

**TECHNICKÁ UNIVERZITA V KOŠICIACH**  
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

**Bezpečná aktualizácia firmvéru v senzorovej sieti na báze  
ESP32  
Diplomová práca**

**2021**

**Martin Chlebovec, Bc.**

**TECHNICKÁ UNIVERZITA V KOŠICIACH**  
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

**Bezpečná aktualizácia firmvéru v senzorovej sieti na báze  
ESP32**  
**Diplomová práca**

Študijný program: Počítačové siete  
Študijný odbor: Informatika  
Školiace pracovisko: KEMT  
Školiteľ: prof. Ing. Miloš Drutarovský, CSc.

**2021 Košice**

**Martin Chlebovec, Bc.**

## Abstrakt v SJ

Hlavnou témou diplomovej práce je využitie mikrokontroléra ESP32 v úlohe sensorového uzla, ktorý dokáže vykonávať vzdialenú aktualizáciu firmvéru cez internet s využitím WiFi konektivity cez bezpečný prenosový kanál. Proces vzdialenej aktualizácie kladie prioritu najmä na bezpečnosť, keďže sa dáta prenášajú cez internet, čo je nebezpečné prostredie pre prenos citlivých informácií a údajov. ESP32 umožňuje implementovať mechanizmy, ktoré dokážu zabezpečiť integritu (nepozmenenosť) prevzatého firmvéru zo servera dosiahnuteľného cez internet a tiež pri procese jeho bootovania. Pre prevenciu a demonštráciu ochrany pred možným typom najčastejších útokov na ESP32 boli v diplomovej práci implementované mechanizmy, ktoré dokážu zabezpečiť bootovací proces s využitím hardvérových funkcionalít, ktoré pracujú na princípe reťaze dôvernosti a využívajú kľúč pre kryptografické operácie, ktorý je zapísaný v jednorazovo programovateľnej pamäti eFuse, kde má prístup iba hardvérová funkcionálnosť. Funkcionálnosť zabezpečenia bootovacieho procesu umožňuje verifikovať, že je bootloader (zavádzač) zapísaný vo flash pamäti dôveryhodný a umožní mu prístup k ďalšej fáze bootovacieho procesu, ktorý spočíva v bootovaní a zavedení firmvéru z aplikačnej partície do RAM pamäte pre jeho úspešné spustenie. Keďže ESP32 skorších výrobných revízií neumožňuje zabrániť prevzatiu firmvéru, resp. obsahu flash pamäte cez fyzické USB-UART rozhranie, pre tento typ možného odcudzenia firmvéru bola v diplomovej práci implementovaná funkcionálnosť šifrovania flash pamäte, ktorá umožní útočníkovi prevziať len šifrovaný firmvér, ktorý nie je spustiteľný na ESP32 bez potrebného symetrického kľúča, ktorý tieto kryptografické operácie šifrovania a dešifrovania vykonáva a je bezpečne chránený v jednorazovo programovateľnej pamäti ESP32. Demonštrácia sensorového uzla využíva záznam dát z meteorologického senzora a prenos údajov na vzdialené webové rozhranie s vizualizáciou nameraných údajov v grafickom rozhraní.

## Kľúčové slova v SJ

ESP32, digitálny podpis, MCU, OTA, WiFi, mikrokontrolér, IoT, firmvér, aktualizácia

## **Abstrakt v AJ**

The main topic of the diploma thesis is the use of the ESP32 microcontroller in the role of a sensor node, which can perform remote firmware updates via the Internet using WiFi connectivity via a secure transmission channel. The remote update process places particular priority on security, as data is transmitted over the Internet, which is a dangerous environment for the transmission of sensitive information and data. ESP32 allows you to implement mechanisms that can ensure the integrity (immutability) of the firmware downloaded from the server accessible via the Internet and also during the boot process. To prevent and demonstrate protection against the possible type of the most common attacks on ESP32, mechanisms were implemented in the thesis that can secure the boot process using hardware functionalities that work on the principle of confidentiality and use the key for cryptographic operations, which is written in one-time programmable memory eFuse , where only hardware functionality has access. Boot process security functionality allows you to verify that the bootloader written to flash memory is trusted and allows it to proceed to the next phase of the boot process, which consists of booting and loading firmware from the application partition into RAM for successful startup. Since the ESP32 of earlier production revisions does not allow to prevent the download of firmware, resp. content of flash memory via physical USB-UART interface, for this type of possible theft of firmware was implemented in the thesis the functionality of flash memory encryption, which allows an attacker to download only encrypted firmware that is not executable on ESP32 without the necessary symmetric key that these cryptographic encryption operations and decryption is performed and is securely protected in the ESP32 one-time programmable memory. The demonstration of the sensor node uses the recording of data from the meteorological sensor and the transfer of data to a remote web interface with the visualization of the measured data in a graphical interface.

## **Klíčové slova v AJ**

ESP32, digital signature, MCU, OTA, WiFi, microcontroller, IoT, firmware, update

# Zadanie práce

60848

**TECHNICKÁ UNIVERZITA V KOŠICIACH**  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY  
Katedra elektroniky a multimediálnych telekomunikácií

## ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**  
Študijný program: **Počítačové siete**

Názov práce:

**Bezpečná aktualizácia firmvéru v senzorovej sieti na báze ESP32**  
Secure firmware update in sensor network based on ESP32


Študent: **Bc. Martin Chlebovec**  
Školiteľ: **prof. Ing. Miloš Drutarovský, CSc.**  
Školiace pracovisko: **Katedra elektroniky a multimediálnych telekomunikácií**  
Konzultant práce:  
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

Na základe dostupných informácií naštudujte mechanizmy umožňujúce realizovať bezpečnú aktualizáciu firmvéru na platforme mikropočítača ESP32. V práci opíšte hardvérové aspekty ESP32, ktoré sú pri aktualizácii využívané. Opíšte tiež využité kryptografické algoritmy a softvérové nástroje, ktoré má vývojár pre platformu ESP32 na aktualizáciu firmvéru k dispozícii. S využitím dostupných technických prostriedkov na báze ESP32 vytvorte demonstračnú aplikáciu senzorového uzla, ktorá umožní aktualizovať firmvér s využitím WiFi rozhrania a snímať údaje z vhodného senzora. Experimentálne overte funkčnosť navrhnutého riešenia a analyzujte možnosti zníženia celkovej spotreby senzorového uzla pri zachovaní možnosti bezpečnej aktualizácie firmvéru.

Jazyk, v ktorom sa práca vypracuje: slovenský  
Termín pre odovzdanie práce: 23.04.2021  
Dátum zadania diplomovej práce: 30.10.2020



  
.....  
prof. Ing. Liberios Vokorokos, PhD.  
dekan fakulty

## Čestné vyhlásenie

Vyhlasujem, že som celú diplomovú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice, 23. apríla 2021

.....

vlastnoručný podpis

## **PodĎakovanie**

V prvom rade sa chcem poĎakovať vedúcemu práce prof. Ing. Milošovi Drutarovskému, CSc. za cenné rady, vysvetlenie odborných termínov a konzultácie súvisiace s technickou stránkou diplomovej práce.

# Obsah

Zoznam obrázkov .....	10
Zoznam tabuliek .....	12
Zoznam symbolov a skratiek .....	13
Úvod .....	16
1. ESP32 .....	18
1.1. Cyklus spustenia mikrokontroléra ESP32 .....	19
1.2. Hardvérové verzie ESP32 .....	21
2. Pamäť mikrokontroléra ESP32 .....	25
2.1. Vstavaná pamäť .....	26
2.2. Externá pamäť .....	27
2.3. Tabuľka partícií .....	28
2.4. Jednorazovo programovateľná pamäť eFuse .....	30
3. Nízkopríkonový režim mikrokontroléra ESP32 .....	32
4. Vývojársky framework ESP-IDF .....	34
4.1. Vývojárske nástroje v ESP-IDF .....	34
4.2. Vývoj frameworku ESP-IDF .....	35
5. Arduino Core .....	38
6. Vzdialená aktualizácia firmvéru .....	39
6.1. Aktualizácia firmvéru v Arduino Core .....	39
6.2. Aktualizácie firmvéru v ESP-IDF .....	43
6.3. Metóda digitálneho podpisu .....	49
6.3.1. Digitálny podpis – implementácia v prostredí ESP-IDF .....	51
6.4. Secure Boot v ESP-IDF .....	56
6.4.1. Metóda zabezpečenia bootovacieho procesu – implementácia v ESP-IDF .....	58
6.4.2. SBDA algoritmus .....	61
6.5. Šifrovanie flash pamäte .....	65
6.5.1. Proces šifrovania flash pamäte .....	67



6.5.2.	Algoritmus šifrovania flash pamäte .....	68
6.5.3.	Šifrovanie flash pamäte – implementácia v prostredí ESP-IDF .....	69
7.	Experimentálny program pre aktualizáciu senzorového uzla .....	73
7.1.	Realizácia programu pre senzorový uzol.....	74
7.2.	Úpravy webového rozhrania .....	82
7.3.	Minimálna schéma zapojenia.....	86
Záver	.....	88
Zoznam použitej literatúry	.....	90
Prílohy	.....	97

## Zoznam obrázkov

Obr. 1 Jednotlivé fázy spustenia ESP32.....	20
Obr. 2 ESP32-DevKitC z produkcie Espressif Systems [22].....	22
Obr. 3 Zjednodušená bloková schéma mikrokontroléra ESP32 .....	24
Obr. 4 Mapa pamäte mikrokontroléra ESP32 .....	25
Obr. 5 Príklad rozdelenia partícií vo flash pamäti .....	30
Obr. 6 Sieťový OTA port v prostredí Arduino IDE.....	40
Obr. 7 OTA Web Updater .....	42
Obr. 8 Menuconfig – konfiguračné menu projektov v ESP-IDF.....	45
Obr. 9 Úspešná aktualizácia firmvéru, reštart systému .....	47
Obr. 10 Stiahnutie identického firmvéru, čakanie v slučke na reštart .....	47
Obr. 11 ESP32 – bloková schéma Native OTA – proces aktualizácie .....	48
Obr. 12 Podpísanie firmvéru súkromným kľúčom dôveryhodného vydavateľa .....	50
Obr. 13 Overenie digitálneho podpisu verejným kľúčom podpisujúceho .....	50
Obr. 14 Nastavenie ofsetu tabuľky partícií v Menuconfig .....	52
Obr. 15 Podpísanie firmvéru súkromným kľúčom v konzolovej aplikácii ESP-IDF .....	54
Obr. 16 Úspešné overenie digitálneho podpisu firmvéru pri bootovaní firmvéru .....	55
Obr. 17 Neúspešné bootovanie firmvéru zo všetkých aplikačných partícií .....	55
Obr. 18 Neúspešné overenie dig. podpisu pri jednej z partícií .....	56
Obr. 19 MZBP – neoverený softvérový bootloader, zakázaná fáza bootovania .....	57
Obr. 20 Nastavenie MZBP a verejného kľúča pre bootloader v Menuconfigu .....	58
Obr. 21 Sumár eFuses zobrazených cez nástroj esepfuse.py.....	59
Obr. 22 Bloková schéma procesu MZBP po bootovanie firmvéru .....	65
Obr. 23 Metóda šifrovania flash pamäte – Release režim .....	66
Obr. 24 Konfigurácia bezpečnostných makier pre MZBP a MŠFP.....	70
Obr. 25 Problém s načítaním obsahu flash pamäte .....	71
Obr. 26 Bosch BME280 - senzor teploty, tlaku, vlhkosti vzduchu.....	75
Obr. 27 Konzistentnosť meraní senzora BME280 v NORMAL móde.....	77
Obr. 28 Konzistentnosť meraní senzora BME280 vo FORCED móde .....	77
Obr. 29 Webové rozhranie termostatu pre ovládanie relé.....	78
Obr. 30 Vlastné konfiguračné menu pre voľbu I2C parametrov a adresy, režimu BME280 .....	81
Obr. 31 Bloková schéma riešenia senzorového uzla na báze ESP32.....	82
Obr. 32 Vizualizácia nameraných údajov vo webovom rozhraní .....	83
Obr. 33 Grafické rozhranie webstránky pre nahranie OTA firmvéru cez HTML formulár .....	83

