

[This document contains detailed information about the project and was created as a go-to place to find answers to all of your HOWs and WHYs, so the software support and further project development are easier.]

[ March 2025 ]

[42 smart cluster sigN.]

[technical documentation.]

[Table of contents.]

[SUMMARY.]

|  |  |
| --- | --- |
| 2  3  5  6  9  11  12  15  16  18  22  23  24  25  26  30  31  32 | Vocabulary of terms  About the project //TO-DO  Contractor's requirements  General description of the program run  Program run step-by-step //TO-DO  How to build the Sign yourself //TO-DO  Getting ready to maintain and develop the project  Hardware maintenance  Program files description  The config file  Functions description  How to get exams info from Intra  Exam Simulation and how to use it  Create your own graphics  How to draw on the display //TO-DO  The intricacies of time keeping  Service messages meaning //TO-DO  Libraries and their use  Bugs and suggestions how to fix them  New bugs and Future development suggestions  Suggestions for dealing with confidential information  Sources  //TO-DO:  Hardware description  Circuit diagrams and schematics  Description of the program constants  Architectural decisions explained  Power management  Safety considerations  The Don’ts of changing the program  Used terms |

[vocabulary of terms.]

**42 Smart Cluster Sign** — in the project files may be refered to as „the Sign“, „the device“.

**Campus** — physical premises of a 42 school, mainly consist of clusters.

**Cluster** — an auditorium, a study room, usually full of workspaces with computers. For example, C3 is short for Cluster 3.

**Intra** — a website of the 42 school internal information system, but in this project „Intra“ rather means the school‘s internal information system server, as the Sign never accesses the website as a common user, but only the server via its API.

**Secret** — a credential. Along with a UID, it allows to log into Intra via its API. Usually, good for a month, then it expires. Not to be confused with „a security token“ or just „a token“.

**Security token** — also refered to as „token“, is a unique short-lifespan key given by Intra that allows the device to access data stored on the server. Usually, good for a few hours.

**Seeed Studio XIAO ESP32-C3** — a development board with an ESP32-C3 microcontroller and a battery charging IC under the shield. Has two buttons: „B“ stands for „BOOT“, and „R“ stands for „RESET/REBOOT“. Has a red LED indicating the states of the battery charging process.

**ESP32-C3** — the microcontroller onboard the Seeed Studio XIAO ESP32-C3 development board, located under the shield.

**Deep Sleep** — a functionality of an ESP32-C3 microcontroller allowing it to stay ON while consuming almost no power. The most effective way to save battery charge, but reboots the microcontroller causing all the temporary data in RAM to be lost.

**RTC memory** — a small section of the ESP32-C3 memory that stays powered over Deep Sleep. Data put into this section will survive Deep Sleep, but will not survive resetting the device with the reset button. In this project, only data put into the file system can survive both Deep Sleep and resetting the device with the reset button.

**Light Sleep** — a functionality of an ESP32-C3 microcontroller similar to Deep Sleep. Light Sleep is far less effective in saving battery charge, but does not lose RAM data and allowes to contunue executing the program after sleep.

**OTA** — stands for „Over-The-Air“, functionality that allows updating the software of the microcontroller wirelessly.

**SPIFFS** — stands for Serial Peripheral Interface Flash File System, is one of the ESP32-C3 microcontroller memory partitions dedicated to storing files. In this project (and often on the Internet) the term is used as a synonym to „a file system“. The term also serves as a name to the formerely widly-spread SPIFFS library. Instead of it, this project employs the LittleFS library as more modern and light-weight.

[ABOUT THE PROJECT.]

This project is designed to manage and display information on a smart sign for a 42 campus cluster. The sign displays various messages and images, including exam schedules, battery status, and OTA updates. It communicates with the 42 Intra API and a Telegram bot to fetch and display relevant information.

[**Contractor's Requirements**.]

[and how the sign matches them.]

The finished device must fulfill the following requirements:

1. **Auditorium status display**: the device must always display the current status of the auditorium, indicating whether it is free or if an exam is in progress.  
     
   — *The Sign displays the number of the auditorium in calm black and white colours to indicate that the room is available for everyone or it displays the „exam in progress“ sign in bright red and black colours to show that the room shall not be entered by those who do not participate in the exam. The Sign is never blank.*
2. **Advance notification**: the device should notify students in advance on exam days about the need to vacate the auditorium.   
     
   — *The Sign displays a note with the exact time of the upcoming exam right from the morning of that day. 1 hour before the exam the black-white-and-red „reserve for exam“ sign with time left starts being displayed.*
3. **Autonomy**: the device must operate independently, performing its tasks and solving problems that arise without requiring the time or intervention of the educational institution's staff.   
     
   — *The Sign can connect to the Internet, access the institution server API and pull exams date and time. This is how it knows when to display the apropriate state of the auditorium. The Sign software is designed with all the common negative situations in mind which ensures the Sign does not bother anyone unless it absolutely needs to.*
4. **Problem reporting**: the device must be capable of reporting issues that cannot be resolved without assistance from the educational institution's staff.   
     
   — *When the Sign fails to resolve an unordinary situation itself and requires assistance, it displays an apropriate error message on its display as well as sends a detailed error report to its Telegram chat.*
5. **Support and expandability**: the device should be designed for support and expansion, allowing any student of the educational institution to contribute to the project, develop new functionality, and upload new software onto the device.   
     
   — *The Sign project was chosen to be made using Arduino IDE as the most beginners-friendly developing platform. The software was developed using a straightforward bare-metal approach to maintain easy-to-follow program logic. The Sign has a standart USB-C port for flashing its software, monitoring its Serial port and charging its battery. The microcontroller used in the project has inner USB controller which eliminates the need of using a UART-TTL adapter. The microcontroller pins are equipped with standart Dupont sockets which allows anyone to change used pins and add new hardware by simply connecting it to the microcontroller with Dupont cables. The Sign is securelly fixed on the wall with 4 furniture double ball catches, which at the same time allow to take the Sign off the wall for maintanance.*
6. **Safety**: the device must ensure safety by incorporating protective elements for potentially dangerous electronic components.   
     
   — *The battery used in the project has an embedded protection against shortcuts, overcharging and ungercharging, which has the ability of completely disconnecting the battery from the rest of the circuit. The battery charging IC is capable of adapting its charging power, applying lower voltages when the battery is low on charge or close to being fully charged. The charging IC generally uses slightly lower charging rate than the standart charging rate for this particular battery, which will result in the battery longer life.*
7. **Rechargeable operation**: the device must operate on a rechargeable battery, include a common charging connector, and provide indications for the charging process and its completion.   
     
   — *The Sign employs an internal high-capacity rechargeable battery, which insures a long lasting operation without the need of changing batteries. The battery can be recharged by plugging the Sign into any 5V power adapter. The Sign has a USB-C connector for recharging the battery as well as for flashing the software. The Sign employs a red LED to indicate an ongoing charging process.*
8. **Design compatibility**: both the graphical user interface (GUI) and the physical appearance of the device must align with the established style of the educational institution's interior design.   
     
   — *The body of the Sign is made of black wooden frame with mate finish, which perfectly matches the mate-black profile of the auditorium glass door. The GUI was designed inspired by the painings on the walls of the institution as well as the „42“ logo, thus nicely matching the overal style of the interiors.*

[**General description of the program run**.]

Let's take a look at one day of the 42 Smart Cluster Sign‘s life. Let’s say that on this particular day there is an exam scheduled for 13:00 and it will last for 3 hours.

It is still night and the Sign is showing the cluster number with the default pictograms from the day before, while still being asleep. It is normal for the Sign to sleep all the time. It has its scheduled hours to wake up and check if something needs to be done. They are 6, 9, 12, 15, 18 and 21 o’clock (could have been changed in the program). But it is still too early.

Time passes by. Now, it is 6:00 in the morning. The Sign wakes up to check if there are exams today. It goes online, pulls the information from the Intra server and sees that there will be an exam starting at 13:00 and ending at 16:00. The Sign replaces the default pictograms from yesterday with a note “*The cluster will be reserved for an exam today at 13:00*” while still displaying the cluster number. Since there is nothing more for the Sign to do, it sets its alarm clock for 12:00 - an hour before the exam - and goes back to sleep.

It is 12 o'clock and the Sign wakes up again to get ready for the exam. It checks Intra to make sure the exam was not canceled during its sleep and that there is at least 1 person attending. If everything checks, it replaces the cluster number with a big warning sign that says: “*RESERVATION! The cluster is reserved for an exam. Please, vacate it in due time. You have XX minutes left*”. Instead of XX it first says 50 minutes, then 25 minutes and finally 5 minutes left.

Finally, it is 13:00. The exam begins. The Sign changes the previous warning sign for a new one, saying “*DO NOT ENTER! Exam in progress!*”. At this point the Sign has nothing else to do, so again it sets its alarm clock for 16:00 - the end of the exam - and goes to sleep.

At 16 o'clock the Sign wakes up, checks Intra, finds no more exams for today, replaces the warning sign with a cluster number with the default pictograms, sets its alarm clock until the next scheduled wake up - in this case 18:00 - and goes back to sleep.

At 18 o'clock the Sign wakes up, checks Intra and finds no more exams, so it does nothing. Later, at 21 o’clock it wakes up for the last time today, finding nothing more to be done. Its work for today is over. It goes to sleep to wake up again the next morning at 6.

[PROGRAM RUN STEP-BY-STEP.]

STARTING THE PROGRAM

1. Turning on

2. Initializing the Serial Port

• Outputting a dash to the serial port gives time to synchronize data transfer with the computer and avoid losing important data

3. Initializing the file system

• If the file system initialization fails, the following functions will not be available: restoring the last used Telegram chat, the Secret value, and the OTA flag value after reinstalling the program, after a power outage, or after a software reset (i.e. after all cases when RTC memory data is lost); record the value of the last used Telegram chat, the Secret value and the OTA flag value

4. initialize the buttons (disabled due to bugs)

5. initialize ADC for battery measurements

6. initialize the SPI port of the display

7. check the reboot reason

• also restores the value of the last used Telegram chat, the Secret value and the OTA flag value after reinstalling the program, after a power outage or after a software reset (i.e. after all cases when RTC memory data is lost)

• also puts the device to sleep for 24 hours if the BROWN OUT detector is triggered. It is triggered if the battery charge is insufficient to continue operation. In this case, [DEVICE OPERATION ENDS HERE] until the battery is charged.

