, we can use this to our advantage and have one core perform one „task“. Tasks are mechanisms that we might know from PCs, threads. However, here we can attach one task to a specific core, this functionality is implemented by freeRTOS (library for ESP32). Since we are working with parallelism, we should ensure that data are being correctly guarded from consequent reading/writing, therefore, we will use Mutex to guard data that will be used by both tasks.

3.2 Drivers for display

Files: *driver.c driver.h*

3.2.1 Display initialization

Display driver drivers are simply said to be an implementation of SPI communication between ESP32 and display. Information as to how to use it, what codes and format to use can be found in displays datasheet. First, we need to change the display behavior with TFT_DC pin (A0 on the PCB). This pin controls whenever data that are being sent over serial communication are Commands or Data (commands are when pin is on 0).

Initializing is done by setting SPI and GPIO (general purpose input output) pins to correct values, we need to follow the scheme that we used in when we were physically connecting boards. After that we set up display settings like color mode (more on this later), orientation of display, etc. Process is always like this, we switch to Command sending mode (DC pin to low or zero) and send correct command over SPI, after that we can send additional info with data transfer (setting DC to high or 3,3V).

3.2.2 Display color

One of the more interesting setup configurations for display are color mode, ILI9163C is an 18-bit display, meaning it can display colors with an 18-bit depth, however I chose to

use 16-bit instead (one of possible configurations). This has lots of benifits, first would be less data needed for a matrix (2D array) that would represent each pixel in the program (more on this later). The second big factor is alignment, since 16 bits are exactly 2 bytes it will be a bit easier to align data and overall work with them.

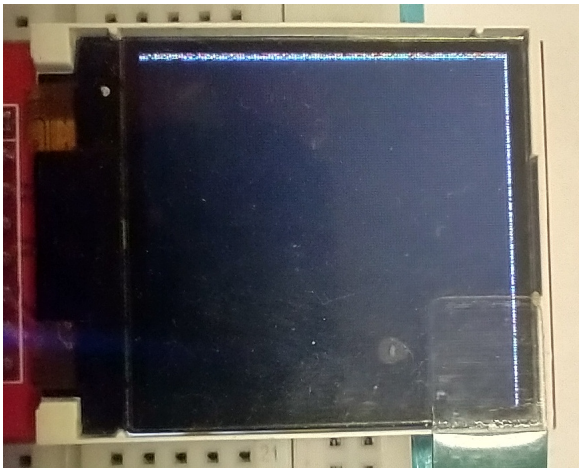
16-bit means that we have only two bytes to store three different colors (Red, Green and Blue), which will make overall color of each pixel on the display. The recommended and supported way of doing this is RGB565, which means that to store red and blue five bits will be used, and for green a six bits will be used. More in-depth information how exactly this works can be found in Chapter 6.8 of display datasheet.

3.2.3 Display bitmap

It would be convenient if we knew about the state of our display (remember, display cannot communicate back to the ESP32), therefore I used a matrix (2D array) to store the information about display; this bitmap would then be sent over to the display over serial communication and correct colors and patterns will be displayed on it.

$$width \cdot height \cdot pixel\_size \cdot color\_depth = 128 \cdot 128 \cdot 2B = 32768B = 32KiB$$

This means that we need 32KiB of memory space for our display matrix. We are keeping track of this size because we are working on an embedded system and memory is not „sky high“ like on modern day PCs. However, there is a small problem, if we were to create a project with just 128x128 pixels we would see buggy edges around the the display.



I did not study the reason for this, however, after some experiments I came to conclusion that when I change the size of display to 131*129 (width * height) the issue seemed to resolve itself. This makes me believe that the display probably has more pixels than it advertises. This is not a bad thing, and we could use these extra pixels, so let us change our matrix size equation.

$$width \cdot height \cdot pixel\_size \cdot color\_depth = 131 \cdot 129 \cdot 2B = 33798B$$

Our overall size of the matrix will then be roughly 33KiB of data.

3.3 WiFi

Files: *wifi_task.c wifi_task.h*

This project used one of Espressif IDF examples as a backbone, there I do not claim ownership over these files, I just modified them slightly to match my use-case.

3.4 UDP server

Files: *udp_server.c udp_server.h*

This file contains a simple loop that will keep cycling until end of the existence of the programs. In loop the program is waiting for a UDP communication on port „50001“, once a message is detected, it will be processed.

3.4.1 Custom communication protocol

In order for our ESP32 board to communicate with other devices through WiFi we need to use some protocol, we could use a HTTP protocol and host a simple „website“ that could server a simple GUI in the form of HTML file, however, I chose a different method, build custom protocol on top of UDP protocol. This might be more inconvenient, however, we will have less data overhead and more control code.

Therefore, I chose the most simple protocol that I could come up with. Each command starts with command code, | Data | represents one byte of data, [DATA1 | DATA2] is an array of pairs where two values are repeating, if it was [DATA1 | DATA2 | DATA3] array of three repeating values etc.

3.4.2 Confirm - 0x00

Confirmation of received message, UDP server will send confirmation of all received messages

3.4.3 Fill - 0x11

Fills the whole display with the provided color, `0x11` followed by `Red`, `Green`, `Blue`, each of colors is one byte long and represents color value.

