# Table of Contents

**1. Laboratory on the Internet of Things - Advanced IoT Hardware - on-site** ..... 2
***1.1. Hardware*** ..... 2
1.1.1. ESP32 WROVER DEVKIT ..... 2
1.1.1.1. Built-in integrated LCD (TFT) display ..... 3
1.1.1.2. Configuring VREL - platformio.ini ..... 4
1.1.2. M5 Stack ESP32 Core Ink DEVKIT ..... 5
1.1.2.1. Built-in integrated e-Paper display ..... 7
1.1.2.2. Configuring VREL - platformio.ini ..... 7
1.1.3. A special note on PWM in ESP32 ..... 9
***1.2. Laboratory scenarios*** ..... 10
1.2.1. Task 0 ..... 10
1.2.2. Task 1 ..... 10
1.2.3. Task 2 ..... 10
1.2.4. Task 3 ..... 11
1.2.5. Task 4 ..... 11
1.2.6. Report ..... 11
***1.3. Resources*** ..... 11

# 1. Laboratory on the Internet of Things - Advanced IoT Hardware - on-site

Laboratory guide for IoT AIH students on ESP32 graphical display programming:

- ESP32 WROVER DEVKIT module - that includes (among others) dot-matrix LCD TFT display
- M5 Stack CORE Ink DEVKIT Module - treat contains (among others) dot-matrix e-paper (e-ink) display

## 1.1. Hardware

* ESP32 WROVER DEVKIT with LCD TFT module * ESP32 PICO D4 M5 Stack with e-Paper (e-Ink) module

Below, we present hardware details on those two modules:

### 1.1.1. ESP32 WROVER DEVKIT

This board is double-sided; on one side, you will find an LCD screen (so-called **back**); on the other side (**front**), there are GPIO connectors you connect sensors and actuators to, TF (microSD) card slot with SPI reader and an RGB LED (among others).

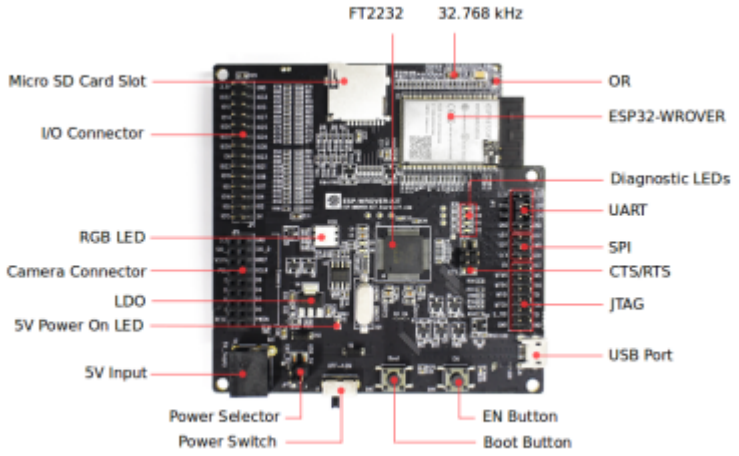
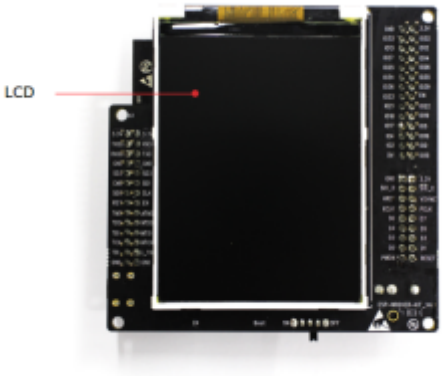



Fig. 1: ESP32 WROVER Dev Kit front side [1)]

Fig. 2: ESP32 WROVER Dev Kit back side [2)]




ESP32 provides many GPIOs to the developers, but some are hard-bound to their features as connected on the development board.

Some GPIOs are dedicated to implementing a limited number of features, while some are dedicated to using a particular protocol on this development board. As this development board contains some peripherals (RGB LED, LCD, TF card reader), do not use GPIOs that are controlling them.