Obr. 34 Jednoduché overenie prístupu menom a heslom.....	84
Obr. 35 Minimálna schéma senzorového uzla .....	87

## Zoznam tabuliek

Tab. 1 Prevádzkové režimy mikrokontroléru ESP32 a stav periférii .....	32
Tab. 2 Metódy zabezpečenia mikrokontroléru .....	49
Tab. 3 Rozdiel ofsetu aplikačných partícií po a pred posunutím .....	52
Tab. 4 Pripojenie vývodov ESP32 k vývodom použitých periférii .....	78
Tab. 5 Úlohy sensorového uzla pre odosielanie a načítanie dát z webového rozhrania .....	79

## Zoznam symbolov a skratiek

ADC	Analog-to-Digital Converter (analogovo-digitálny prevodník)
AES	Advanced Encryption Standard (štandard pokročilého šifrovania)
ALU	Arithmetic Logic Unit (aritmeticko-logická jednotka)
AP	Access Point (prístupový bod)
API	Application Programming Interface (rozhranie pre programovanie aplikácií)
BLE	Bluetooth Low Energy (technológia Bluetooth s nízkou spotrebou energie)
BLK	Označenie bloku jednorazovo programovateľnej pamäte eFuse v ESP32
COM	Communication port (sériová linka)
CSS	Cascading Style Sheets (kaskádové štýly)
DMA	Direct Memory Access (priamy prístup do pamäte)
DNS	Domain Name System (systém názvov domén)
DPS	Doska Plošných Spojov
DRAM	Data Random Access Memory (operačná pamäť dát)
DTR	Data Terminal Ready (pripraviť dátový terminál)
eFuse	electronic Fuse (elektronická poistka – jednorazovo programovateľná pamäť)
ESP-IDF	Espressif – IoT Development Framework
ESP-NOW	Protokol pre komunikáciu zariadení ESP(8266, 32), nevyžaduje smerovač, prepínač
FTDI	Future Technology Devices International
FTP	File Transfer Protocol (protokol pre prenos súborov)
GPIO	General-purpose input/output (vstupno-výstupný vývod)
HTML	HyperText Markup Language (hypertextový značkovací jazyk)
HTTP	HyperText Transfer Protocol (hypertextový prenosový protokol)
HTTPS	HyperText Transfer Protocol Secure (zabezpečený HTTP protokol)
I2C	Intel Integrated Circuit (dvojvodičová obojsmerná zbernica)

---

IIR	Infinite Impulse Response (filter s nekonečnou impulznou odozvou)
IO	Input-Output (vstup-výstup)
IoT	Internet of Things (internet vecí)
IP	Internet Protocol (komunikačný protokol, logická adresa zariadenia v sieti)
IRAM	Instruction Random Access Memory (operačná pamäť inštrukcií)
JTAG	Joint Test Action Group (štandard pre testovanie plošných spojov)
LAN	Local-Area-Network (lokálna počítačová sieť malej geografickej oblasti)
LSB	Least Significant Bit (najmenej významný bit)
MAC	Media Access Control (riadenie prístupu k médiu, fyzická adresa zariadenia)
MD5	Message-Digest algorithm 5 (algoritmus odtlačku správy 5, hašovacia funkcia)
mDNS	multicast Domain Name System (multicastový systém názvov domén použiteľný v lokálnych sieťach pre identifikáciu zariadení doménovým menom)
MHz	Megahertz
MMU	Memory Management Unit (jednotka správy pamäte)
MŠFP	Metóda Šifrovania Flash Pamäte
MZBP	Metóda Zabezpečenia Bootovacieho Procesu
OTA	Over-The-Air (metódy opisujúce spôsoby aktualizácie firmvéru, konfigurácie)
PHP	Hypertext Preprocessor (hypertextový preprocesor – skriptovací jazyk)
PHY	PHYsical layer (fyzická vrstva)
PoE	Power over Ethernet (napájanie cez dátové vodiče sieťového kábla)
PSRAM	Pseudo-Static Random Access Memory (dynamická operačná pamäť typu DRAM s preklápacími obvodmi, vyžaduje refresh)
QSPI	Quad Serial Peripheral Interface (štandard pre komunikáciu s externou pamäťou)
RAM	Random Access Memory (operačná pamäť, energeticky závislá)
RMII	Reduced Media-Independent Interface (rozhranie nezávislé od média)

ROM	Read-Only Memory (pamäť len na čítanie, energeticky nezávislá)
RTC	Real Time Clock (hodiny reálneho času)
RTOS	Real Time Operating System (operačný systém reálneho času)
RTS	Request To Send (požiadavka na odoslanie)
SBDA	Secure Boot Digest Algorithm (algoritmus zabezpečeného bootovacieho procesu)
SCL	Synchronous Clock (synchronizačné hodiny, hodinový signál)
SDA	Synchronous Data (synchronizované dáta)
SHA	Secure Hash Alogithm (bezpečný hašovacie algoritmus)
SPI	Serial Peripheral Interface (synchronne sériové periférne rozhranie)
SPIFFS	Serial Peripheral Interface Flash File System (súborový systém na flash disku)
SRAM	Static Random Access Memory (statická operačná pamäť s preklápacími obvodmi, nevyžaduje refresh)
SSR	Solid State Relay (polovodičový typ relé s triakom bez mechanickej časti)
UART	Universal Asynchronous Receiver-Transmitter (univerzálny asynchrónny prijímač a vysielateľ)
ULP	Ultra Low Power (ultra-nízky režim príkonu)
USB	Universal Serial Bus (univerzálna sériová zbernica)
USB-UART	Universal Serial Bus - Universal Asynchronous Receiver-Transmitter
XOR	eXclusive OR (exkluzívny súčet, kombinačné hradlo)

## Úvod

Senzorové siete sú rozšírené v oblastiach priemyslu, automatizácie a internetu vecí (IoT). Poskytujú zber dát na základe ktorých je možné riadiť výkonové spotrebiče, stroje a automatizovať výrobné procesy a zjednodušiť tak každodenný život. Do kategórie mikrokontrolérov môžeme zaradiť aj platformu ESP32 [1] z produkcie Espressif Systems, ktorá je použitá v tejto diplomovej práci pre demonštráciu riešenia sensorového uzla s možnosťou vzdialenej aktualizácie firmvéru a metód jeho zabezpečenia.

ESP32 má vstavanú WiFi a Bluetooth konektivitu, čo ho predurčuje na použitie v bezdrôtových aplikáciách sensorového uzla s možnosťou prenosu nameraných údajov dát cez WiFi rozhranie, ktoré môže byť použité aj pre prevzatie vzdialenej aktualizácie firmvéru z distribučného servera. Kapitola 1 opisuje základné parametre mikrokontroléra ESP32, poukazuje na možnosti využitia samostatného čipu, alebo vývojových kitov s osadenými komponentami pre programovanie a signalizáciu. V kapitole 2 sú opísané pamäte ESP32 s kategorickým rozdelením vrátane ich mapovania v adresnom priestore na dátovej a inštrukčnej zbernici.

ESP32 je možné prevádzkovať v režime s nízkym príkonom, čo nájde využitie najmä pre aplikácie internetu vecí pre prevádzku na batériu. Tento režim opisuje kapitola 3, ktorá poukazuje aj na dostupné režimy spánku hlavného procesora ESP32, obsluhu zberníc a vstupov, ktoré je možné v tomto režime prevádzky použiť z pohľadu zdrojov prebudenia hlavného procesora, alebo ich použiť s koprocesorom, ktorý je aktívny práve v tomto režime nízkeho príkonu. ESP32 je možné programovať v rôznych vývojových prostrediach, ktoré sa líšia možnosťami vývojárskych nástrojov, implementovanými ukázkovými zdrojovými kódmi pre obsluhu zberníc a iných funkcionalít.

Pre návrh a implementáciu výslednej aplikácie som využil framework ESP-IDF (Espressif IoT Development Framework) [2] z produkcie Espressif Systems v ktorom som vytvoril finálnu aplikáciu pre demonštráciu sensorového uzla s možnosťou vzdialenej aktualizácie firmvéru bezpečným prenosovým kanálom. Framework ESP-IDF spolu s vývojárskymi nástrojmi balíka ESPTOOL [3], aktuálne vyvíjanými verziami, softvérovou podporou je opísaný v kapitole 4.

Vývojové prostredie Arduino IDE [4] som využil iba na testovanie a porovnanie podporovaných metód aktualizácii firmvéru s prostredím ESP-IDF. Kapitola 5 opisuje prostredie Arduino IDE, jeho výhody, nevýhody pre návrh a vývoj aplikácií. Samotné metódy vzdialenej aktualizácie firmvéru podporované v jednotlivých vývojových prostrediach sú opísané v kapitole 6.



Kapitola ďalej obsahuje opis metód pre zabezpečenie firmvéru v prostredí ESP-IDF s využitím digitálneho podpisu, zabezpečenia bootovacieho procesu s overením bootloadera hardvérovou funkcionalitou a metódy pre šifrovanie flash pamäte, ktoré slúžia ako prevencia pred možnými typmi útokov, ktorým môže byť ESP32 vystavené v plnej prevádzke. Kapitola 7 opisuje realizáciu finálnej aplikácie na ESP32 pre senzorový uzol, použité hardvérové komponenty a periférie, úpravy hlavnej aplikácie mikrokontrolera, webového rozhrania, problémy, na ktoré som pri vývoji diplomovej práce narazil.

## 1. ESP32

ESP32 [1] je mikrokontrolér určený predovšetkým pre aplikácie internetu vecí a rôzne projekty vyžadujúce konektivitu so zariadeniami v sieti, internetom. Platforma podporuje WiFi (2.4 GHz) a Bluetooth konektivitu, pričom obe technológie zdieľajú spoločnú plošnú anténu na doske plošných spojov (DPS), čím sa výrazne redukuje aj samotná veľkosť modulu. Použitie oboch technológií je univerzálne s kompatibilným hardvérom.

WiFi modem ESP32 podporuje štandardy 802.11 b/g/n, čím dosahuje teoretickú rýchlosť prenosu až 300 Mbit/s. Podporuje protokol ESP-NOW [5], ktorý umožňuje komunikáciu medzi mikrokontrolérmi z produkcie Espressif Systems. Zaujímavosťou technológie je, že nevyžaduje smerovač, ani prístupový bod. Vyžaduje však párovanie obdobne ako pri technológii Bluetooth. Použitie protokolu ESP-NOW je energeticky výhodné aj pre aplikácie senzorových sietí s mesh prepojením, kde je nutné garantovať doručenie informácie, správy akoukoľvek cestou.