Example of a byte array:

```
0x11, 0xFF, 0x00, 0x00 - set display to bright red color
```

„ , “ represents one byte

3.4.4 Set row - 0x2x

Sets values of given row. `0x2x` followed by `Row` which will be affected and array of values color values, values change based on command:

- 0x20 (Set row): array is made of `Red`, `Green`, `Blue`, each element has 1 byte, array is of size display width

- 0x21 (Set row normalized) : same as 0x21 but values are now expected to be normalized, meanaing that red and blue values wont be bigger than 31 and green wont be bigger than 63
- 0x22 (Set row normalized and compressed): array is now made of `Byte0` and `Byte1` which are three colors compressed into RGB565 format

Example of a byte array:

```
0x23, 0x02, [0x00, 0x00] ... - sets 3rd row to black color, where ... is meant
as continuation of [0x00, 0x00] bytes in length of display width
```

„ , “ represents one byte

3.4.5 Set pixel - 0x3x

Sets the value of given pixel. `0x3x` followed by `Row` and `Col` (position of the pixel) which will be affected and `Red`, `Green` and `Blue` color values

- 0x30 (Set pixel): color values are expected to be one byte long each
- 0x31 (Set pixel normalized) : color values are replaced with `Byte0` and `Byte1` which are in compressed RGB565 format

Example of a byte array:

```
0x31, 0x02, 0x03, [0xFF, 0xFF] - sets pixel in 3rd row and 4th column
to white color
```

„ , “ represents one byte

3.4.6 Set vector - 0x40

Sets pixels that are to its colors, it is an extension of Set pixel command focused on performance. `0x40` followed by `Number_of_pixels` which is a number of pixels that will be in the provided array. The array is made of `Row` and `Col` (position of the pixel) which will be affected and `Byte0` and `Byte1` which is compressed RGB565 color value.

Example of a byte array:

```
0x40, 0x02; 0x00, 0x01, 0x00, 0x00; 0x08, 0x04, 0xFF, 0xFF
- sets pixel in 0rd row and 2th column to black color and pixel
in 9th row and 5th colum to white color
```

„ , “ represents one byte, „;“ is the same as „“ but is meant to distinguish between pixels

This command is a performance focused extension of the Set pixel command, by packing more data into a single packet less packet overhead is needed, however, do bear in mind that maximum size of buffer is only 1024+32 bytes, which means that approximately 255 pixels at once can be sent.

Chapter 4

Install and run

Tested configuration:

- OS: Linux Mint 21.3 Cinnamon
- Qt: Qt version 6.8.1
- CMake: cmake version 3.22.1

4.1 Installation

4.1.1 Platformio

Make sure to have installed PlatformIO, ideally extension to Visual Studio Code. Before running the project run Makefile command:

```
$ make pio
```

Run project using pio command line or VS extension

4.1.2 ESP-IDF

Make sure to have installed ESP IDF and it is in environment.
Run `. export.sh` from installed esp-dif

```
esp-idf$ . ./export.sh
```

```
imp_project$ make idfpy
imp_project$ idf.py build
```

this might wary from system to system

```
imp_project$ idf.py -p /dev/ttyUSB0 flash
imp_project$ idf.py monitor
```

4.1.3 Testing with GUI application

Clone, install and run GUI application from: https://github.com/denis-fekete/imp_bitmap_editor.

Git repository is added as a submodule in this repository under `/gui` directory. Make sure to have installed Qt, tested on version 6.8.1.

Navigate to `/gui` directory:

```
imp_project$ cd gui
```

Create and navigate to build directory:

```
imp_project/gui$ mkdir build
imp_project/gui$ cd build
```

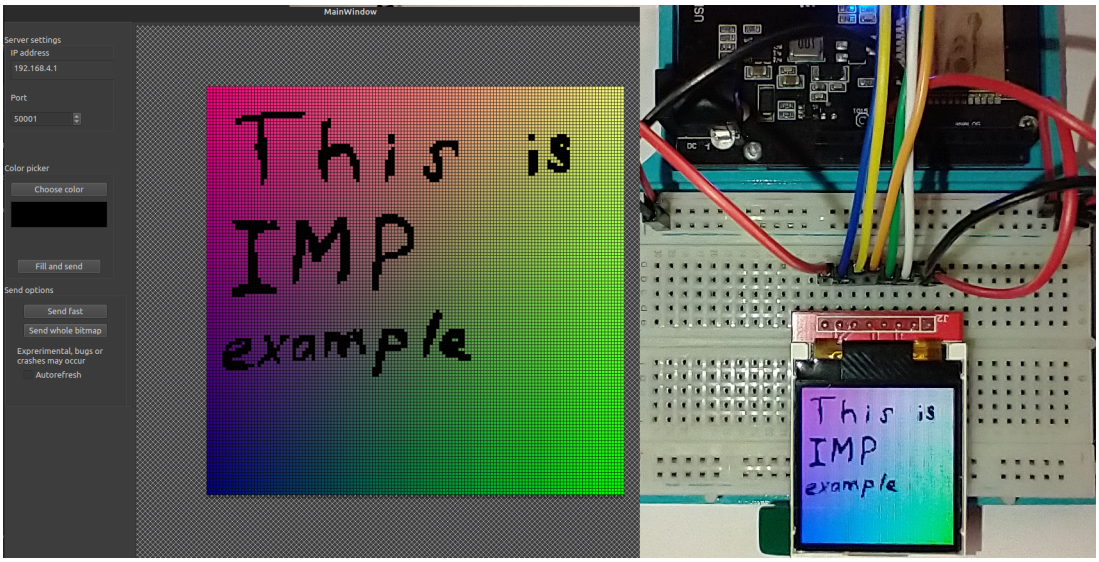
Run cmake, after than run make:

```
imp_project/gui$ cmake ..
imp_project/gui$ make
```

Run compiled binary `imp_drawing`

```
imp_project/gui$ ./imp_drawing
```

4.1.4 Real world example



Video showing basic functionality and installation process: https://youtu.be/P-EEpC1dRDg