8. battery check

• due to the technical features of the device, we can determine the battery charge level only when it is almost discharged. Accurate battery measurements can be taken approximately between 3% and 0% of the battery charge. In ADC readings, this corresponds to 800 and 400.

• take 5 measurements of the charge level and calculate their average value

• all readings above 800 mean that the battery is sufficiently charged and there is no need to report a low battery - exit the function

• connect to Wi-Fi to report the battery status to Telegram

• all readings below 400 mean a completely discharged battery. Despite the fact that the BROWN OUT detector did not work in the previous step, you cannot continue working with such a low charge. We report a low battery in Telegram, display the message "Low battery" on the display and put the device to sleep for 24 hours. In this case, [DEVICE OPERATION ENDS HERE] until the battery is charged.

• indicators between 700 and 600 may mean that the device is charging

• if we still haven't exited the function, but the indicators are below 800, then the battery is already discharged, but we can still continue working. We report the discharged battery in Telegram, display the message "Low battery" on the display and continue executing the program.

9. initializing the OTA function (disabled due to blocking by the firewall)

10. switching to OTA mode (disabled due to blocking by the firewall)

11. choosing which mode to continue working in: in cluster number mode (default mode) or in exam mode

• Cluster number mode displays the cluster number + icons (on a normal day) or a warning message about the exam (on the day of the exam) or error messages (inability to receive exam data, expired Secret, low battery). This mode is active 99% of the time.

• The exam mode is activated 1 hour before the exam, shows a warning about the exam starting soon, then switches to a warning about the exam in progress and after the exam is over, switches back to the Cluster Number Mode. This mode is active only on the exam day, 1 hour before the exam + the entire exam time.

• Double checking the exam status flag in this function is necessary to switch from one mode to another. Do not change!

IN CLUSTER NUMBER MODE

1. connecting to a Wi-Fi network

2. checking incoming messages in the Telegram chat

• A new Secret may arrive via Telegram chat, which will be useful later when requesting exam data

3. synchronizing time, date and summer/winter time with the NTP server

• Without time data, reliable operation of the device cannot be ensured. If after several attempts to receive time data it was not possible, an error is displayed on the display and the device itself is put to sleep until the next scheduled awakening. In this case, [DEVICE OPERATION ENDS HERE], until we manage to get the time data during one of the future scheduled awakenings.

4. get exam data for the current day from Intra

• Without exam data, it is impossible to ensure reliable operation of the device. If after several attempts to get exam data it was not possible, an error is displayed on the display and the device itself is put to sleep until the next scheduled awakening. In this case, [DEVICE OPERATION ENDS HERE], until we manage to get exam data during one of the future scheduled awakenings.

• go to the Intra website

• log in to the Intra website

• go to the schedule page for today

• read the received HTML code until we find exam data

• clear the data from unnecessary garbage

• compare the received data with the existing data

If the data differs, we change it on the display;

If not, we leave it as it is.

Setting the time for the next activation.

Turning off the display power.

Power off.

[How to Build the Sign Yourself.]

YOU WANT TO MAKE YOUR OWN SIGN FOR A 42 CAMPUS BUT THIS INSTRUCTION IS NOT COMPLETE YET?

CONTACT THE AUTHOR FOR PERSONAL FREE-OF-CHARGE ONLINE CONSULTATIONS!

LINKEDIN: <https://www.linkedin.com/in/roman-alexandrov-a75b89195/>

Building the 42 Smart Cluster Sign involves assembling the hardware components, setting up the software, and configuring the device. Follow these instructions to build your own 42 Smart Cluster Sign.

## Hardware Components

The detailed information about the hardware components can be found in the Bill\_of\_Materials.xlsx file, located in the docs folder of the project, but here is a simple list:

1. Seeed Studio XIAO ESP32C3 Wi-Fi module
2. External WiFi antenna 2.4G with IPEX1 connector
3. Good Display GDEY075Z08 7.5" 800x480 ePaper black/red/white SPI display
4. Good Display DESPI-C02 universal SPI e-Paper adapter
5. Dupont male-to-female wires for internal wiring
6. Push buttons
7. 4000mAh Li-ion battery with overcharge and undercharge protection
8. custom 3D-printed board
9. IKEA RÖDALM photo frame, 13x18 cm
10. furniture ball catches
11. transparent plexiglass pannel, 2.5mm or thiker, 13x18 cm
12. transparent double-sided sticky tape

## Tools Required

* soldering iron and solder
* screwdrivers
* 3D printer
* computer with any OS
* USB to USB-C data cable compatible with your computer
* 5V power adapter

## Needed software

* Arduino IDE with installed ESP-IDF plug-in,
* Telegram (smartphone app or its desktop version).

## Display Pins

|  |  |  |
| --- | --- | --- |
| XIAO ESP32C3 Pins | Display adapter Pins | Pin Description |
| D4 | **BUSY** | busy / free status line |
| D5 | **RES** | reset line |
| D6 | **D/C** | Data mode / Command mode select |
| D7 | **CS** (also called **SS**) | Chip Select |
| D8 | **SCK** | common clock |
| D10 | **SDI** | data line |
| GND | **GND** | ground line |
| 3.3V | **3.3V** | power line |

## Building Steps

1. Order the hardware components
2. Print the custom 3D-printed board
3. Create an Intra API app for your Sign
4. Create a Telegram bot for your Sign
5. Install Arduino IDE and flash XIAO ESP32C3 with the program

## Alternatives For The Hardware Components

Compatibility with any alternatives to the hardware stated in the hardware list above was not tested. Using alternative hardware components may complicate building the Sign in an unexpected way. Still, here are some advices if you decide to seek alternative hardware:

* if you want to replace the Seeed Studio XIAO ESP32C3 Wi-Fi module with any other Seeed Studio ESP32 module, make sure that it has at least the same ammount of RAM, since the project software is RAM intensive. Replacing it with other ESP32 modules may also require to adjust the custom 3D-printed board;
* you may use any Wi-Fi antenna as long as it fits the module;
* the project software uses Jean-Marc Zingg's GxEPD2 library for e-paper displays. The library has its list of supported e-paper displays and your alternative display should be on the list. Using any alternative dislay will always require adjusting the project software. Using an alternative display with different resolution will also require to remake all the GUI images used in the project. Using an alternative display with different physical size and/or different physical proportions will also require to adjust the custom 3D-printed board. Using an alternative display adapter would probably require to adjust the custom 3D-printed board, too;
* if you do not need modularity in your version of the Sign, it would be easier to solder ordinary wires instead of using Dupont wires. If needed, Dupont male connectors may be soldered, too;
* you may use any push-buttons as long as you are willing to adjust the custom 3D-printed board to fit them;
* you may use any battery as long as it is rated 3.7V and is capable of constantly outputting at least 250mA. Smaller alternative batteries may be simply fixed on the 3D-printed board with a double-sided sticky tape. Bigger alternative batteries may require to adjust the custom 3D-printed board to fit them;
* you may avoid printing the custom 3D-printed board and simply glue all the electronic components inside of the frame. In that case, be adviced that e-paper displays do not like ANY heat and an ESP32 can get very warm during work and even hotter when the battery is being charged, so make sure to have some good insulation between the two;
* any frame would work as long as it is deep enough to accomodate all the electronics. It does not even has to be a frame, but any enclosure you can produce;
* as the furniture ball catches can be quite pricy, you may want to 3D-print the catches yourself. Luckily, there is plenty of ready-to-print models on the Internet.
* if you do not need your Sign to be detachable from the wall, you will not need the furniture ball catches and the transparent plexiglass pannel.

[GETTING READY TO MAINTAIN AND DEVELOP THE PROJECT.]

## Needed hardware:

* Computer with any OS,
* USB to USB-C data cable compatible with your computer.

## Needed software:

* Arduino IDE with installed ESP-IDF plug-in,
* Telegram (smartphone app or its desktop version).

## Preparing the software tools

1. Install Arduino IDE and add the ESP-IDF extension. User-friendly instructions on how to do it may be found here:

<https://randomnerdtutorials.com/getting-started-with-esp32/#esp32-arduino-ide>

The online instruction suggests to download and install the latest version, but the project was built using Arduino IDE version **1.8.19** and the „esp32“ board version **3.0.7**. Compatibility with the later versions was not tested, this is why it is recomended to use these, even though outdated, versions of the tools.

1. Create a folder for Arduino IDE projects. This folder will contain all the projects ever created in your Arduino IDE as well as all the installed libraries. The folder may be created anywhere on your computer and may be called any name you give to it (do not use spaces in the folder name, it may cause problems with the IDE).   
   Now, in your Arduino IDE, go to **Arduino > Settings** and at the top of the opened Settings window add the created folder path.
2. Make sure all the required libraries from the „LIBRARIES AND THEIR USE“ list are installed. To do so, in your Arduino IDE, go to **Arduino > Add library > Manage libraries**. In the opened window of the libraries manager you may find all the installed libraries as well as all the available libraries on the Internet.

It is recommended to install the libraries versions stated in the list even though they might be outdated.

1. Set the compilation target. The compiler has to be told what exact model of an ESP32 board to compile for. In case of this project it is „**XIAO\_ESP32C3**“. To do so, in your Arduino IDE, go to **Tools > Boards > ESP32 Arduino > XIAO\_ESP32C3**.

## Opening the project

1. Open the folder for Arduino IDE projects (the one created in step 2 above) in your terminal and use the following command to get yourself a copy of the project:

**git clone https://github.com/RomanAlexandroff/42-Smart-Cluster-Sign.git**

1. Open your Arduino IDE and go to **File > Projects > 42-Smart-Cluster-Sign > src**. Your project will open.
2. The project comes without any security-sensitive credentials. They may be found printed on the back of the Sign or obtained from the Bocal team. Rename the „**credentials-example.h**“ file included in the project into „**credentials.h**“ and fill-in the credentials from the Sign.