Bluetooth technológia vo verzii 4.2 s podporou BLE (Bluetooth Low Energy) [6] je vhodná pre aplikácie pre prevádzku na batériu a má spätnú kompatibilitu aj so zariadeniami so staršími verziami technológie Bluetooth. Rovnako tak novšie zariadenia s verziou Bluetooth 5.0 majú spätnú kompatibilitu a môžu komunikovať s ESP32. BLE je využívané predovšetkým pri operáciách skenovania Bluetooth zariadení v dosahu, pri vysielaní v cielenom, prípadne v Beacon (jednosmerný maják) [7] režime, podporuje viacnásobné spojenia.

V závislosti od verzie ESP32 čipu [8] sa mikrokontrolér líši veľkosťou externej flash pamäte [9], ktorá má veľkosť 4 až 16 MB. Čip môže mať podporu, alebo aj dostupnú externú RAM (Random Access Memory) pamäť osadenú na plošnom spoji. Medzi ďalšie parametre v ktorých sa verzie ESP32 čipu môžu líšiť, vieme zaradiť napríklad počet jadier procesora, frekvencia, verzia hlavného procesora.

ESP32 najčastejšie obsahuje dvojjadrový, pri istých verziách čipu jednojadrový 32-bitový procesor harvardskej architektúry Tensilica Xtensa L6 [10] s nastaviteľným taktom 80, 160, alebo 240 MHz. Architektúra má oddelenú pamäť pre dáta (údaje s ktorými mikrokontrolér pracuje) a inštrukcie (riadiaci program, ktorý mikrokontrolér vykonáva).

Hlavný procesor Xtensa môže realizovať výpočty, komunikáciu so senzormi a perifériami, s ktorými je mikrokontrolér prepojený cez dostupné zbernice. Zároveň však riadi a obsluhuje aj WiFi / Bluetooth zásobník (stack) pre zabezpečenie konektivity. Za správu a obsluhu WiFi / Bluetooth zásobníka zodpovedá jedno z jadier procesora u ktorého má WiFi / Bluetooth zásobník maximálnu prioritu. Jadrá sa najčastejšie označujú ako Core 0 (APP\_CPU) a Core 1 (PRO\_CPU) [1].

Core 0 nazývame aj aplikačné jadro procesora. Spúšťa a obsluhuje prioritne používateľskú aplikáciu (firmvér). Druhé z jadier – Core 1 nazývame jadro protokolu, ktoré sa primárne stará o WiFi / Bluetooth zásobník, udržiava spojenie s AP (prístupovým bodom) a zabezpečuje konektivitu. Zároveň však môže obsluhovať aj používateľskú aplikáciu, ak ju dokáže v reálnom čase obslúžiť. Výber APP a PRO procesora je voliteľný, štandardne je Core 0 APP\_CPU a Core 1 PRO\_CPU. Oba procesory zdieľajú spoločnú cache (vyrovnávaciu) pamäť s veľkosťou 64 kB a môžu pristupovať na rovnaké miesto v pamäti.

Procesor dokáže spustiť používateľské programy napísané v rôznych vývojových prostrediach s podporou kompilácie programu pre túto platformu, napríklad Arduino Core (prog. jazyk Wiring), ESP-IDF, Mongoose OS, MicroPython [11], Lua [12], Node.js [13], NodeMCU. Espressif Systems zabezpečuje a dohliada na vývoj vlastného frameworku ESP-IDF a rovnako aj pre Arduino Core, ktoré nebolo producentom čipov ESP32 podporované pri generácii predchádzajúcej platforme ESP8266 [14] od začiatku.

Arduino Core pre ESP8266 vyvíjali fanúšikovia a nadšenci mikrokontrolérov Arduino na Githube [15], neskôr ho zastrelila firma Espressif Systems. Posledná produkčná verzia Arduino Core pre ESP8266 je 2.7.4. z 2. Augusta 2020.

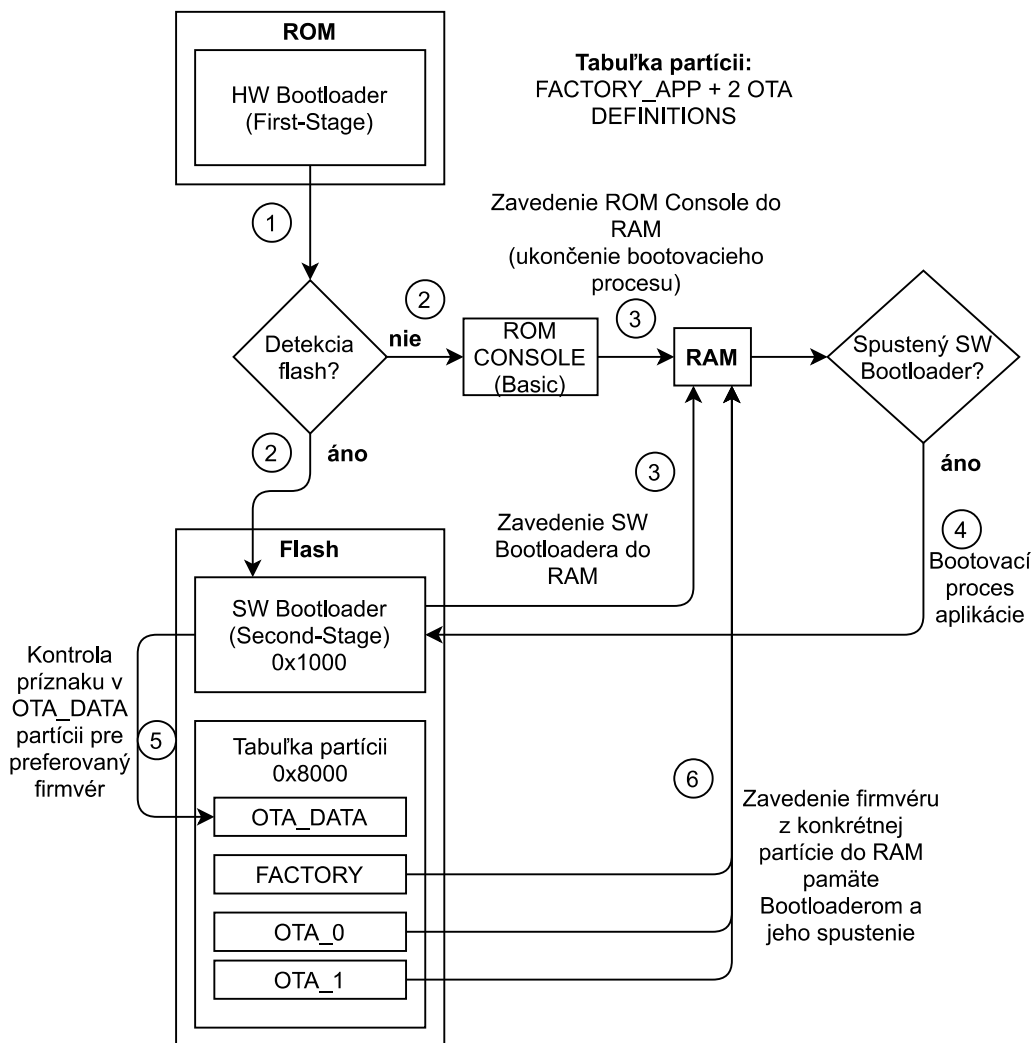
### 1.1. Cyklus spustenia mikrokontroléra ESP32

Spustenie ESP32 [16] nastáva pri každom štartovacím cykle v dôsledku reštartu, alebo po pripojení napájania. Proces pozostáva z viacerých fáz, ktoré na seba nadväzujú a majú jasné poradie vykonávania:

1. hardvérový bootloader (zavádzač) zavedie do RAM pamäte softvérový bootloader, ktorý sa nachádza na ofsete 0x1000 (hexadecimálna hodnota) flash pamäte,
2. softvérový bootloader zavedie tabuľku partícií (ofset 0x8000) a hlavnú aplikáciu (ofset 0x10000) z dostupnej (aplikačnej) partície s podporou bootovania do RAM pamäte. Aplikácia obsahuje dátové a inštrukčné segmenty v RAM pamäti a určité segmenty (iba na čítanie) mapované vo flash pamäti, ku ktorým pristupuje jadro procesora za behu aplikácie,
3. aplikácia sa spustí, následne aj druhé jadro procesora a plánovač operačného systému reálneho času (RTOS).

Jednotlivé fázy spustenia ESP32 sú bližšie opísané blokovou schémou na Obr. 1. Úspešný proces bootovania závisí v prvom rade od detekcie externého úložiska – flash pamäte, kde je uložený softvérový bootloader (Second-Stage bootloader) a spustiteľný firmvér, ktorý v ďalšej fáze cyklu

spustenia dokáže bootloader zaviesť do RAM pamäte a spustiť. Samotnú detekciu flash pamäte realizuje hardvérový bootloader (First-Stage bootloader) uložený v ROM pamäti.



Obr. 1 Jednotlivé fázy spustenia ESP32

### Hardvérový bootloader

Po reštarte mikrokontroléra ESP32 sa okamžite spustí jadro procesora PRO\_CPU, pričom APP\_CPU je udržiavané v resete. PRO\_CPU vykonáva celú inicializáciu mikrokontroléra. Na základe dôvodu reštartu – tzv. trigger kódu (Softvérový reštart, Watchdog reštart, reštart pre potrebu prebudenia hlavného čipu ESP32) dokáže PRO\_CPU prispôbiť výpis na UART (Univerzálny asynchrónny prijímač a vysielač) monitor, ktorý má informačnú hodnotu aj pre programátora v procese ladenia programu (debugovanie). Prostredníctvom trigger kódu dokáže reštart upozorniť na pretečenie zásobníka, „brownout“ reštart v dôsledku nedostatočného zdroja napájania, pretečenie Watchdogu a pre iné (prípadové) dôvody reštartu.

Zároveň dokáže po reštarte jadro PRO\_CPU prepnúť ESP32 do určitého režimu, napríklad pre prevzatie obsahu flash pamäte, nahrávanie firmvéru, alebo prevádzkového režimu. Ak zavedenie bootloadera z flash pamäte do RAM pamäte zlyhá, hardvérový bootloader zavedie Basic (TinyBasicPlus) interpreter (ROM Console) do RAM pamäte a spustí ho [17]. Nástroj slúži predovšetkým na ladenie, dokáže vykonávať základnú obsluhu GPIOs (vstupno-výstupné vývody), dokáže zapisovať a čítať z pamäte.

### **Softvérový bootloader**

Druhá fáza cyklu spustenia ESP32. Softvérový bootloader načíta tabuľku partícií na preddefinovanom ofsete (štandardne 0x8000). Nájde aplikačné partície (s podporou bootovania) a ich ofsety v pamäti. Ak sa využíva tabuľka partícií s OTA (Over-The-Air) definíciami, existuje aj partícia OTA\_DATA, ktorá definuje príznak pre bootloader pre bootovanie konkrétnej aplikačnej partície, kde sa nachádza spustiteľný firmvér.