The following list presents GPIOs used with the integrated hardware:

Tab. 1: ESP32 WROVER DEVKIT GPIO and features

| No | Feature | ESP32 GPIO Pin | Notes |
|---|---|---|---|
| 1 | JTAG | GPIO12-15 | |
| 2 | Camera | GPIO0, GPIO4-5, GPIO18-19, GPIO21-23, GPIO25-27, GPIO34-36, GPIO39 | If you do not use the camera, feel free to use those GPIOs unless they're not shared with another feature. |
| 3 | RGB LED | GPIO0, GPIO2, GPIO4 | |
| 4 | MicroSD card reader | GPIO2, GPIO4, GPIO12-15, GPIO21 | |
| 5 | LCD | GPIO5, GPIO18-19, GPIO21-23, GPIO25 | |

Note - GPIOs 34, 35, 36 and 39 do not have internal pull-up nor pull-down resistors

#### 1.1.1.1. Built-in integrated LCD (TFT) display

The built-in display is ILI9341 TFT display, 320×240 pixels (240×320 actually), RGB, 262k colours. Although it is possible to program it raw, we will use a ready software library to control it. This TFT display uses 16-bit per pixel memory organisation, frequently referenced as 565 (RGB, R-5-bits, G-6-bits, B-5-bits).

#### 1.1.1.2. Configuring VREL - platformio.ini

Correct configuration is necessary to build and deploy your code.
For ESP32 WROVER DEVKIT, a good starting point is the following content for the `platformio.ini` file:

```
[env:d1_mini]
platform = espressif32
board = esp-wrover-kit
framework = arduino
lib_ldf_mode = deep+
lib_deps =
   adafruit/Adafruit GFX Library
   Wire
   espressif/WROVER KIT LCD @ ^1.0.3
```

Configuring the serial port name used for uploading your firmware and monitoring is also necessary. In the case of this microcontroller, it is the same port, so the `platformio.ini` file should be modified with the following:

```
...
upload_port = <some port name comes here>
monitor_port = <some port name comes here>
...
```



Check your COM port in the Device Manager (Windows) or terminal (dev for Linux / macOS). It can be different than in the example above!

##### 1.1.1.2.1. Practice: handling LCD display

Write Hello World on the LCD screen.

Type the textual “Hello World” on the screen. Then, you're welcome to decorate it, change colours, size, orientation, etc.

You must include the correct libraries for this LCD model to use LCD. The includes section of your source code file may then look the following way:

```
...
#include <Arduino.h>
#include <SPI.h>
#include <Adafruit_GFX.h>
#include <WROVER_KIT_LCD.h>
...
```

Then, define a controller class that manages your LCD:

```
...
WROVER_KIT_LCD tft;
...
```

Remember to initiate the LCD controller class:

```
...
  tft.begin();
...
```

Some useful functions to handle LCD operations are presented below (but not limited to):

```
...
  tft.fillScreen(WROVER_BLACK); //Clear screen
  tft.setCursor(0, 0); //Set upper left corner
  tft.setTextColor(WROVER_WHITE);  //Set text colour
  tft.setTextSize(2); //Set font size
  tft.println("Hello World!"); // Print the text
...
```



Have a minute and jump into the `WROVER_KIT_LCD.h` and `Adafruit_GFX.h` headers to find more useful functions, i.e. drawing a bitmap, scrolling the display, etc.

### 1.1.2. M5 Stack ESP32 Core Ink DEVKIT

This M5 Stack module is an integrated ESP32 microcontroller (PICO D4 version, full specification is available on the manufacturer's page). This module, even if compact, also contains expansion ports, an internal battery powering RTC (BM8563) when no external power is present, 3 physical buttons: reset, power and programmable, a green LED and a passive buzzer.



Fig. 3: M5 Stack ESP32 Core Ink DEVKIT

ESP32 provides many GPIOs to the developers, but some are hard-bound to their features as connected on the development board.


Following GPIOS are exposed: G25, G26, G36, G23, G34, G18, G21, G22, G14, G13



Fig. 4: M5 Stack ESP32 Core Ink DEVKIT GPIOs


Some GPIOs are dedicated to implementing a limited number of features, while some are dedicated to using a particular protocol on this development board. This development board contains some peripherals (RGB LED, LCD, TF card reader) that do not use GPIOs to control them.