**DO NOT COMPROMISE THE CONFIDENTIALITY OF THE CREDENTIALS !!!**

**REPORT ALL THE OCCURED LEAKS TO THE BOCAL TEAM IMMEDIATELY !!!**

## Uploading the changes

1. Connect the Sign to your computer if you have not done so by this time.
2. Activate the software update mode. To do that, on the back of the Sign locate button **B** and button **R**. First, press and hold button B. While holding button B, press and release button R once. Then release button B. Software update mode is now active.
3. In Arduino IDE, go to **Tools** and set the following settings as follows:

* Upload speed: 115200
* CPU Frequency: 160 Mhz
* Flash Frequency: 80 Mhz
* Flash Mode: "QIO"  
   the fastest mode for the flash memory
* Partition Scheme: "Minimal SPIFFS"  
   do not use partition schemes marked with "No OTA"
* Core Debug Level: "Verbose"  
   the most detailed debugging output into the Serial monitor
* Erase All Flash Before Sketch Upload: "Disabled"
* Port: choose the development board port.

1. In Arduino IDE, click the Upload button to start uploading.

[hardware MAINTENance.]

The Sign generally does not require any assistance, but it needs to be charged once every so often. If you have not ever charged the Sign before, it is OK, since its high-capacity inner battery allows the Sign to operate up to half a year on a single charge.

If the Sign needs to be recharged, it will indicate it by displaying the „LOW BATTERY“ note next to the cluster number as well as sending a notification into its Telegram chat.

Needed hardware:

* **5V** power adapter,
* USB-C cable, compatible with the power adapter.

To charge the Sign:

1. locate a round opening on the side of the wooden frame;
2. look inside the opening - there you should see a USB-C female connector;
3. find a cable with a USB-C male connector that fits the round opening;
4. if the cable is too short to reach the power socket, carefully take the Sign off the wall.   
   To do that use both of your hands! Grab the wooden frame from its top and its bottom – that will prevent it from falling. Do not pull the Sign from the wall with your arms. Instead start pushing your fingers deeper between the frame and the wall – at some point the Sign will simply detach from the wall and stay in your hands;
5. connect the Sign to the 5V power adapter with the cable;
6. plug the 5V power adapter into a power socket;
7. look at the opening in the frame again - you should see red light comming out of it, which means that the Sign is now charging. If after a few seconds there is no red light, that may indicate a poor connection or lack of electricity.
8. wait for the red light to turn off – typically about 10 hours – that indicates that the Sign is fully charged.

[PROGRAM FILES DESCRIPTION.]

|  |  |
| --- | --- |
| src.ino | Main file. Despite being written in C, it has to have the .ino extention as it is the Arduino IDE file format. Other source files may have other extentions. The file name has to be the same as the folder name it is contained in – as you may see in this project. |
| 42-Smart-Cluster-Sign.h | Main header file that includes all necessary libraries and declares functions used across the project. Including ota.h at the bottom of the file is not coinsidential – it has to be kept below the OTA functions declarations for the OTA update to work. |
| battery\_management.cpp | Initialize inner ADC module, measure battery voltage level, assign the results to a battery state, act upon the battery state. |
| bitmap\_library.h | Contains all the images to be displayed on the device screen in bit-map form. The complete images list can be found in the file. |
| buttons\_handling.cpp | Buttons initialisation and interrupt service routines. |
| cluster\_number\_mode.cpp | Everything that the Sign does outside of the exam time: gets exact time, checks exams, displays system warnings if there are any, displays cluster number. |
| config.h | Constants to adjust and tune the program behaviour. E.g. software version number, device name, DEBUG macro, Serial port baud rate, the Sign’s wake-up hours, Wi-Fi connection time limit, etc. More about it in the **[THE CONFIG FILE.]** chapter. |
| constants.h | Constants that are not expected to be ever changed. General constants, buttons and display SPI port configurations, display driver coniguration, images and errors enumerators. |
| credentials.h | Contains project confidential information, such as Intra API authorisation data, Telegram bot token, Wi-Fi access point SSID and password. If instead of credentials.h there is only credentials-example.h, then in the **[GETTING READY TO MAINTAIN AND DEVELOP THE PROJECT.]** chapter, please, navigate to „Opening the project“, point 3. |
| display\_handling.cpp | Functions for outputting images and text onto the display, cluster number drawing logic, display initialization. Using display.powerOff() in the drawing functions may trigger the watchdog with high probability. |
| exam\_mode.cpp | Everything to handle informing students about an exam and the pre-exam time. |
| file\_system.cpp | Manages file system operations, including reading from and writing to files. |
| globals.h | Declares global variables and includes necessary libraries. |
| globals.cpp | Defines global variables used across the project. |
| intra\_interaction.cpp | Handles interactions with the 42 Intra API, including fetching exam schedules. |
| ota.h | Manages OTA updates, including initializing and handling OTA update processes. |
| utils.cpp | Contains functions like sleep, delay and serial initialisation. |
| power\_down\_recovery.cpp | Handles power-down recovery, including reporting reboot reasons. |
| telegram\_bot.cpp | Manages interactions with the Telegram bot, including checking and responding to messages. |
| telegram\_compose\_message.cpp | Composes messages to be sent via the Telegram bot. |
| time\_utilities.cpp | Contains time-related utilities, including fetching and calculating time. |
| watchdog.cpp | Manages the watchdog timer, including starting, stopping, and resetting the watchdog. |

[The config file.]

The config.h file contains configurable parameters for tuning the software behavior of the 42 Smart Cluster Sign project. This file allows you to adjust various settings, including software version, device name, debugging options, and time-related configurations. To customize the behavior of the 42 Smart Cluster Sign project, modify the values in the config.h file according to your requirements. Ensure that the changes you make are consistent with the overall project requirements and do not conflict with other configurations.

**SOFTWARE\_VERSION**: Defines the current version of the software. The software version number can be seen in the debugging output in the Serial monitor as well as in the Telegram chat message when the Sign receives the „/status“ command. The software version number is an essential tool for tracking bugs and the program installed on the physical device.  
  
 #define SOFTWARE\_VERSION 4.32

**DEVICE\_NAME**: Specifies the name of the device, that can be seen in the Ports list when updating via OTA.  
  
 #define DEVICE\_NAME "42 Prague C3 Smart Sign"

**DEBUG** macro: Enables or disables serial output for debugging. Comment out the „#define DEBUG“ line to turn off serial output or uncomment it to turn it on again. The macro does not affect the Core Debug Level, which you set in the Arduino IDE Tools. Changing the DEBUG macro and/or the Core Debug Level will change the dynamic of the whole program execution which might introduce new bugs or solve existing ones.  
  
 #define DEBUG  
 #ifdef DEBUG  
 #define DEBUG\_PRINTF(...) Serial.printf(\_\_VA\_ARGS\_\_)  
 #define WD\_RESET\_INFO true  
 #else  
 #define DEBUG\_PRINTF(...)  
 #define WD\_RESET\_INFO false  
 #endif

**EXAM\_SIMULATION** macro: Uncomment this line to simulate an exam starting at specified time. Useful for testing exam mode execution when there are no actual exams. The macro injects fictitious information about an exam (by default, scheduled for today from 18:00 till 21:00, 4 students attending) into the message from Intra, which causes the Sign to believe that there is an actual exam that day. If needed, the fictitious exam information may be changed in the exam\_simulation() function, located in the utils.cpp file.

***When you are finished with the tests, do not forget to comment out this line and flash the software onto the Sign again to turn off the simulation. Otherwise the Sign will be showing the fictional exam every day.***  
  
 #define EXAM\_SIMULATION  
  
  
**GCC Optimization** macro: Lets you expicitly tell the compiler how to optimise the program code. Be adviced that different levels of optimisation may influence the dynamic of the whole program execution in different ways. In practice, it means that one level of optimisation may introduce new bugs to the project; another level of optimisation may solve those bugs, but introduce completely new ones. Settle on one level of optimisation before starting debugging the project.

* O0 — no optimization (default for the project);
* O1 — basic optimization;
* O2 — moderate optimization: slight program performance increase;
* Os — optimize for size: O2 level + optimizations to reduce program size;
* O3 — high-level optimization: better performance, but bigger program size;
* Ofast — optimize for speed: highest possible performance but may break standards compliance;
* Og — optimize for debugging.

#pragma GCC optimize ("O0")

**BAUD\_RATE**: Sets the speed of the serial communication.  
  
 #define BAUD\_RATE 115200

**WAKE\_UP\_HOURS**: Defines the hours at which the Sign should wake up and check if there are any new exams. The format is a comma-separated list of hours (24-hour format), e.g. 9 means „at 9:00“ (9AM), 18 means „at 18:00“ (6PM), and so on. Note, that every hour is separated by a comma, but there is no comma after the last wake-up hour. Adhere to this format when removing or adding wake-up hours. You may change the existing wake-up hours, delete them or add new wake-up hours. There always should be at least one wake-up hour. Maximum number of wake-up hours is 24. Please, keep in mind that there is no such time as 24:00, instead use 0 (for 0:00) if you want the Sign to wake up at midnights.  
  
 #define WAKE\_UP\_HOURS 6, 9, 12, 15, 18, 21  
  
**RETRIES\_LIMIT**: Sets the maximum number of retries for getting time and exam information. Be adviced, that every next retry is 5 minutes longer than the previous one. E.g. the 1st retry will occur 5 minutes after the default try, the 2nd retry will occur 10 minutes after the 1st retry, the 3rd retry will occur 15 minutes after the 2nd retry, the 4th retry will occur 20 minutes after the 3rd retry, the 5th retry will occur 25 minutes after the 4th retry, and so forth. It is recomended not to exceed the number of 5 retries.  
  
 #define RETRIES\_LIMIT 3  
  
  
**TIME\_ZONE**: Specifies the campus time zone according to the GMT / UTC standard. ***IMPORTANT:*** If in the country of the campus location it is common to switch between winter time and summer time, use the WINTER TIME time zone only! Include "-" sign if it applies to the time zone of your cluster. Do not include "+" sign.   
  
 #define TIME\_ZONE 1

**CONNECT\_TIMEOUT\_S**: Sets the timeout for Wi-Fi connection attempts (in seconds).  
  
 #define CONNECT\_TIMEOUT\_S 5

**DEBOUNCE\_DELAY\_MS**: Defines the debounce delay for button presses (in milliseconds). The lower the number, the faster the button responds, but the probability of errors to occur grows.  
  