Ak sa tabuľka partícií využíva bez OTA definícií s jedným firmvérom, OTA\_DATA partícia nie je vytvorená a alokovaná, keďže existuje iba jedna aplikačná partícia. Po nájdení ofsetu cieľového firmvéru zavedenie bootloader firmvér do RAM pamäte. Inštrukcie sú priradené do pamäte IRAM (RAM pamäť inštrukcií) a dáta do DRAM (RAM pamäť dát).

### **Spustenie aplikácie**

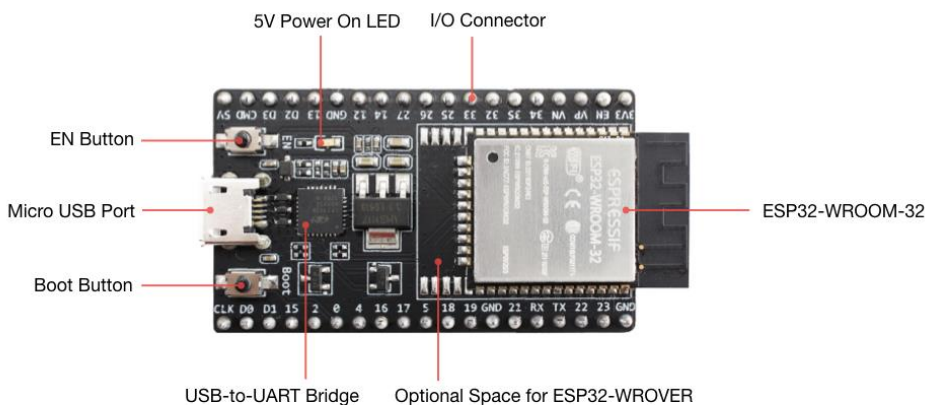
Pri spustení aplikácie (firmvéru) sa spustí APP\_CPU (doteraz udržiavané v resete). Po inicializácii hlavných základných komponentov sa vytvorí hlavná úloha a spustí sa plánovač FreeRTOS [18]. Hlavná úloha je funkcia `app_main()`, ktorú musí obsahovať každý program v prostredí ESP-IDF. Táto úloha má návratový typ (int) a pri návratovej hodnote 0 sa ukončí.

V plánovači FreeRTOS je možné spúšťať samostatné funkcie ako úlohy (tasky). Úlohy sa štandardne vytvárajú priamo v hlavnej aplikácii, ktorá ich inicializuje a následne sa ukončí. Aby dokázali úlohy bežať nepretržite musia byť navrhnuté s využitím cyklu, aby nedošlo k jej ukončeniu.

## **1.2. Hardvérové verzie ESP32**

Pre vývoj užívateľských projektov je možné využiť samostatný čip ESP32 (Standalone) [19], ktorý je možné programovať cez externý USB-UART (Prevodník USB signálov na UART rozhranie) prevodník, napríklad FTDI (Future Technology Devices International Limited) s vyvedenými vývodmi pre pripojenie k ESP32. Taktiež existuje aj vývojový kit (nazývaný aj DevKit), ktorý je pri vývoji projektov viac obľúbený, keďže práca s ním je pohodlnejšia.

Vývojový kit je vybavený USB-UART prevodníkom, najčastejšie CP2102 [20] od Silicon Labs pre možnosť napájania a programovania ESP32 cez USB (Univerzálna sériová zbernica) kábel s dátovými vodičmi. Najpopulárnejší DevKitC V4 od Espressif Systems [21] je možné nájsť na Obr. 2 aj s opisom základných hardvérových prvkov a komponentov, ktorými je tento kit vybavený.



Obr. 2 ESP32-DevKitC z produkcie Espressif Systems [22]

DevKit je osadený aj indikačnými LED (luminiscenčné svetlo) diódami a má vyvedené vývody na ktoré je možné prispájkovať vývodové lišty. Najčastejšie je to verzia DevKitu s 30, alebo 38 vývodmi. Vývodové lišty umožňujú lepšiu manipuláciu s mikrokontrolérom a ľahšie pripojenie periférii napríklad v prepojovacom poli (Breadboarde). DevKity majú vyvedené aj signálové tlačidlá označené ako BOOT a EN. Stlačením tlačidla BOOT je možné nastaviť programovací mód čipu ESP32 aktiváciou DTR (Data Terminal Ready) signálu, ktorý umožní nahrávanie programu.

Stlačením EN tlačidla je možné systém reštartovať RTS (Request To Send) signál. Kombináciou stlačení oboch tlačidiel súčasne je možné dosku prepnúť do módu čítania obsahu flash pamäte, kedy je možné prevziať celý obsah uložený vo flash pamäti, alebo jeho časť s definovaním začiatočného a koncového offsetu.

Načítať obsah flash pamäte v tomto režime je možné cez nástroj `esptool.py` [23], ktorý slúži pre prácu s flash pamäťou mikrokontroléra ESP32. S použitím príkazu `esptool.py -b 921600 read_flash 0 0x400000 flash_dump.bin` je možné načítať obsah celej flash pamäte s veľkosťou 4MB.

V príkaze je definovaná rýchlosť čítania flash pamäte – 921600 baud/s (znakov za sekundu), začiatočný a koncový offset, výstupný .bin súbor do ktorého bude prevzatý obsah flash pamäte zapísaný. Vývojové kity sú najčastejšie osadené najlacnejším a najdostupnejším ESP32 čipom ESP32-WROOM-32 [24] so 4 MB externou flash pamäťou bez osadenej PSRAM (Pseudo Static RAM)

na DevKit doske. Nie všetky dosky majú oblasť na DPS pre priame osadenie externej PSRAM pamäte.

V kooperácii s Espressif Systems vyrábajú čipy ESP32 aj iní výrobcovia (Vendors), ktorí využívajú vlastné označenia svojich ESP32 čipov. Jedným z výrobcov s najvyššou produkciou ročne je firma AI-Thinker. Čip ESP32-WROOM-32 z ich produkcie je označený ako ESP32-S [25] a je 100% ekvivalent čipu od Espressifu.

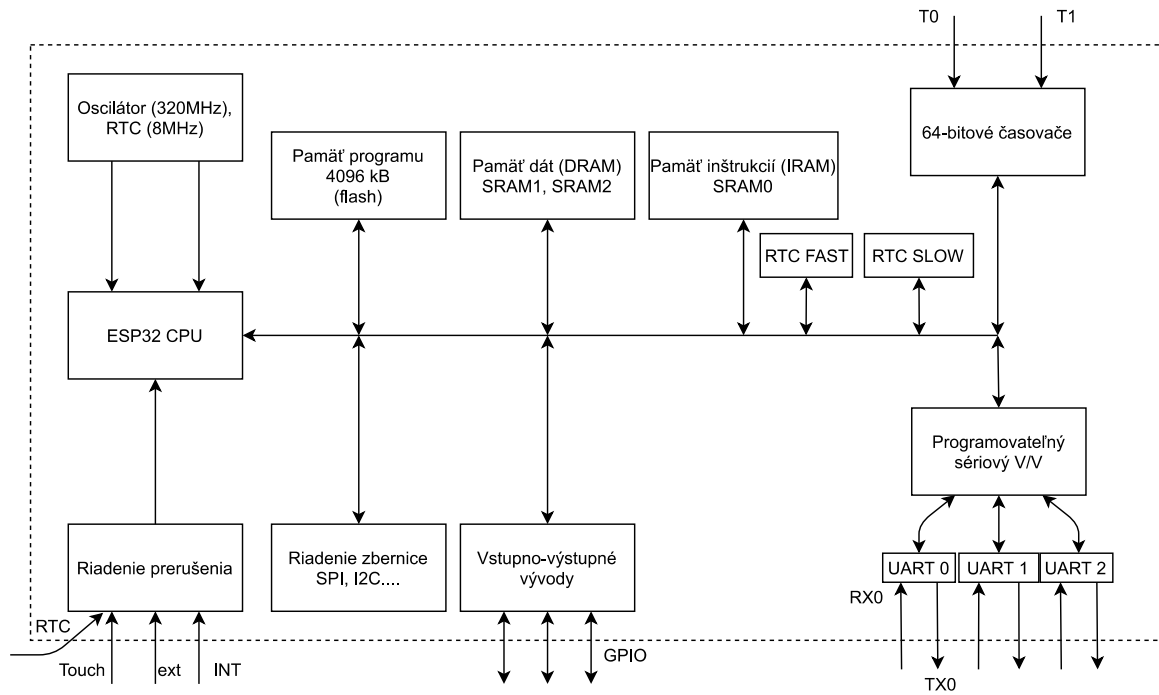
Čip má na plošnom spoji integrovaný u.FL konektor, ktorý umožňuje k ESP32 pripojiť externú anténu pre zvýšenie dosahu pre Bluetooth, WiFi konektivitu. Vyžaduje však aj zmenu fyzickej prepajky, ktorá volí plošnú anténu na DPS / externú anténu pre dosiahnutie efektivity prenosu a zníženie interferencie.

Čipom ESP32-S sú osadené rôzne vývojové kity od AI-Thinker, napríklad ESP32-S Dev Board, aj špeciálne dosky osadené kamerou – napr. ESP-CAM [26]. Populárnym výrobcom kitov je aj bulharská firma OLIMEX [27], ktorá využíva čipy ESP32 od Espressif Systems a ponúka riešenia pre oblasť internetu vecí, riadenia výkonových spotrebičov na vlastných DPS. Firma produkuje hotové DPS s podporou Ethernetu, PoE (Power over Ethernet), prípadne modemom mobilnej siete, ktorý je možné popri WiFi konektivitě ESP32 používať.

Zjednodušená bloková schéma mikrokontroléra ESP32 od Espressif Systems je znázornená na Obr. 3. Schéma opisuje základné typy pamätí, zbernice, časovače, interné a externé prerušenia, ktorými je ESP32 vybavené.

DevKity priamo z produkcie Espressif Systems sú [28]:

- ESP32-DevKitC,
- ESP-WROVER-KIT,
- ESP32-PICO-KIT,
- ESP32-Ethernet-Kit,
- ESP32-DevKit-S(-R),
- ESP32-PICO-KIT-1,
- ESP32-PICO-DevKitM-2,
- ESP32-DevKitM-1.