Tab. 2: M5 Stack Core Ink Development Kit features and connections

| No | Feature | ESP32 GPIO Pin | Notes |
|---|---|---|---|
| 1 | EPaper (e-Ink) SPI display controller | GPIO4, GPIO0, GPIO15, GPIO9, GPIO18, GPIO23 | In order: EPaper BUSY, ePaper RST, EPaper DC, EPaper CS, EPaper SCLK, EPaper MOSI |
| 2 | Multifunction button, a physical button, led and buzzer | GPIO37, GPIO38, GPIO39, GPIO5, GPIO10, GPIO2, GPIO12 | In order: Left (up) multifunction, Button press, Right (down) multifunction, Physical button, LED, Buzzer, Power control MOS |
| 4 | Internal I2C for BM8563 (RTC) | GPIO21, GPIO22 | SDA, SCL, module I2C address is 0x51) |
| 5 | External port (I2C) | GPIO32, GPIO33 | software-defined M-BUS |
| 6 | ESP32 Serial (via USB to serial chip, CP2104) | GPIO1, GPIO3 | RXD, TXD |

#### 1.1.2.1. Built-in integrated e-Paper display

The e-Paper (e-Ink) display is a bi-coloured, 200×200 pixels with an SPI interface module GDEW0154M09. It has an integrated SPI controller.

#### 1.1.2.2. Configuring VREL - platformio.ini

Correct configuration is necessary to build and deploy your code.
For the M5 Stack Core Ink Development Kit, a good starting point is the following content for the `platformio.ini` file:

```
[env:m5stack-core-esp32]
platform = espressif32
board = m5stack-core-esp32
framework = arduino
lib_deps =
    m5stack/M5Core-Ink@^0.0.7
```

Configuring the serial port name used for uploading your firmware and monitoring is also necessary. In the case of this microcontroller, it is the same port, so the `platformio.ini` file should be modified with the following:

```
...
upload_port = <some port name comes here>
monitor_port = <some port name comes here>
...
```



Check your COM port in the Device Manager (Windows) or terminal (dev for Linux / macOS). It can be different than in the example above!

#### 1.1.2.2.1. Practice: handling e-Paper display

Write Hello World on the e-Paper screen textual, then present any image (be respectful when choosing one, please :) ).
To start using e-Paper display easily, you may use a library that can handle display operations. So your source code includes a section that may look somehow like this:

```
...
#include <Arduino.h>
#include "M5CoreInk.h"
...
```

The display initialisation uses a singleton class controller of the M5 stack, including an e-Paper display (static object, already present in the library; you do not need to declare any variable of its type):

```
M5.begin();
  if (!M5.M5Ink.isInit())
  {
    Serial.printf("e-Paper init faild");
    while (1) delay(100);
  }
```



e-Paper is relatively slow. Mind giving it some relaxation time. To clear a display (a good idea to start from), you may use the following code:

```
...
  M5.M5Ink.clear();
  delay(1000);
...
```

This library uses (among others) a “sprite” approach to handle the display layout. A sprite is simply a buffer that you draw virtually all graphics you want to draw; it is still located in the RAM. Once your composition is ready, you can make it visible all at once, by simply uploading it to the display's memory. it enables you to easily handle updates of the part of the display rather than the whole area, as e-Paper (e-Ink) displays are slow in displaying, according to their nature.
To define a sprite, use its starting coordinates (x,y), width and height. The following “sprite” definition creates a full-screen one:

```
...
Ink_Sprite InkPageSprite(&M5.M5Ink); //declare a sprite object
...
 if (InkPageSprite.creatSprite(0, 0, 200, 200, true) != 0) //initialize it
for full screen area (0,0) -> (200px,200px)
  {
        Serial.printf("Ink Sprite create failed");
  }
...
```

There are several functions to handle various tasks so that M5 Stack does not split HAL and GFX layers separately but instead integrates them into a single library. Sample calls to “draw” into the sprite (note it does not show anything on display, draws virtually to the sprite's memory are presented below:

```
...
  InkPageSprite.drawString(10,10, "Hello AIH labs!");  //draw a text
...
```

```
...
  InkPageSprite.drawBuff(0,0,index__w,index__h, _picture); //draw an image, binary defined in the ''_picture'' variable (an array of unsigned bytes)
...
```



To display a “sprite”, use:

```
  InkPageSprite.pushSprite();
```

### 1.1.3. A special note on PWM in ESP32

PWM frequently controls analogue-style, efficient voltage on the GPIO pin. Instead of a resistance driver, PWM uses pulses to change adequate power delivered to the actuator. It applies to motors and LEDs, bulbs, heaters, and indirectly to the servos (but that works another way; see lectures).
The classical `analogWrite` method known as Arduino (Uno, Mega) and ESP8266 do not work here. ESP32 has sixteen (0 to 15) PWM channels (controllers) that can be freely bound to any of the regular GPIOs.
Moreover, ESP32 can use various “resolutions” of the PWM: from 8-bit (0..255) to 15-bit (0..32767), while regular Arduino uses only 8-bit one.