 #define DEBOUNCE\_DELAY\_MS 1000ul

**WD\_TIMEOUT\_MS**: Sets the timeout for the watchdog timer (in milliseconds). Limited to a maximum timeout of 8 seconds (8000 milliseconds).  
  
 #define WD\_TIMEOUT\_MS 8000

**OTA\_WAIT\_LIMIT\_S**: Defines the maximum wait time for OTA updates (in seconds).  
  
 #define OTA\_WAIT\_LIMIT\_S 600

[FUNCTIONS DESCRIPTION.]

# main file functions

setup(): initializes various components, including the watchdog timer, display, file system, buttons, battery, power-down recovery, battery check, Telegram bot, and OTA updates.

loop(): handles the OTA update waiting loop and calls the pathfinder() function to determine the next action based on the current status.

pathfinder(): the function determines the next action based on the current status, including handling exam mode and cluster number mode, and then puts the device to sleep for the calculated time. The function deliberately uses two if-statements since the exam\_status may change its value inside the first if-statement.

# Display Handling

draw\_text(String output, uint16\_t x, uint16\_t y): Draws text on the display at the specified coordinates.

draw\_exam\_start\_time(): Draws the exam start time on the display.

draw\_bitmap\_partial\_update(const unsigned char\* image, uint16\_t width, uint16\_t height): Draws a partial bitmap image on the display.

draw\_colour\_bitmap(const unsigned char\* black\_image, const unsigned char\* red\_image): Draws a full-color bitmap image on the display.

draw\_bitmap\_full\_update(const unsigned char\* image, uint16\_t width, uint16\_t height): Draws a full bitmap image on the display.

display\_cluster\_number(IMAGE\_t mode): makes decisions of whether to draw something on the display or not to draw; calls drawing functions when necessary. REFACTORING RECOMENDED. If you refactor the function, please, do not forget to change the documentation accordingly.

clear\_display(): Clears the display.

display\_init(): Initializes the display.

# Exam Mode

exam\_mode(): Handles the exam mode, including displaying exam-related messages and images.

# File System

secret\_verification(String input): Verifies the secret token.

data\_restore(const char\* file\_name): Restores data from the specified file.

data\_integrity\_check(): Checks the integrity of the file system and restores necessary data.

write\_to\_file(const char\* file\_name, char\* input): Writes data to the specified file in the FS.

read\_from\_file(const char\* file\_name, char\* output): Reads data from the specified file in the FS.

file\_sys\_init(): Initializes the File System.

# Intra Interaction

fetch\_exams(): Fetches exam schedules from the 42 Intra API.

# Telegram Bot

telegram\_check(): Checks for new messages from the Telegram bot and handles them.

compose\_message(int32\_t subject, int16\_t days\_left): Composes messages to be sent via the Telegram bot.

# Time Utilities

expiration\_counter(): Calculates the number of days left until the secret token expires.

unix\_timestamp\_decoder(uint8\_t\* p\_day, uint8\_t\* p\_month, uint16\_t\* p\_year): Decodes a UNIX timestamp into day, month, and year.

get\_time(): Fetches the current time from an NTP server.

time\_till\_wakeup(): Calculates the time until the next wake-up.

time\_till\_event(int8\_t hours, uint8\_t minutes): Calculates the time until a specified event.

time\_sync(unsigned int preexam\_time): Synchronizes time before an exam.

# Watchdog

watchdog\_start(): Starts the watchdog timer.

watchdog\_reset(): Resets the watchdog timer.

watchdog\_stop(): Stops the watchdog timer.

watchdog\_init(): Initializes the watchdog timer.

# Other Functions

go\_to\_sleep(uint64\_t time\_in\_millis): Puts the device to sleep for the specified time.

ft\_delay(uint64\_t time\_in\_millis): Delays execution and puts the device into light sleep. This function helps saving battery power by turning off the Wi-Fi and the Bluetooth modules of the device. Do not use it there where you need to maintain wireless connection.

wifi\_connect(): Connects to Wi-Fi.

serial\_init(): Initializes the serial communication.

power\_down\_recovery(): Handles power-down recovery, including reporting reboot reasons.

exam\_simulation(): Creates a fictitious exam information for the EXAM\_SIMULATION macro.

[Architectural decisions explained.]

Even though the program code of the Sign is pretty straight forward, there are some architectural solutions that are not that clearly obvious. Here we will try to address these uncertainties and make sure the program code is fully understandable. Be adviced, that the article assumes that you have already seen the source code as well as read some previous articles here, namely General description of the program run, Program run step-by-step, Program files description, The config file, Functions description, How to get exams info from Intra.

# src.ino file

If you are new to Arduino IDE, setup() and loop() functions are standart for this workframe. Treat them as you would treat main() in a .c file, it’s just here there are 2 functions instead of 1. Do not swap these two functions places, do not place anything between them, do not rename them.

First, **setup()**. I have been told that the order of functions inside setup() seems somewhat random, that you can change the order and nothig would happen. It was just one opinion, but I must address it, since this misconception may cause big problems. The placement of every single function inside setup() was achieved through hours of testing and debugging, so the order is in fact very much intentional and strict. Misplacing them may cause a whole variety of bugs.

* watchdog\_init() shall be the first as it makes sure the whole program runs or resets after fail;
* display\_init() has to be called before serial\_init(). Otherwise the SPI initialisation inside display\_init() will interfere with the start of the serial communication and some debugging messages in the beginning of the program run will not be shown correctly in the Serial monitor;
* generally, in projects, the serial communication shall be established as soon in the runtime as possible, to show even the earliest runtime bugs. This is why some people would expect serial\_init() to be called as the very first. Yet, watchdog\_init() has a priority because not only watchdog detects problems, but also can solve them by restarting the device. The serial\_init() function itself is not imune to problems at all. Then there is display\_init() which is explained in the previous bullet;
* file\_sys\_init() does not have to be exactly in this place, but it surely should be called after serial\_init() since it may output some Serial debugging messages. Also, file\_sys\_init() shall be called before power\_down\_recovery() since this function may initiate the process of lost variables recovery from the file system – and for that, as you may imagine, it needs the file system to be already initialised;
* buttons\_init() placing is not strictly dependant on other functions, but still has some logic to it. From one side, it is nice to give the user control over the device as soon as possible, hence the function gets called relatively early in the runtime. From the other side, this function initialises Interrupt Service Routines (meaning that pushing the buttons interrupts the normal program execution), that is why it was decided to allow all the communication ports and the file system to initialise before ISRs do – without interrupts. I am not saying that an interrupt happening during e.g. the file system initialisation may necessarily cause fails, but there is a chance. And since ISR bugs are also tricky to debug, it is better to simply avoid this risks no matter how small they are.
* battery\_init() has a somewhat similar logic to it: it has to be called before battery\_check() for obvious reasons, but other than that it could be moved around. Inside of it, battery\_init() calls for ADC initialization. It has been rumoured that in certain cases the actual ADC hardware initialization, which happens behind the scenes, may take longer than the initialization function execution, which in turn may ruine ADC measurements made exactly after ADC initialization call. This claim was not tested in this particular project. But, just to be safe, it was decided to separate battery\_init() (which makes the ADC initialization call) and battery\_check() (which makes ADC measurements) with another function – in this case power\_down\_recovery() – in order to create an artificial time delay;
* power\_down\_recovery() can detect and handle brown-outs. Since brown-outs limit the program runtime to a split of a second, power\_down\_recovery() has to be called as early as possible. On the other hand, in reality, a brown-out would be an extremely rare occurance. In the software, there are other mechanics implemented to prevent the battery from discharging down to the level where brown-outs may start happening.  
  In any case, power\_down\_recovery() shall be called before battery\_check(), because if there is a brown-out, it is way too late to measure the battery charge. Also, since power\_down\_recovery() triggers recovery of lost variables from the file system after device hard reset, it shall be called after file\_sys\_init();
* battery\_check() can put the device into extensive sleep if the battery charge is too low. This is why this function should be called before any serious work, like connecting to Wi-Fi and checking the Telegram chat, starts being executed. On the other hand, it should not be put before power\_down\_recovery() – explained in the previous bullet;
* telegram\_check() uses Wi-Fi connection, which is power-demanding – this is why the function shall be called after battery\_check(), so we already know that the device has enough battery charge for a wireless connection. telegram\_check() works with the file system, so it shall be called after file\_sys\_init(), too. telegram\_check() can change the state of the OTA flag (essentially, activation or deactivating OTA), so the function shall be called before ota\_init();
* ota\_init() has to be as close to ota\_waiting\_loop() as possible and it has to be inside setup(). The position of this function is so optimal, that I would not consider moving it under any circumstances.

Now, about **loop()**.

In a typical Arduino program, execution starts when the device is powered on. First, the code inside setup() runs once, and then the code inside loop() runs repeatedly, over and over, until the device is powered off. In other words, in most Arduino projects, the program cycles endlessly inside loop().

In this project, things work differently. The device uses a power-saving mode called Deep Sleep, which allows it to turn itself off and back on automatically. Each time the device wakes up, it starts fresh: first running setup(), then running loop() once, and finally going back to sleep again. This means the program cycle includes both setup() and loop(), instead of looping only inside loop().

It is important to note that the name loop() is required by the Arduino framework. Even though in this project loop() executes only once per cycle, the function must still be called loop().

Inside loop() you may find ota\_waiting\_loop() and pathfinder(). ota\_waiting\_loop() has to be as close to ota\_init() as possible while still remaining inside loop(). I do not have a better explanation for this arrangement other than this is how it is required to be by the ArduinoOTA library. Since pathfinder() also puts the device into Deep Sleep, it shall be the last function to be called. Anything put after pathfinder() will not ever be executed.

Finally, **pathfinder()**. It uses rtc\_g.exam\_status boolian variable as a flag to decide if to run exam mode that program cycle or not. Cluster number mode gets to run always since it is the default mode. Exam mode fully handles all that related to an exam including calculating how long to sleep for to wake up from Deep Sleep exactly after the exam. If the Exam mode runs, it will put the device to sleep itself, meaning that the program cycle would end in the Exam mode. There are a few points in the program where a program cycle ends and the device goes into Deep Sleep – here, in pathfinder() it is one of them.