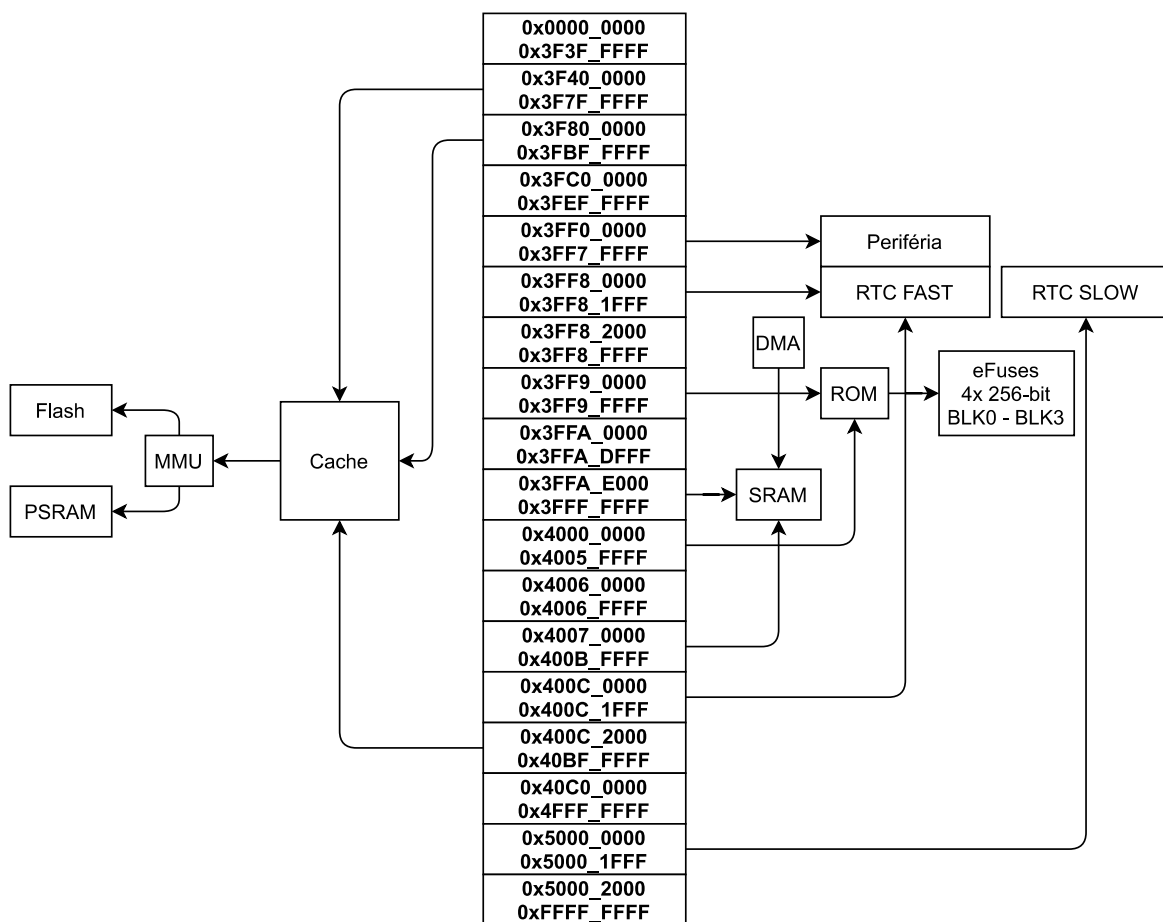
Obr. 3 Zjednodušená bloková schéma mikrokontroléra ESP32



## 2. Pamäť mikrokontroléra ESP32

ESP32 má rôzne typy pamätí, ktoré používa. Z pohľadu ich umiestnenia môžeme pamäte kategorizovať na interné a externé. Interná pamäť je súčasťou hlavného čipu Xtensa a poskytuje rýchlu komunikáciu s procesorom. Externá pamäť je obsiahnutá v samostatnom čipe komunikujúcom po SPI zbernici.

Rýchlosť komunikácie a prenosu externých pamätí v porovnaní s internými je nižšia. Pamäte majú bohaté členenie aj z pohľadu ich logického rozdelenia pamäťových oblastí [1]. Na Obr. 4 je dostupná blokovaná schéma mapy pamäte, ktorá približuje jednotlivé typy externých a interných pamätí a ich ofsety v adresnom priestore, kde sú mapované na dátovej, alebo inštrukčnej zbernici.



Obr. 4 Mapa pamäte mikrokontroléra ESP32

Vlastnosti adresného priestoru ESP32:

- symetricky mapovaná pamäť,
- 4 GB (32-bit) adresného priestoru,
- 1296 kB adresný priestor vstavanej (embedded) pamäte,
- 19704 kB adresný priestor externej pamäte,

- 512 kB adresný priestor periférie,
- 328 kB DMA (priamy prístup do pamäte) adresný priestor.

Vstavaná pamäť :

- 448 kB internej ROM pamäte (pamäť len na čítanie, energeticky nezávislá),
- 520 kB internej SRAM (Static Random Access Memory) pamäte,
- 8 kB RTC FAST pamäte,
- 8 kB RTC SLOW pamäte,
- \* – niektoré verzie ESP32 majú flash pamäť integrovanú (napr. ESP32-PICO-D4 [29]).

Externá pamäť:

- podpora až 16 MB SPI flash pamäte,
- podpora až 8 MB SPI SRAM pamäte.

## 2.1. Vstavaná pamäť

Vstavaná pamäť mikrokontroléra ESP32 má štyri segmenty: ROM, SRAM, RTC (hodiny reálneho času) FAST a SLOW pamäť, ktoré sú logicky rozdelené. Pamäť ROM obsahuje funkcie jadra ESP32 (operačné kódy), ktoré zodpovedajú za správne bootovanie systému ESP32. Obsahuje spodné vrstvy konektivity, obsluhu ich zásobníka, systémové funkcie, Stub a iné...

Funkcia Stub [30] môže spustiť kód uložený v ROM pamäti, ale taktiež spustiteľný kód uložený v RTC SLOW pamäti. Využíva sa napríklad v operáciách Deep Sleep (režim hlbokého spánku) hlavného procesora ESP32 pre výpis informácií na UART rozhranie.

Funkcia pre stub využíva v definícii funkcie v zdrojovom kóde atribút „RTC\_IRAM\_ATTR“, aby bola funkcia dostupná v RTC SLOW pamäti (zabezpečí linker v procese kompilácie) a mohla byť obslužená aj pri režime spánku hlavného procesora ESP32.

V ROM pamäti sú obsiahnuté rutiny [1], ktoré sa dokážu spúšťať priamo z tejto pamäte a nezaberajú miesto v RAM pamäti. Spustenie je rýchlejšie ako z externej flash pamäte, nakoľko ide o internú pamäť. K ROM pamäti môžu pristupovať obe jadrá procesora Xtensa.

ROM pamäť je rozdelená na oddiel ROM0 (384 kB), ktorý je mapovaný v pamäti inštrukcií v rozsahu 0x4000\_0000 až 0x4005\_FFFF (vyjadrené v hexadecimálnej hodnote). Zostávajú časť sa označuje ako ROM1 (64 kB) a je mapovaná na dátovej zbernici pre rozsah 0x3FF9\_0000 až 0x3FF9\_FFFF. Pamäť SRAM je rozdelená na tri oddiely – SRAM0 (192 kB), SRAM1 (128 kB), SRAM2 (200 kB).

Z pamäte SRAM0 je možné vyhradiť časť 64 kB pre cache externej flash pamäte. Zostávajúca časť SRAM0 (128 kB) slúži na zápis a čítanie hlavným procesorom Xtensa ESP32, pristupovať k nej môžu obe jadrá procesora. V prípade, že sa alokovaná časť 64 kB na cache nepoužíva, je ju možné používať ako štandardnú RAM pamäť. Pristupovať k nej môžu obe jadrá procesora. 64 kB časť SRAM0 je mapovaná na zbernici inštrukcií v rozsahu 0x4007\_0000 až 0x4007\_FFFF.

Zostávajúcich 128 kB SRAM0 pamäte je mapovaných na zbernici inštrukcií v rozsahu 0x4008\_0000 až 0x4009\_FFFF. Pamäť SRAM1 má veľkosť 128 kB. Je mapovaná na inštrukčnej i dátovej zbernici zároveň. Na dátovej zbernici je SRAM1 mapovaná v rozsahu 0x3FFE\_0000 až 0x3FFF\_FFFF a na inštrukčnej pre rozsah 0x400A\_0000 až 0x400B\_FFFF. Pamäť SRAM2 veľkosti 200 kB je mapovaná na dátovej zbernici pre rozsah 0x3FFA\_E000 až 0x3FFD\_FFFF s možnosťou prístupu oboch jadier procesora.

DMA používa rovnakú adresáciu ako procesor dátovej zbernice pre zápis a čítanie do SRAM1 (0x3FFE\_0000 až 0x3FFF\_FFFF) a SRAM2 (0x3FFA\_E000 až 0x3FFD\_FFFF). Priamy prístup k pamäti využíva celkom trinásť zberníc: UART0, UART1, UART2, SPI1 (Synchronne sériové periférne rozhranie), SPI2, SPI3, I2S0, I2S1, SDIO, Slave SDMMC, EMAC, Bluetooth, WiFi. RTC FAST je pamäť typu SRAM a má veľkosť 8 kB. Čítať a zapisovať do nej môže iba jadro protokolu – PRO\_CPU. Aplikačné jadro nemá k pamäti prístup. Pamäť RTC FAST je adresovaná v rozsahu 0x3FF8\_0000 až 0x3FF8\_1FFF na dátovej zbernici, resp. na 0x400C\_0000 až 0x400C\_1FFF na zbernici inštrukcií. RTC SLOW pamäť má rovnakú veľkosť ako RTC FAST (8kB).

Je adresovaná v rozsahu 0x5000\_0000 až 0x5000\_1FFF na dátovej i inštrukčnej zbernici zároveň. K tejto pamäti môžu však pristupovať obe jadrá procesora Xtensa. V prípade využívania ULP (režim nízkeho odberu elektrickej energie) koprocesora je táto pamäť vyhradená pre neho a slúži na uloženie jeho programu, odkiaľ ho dokáže spustiť, prípadne môže byť využitá pre funkciu Stub.

## 2.2. Externá pamäť

K mikrokontroléru ESP32 je možné pripojiť externú pamäť typu flash a SRAM cez QSPI (štandard pre štvornásobné zrýchlenie SPI zbernice) zbernicu. Hlavný procesor Xtensa pristupuje k externej pamäti cez Cache a MMU (jednotku správy pamäte). Na základe dostupného adresného priestoru je možné využiť maximálne 16 MB externej flash pamäte a 8 MB externej SRAM pamäte.

Externá SRAM pamäť sa pripája paralelne k existujúcej externej flash pamäti. ESP32 podporuje viacero typov SRAM pamätí, avšak vývojársky framework ESP-IDF na základe dokumentácie podporuje iba externú SRAM z vlastnej produkcie Espressif Systems s výrobným označením ESP-PSRAM32 (veľkosť pamäte 32 Mb), ESP-PSRAM64 (64 Mb) [31].

Externú SRAM pamäť označujeme aj ako PSRAM. Alokácia PSRAM pamäte začína od offsetu 0x3F80\_0000 (v prípade 4 MB flash pamäte). PSRAM je obslužená aplikačným programom, ktorý zodpovedá za správu pamäte, obsluhu vyrovnávacej pamäte. Externá flash pamäť je typom energeticky nezávislej pamäte, ktorá dokáže uchovať dáta aj po odpojení napájania. Pamäť typu flash je v prípade mikrokontroléru ESP32 obsiahnutá v samostatnom (externom) čipe, ktorý je s hlavným procesorom prepojený cez SPI zbernicu, ktorou sa dáta prenášajú.

Veľkosť flash pamäte, ktorá je na ESP32 osadená je najčastejšie 4 MB s možnosťou jej rozšírenia na 16 MB. Do flash pamäte je možné zapísať aplikačný program (aj viacero zároveň, ak existujú dostupné aplikačné partície s podporou bootovania), tabuľku partícií, ktorá zabezpečuje logické rozdelenie flash pamäte na bootovateľné partície s firmvérom, oddiel bootloadera a SPIFFS (súborový systém na flash disku) filesystemu a iné oddiely...