To use PWM in ESP32, one must perform the following steps:

- Configure GPIO pin as OUTPUT
- Initiate PWM controller with fixing PWM frequency and resolution
- Bind the controller to the GPIO pin
- Write to the controller (not to the PIN!) providing a duty cycle related to the resolution selected above - every call persistently sets the PWM duty cycle until there is the next call to the function setting duty cycle.

# 1.2. Laboratory scenarios

In general, you will program using the Arduino framework.
As presented during the lecture, you will also use FreeRTOS functions to handle multicore parallel programming to perform tasks, as demonstrated during the lecture.



ESP32 framework uses C++ optimiser while compiling the source code. It is common that C++ traces unused code and removes/optimises it to minimise CPU effort. As your tasks to be completed involve integer and floating-point performance evaluation, it is essential to use the result somewhere, i.e. do a dummy display of the final value on the screen. Otherwise, the C++ compiler may remove this section of the code, and you won't be able to obtain reasonable results because your code may be already trunk and not executed at all!



FreeRTOS, once the task is created and scheduled for execution, executes the task infinitely! It means your function implementing the task will be executed again and again. If you intended to run a looped task, that is pretty OK. If you intend to run the one-off task, then you need to call explicitly:

```
...
vTaskDelete(NULL);
...
```

as a last line of your function implementing the task.

## 1.2.1. Task 0

This warm-up lab scenario lets you understand and prepare a code for presenting your data on the display (LCD and e-Paper).
Implement a program that on the first screen presents, alternatively (each for 5s), your team number and names of the team members then presents a picture of choice. (logo, photo, meme).

## 1.2.2. Task 1

Evaluate single-core performance for integer computation. Perform two experiments: with the task bound to the APP core and separately to the PRO core. First, observe if there is a difference in measurements. Then, propose an algorithm that can generate a complexity of integer computation observable and measurable. Perform at least 10 measurements for each experiment. Consider using parts of code for the Dhrystone benchmark https://en.wikipedia.org/wiki/Dhrystone.

## 1.2.3. Task 2

Evaluate multicore performance for integer computation. Perform single-core computation, running your algorithm at least 4 times (any even number works here). Then execute in parallel (2×2). Observe if there is a difference in measurements. Propose an algorithm to generate a complex, observable, measurable integer computation. Perform at least 20 measurements for each experiment (10 serial + 10 parallel). Consider using parts of code for the Dhrystone benchmark https://en.wikipedia.org/wiki/Dhrystone.

### 1.2.4. Task 3

Evaluate single-core performance for floating-point computation. Perform two experiments: with the task bound to the APP core and separately to the PRO core. First, observe if there is a difference in measurements. Then, propose an algorithm that can generate a complex enough floating points computation that is observable and measurable. Perform at least 10 measurements for each experiment. Consider using FP parts of code for the Whetstone benchmark https://en.wikipedia.org/wiki/Whetstone_(benchmark).

### 1.2.5. Task 4

Evaluate multicore performance for floating-point computation. First, perform single-core computation, running your algorithm at least 4 times (any even number works here) your algorithm. Then execute in parallel (i.e. 2×2). Finally, propose an algorithm to generate a complex enough floating-point, observable and measurable computation. Perform at least 20 measurements for each experiment (10 serial + 10 parallel). Consider using FP parts of code for the Whetstone benchmark https://en.wikipedia.org/wiki/Whetstone_(benchmark).

### 1.2.6. Report

Include algorithm definition and measurements, statistical analysis of the results (at least average and standard deviation) and a summary.

## 1.3. Resources

1. ESP32 WROVER kit starting page
2. Hardware definition (pinouts, GPIOs, etc.)
3. ESP-WROVER-KIT V4.1 schematic
4. M5 Stack Core Ink (e-Paper) module documentation
5. On-line bitmap (image) to memory content (cpp) converter
6. FreeRTOS - multitasking with ESP32 explained

[1)] , [2)]
https://docs.espressif.com/projects/esp-idf/en/latest/hw-reference/get-started-wrover-kit.html

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
**https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:lab:gfxonsite?rev=1699607479**

Last update: **2023/11/10 10:11**