# 42-smart-cluster-sign.h file

One thing in the file that stands out is the "ota.h" inclusion command being at the very bottom. It is not a mistake. It is the result of a workaround that allowed to use the basic Arduino Over-The-Air functionality with Deep Sleep. There might be a separate chapter on this workaround in this documentation. But here and now, it is important to say only that if the inclusion command was moved elsewhere, OTA functionality would not work any more.

# battery\_management.cpp file

—

# bitmap\_library.h file

Every unsigned char array is an image. In the head of the file you may see the table with the images information. Location represents the number of the row the image array starts from.

# buttons\_handling.cpp file

The pins D3 and D9 are the only two pins which are not physically connected to anything, in other words they are floating. Untreated floating pins may cause hardware related bugs. Usually, such pins would be physically connected to the ground with resistors. Luckly, ESP32 has internal resistors that can be programmed to connect the pins to the ground with one line of code per pin. This is why in the buttons\_init() function these two particular pins are set to be pulled down to the ground. Like this, these pins are much less likely to cause any troubles. If at some point you decide to use one of the pins, you absolutely can do so, just reprogram them the way you need it.

# cluster\_number\_mode.cpp

—

# config.h

—

# constants.h

—

# credentials-example.h

—

# display\_handling.cpp

The display\_cluster\_number() is a big and complicated state machine. It is one of the paramount functions in the project. In order to understand how the project works, it is crucial to understand how this function works.

The function manages states of the two parts of the cluster number display: the cluster number image on the left and the side notes area on the right. In order to achieve that, the function not only accepts input with information what to draw on the display, but also always remembers what is currently drawn on the display.

The display\_cluster flag is how the function remembers whether the cluster number image is on the display or not. The displaying\_now variable is how the function remembers what is currently drawn in the side notes area. Both variables are static and employ the RTC\_DATA\_ATTR attribute – that allows them to keep their data even over Deep Sleep.

At the very top, the function decides if anything needs to be drawn at all. It is possible that the function is asked to draw something that is already drawn on the display. In such case, drawing will be simply skipped.

Then, for the cluster number image, the function checks if the image is currently on the display. If not, the image gets drawn and the event gets remembered. If the cluster number image was not on the display until now, then nothing is currently drawn on the side notes area either. So, with the next line of code, the function unblocks drawing of all of the side notes.

Finally, the function decides what side note to draw and records what has been drawn into the displaying\_now variable.

The clear\_display() function does not actually get used, as there always should be something drawn on the display.

# exam\_mode.cpp

The exam\_mode() function contains three time comparisons, that may be confusing at the first glance. We need them just to be sure that the device displays correct image at correct time even if the Exam mode was accidentally entered at an unusual time, e.g. mere minutes before the exam starts.

Drawing on colored E-ink displays takes substential time (around 25 seconds), so if there is less that 10 minutes (600.000 milliseconds) before an exam, there is no point in showing the pre-exam warning sign just to replace it with the exam sign a couple of minutes later. Because when the people standing around see the display flashing for 25 seconds and then flashing for 25 seconds again after a couple of minutes, they will be thinking the Sign is broken.

This is why we check if there is more than 10 minutes left till an exam. If so, the pre-exam warning sign gets drawn onto the display. If it is less than 10 minutes, the device simply waits till 25 seconds before the exam with whatever is already on the display. If it is even less than 25 seconds, it means that the device is already late and it needs to draw the exam sign as soon as it is physically possible.

The call for an exam to start is in the very last line of the function. The call returns the exact time the device shall sleep for in order to wake up precisely at the end of the exam.

# file\_system.cpp

—

# globals.h

The relations between the globals.h and globals.cpp files might not be completely obvious, so here is the description of what is going on there. Global objects and configuration structures are declared as extern in the header (globals.h) file to make them accessible across multiple translation units. Their definitions and initial values are provided in the corresponding globals.cpp file.

# globals.cpp

The relations between the globals.h and globals.cpp files might not be completely obvious, so here is the description of what is going on there. Global objects and configuration structures are declared as extern in the header (globals.h) file to make them accessible across multiple translation units. Their definitions and initial values are provided in the corresponding globals.cpp file.

# intra\_interaction.cpp

In the request\_exams\_info() function, the time section in hours is hardcoded (currently, from 05:00:00 in the morning till 21:00:00 in the evening) in the request message to the server. It should be kept in mind that these hours refer to the Intra server time zone (which is Paris, France – GMT+1 in winter and GMT+2 in summer), not to the time zone the Sign is located in. This is not a problem if the Sign is in an European campus, but it would completely mess everything up for campuses in Asia and Australia. E.g. 05:00 in Paris is 13:00 in Seoul, South Korea and 21:00 in Paris is 4:00 in the morning of the next day for Seoul, South Korea. If you ever decide to refactor that part into an actual logic, do not forget that some countries do not switch between winter and summer time, but France surely does.

# ota.h

Despite containing function bodies instead of what you usually expect to find in a header file, ota.h has to be a header file format. In short, it is made this way, so this file can be included in the project from the main header file 42-Smart-Cluster-Sign.h, and so it is possible to use include guards to prevent the OTA code to be copied into every single cpp file in the project.

In detail, because of the intricacies of how the OTA library works, the functions of the OTA functionality (the ones you may see in the ota.h file) must be placed right above the setup() function. That would have absolutely bloated the src.ino file, so it was decided to alocate the functions of the OTA functionality into a separate file and include it from the src.ino file. Unfortunately, despite multiple tries it did not work this way, probably due to how Arduino IDE works. Through trial and error the solution you see now in the project was found. It is not the most elegant one, but it checks the main boxes: the functions of the OTA functionality are located in a separate file for readability, during compilation those functions get included right above the setup() function, the solution does not cause compilation issues, OTA functionality actually works (when not blocked by a firewall).

# power\_down\_recovery.cpp

The functionality mostly relies on the ESP-IDF API functions. More on that exact functionality here: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-reference/system/misc_system_api.html#_CPPv418esp_reset_reason_t> — under esp\_reset\_reason() and Enumerations.

# telegram\_bot.cpp

You might see the following call *bot.sendMessage(String(rtc\_g.chat\_id), message, "")* throughout the file a lot, less elsewhere in the project files. That may make you think why not refactor the *rtc\_g.chat\_id* variable into a *String* type and remove all of those explicit casts into String objects. Do not do that! The *rtc\_g.chat\_id* variable is stored in the RTC memory that has got extremely limited space. String objects take simply too much space to be stored there. RTC memory overflow will inevitably cause a wide range of issues throughout the whole project!

# telegram\_compose\_message.cpp

—

# time\_utilities.cpp

—

# utils.cpp

—

# watchdog.cpp

All the functions here are wraps around ESP-IDF watchdog API with added debugging messages and improved readability. More on ESP-IDF watchdog here under “Task Watchdog Timer (TWDT)” and “API Reference”: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-reference/system/wdts.html>

[How to Get Exams Info from Intra.]

The Smart Sign does it in the following 6 steps:

1. connects to Wi-Fi,
2. connects to the 42 Intra server,
3. asks the server for a temporary access token using the UID and the Secret,
4. retreives the temporary access token from the server response,
5. asks the server for exam information for a particular campus, a particular cluster, on a particalar date,
6. retreives the exam information from the server response.

For testing purposes, this process can be recreated on a computer, in Terminal using Curl:

1. enter these variables into the Terminal

**CLIENT\_ID=**put\_your\_42\_API\_app\_UID\_number\_here

**SECRET\_ID=**put\_your\_42\_API\_app\_Secret\_token\_here

2. ask the 42 Intra server for a temporary access token

**curl -X POST --data "grant\_type=client\_credentials&client\_id=${CLIENT\_ID}&client\_secret=${SECRET\_ID}" https://api.intra.42.fr/oauth/token**

3. copy the access token from the server response and enter it as a variable into the Terminal

**TKN=**put\_received\_access\_token\_here

4. ask the server to send you the information about exams in the cluster C3 and put it into a .json file. 56 is the ID of the 42 Prague campus. Curl does not like square brackets [ ] in its calls, so they need to be escaped with a backslash \.

**curl -H "Authorization: Bearer $TKN" "https://api.intra.42.fr/v2/campus/56/exams&filter\[location\]=C3" > c3\_exams.json**

If you want to filter the results down to the exact date, as the Smart Sign does, use the following call instead.

**curl -H "Authorization: Bearer $TKN" "https://api.intra.42.fr/v2/campus/56/exams?filter\[location\]=C3&range\[begin\_at\]=2024-07-12T05:00:00.000Z,2024-07-12T22:00:00.000Z" > c3\_exams1.json**

5. this command opens the .json file in the Terminal

**python -m json.tool < prague\_exams.json | grep "begin\_at" | tr -d " ," | awk -F '"begin\_at":' '{print("["++count"]:", $2)}'**

[example of the 42 server access token response as the smart sign sees it.]

HTTP/2 200

date: Thu, 11 Jul 2024 13:19:37 GMT

content-type: application/json; charset=utf-8

cache-control: no-store

etag: W/"77a2df7a4e20f5f76e6364d36bc76e8a"

pragma: no-cache

set-cookie: \_mkra\_stck=15e20a8020c702e70007eb1e185a06fb%3A1720703982.2018037; path=/; max-age=10; expires=Thu, 11 Jul 2024 13:19:47 -0000; HttpOnly

status: 200 OK

vary: Origin,Accept-Encoding

x-rack-cors: preflight-hit; no-origin

x-request-id: 3d153728-82b5-48a0-84e7-7c1f1efe598a

x-runtime: 0.076367

cf-cache-status: DYNAMIC

report-to: {"endpoints":[{"url":"https:\/\/a.nel.cloudflare.com\/report\/v4?s=5%2Bb21KLqrzLETXPtKW2gerMAMrEPjiLAWT6eRUKeyuOVy3b5pvEr6Tc7D%2BMB%2BB4gqUHrTyXWaYy01CmZjQqUGReP7COyDKfBhKpl75Kwd%2FWrMWCVZD%2FkWhvM1iHF0V43hw%3D%3D"}],"group":"cf-nel","max\_age":604800}

nel: {"success\_fraction":0,"report\_to":"cf-nel","max\_age":604800}

server: cloudflare

cf-ray: 8a191610cec6bc03-FRA

{"access\_token":"03e4cb9b861dad6c49f2267cf97bd18a942507efa7840dc971008d264596cf89","token\_type":"bearer","expires\_in":6564,"scope":"public","created\_at":1720703340,"secret\_valid\_until":1722585613}

[example of the 42 server EXAM INFORMATION response as the smart sign sees it.]