Do flash pamäte je možné zapísať aj rôzne dáta a súbory (do SPIFFS filesystemu), ku ktorým môže pristupovať mikrokontrolér v aplikačnom programe. Najčastejšie sú to konfigurácie, obrázky, textové súbory, kaskádové štýly, súbory Javascriptu, ktoré je možné použiť vo vlastnom aplikačnom programe, napríklad pri implementácii HTTP (Hypertextový prenosový protokol) webservera bežiaceho na platforme ESP32, ktorá má všetky zdroje HTML (Hypertextový značkovací jazyk) stránky uložené vo vlastnej pamäti.

### 2.3. Tabuľka partícií

ESP32 dokáže mať niekoľko na sebe nezávislých partícií uložených vo flash pamäti. Na preddefinovanom offsete vo flash pamäte (štandardne 0x8000) je zapísaná tabuľka partícií [32], ktorá rozdeľuje priestor vo flash pamäti na logické celky, kde je každá partícia opísaná svoji názvom, typom (rozdeľujeme systémové, aplikačné, dátové partície a iné druhy), začiatočným offsetom a veľkosťou.. Tabuľka má pevnú veľkosť, do ktorej sa zmestí opis až 95 partícií.

Tabuľka partícií v rozhraní ESP-IDF využíva dva štandardné typy, ktoré sa najčastejšie používajú – „Single Factory App, no OTA“ a „Factory App, two OTA definitions“. Prvý menovaný typ je vhodný pre jeden firmvér, ktorý tvorí aplikačný program. Pre metódy aktualizácie firmvéru s možnosťou jeho uloženia do inej dostupnej aplikačnej partície je vhodný typ „Factor App, two OTA definitions“, ktorý umožňuje okrem hlavnej aplikácie firmvéru spúšťať aj firmvér uložený na dvoch ďalších aplikačných partíciách (označených ako OTA), celkovo tri firmvéry, každý s veľkosťou 1MB. Každá aplikačná partícia je bootovateľná (spustiteľná) v RAM pamäti.

Tento typ schémy rozdelenia partícií som využil aj v mojej diplomovej práci, keďže OTA aktualizáciu využívam pre finálnu demonštráciu a opis vzdialenej aktualizácie firmvéru. V prípade tohto typu

schémy partícií s viacnásobnou bootovateľnou partíciou je do tabuľky pridaná aj partícia OTA\_DATA, ktorá definuje príznak pre bootloader a rozhoduje tak o tom, ktorý firmvér bude prioritne bootovaný.

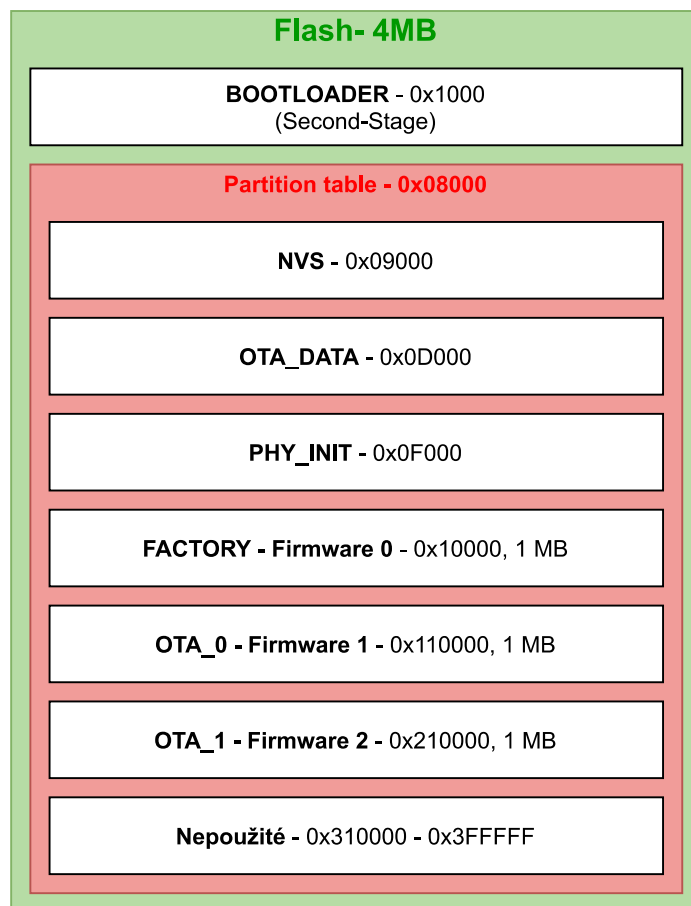
Ak je príznak v partícií OTA\_DATA prázdny, bootuje sa továrenská verzia firmvéru z partície FACTORY. Firmvér do tejto partície je možné nahráť iba prostredníctvom USB-UART rozhrania. Okrem aplikačných partícií sú v tabuľke partícií zapísané aj iné partície dátového a konfiguračného významu. Štandardná tabuľka partícií pre OTA aplikácie má celkovo tri bootovateľné partície označené ako aplikačné (APP).

#### **Tabuľka partícií pre schému partícií „Factory App, two OTA definitions“ v .csv formáte:**

```
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x4000,
otadata, data, ota, 0xd000, 0x2000,
phy_init, data, phy, 0xf000, 0x1000,
factory, app, factory, 0x10000, 1M,
ota_0, app, ota_0, 0x110000, 1M,
ota_1, app, ota_1, 0x210000, 1M,
```

Tabuľku partícií je možné plne prispôsobiť pre potreby finálnej aplikácie. V prostredí frameworku ESP-IDF je možné tabuľku partícií importovať s vlastnou štruktúrou logického rozdelenia flash pamäte v .csv formáte. Tabuľka partícií je v procese kompilácie skompilovaná do binárnej podoby, ktorá sa následne zapíše do flash pamäte na nastavený ofset.

Každá partícia môže mať priradený príznak (Flag), ktorým je možné nastaviť partícii systémovú funkcionálnosť, napríklad šifrovanie jej obsahu na danom ofsete, ak sa šifrovanie flash pamäte používa. Bootovateľné partície (typ app) sú šifrované vždy bez nutnosti nastaviť príznak, ak je povolené šifrovanie flash pamäte. Na Obr. 5 je bloková schéma flash pamäte s príkladom možného rozdelenia partícií pre schému „Factory App, two OTA definitions“.



Obr. 5 Príklad rozdelenia partícií vo flash pamäti

## 2.4. Jednorazovo programovateľná pamäť eFuse

Mikrokontrolér ESP32 obsahuje niekoľko interných pamäťových blokov – eFuses (elektronická poistka) [33]. Tieto bloky sú jednorazovo programovateľnou pamäťou a majú veľkosť 256 bitov. Každý z blokov eFuses je rozdelený do ôsmich 32-bitových registrov.

Každý register funguje aj ako 1-bitové pole, ktoré umožňuje jednorazový zápis do tejto pamäte (v prípade niektorých eFuses je možný aj viacnásobný prepis s obmedzeným počtom prepisov). ESP32 má celkovo štyri eFuse bloky. Sú označené názvom Block, resp. BLK (Označenie bloku jednorazovo programovateľnej pamäte eFuse) a indexom, ktorý ich charakterizuje 0 až 3.

Z pohľadu použitých eFuses v mikrokontroléri ESP32 ich môžeme rozdeliť na:

- kalibračné (kalibrácia napäťovej referencie, ADC – analógovo-digitálny prevodník),
- konfiguračné (konfigurácia SDIO rozhrania, signálov),
- poistkové eFuses (rezervácia BLK3, maska bitových polí eFuses),
- eFuses pre identitu (MAC – fyzická adresa zariadenia, revízia čipu ESP32),

- bezpečnostné (JTAG – štandard pre testovanie plošných spojov, symetrický kľúč šifrovanie flash pamäte, symetrický kľúč pre bezpečný proces bootovania).

Po zapísaní do systémových eFuse nie je možné jej obsah softvérovo prečítať, používa ich výhradné program z ROM pamäte, ktorý plní bezpečnostnú funkcionality. Pri procese zabezpečeného bootovacieho procesu využíva kľúč, ktorý je zapísaný v eFuse BLK2. Blok eFuse BLK0 má niekoľko polí do ktorých je možné zapísať bit, prípadne viac-bitovú hodnotu. Niektoré z registrov v eFuse BLK0 má aj systémový význam.

Potvrdzovacia poistka ABS\_DONE\_0 (1-bitové pole) po zápise bitu permanentne spúšťa metódu zabezpečenia bootovacieho procesu bez možnosti jeho vypnutia. Dva bloky – eFuses BLK1 a BLK2 sa nazývajú aj systémové poistky. Blok eFuse BLK1 je úložiskom symetrického AES kľúča pre šifrovanie / dešifrovanie flash pamäte. Veľkosť tejto eFuse pamäte je 256 bitov.

Pamäťový blok eFuse BLK2 má rovnakú veľkosť 256 bitov. Je úložiskom pre symetrický AES kľúč, ktorý sa používa pri výpočte odtlačku bootloadera zapísaného vo flash pamäti, ktorým je možné verifikovať softvérový bootloader a zabezpečiť tak bootovací proces. Posledným eFuse blokom je BLK3, ktorým je možné nastaviť špecifickú MAC adresu ESP32.

### 3. Nízkopříkonový režim mikrokontroléra ESP32

Nízka spotreba je jednou z priorit IoT aplikácií [34]. Tieto aplikácie nazývame aj ULP (Ultra-Low Power), teda aplikácie s ultra-nízkou spotrebou elektrickej energie, alebo s nízkopříkonovým režimom. Využitie týchto aplikácií je predovšetkým v implementáciách s prevádzkou na batériu. ESP32 umožňuje uspať hlavný procesor Xtensa a vypnúť WiFi a Bluetooth modem v nečinnosti zariadenia. Spotreba ESP32 dosahuje podľa dokumentácie v tomto režime spotrebu na úrovni desiatok mikroampérov.

Tento jav nazývame aj režimom spánku ESP32, ktorý rozlišujeme na základe toho, ktoré periférie a hlavné časti mikrokontroléru sú vypnuté, neaktívne. Jednotlivé podporované režimy spánku pre ESP32 sú opísané v Tab. 1. Pre opätovné zapnutie hlavného procesora, modemu WiFi / Bluetooth konektivity je potrebné prebudiť hlavný procesor vykonaním reštartu.

Tab. 1 Prevádzkové režimy mikrokontroléru ESP32 a stav periférií

Prevádzkový režim	Aktívny	Modem-sleep	Light-sleep	Deep-sleep
<b>Hardvér</b>	–	–	–	–
Xtensa CPU	ZAP	ZAP	PAUSED	VYP
WiFi/Bluetooth modem	ZAP	VYP	VYP	VYP
RTC pamäť a periférie	ZAP	ZAP	ZAP	ZAP
ULP koprocessor	ZAP	ZAP	ZAP	ZAP / VYP

**Pre prebudenie čipu ESP32 v režime ľahkého spánku (Light Sleep) je možné využiť tieto druhy zdrojov:**

- **UART** – prebudí čip ESP32, ak sa v RX (prijímači) objaví určitý počet signálov v logickej 1, ktoré indikujú príjem dát. Používa počítadlo v prerušení na nábežnú hranu (RISING) pri prechode z log. 0 do log. 1 (3,3V). Na základe rozhodovacieho prahu – počet signálov UART prebudí a môže prijímať dáta (existujúce dáta – signály, ktoré realizovali prebudenie nie sú uložené v buffri UART zbernice),
- **GPIO** – umožňuje využiť akýkoľvek GPIO vývod a definovať na ňom očakávanú logickú úroveň, ktorá prebudí hlavný procesor Xtensa mikrokontroléru ESP32, ak sa na danom takáto logická úroveň objaví.