HTTP/1.1 200 OK

Date: Thu, 28 Nov 2024 06:10:13 GMT

Content-Type: application/json; charset=utf-8

Transfer-Encoding: chunked

Connection: close

Cache-Control: max-age=0, private, must-revalidate

etag: W/"4dc462b36f78c9e055076113bae0d605"

status: 200 OK

vary: Origin,Accept-Encoding

x-application-id: 67990

x-application-name: 42PRAGUEAPI SCREEN

x-application-roles: None

x-content-type-options: nosniff

x-fast: false

x-frame-options: SAMEORIGIN

x-hourly-ratelimit-limit: 1200

x-hourly-ratelimit-remaining: 1199

x-page: 1

x-per-page: 30

x-rack-cors: preflight-hit; no-origin

x-request-id: 1d409b60-b763-4e10-af5a-8fe35593b80d

x-runtime: 0.196901

x-secondly-ratelimit-limit: 2

x-secondly-ratelimit-remaining: 1

x-total: 1

x-xss-protection: 1; mode=block

cf-cache-status: DYNAMIC

Server: cloudflare

CF-RAY: 8e9831883e36b353-PRG

9fa

[{"id":21213,"ip\_range":"10.11.0.0/16,10.12.0.0/16,10.13.0.0/16","begin\_at":"2024-11-28T14:00:00.000Z","end\_at":"2024-11-28T17:00:00.000Z","location":"C3","max\_people":25,"nbr\_subscribers":4,"name":"EXAM STUD","created\_at":"2024-11-22T08:20:02.386Z","updated\_at":"2024-11-27T23:40:57.806Z","campus":{"id":56,"name":"Prague","time\_zone":"Europe/Prague","language":{"id":2,"name":"English","identifier":"en","created\_at":"2015-04-14T16:07:38.122Z","updated\_at":"2024-11-18T11:22:47.733Z"},"users\_count":1335,"vogsphere\_id":52,"country":"Czech Republic","address":"AFI CITY TOWER Kolbenova 1021/9 Praha 9 - Vysočany","zip":"19000","city":"Prague","website":"https://42prague.com","facebook":"https://www.facebook.com/42Prague","twitter":"","active":true,"public":true,"email\_extension":"42prague.com","default\_hidden\_phone":false},"cursus":[{"id":21,"created\_at":"2019-07-29T08:45:17.896Z","name":"42cursus","slug":"42cursus","kind":"main"},{"id":21,"created\_at":"2019-07-29T08:45:17.896Z","name":"42cursus","slug":"42cursus","kind":"main"},{"id":21,"created\_at":"2019-07-29T08:45:17.896Z","name":"42cursus","slug":"42cursus","kind":"main"},{"id":21,"created\_at":"2019-07-29T08:45:17.896Z","name":"42cursus","slug":"42cursus","kind":"main"},{"id":21,"created\_at":"2019-07-29T08:45:17.896Z","name":"42cursus","slug":"42cursus","kind":"main"}],"projects":[{"id":1320,"name":"Exam Rank 02","slug":"exam-rank-02","difficulty":0,"parent":null,"children":[],"attachments":[],"created\_at":"2019-07-29T09:05:05.890Z","updated\_at":"2024-11-25T08:53:45.680Z","exam":true,"git\_id":null,"repository":null},{"id":1321,"name":"Exam Rank 03","slug":"exam-rank-03","difficulty":0,"parent":null,"children":[],"attachments":[],"created\_at":"2019-07-29T09:05:15.263Z","updated\_at":"2024-11-25T08:56:10.466Z","exam":true,"git\_id":null,"repository":null},{"id":1322,"name":"Exam Rank 04","slug":"exam-rank-04","difficulty":0,"parent":null,"children":[],"attachments":[],"created\_at":"2019-07-29T09:05:24.256Z","updated\_at":"2024-11-25T08:56:32.456Z","exam":true,"git\_id":null,"repository":null},{"id":1323,"name":"Exam Rank 05","slug":"exam-rank-05","difficulty":0,"parent":null,"children":[],"attachments":[],"created\_at":"2019-07-29T09:05:32.360Z","updated\_at":"2024-11-25T08:56:53.071Z","exam":true,"git\_id":null,"repository":null},{"id":1324,"name":"Exam Rank 06","slug":"exam-rank-06","difficulty":0,"parent":null,"children":[],"attachments":[],"created\_at":"2019-07-29T09:05:39.838Z","updated\_at":"2024-11-25T08:57:29.269Z","exam":true,"git\_id":null,"repository":null}]}]

0

[Exam Simulation and how to use it.]

The project code contains a so-called Exam Simulation logic, so it is possible to test how the device behaves during exams without the necessity to wait for an actual exam. Here is how to do it.  
  
1. In the project files go to config.h and uncomment the following line

//# define EXAM\_SIMULATION

2. Now, go to the utils.cpp file and scroll all the way down. There you will find the exam\_simulation() function.

3. In the function, manually change the exam beginning and ending time the way you want it.   
  
When changing time, mind the time zone difference between your campus location and the Intra server location. The exam beginning and ending time must be stated in the time of the Intra server location. Let’s demonstrate on an example:

- Your are testing a Sign for the 42 SEOUL campus, located in Seoul, South Korea;

- By googling, you discover that Seoul, South Korea has a time zone UTC+9. Also, South Korea does not switch between summer and winter time;

- By this moment, you already know that the Intra server is located in France. France DOES switch between summer and winter time and you should account for that. In winter, France uses UTC+1 time zone, and in summer France uses UTC+2 time zone. Let’s assume that at the time of this example the summer time applies. So, we will use UTC+2 for the Intra server;

- Simple math operation shows that the time difference between the 42 SEOUL campus and the Intra server location is (9 – 2 =) 7 hours. It means, that at the time of this example, the Intra server clock is 7 hours behind the 42 SEOUL campus clock;

- Let’s assume that you decided to test the Sign for a simulated exam starting at 14:00 (Seoul time) and ending at 15:30 (again, Seoul time). Since we already know that the Intra server time is 7 hours behind the Seoul time, we need to deduct these 7 hours from the time of our simulated exam:

14:00 - 7 = 7:00 and 15:30 - 7 = 8:30

- So, now you know that to simulate an exam starting at 14:00 and ending at 15:30 Seoul time, in the exam\_simulation() function you need to state 07:00 as the exam begin time and to state 08:30 as the exam end time.

Of course, you need to go throug this kind of analysis only for the first time you decide to use the Exam Simulation — next time you will already know how many hours to adjust for. For the Seoul campus it is slightly more complicated, since the South Korea does not switch between summer and winter time but France does, the time difference will be constanty shifting and the Sign developer from the South Korea will always have to keep track of that. But if you are from a country that, like France, switches between summer and winter time, you do not even have to worry about that, as the switch happens roughly at the same time worldwide.

[CREATE YOUR OWN graphics.]

You can create your own graphics to be displayed on the Sign. It may look complicated at first, but once you do it at least once, it will all become easy.

Why is it complicated? Well, you cannot just drop an image into the Sign’s memory and display it. First, you have to convert your image into a C code and then put that C code into the Sign’s memory. The C code that represents an image is nothing else but a simple unsigned char array. In this array, each element represents the colour of each pixel in a bit format. Since there are a lot of pixels in an 800 x 480 image, the array will be very big, yet it will be just a simple array. Usually, they call this arrays bitmaps. If you would like to learn more, „bitmaps“ is the term you should google.

So, here is the simple instruction:

**get an image** > **convert into bitmap** > **put the bitmap into the project code** > **upload to the Sign**.

Here are the steps in details:

1. **Create your image**

- it has to be 800 x 480 pixels or smaller,

- JPEG/JPG format only,

- it can be only Black-and-White or Red-Black-and-White.

2. **Convert your image into a bitmap**. To do that, you may use the tool located in this repository:

tools > epd\_image\_converter

- download the tool from GitHub onto your local machine,

- put all the images you want to convert into the epd\_image\_converter folder,

- open the epd\_image\_converter folder in the Terminal,

- use the following command to create Black-and-White image bitmap:

./epd\_image --BW --DITHER Your\_Image\_Name.jpg Any\_Name.h

- use the following command to create Red-Black-and-White image bitmap:

./epd\_image --BWR --DITHER Your\_Image\_Name.jpg Any\_Name.h

- Your\_Image\_Name.jpg is the image you want to convert into a bitmap,

- Any\_Name.h is the file where the bitmap array will be generated,

- Black-and-White images get converted into 1 bitmap,

- Red-Black-and-White images get converted into 2 bitmaps: Red-White and Black-White.

3. **Put your image bitmap into the project code**

- open all the Any\_Name.h files that you generated in the previous step,

- give the arrays apropriate names,

- open the project code files and find the bitmap\_library.h file,

- copy-paste the arrays from the the Any\_Name.h files into the bitmap\_library.h file of the project.

4. **Use your bitmap to be displayed on the Sign**

- go to the display\_handling.cpp file of the project and find the apropriate function to display your image,

- implement when and how your image shall be displayed.

5. **Upload the new software to the Sign and watch it run.**

[HOW TO DRAW ON THE DISPLAY.]

COORDINATES, SIZES IN PIXELS AND ROTATION

In the **setRotation(uint8)** function you may set how the window you want to output on the display will be turned compared to the display physical orientation. All the displays have their default physical orientation, but it may vary from display to display, so finding out the display's default physical orientation is the starting point for any work with display graphics. It is easy to do: take a look at the display's physical appearence, note the side where a ribbon of wires sticks out of it — that's the bottom side of this display's default physical orientation. Knowing that, we now can set the rotation:

setRotation(0) — for the default rotation, it is equal to the display default physical orientation

setRotation(1) — will rotate the window 90 degrees clockwise comparing to the display default physical orientation

setRotation(2) — will rotate the window 180 degrees (or simply flips it up-side-down) comparing to the display default physical orientation

setRotation(3) — will rotate the window 270 degrees clockwise comparing to the display default physical orientation

Keep in mind that however you rotate the window, its system of coordinates always rotates with it. So, the 0,0 coordinates are always in the TOP LEFT corner of your window, but not necessarily in the top left corner of your display.