**Pre prebudenie hlavného procesora ESP32 v režime hlbokého spánku (Deep Sleep) je možné využiť až štyri druhy zdrojov:**

- **Timer** – RTC kontrolér má Timer (časovač), prostredníctvom ktorého dokáže zobudiť hlavný procesor mikrokontroléru ESP32 po určitom zadanom čase, kedy dôjde k pretečeniu časovača. Časovač je možné nastavovať prostredníctvom parametra v mikrosekundách,
- **Dotyk** – využíva detekciu zmeny analógovej hodnoty na preddefinovanom vstupnom vývode s podporou TOUCH (snímania dotyku) funkcie. Dotykom sa vykoná kapacitná zmena, na ktorú dokáže mikrokontrolér reagovať čítaním analógovej hodnoty vývodu a pri prekročení rozhodovacieho prahu prebudí hlavný procesor Xtensa mikrokontroléru ESP32,
- **Externé prebudenie** – RTC IO (vstupno-výstupný) modul pracuje s logikou, ktorá dokáže zobudiť hlavný procesor Xtensa, ak je RTC GPIO nastavený na požadovanú logickú úroveň. Existuje aj rozšírenie tohto prebudenia o skupinu vývodov, kde sa skúma logická úroveň všetkých vývodov v tejto skupine a po nadobudnutí špecifickej úrovne sa hlavný procesor Xtensa prebudí,
- **ULP koprocesor** – spúšťa svoj program uložený v RTC SLOW pamäti v režime spánku hlavného procesora Xtensa. Dokáže obsluhovať ADC, vykonávať merania zo senzorov, dokáže pristupovať k RTC registrom a čítať / zapisovať do nich. Pri určitej udalosti – napríklad prekročení rozhodovacieho prahu ADC prevodníka dokáže hlavný procesor zobudiť.

ULP koprocesor [35] je jednoduchý automat konečných stavov. Využíva 32-bitové inštrukcie, dokáže pristupovať k istej časti pamäte RTC SLOW. Má vlastné štyri 16-bitové registre do ktorých môže ukladať dáta. Majú označenie R0 až R3. Koprocesor má vlastný 8-bitový čítačový register, ktorý môže byť použitý pre vykonávanie slučiek. Koprocesor k registru pristupuje cez špeciálne inštrukcie.

Programový kód, ktorý dokáže vykonávať je najčastejšie zapísaný v jazyku Assembler [36] a kompiluje sa spoločne s hlavným (aplikačným) programom. Koprocesor dokáže obslúžiť ALU (aritmeticko-logická jednotka) inštrukcie, obsluhu registrov RTC\_CNTL, RTC\_IO, SENS a inštrukcie pre prácu s pamäťou dokáže obslúžiť v jednom cykle. Pre obsluhu externých zberníc (napríklad ADC) potrebuje 2 až 4 cykly na základe vykonávanej operácie.

## 4. Vývojársky framework ESP-IDF

Framework pre vývoj IoT aplikácií založený na jazyku C pre platformu ESP32. Obsahuje API (rozhranie pre programovanie aplikácií). Zahŕňa knižnice a ukážkové zdrojové kódy – projekty pre základnú obsluhu zberníc, rozhraní a komponenty pre obsluhu konkrétnej aplikácie. Obsahuje balík programov (toolchain) pre kompiláciu programu s build systémom CMake a Ninja.

Cieľom frameworku je ponúknuť základné implementácie pre vývojárov IoT aplikácií s Bluetooth a WiFi konektivitou, ktoré sú bezpečné, poskytujú možnosť riadenia spotreby elektrickej energie zariadenia, ktorá je častokrát kľúčovým parametrom úspešnej (kritickej) aplikácie prevádzkovej na batériu pre IoT aplikácie.

Súčasťou ESP-IDF je aj jednoduché konzolové rozhranie, ktoré slúži pre výber cieľového projektu, možnosť jeho kompilácie, konfigurácie aktuálne otvoreného projektu. Umožňuje využiť vývojárske nástroje pre prácu s flash pamäťou, eFuses ESP32, či pre kryptografické operácie.

### 4.1. Vývojárske nástroje v ESP-IDF

Framework ESP-IDF má viacero vývojárskych nástrojov, ktoré sú realizované pomocou Python scriptov. Scripty sú spustiteľné prostredníctvom konzolovej aplikácie ESP-IDF a dokážu zároveň prevziať a spracovať aj vstupné argumenty s ktorými sú spustené. Na základe vstupného argumentu dokáže script spustiť funkciu, ktorá je týmto argumentom definovaná priamo v scripte, prípadne využíva import iných (externých) Python scriptov, ktoré majú danú funkciu definovanú a dokážu ju vykonať.

Základným nástrojom v prostredí ESP-IDF je `idf.py` [37]. Obsluhuje príkazy súvisiace s kompiláciou programu, ale aj pre spustenie podprogramov prostredia. Príkazom `idf.py menuconfig` umožňuje vyvolať Menuconfig (konfiguračné menu), ktorým je možné nastaviť aktuálne otvorený projekt pre potrebu vývojára. Dokáže vykonať zápis skompilovaného firmvéru a ďalších súčastí do flash pamäte.

Nástroj `esptool.py` slúži pre prácu s flash pamäťou ESP32. Dokáže vykonať priamy zápis do flash pamäte (najčastejšie je spúšťaný nástrojom `idf.py`, ak nie je vyvolaný priamo príkazom). Označenie „ESPTOOL“ je zároveň aj označením pre balík vývojárskych nástrojov dostupných v prostredí ESP-IDF (`esptool.py`, `espsecure.py` a `espefuse.py`).

Pre kryptografické operácie je v prostredí ESP-IDF dostupný nástroj `espsecure.py`. Nástroj `espefuse.py` slúži pre prácu s jednorazovo programovateľnými pamäťami eFuses. Všetky nástroje je možné obsluhovať z príkazového riadku ESP-IDF. Keďže nástroj `espsecure.py` a `espefuse.py` nemá podporu spustenia v celom systéme Windows ale iba z priečinka, kde sú umiestnené, bolo potrebné balík

nainštalovať opätovne s celosystémovou podporou, čo zaručí, že je možné nástroj spustiť kdekoľvek v operačnom systéme z príkazového riadku. Po stiahnutí balíka ESPTOOL z Githubu je potrebné spustiť súbor `setup.py`, ktorý nástroje nainštaluje pre možnosť spustenia v celom operačnom systéme.

Do frameworku ESP-IDF sú implementované aj komponenty vďaka ktorým je možné konfigurovať špecifické požiadavky aplikácie, napríklad – využitie hardvérového kryptografického akcelerátora AES (Advanced Encryption Standard), využitie koprocesora pre aplikácie s nízkym príkonom, vyhradenie zásobníka pre hlavnú úlohu, umožnenie kompilácie programu so spätnou kompatibilitou s predchádzajúcimi verziami ESP-IDF (týka sa najmä nepodporovaných a zastaralých funkcií v programových implementáciách so starším API), nastavenie ignorácie a obídenie bezpečnostných prvkov pre vývojárske účely a iné...

## 4.2. Vývoj frameworku ESP-IDF

Espressif Systems vyvíja framework ESP-IDF na Githubu v repozitári projektu. Projekt je vyvíjaný v niekoľkých verziách zároveň v príslušných vetvách projektu – tzv. branches. Vývojár si tak môže nainštalovať konkrétnu verziu ESP-IDF, ktorú chce využívať vrátane aktuálne vyvíjanej / release (produkčnej – stable) verzie projektu, alebo pre testerov je možné využiť pre-release verzie.

Každá z verzií ESP-IDF má samostatnú dokumentáciu na webových stránkach frameworku: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/>. V mojej diplomovej práci som využil na začiatku release verziu ESP-IDF 3.3, neskôr po vydaní verziu 4.0 a pre finálnu aplikáciu ESP-IDF vo verzii 4.2 [38], čo aktuálna verzia frameworku zo 7. Decembra 2020. V aktívnom vývoji je verzia 3.3.X, 4.1.X a verzia 4.3, ktorá je aktuálne v pre-release testovacej verzii.

Každá z posledných release verzií má podporu po určitý čas pokiaľ bude dostávať aktualizácie a budú v nej vyriešené chyby a bude dostávať bezpečnostné záplaty vývojármi Espressif Systems a komunitou vývojárov. Dĺžka podpory pre verzie 4.X je 30 mesiacov. Verzia ESP-IDF 4.2 má softvérovú podporu do júna 2023. Vývoj aktuálnych verzií frameworku ESP-IDF a nasledujúcich sa orientuje aj na podporu nových platforiem ESP32-C3 a ESP32-C6, ktoré sú najnovším prírastkom do rodiny procesorov ESP32 a majú novšie funkcie, ktoré nájdu využitie predovšetkým v bezpečných IoT aplikáciách.

Hlavnou výhodou frameworku ESP-IDF v porovnaní s prostredím Arduino IDE je možnosť programovania bližšie k fyzickej vrstve zariadenia, čo umožňuje vyvíjať komplexnejšie programy. Využíva operačný systém reálneho času – FreeRTOS [18]. Základnou ideou operačného systému

reálneho času je možnosť spustiť samostatné podprogramy ako úlohy, ktoré sa môžu nezávisle na sebe vykonávať na (dvojjadrovom) procesore Xtensa.

Každý úlohe je možné priradiť veľkosť pamäte – zásobníka, prioritu a aj priradenie konkrétnemu jadru, ak je to pre aplikáciu potrebné. Úlohy môžu medzi sebou vzájomne komunikovať. Tento typ komunikácie sa označuje aj ako medzi-úlohová (inter-task). Jednotlivé úlohy si môžu medzi sebou vymieňať dáta podmieňovať tak spúšťanie určitej úlohy, ktorá čaká na dáta z inej úlohy, napríklad úloha pre odoslanie dát na webserver čaká na vykonanie merania druhou úlohou.

Existujú rôzne druhy blokovania úlohy, ktorá čaká na dáta, napríklad front (Queue), semafor (Semaphore). Blokovanie úlohy je možné používať aj v operáciách, kedy viacero úloh pristupuje k rovnakému médiu, ktoré nie je schopné obslúžiť všetky úlohy súčasne – napríklad pri zápise do pamäte na konkrétny ofset.

Úloha, ktorá môže pristupovať k médiu má takzvaný MUTEX (záмок), ktorý úlohu oprávňuje k prístupu k médiu. Po ukončení operácie daná úloha MUTEX ukončí a dovolí ho použiť inej úlohe, ktorá má záujem pristúpiť k médiu. Je to riadiaci mechanizmus prístupu k pamäti, médiu.