When dealing with the Partial Update windows, it is important to remember about one unobvious limitation of the e-paper controller: partial update window size and position are on byte boundary with physical x direction. Meaning that the controller may set the horisontal window boundaries only on every 8th pixel comparing to its default physical orientation. Let's summ it up with a simple rule that is easy to follow:

The value of x and the value of width should be multiple of 8, for rotation 0 or 2,

the value of y and the value of hight should be multiple of 8, for rotation 1 or 3.

The function **setPartialWindow(x, y, width, height)** allows us to create a window of any size and, instead of the whole display, update only the content of this window. It is very useful feature since it is much faster than the conventional full display update with the setFullWindow() function. Moreover, this type of an update does not flicker. So, if you need to get rid off of the flickers between the slides, with setPartialWindow(x, y, width, height) you may set the whole display as a Partial Update window. But be careful as THE FIRST UPDATE AFTER TURNING ON OR RESET SHALL ALWAYS BE THE FULL DISPLAY UPDATE with the setFullWindow() function. Otherwise the display may be irrevertably stuck on the last image for ever.

It may be useful to note the relationship between the setPartialWindow(x, y, width, height) and the setRotation(uint8) functions. Some may think that setRotation(uint8) may turn a Partial Update window created with the setPartialWindow(x, y, width, height) function. And it is true, but only to some extent. setRotation(uint8) does not turn Partial Update windows, but it actually turns the whole display window orientation with Partial Update windows inside of it. That is why when we change the display orientation with the setRotation(uint8) function, we also need to remember that the system of coordinates for x and y in the setPartialWindow(x, y, width, height) also changes. And when the system of coordinates changes, the understanding of width and height changes with it. E.g. if while setRotation(0) the window's width is 800 and the window's height is 480, then while setRotation(1) the window's width becomes 480 and the window's height becomes 800.

For some unexplainable reason when it comes to the text body coordinates, their logic is very different from windows and images. The x and y coordinates point to the bottom left corner of the first line of text even if there are multiple lines of text. In comparison, the common logic for the windows and bitmaps coordinates are that their x and y point to the top left corner of the image. So, if you used display.setCursor(x, y) command to output the word HELLO onto the display, the x and the y there would point to the very bottom left pixel of the letter H.

The coordinates logic is also tricky when it comes to the command

**display.getTextBounds(output, 0, 0, &text\_box\_x, &text\_box\_y, &text\_width, &text\_height)**

This command can view any text as the smallest box that can accomodate the given text and give you all the needed information to create such a box on the screen. It is very useful when you want to update a piece of text or a single character on your display instead of the whole display. The function takes a few parameters: the text in the form of a String varible (here it is "output"), x and y coordinates of the area you want to place the text (here they are "0, 0" - the top left corner of the display), pointers to x and y coordinates (this is how you will get x and y coordinates for the top left corner of the box for the text), pointers to the width and height (this is how you will get width and height values of the box for the text). It is not required to change the "0, 0" coordinates because this function does not place anything anywhere - it uses this values only for calculations. Though, it is important to remember, that when you use the default "0, 0" coordinates, the value in the text\_box\_y variable will be NEGATIVE. Why does this happen? Exactly because of the difference in the coordinates logic between texts and windows/bitmaps/boxes. If you remember, x and y for texts point to the BOTTOM left pixel of the text; but x and y for windows/bitmaps/boxes point to their TOP left pixel. The display.getTextBounds tries to compensate this logic difference. E.g. if you were to place the word HELLO with the 10 size font into the top left corner (0, 0) of the display, you would need to call setCursor(0, 10); but if you were to creare a partial update window in the the top left corner (0, 0) of the display to update that text, you would need to call setPartialWindow(0, 0, text\_width, 10).

But noone has time to deal with different coordinates logic! We want to give the function the text and just one set of x,y coordinates and the text shall simply aline with my partial update window automatically! That's where the getTextBounds may help. In our example our target display coordinates are x = 0, y = 0 (the top left corner), so let's place HELLO in there with a partial display update:

display.setTextSize(10);

display.getTextBounds("HELLO", 0, 0, &text\_box\_x, &text\_box\_y, &text\_width, &text\_height);

// now text\_box\_x has value 0, text\_box\_y == -10, text\_width == HELLO-width value (I don't know it), text\_height == 10

display.setPartialWindow(x, y, text\_width, text\_height);

// that creates a window in the top left corner of the display, HELLO-width pixels wide and 10 pixels high

display.setCursor(x - text\_box\_x, y - text\_box\_y);

// this is how we compensate for the text coordinates logic: for x (0 - 0 = 0) and for y (0 - (-10) = 10) coordinates

display.print("HELLO");

[The Intricacies of time keeping.]

The project has two sources of time-related data. There is an NTP server for actual time and the Intra server for exam time.

Actual time data from the NTP server can be updated at any time by calling the get\_time() function – it gets stored into 5 global variables: com\_g.hour, com\_g.minute, com\_g.day, com\_g.month, com\_g.year. The time data in those global variables is already adjusted to the time zone of the Sign and to the summer/winter time.

Exam time data from the Intra server gets updated only when the fetch\_exams() function is called. The exam time data is then stored in the following 4 global variables: rtc\_g.exam\_start\_hour, rtc\_g.exam\_start\_minutes, com\_g.exam\_end\_hour, com\_g.exam\_end\_minutes. There is no exam date variable, because the time data is searched only for the current day. The exam time data in those global variables is already adjusted to the time zone of the Sign and to the summer/winter time.

It is worth noting that all of the time-related data does not simply come nicely adjusted for this particular device to use. All the adjustments happen on the device. To understand how it works, it is important to have information about the sources of the raw data.

The NTP server provides only standart UTC time. It never adjusts for Summer or Winter time. It does not care about time zones. All of that – both summer/winter time change and time zone adjustment – is handled on the device by the Time library.

The Intra server behaves differently – it always switches between Summer and Winter time automatically and sends already adjusted time, so we do not have to care about this part. However, the Intra server does not care about time zones and additionally uses the time zone of its actual location (UTC+1 — CET, Central European Time in winter and UTC+2 — CEST, Central European Summer Time in summer) forcing the device to adjust to it locally. Time zone adjustment happens in the get\_exam\_time() function.

Since the Intra server provides not only exam time, but also actual time, it is possible to refactor the project to use the Intra server as the sole source for all the time-related data. The need of adjusting for time zones applies.

Located in the config.h file, there is a very important time-related macro TIME\_ZONE. This macro is the place where you have to manually write the time zone of the campus where you use the Sign. To do that correctly, you should know that for the same one country time zone shifts when time gets switched between winter time and summer time. E.g.: in winter, France uses Central European Time and has the UTC+1 time zone; but in summer France uses Central European Summer Time and has the UTC+2 time zone.

So, if in the country of your campus location it is common to switch between winter time and summer time, make sure to use the WINTER-TIME TIME ZONE in the macro. Include "-" sign if it applies to the time zone of your cluster. Do not include "+" sign at all.

[SERVICE MESSAGES MEANING.]

[LIBRARIES AND THEIR USE.]

The project was built in Arduino IDE 1.8.19.

It uses board 'esp32' version 3.0.7

The libraries in bold are explicitly included in the project.

|  |  |  |
| --- | --- | --- |
| **Arduino.h** |  | String variables manipulations |
| **LittleFS** | 2.0.0 | stores data even without electricity (Telegram chat number, Secret, OTA flag value) |
| FS | 2.0.0 | dependency for the LittleFS library |
| **ArduinoOTA** | 2.0.0 | for the Over The Air update functionality |
| **WiFiUdp** | 2.0.0 | dependency for the ArduinoOTA library |
| **ESPmDNS** | 2.0.0 | dependency for the ArduinoOTA library |
| Update | 2.0.0 | dependency for the ArduinoOTA library |
| **time.h** |  | gets NTP Server date and time; deciphers UNIX timestamp for the SECRET expiration date |
| **stdio.h** |  | provides printf() function for the DEBUG macro |
| **stdint.h** |  | provides fixed-width integer types |
| **esp\_system.h** |  | allows to use ESP-IDF native functions |
| **esp\_sleep.h** |  | allows to use the Deep Sleep power-saving functionality |
| **driver/adc.h** |  | for battery charge measurements |
| **esp\_task\_wdt.h** |  | program execution watchdog |
| **Wire** | 2.0.0 | for SPI reconfiguration in the ft\_display\_init function |
| SPI | 2.0.0 | dependency for the Wire library |
| **GxEPD2\_3C** | 1.5.2 | 3-coloured version of the GxEPD2 library for e-paper displays |
| **GxEPD2\_BW** | 1.5.2 | dependency for the GxEPD2\_3C library |
| Adafruit\_GFX\_Library | 1.11.8 | dependency for the GxEPD2\_3C library |
| Adafruit\_BusIO | 1.14.4 | dependency for the GxEPD2\_3C library |
| **Fonts/FreeSansBold24pt7b.h** |  | the fonts come from the Adafruit GFX library which gets called by the GxEPD2 library |
| **WiFi** | 2.0.0 | for Wi-Fi functionality |
| **WiFiClientSecure** | 2.0.0 | for secure HTTPS requests |
| **UniversalTelegramBot** | 1.3.0 | Telegram bot; for wireless SECRET update and low battery notifications |
| ArduinoJson | 6.21.3 | dependency for the UniversalTelegramBot library |

[BUGS AND SUGGESTIONS HOW TO FIX THEM.]

***Display does not work / does not draw an image.***

* you may have miscalculated the image coordinates and the image gets drawn outside of the display field of coordinates. Remember that setRotation also rotates the display field of coordinates. Remember that coordinates always point to the top left corner of an image, but for a text it is the bottom left pixel of the first character in the first line.
* you may have misaligned the image with the partial update window if you are using one. Remember that even when you draw an image in a partial update window, you use the whole display field of coordinates to place it; a partial update window does not have its own field of coordinates.
* the display driver memory may be full. Run your program and open the Serial monitor. When an image gets drawn on the display, in the Serial monitor it says “Updating xxxxxxxxxx” where xxxxxxxxxxxx is a very long number that can be different each program run. Among all the outputted messages find the “Updating” message that corresponds to your image being drawn. Now look higher and find the “Power on” message. Keep looking higher and find the “Power off” message, it should not be separated from your “Updating” message by other “Updating” messages. If you cannot find it, it means that this is a bug. To fix it, in your code use the function to force the display to power off before drawing the image. The display will power back on automatically.

***Flash memory became too small to accomodate the software.***

* ESP32C3 comes with 4MB of onboard memory. This project‘s default memory partition scheme "Minimal SPIFFS (1.9MB APP with OTA/190KB SPIFFS)" provides exactly 1 966 080 Bytes for the software program, another 1 966 080 Bytes as an OTA buffer and 194 560 Bytes for the file system files storage space.
* Commenting out the **#pragma GCC optimize ("O3")** line in config.h will make the compiler optimize the code for lower software image size instead of faster program performance. That may win over about 2-3 KB of memory space.
* Before compiling the software image, in the „Tools“ menu of Arduino IDE, go to the Core Debug Level and switch it from „Verbose“ to „None“ — this simple step can free up to 3% of the memory space.
* Downgrade the project from the esp32 board version 3.0.7 to the esp32 board version 2.0.16. This step may require to refactor the watchdog and the battery management functionalities, but it may free up to 10% of the memory space.
* In the „Tools“ menu of Arduino IDE, go to the Partition Scheme and choose the option „Huge APP (3MB No OTA/1MB SPIFFS)“. Doing so will free up a whopping 30% of the memory space. Unfortunatelly, the Over-The-Air update functionality will not have a buffer for its work and so will no longer be available. At this point the OTA functionality code may well be removed from the project.

***Some RTC variables loose thier values over the Deep Sleep.***

* There could be more than one reason to it, but most likely it is caused by the RTC memory overflow. Try removing some less important variables from the RTC domain. The String Object type variables (or simply String variables) are known to have a big overhead and thus to take a lot of memory space. It is encouraged not to use them in the RTC domain at all and to use C-style char arrays instead.

***DEBUG\_PRINTF does not output a message.***

* the DEBUG\_PRINTF macro cannot output String type variables natively. To do that, you need to explicitly cast the String variable into the C-style string with c\_str() command. Look for examples in the program code.

***Serial monitor is empty / outputs gibberish.***

* check the DEBUG macro in the config.h file. The DEBUG definition should not be commented out for the Serial output to work. Additionally, you can set the Core Debug Level to "Verbose" in the Arduino IDE Tools to get detailed information about the firmware processes.
* make sure that the baud rate in the Serial monitor is set to the same baud rate as in the config.h file.
* you may encounter such behaviour right after the software update. It is normal. Try closing and opening again the Serial monitor window. If that does not help, push the Reset ("R") button on the module.

***Serial monitor skips some messages / does not show some messages.***

* it is a common situation at the beginning of the program. Serial communication between the computer and the microcontroller needs time to stabilise and synchronise itself. ESP32-C3 USB Serial is especially prone to this issue. To overcome it, increase the delay inside of ft\_serial\_init() or add a few empty messages to be outputted after the Serial.begin() command. You may well try to implement both of the suggested solutions at the same time.

***Wi-Fi does not connect / reconnect without apparent reason.***

* thoroughly check your network SSID and password spelling. Surprisingly, it is a very widely spread cause. A single character written small instead of capital may easily prevent you from connecting.
* make sure not to use ft\_delay() in any of your functions responsible for connecting or reconnecting to Wi-Fi. The ft\_delay() function not only delays the program execution but also puts the microcontroller's inner Wi-Fi module to sleep. Using ft\_delay() in functions responsible for retrieving information from the Internet may result in unexpected behaviour. If you are not sure that using ft\_delay() is safe in your particular function, use delay() instead.

***OTA does not work. Cannot see the device in the ports list.***

* make sure that the Sign and your computer are connected to the same Wi-Fi network and to the same Wi-Fi modem within that network. In the Telegram chat prompt the Sign with the „/status“ command to see the MAC address of the Wi-Fi modem it is currently connected to.
* try closing and reopening Arduino IDE.
* the school firewall may be blocking OTA connection. Ask your campus system administrator if it could be overcome.

***Adding multiple Strings together with the “+” command causes compilation error.***

* strangely, sometimes the compiler may not like it in one part of the code and be completely fine with it in another. The solution is to explicitly cast the variable after the first “+” command into String with the String(your\_variable\_or\_text) command. Understandably, it is strange to cast a String variable into String, but it works.

***WARNING: Skipping SSL Verification. INSECURE!***

* not a bug.
* this message appears when connecting to the Intra server and is caused by the following line in the intra\_interaction.cpp file: „client1.setInsecure();“.
* one one hand, it can be solved by getting and setting up a certificate for this connection. On the other hand, it does not affect the program run at all and can be ignored.

***setSocketOption(): fail on 0, errno: 9, "Bad file number"***

* a minor issue and does not necessarily indicate a problem with the program.
* this message may appear when the Smart Sign fails the first attempt to get a server response from the Intra server and goes for the second or third attempt.
* this error can occur when you try to set a socket option on a socket that has already been closed or is in the process of being closed. This can happen during the transition between closing the previous connection and opening a new one. As long as the SSL/TLS communication with the Intra server is functioning correctly after the reconnection, this error can generally be ignored.

***spiAttachMISO(): SPI Does not have default pins on ESP32C3!***

* not a bug.
* This message appears when the microcontroller assigns pins for the display SPI port. In this project we do not use the MISO pin (thus the „-1“ value defined for the SPI\_MISO\_PIN in the constants.h file).

***401 Unauthorized. Error! Server response came without the Access Token.***

* often happens when something is wrong with the Secret token authentication, commonly with the Secret token itself. Most likely, an extra character was added to your Secret token somewhere along the way. The character may even not to be visible in the Serial monitor. It may happen when you write to or read from the filesystem files. Try using trim() on the variable (e.g. your\_string\_variable.trim();), it will remove spaces and/or new line signs at the beginning and at the end of the string.
* rarely may happen due to the Intra server maintenance. There is no solution to it but to wait.

***Compilation error: “Section .dram0.bs 'Will Not Fit In Region Dram0\_0\_seg' Region.`Dram0\_0\_seg 'Overflowed by 9648 Bytes. Collect2: Error: LD Returned 1 Exit Status”***

* it means that the program takes more RAM space than it is available. DRAM stands for Data Random Access Memory and is used for data.
* This error may be caused for example by excessive use of global variables, large arrays, big buffers, etc.
* The most likely reason for this error in this project is the display buffer being too big. The ESP32 and the ESP32-S2 are especially prone to this issue. To overcome this problem the display buffer size should be reduced. It can be done in the display instantiation.  
  Here is the default display buffer instantiation:   
  GxEPD2\_3C<GxEPD2\_750c\_Z08, GxEPD2\_750c\_Z08::HEIGHT> display(GxEPD2\_750c\_Z08(SPI\_SS\_PIN, DC\_PIN, RST\_PIN, BUSY\_PIN));  
  And here is a reduced display buffer instantiation: GxEPD2\_3C<GxEPD2\_750c\_Z08, GxEPD2\_750c\_Z08::HEIGHT/2> display(GxEPD2\_750c\_Z08(SPI\_SS\_PIN, DC\_PIN, RST\_PIN, BUSY\_PIN));
* Less likely reason for this error is an excessive use of the global scope for data. To solve it reduce the number of global variables, use the file system to store the data instead of arrays.

***Software update fails while trying to connect to the microcontroller board***

* Go into the "Tools" menu and change the Upload Speed to 115200. Sometimes the IDE automatically sets the Upload Speed to the highest value and your board may happen not to support it.

***[INTRA] Error! Server response to the Access Token request was not received.***

***[INTRA] Error! Server response to the Exam Time request was not received.***

* If any of these error messages keep appearing in the Serial monitor, it is likely that it is not enough time for the Intra server to proceed the request from the Sign. Try going into the config.h file and incrementally increasing the value of the SERVER\_WAIT\_MS macro by 500 milliseconds each time.

[NEW bugs and Future development suggestions.]

All newly discovered bugs should be documented in the "Issues" tab of the project's GitHub repository. This helps keep track of problems and facilitates community engagement in resolving them.

Any suggestions for future development or enhancements can be added to the "Suggestions for Contributions" section in the README file of the project's GitHub repository. This allows everyone to see potential improvements and participate in developing them.

[Suggestions for dealing with confidential information.]

[External Information Sources.]

NTP (time) server API documentation

<https://cplusplus.com/reference/ctime/tm/>

How to Get Date and Time with the Time.h library — article

<https://randomnerdtutorials.com/esp32-date-time-ntp-client-server-arduino/>

GDEY075Z08 display data sheet

<https://www.laskakit.cz/user/related_files/gdey075z08.pdf>

GDEY075Z08 display code example

<https://github.com/LaskaKit/Testcode_examples/blob/main/Displays/E-Paper/7-50/GDEY075Z08_GxEPD2/GDEY075Z08_GxEPD2.ino>

UC8179 - the display hardware driver - data sheet <https://www.laskakit.cz/user/related_files/uc8179.pdf>

GxEPD2 library - the display software driver - online page

<https://github.com/ZinggJM/GxEPD2>

Online forum for GxEPD2 library troubleshooting discussions

<https://forum.arduino.cc/t/good-display-epaper-for-arduino/419657>

ESP32 RAM issue discussion page

<https://github.com/espressif/arduino-esp32/issues/1163>

XIAO ESP32C3 development board instruction page

<https://wiki.seeedstudio.com/XIAO_ESP32C3_Getting_Started/>

Video tutorial about troubleshooting the ArduinoOTA library

<https://www.youtube.com/watch?v=z_btZfxrS48>

The best instruction ever on the buttons implementation with ESP32

<https://esp32io.com/tutorials/esp32-button>