**Fragment kódu – Queue [39] s čakaním úlohy consumer\_task na vykonanie úlohy producer\_task, výmena inkrementovanej hodnoty – dynamického počítadla so sekundovou inkrementáciou:**

```
QueueHandle_t q=NULL;

//TASK PRIJIMATELA (CAKA POKYM TASK ODOSIELATELA POSLE CISLO)
void consumer_task(void *pvParameter) {
    unsigned long counter;
    if(q == NULL){
        printf("Queue nepripravene ");
        return;
    }
    while(1){ //prijem hodnoty z inter-task komunikacie
        //hodnota sa vypise az po prijati na UART monitor
        xQueueReceive(q,&counter,(TickType_t)(1000/portTICK_PERIOD_MS));
        printf("Hodnota prijata cez queue: %lu \n",counter);
        vTaskDelay(500/portTICK_PERIOD_MS); //wait for 500 ms
    }
}

//TASK ODOSIELATELA (ODOSIELA CISLO KAZDU SEKUNDU)
void producer_task(void *pvParameter){
    unsigned long counter=1;
    if(q == NULL){
        printf("Queue nepripravene \n");
    }
}
```

```
return;
}
while(1){ //odosielanie hodnot do medzi-úlohovej komunikacie
//Hodnota sa na UART monitor vypise pri kazdom odoslani
printf("Odoslana hodnota cez Queue: %lu \n",counter);
xQueueSend(q,(void *)&counter,(TickType_t)0); // add the counter value to the queue
counter++;
vTaskDelay(1000/portTICK_PERIOD_MS); //wait for a second
}
}

void app_main() {
q=xQueueCreate(20,sizeof(unsigned long)); //Vytvorenie Queue
if(q != NULL){
printf("Queue uspesne vytvorene\n");
vTaskDelay(1000/portTICK_PERIOD_MS); //wait for a second
xTaskCreate(&producer_task,"producer_task",2048,NULL,5,NULL);
printf("Producer task spusteny\n");
xTaskCreate(&consumer_task,"consumer_task",2048,NULL,5,NULL);
printf("Consumer uloha spustena\n");
}
else{
printf("Vytvorenie Queue zlyhalo");
}
}
}
```

## 5. Arduino Core

Implementácia podpory mikrokontrolérov ESP32 do prostredia Arduino IDE [4]. Obsahuje nástroj `esptool.py` pre nahrávanie firmvéru do ESP32, kompilátor, sadu knižníc, ktoré sú rozhraním nad ESP-IDF. V paradigme jazyka Wiring [40] – zjednodušený jazyk C, je možné spúšťať jednoduchými funkciami zložitejšie podprogramy funkcií zapísaných v ESP-IDF.

Vývoj Arduino Core podporuje a zastrešuje v súčasnosti Espressif Systems. Celé rozhranie Arduino Core [41] je navrhnuté pre programovanie a vývoj blízko k aplikačnej vrstve mikrokontrolérov. Je tak vhodnou alternatívou pre používateľov a vývojárov s elementárnou znalosťou programovania, umožňuje im vytvárať jednoduché programy aj pre IoT aplikácie bez hlbších znalostí programovania.

Prostredie Arduino IDE je obľúbené najmä pre ľahkú syntax jazyka Wiring. Pre náročnejších užívateľov je možné spúšťať aj konkrétne funkcie frameworku ESP-IDF a jeho komponenty. Neumožňuje však nastavenie špecifickej konfigurácie, vykonanie voľby oscilátora (interný / externý) pre časovače, neumožňuje voľbu aplikačného jadra, či jadra protokolu. Nemá podporu konzolovej aplikácie, či vývojárskych nástrojov, ktoré sú dostupné v ESP-IDF. Vývojár tak nedokáže obsluhovať a vytvoriť program pre ULP koprocesor v rozhraní Arduino IDE bez inštalácie dodatočných nástrojov pre podporu (ULPTOOL pre Arduino Core [42]).

Funkcie Arduino Core sú značne obmedzené a ponúkajú iba možnosť využitia základných možností ESP32, ktoré sú však ďaleko rozsiahlejšie. Nedokáže tak využiť plný potenciál tejto platformy. Použitie Arduino Core sa viaže k aplikačnej vrstve vyvíjanej aplikácie a neumožňuje programovanie bližšie k fyzickej vrstve mikrokontroléra.

V prostredí Arduino IDE je možné pracovať aj s FreeRTOS a využívať úlohy vytvorené prostredníctvom plánovača nezávisle na sebe podobne ako vo frameworku ESP-IDF. Keďže jazyk Wiring nemá funkciu `app_main()`, je potrebné úlohy vytvárať vo funkcii `setup()`, ktorá sa vykoná raz alebo je možné využiť nekonečnú slučku `loop()`, ktorá je označená ako hlavná úloha a je ekvivalent k spomenutej `app_main()` funkcii.

Arduino Core je stále vo vývoji na Githubu. Ukážkové programové implementácie (projekty) sa vytvárajú dodatočne podľa už existujúcich programových implementácií vo frameworku ESP-IDF, kde sú vyladené a ktorého API Arduino Core využíva.

## 6. Vzdialená aktualizácia firmvéru

OTA (Over-The-Air) je termín opisujúci metódy distribúcie aktualizácii softvéru, konfigurácii a firmvéru pre zariadenia [43]. V praxi sa stretávame predovšetkým s takzvanými vzdialenými (remote) OTA aktualizáciami, ktoré sú distribuované prostredníctvom centrálného bodu, najčastejšie servera, ktorý je pre všetky zariadenia dosiahnuteľný cez internet. Tento typ aktualizácie poznáme z operačných systémov Windows, Android, iOS.

Operačný systém Windows dokáže používateľa na novú aktualizáciu upozorniť vyskakovacím (popup) oknom. Mobilné operačné systémy (Android, iOS) používajú oznámenie notifikáciou, prípadne dokážu aktualizáciu prevziať a nainštalovať bez nutnosti akcie používateľa. Aktualizácie môžu byť systémové, programové s rôznou prioritou. Programové aktualizácie sa viažu ku konkrétnemu programu / softvéru, ktorý sa v operačnom systéme používa ako aplikácia.

Aktualizácie poskytujú ochranu pred bezpečnostnými hrozbami, ktoré sa prirodzene vyvíjajú s nárastom výpočtového výkonu a možností počítačov a zariadení. Ponúkajú nové funkcionality operačných systémov a programov, opravujú chyby. Pre vzdialené OTA aktualizácie sú kladené isté špecifiká, keďže musí byť prenos dát realizovaný zabezpečeným kanálom a zariadenie musí dôverovať aktualizácii, ktorú prevezmú z distribučného OTA servera.

OTA aktualizácie našli uplatnenie aj pre IoT zariadenia, ktoré je možné vzdialene aktualizovať cez internet, alebo z LAN (lokálnej) siete na základe podporovanej OTA metódy bez nutnosti fyzického prístupu k zariadeniu [44]. Medzi IoT platformy podporujúce OTA aktualizácie sa radí aj ESP32, ktorý som využil vo svojej diplomovej práci pre experimentálne overenie dostupných OTA.

Prostredia ESP-IDF a Arduino IDE sa líšia v programových implementáciách pre OTA aktualizácie a spôsobe samotnej aktualizácie rôznymi metódami. Pre overenie a otestovanie možností OTA aktualizácii v programových implementáciách som využil Arduino Core pre ESP32 v release verzii 1.0.4 v prostredí Arduino IDE verzie 1.8.4 a pre vývoj finálnej aplikácie framework ESP-IDF vo verzii 4.2. Oba programovacie jazyky používajú špecifickú tabuľku partícií, ktorá definuje logické rozdelenie flash pamäte na viacero bootovateľných partícií.

### 6.1. Aktualizácia firmvéru v Arduino Core

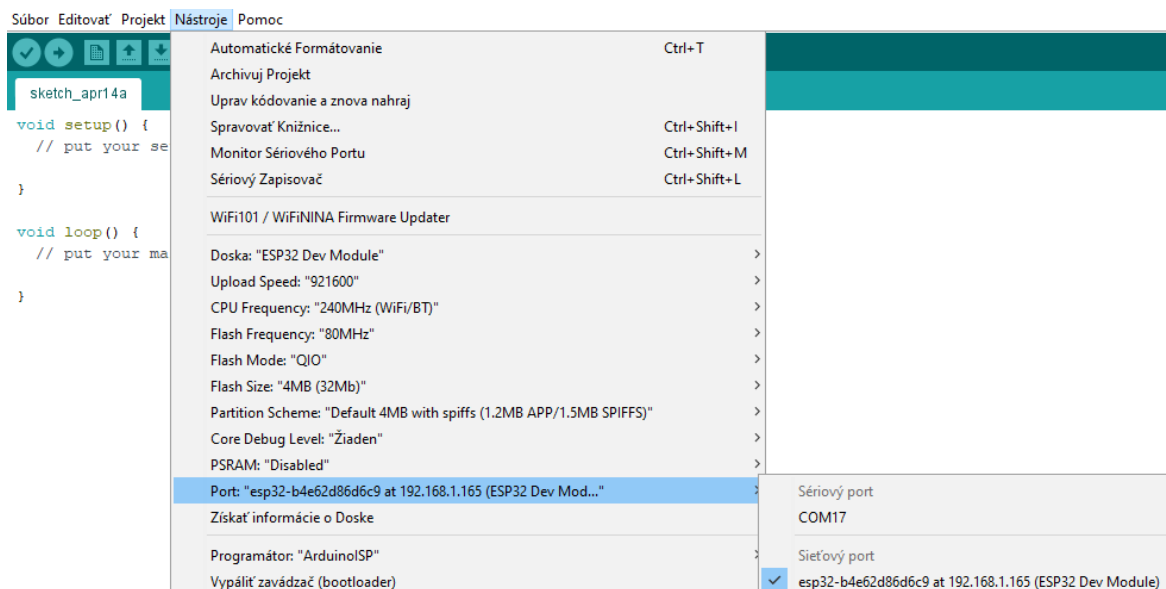
Projekty ukážkových implementácií v jazyku Wiring obsahujú možnosť aktualizovať firmvér cez LAN sieť. V jednom z prípadov podporovaných OTA metód v Arduino IDE je možné využiť OTA sieťový port, na ktorom počúva ESP32 a dokáže cez neho aktualizáciu prijať. Dostupný sieťový OTA identifikátor ESP32 sa zobrazuje v prostredí Arduino IDE v časti Nástroje → Porty, kde sú dostupné

aj fyzické COM (Sériová linka) porty, ktoré slúžia na komunikáciu počítača s ESP32 cez UART-UART rozhranie a najmä pre účely UART monitoru.

Pri využívaní Basic OTA projektu nie je možné funkcionality UART monitoru využiť, ak nie je mikrokontrolér fyzicky pripojený k počítaču cez USB-UART rozhranie. Programová implementácia pre tento typ aktualizácie – Basic OTA [45] vyžaduje, aby používateľ, ktorý firmvér do zariadenia aktualizuje bol v rovnakej sieti, ako ESP32. Údaje o WiFi sieti (meno a heslo WiFi siete) sú nastavené v zdrojovom kóde a sú súčasťou projektu Basic OTA.

Aby bolo možné vôbec aktualizácie vykonávať, prvýkrát musí byť OTA program nahratý do ESP32 fyzicky prostredníctvom USB-UART rozhrania cez USB kábel, alebo cez programátor, ak využívame samostatný čip ESP32 bez USB-UART prevodníka. Následne je možné nahráť aktualizovaný firmvér do ESP32 cez OTA metódu za predpokladu, že má vývojár nainštalovaný program Python minimálne vo verzii 3, ktorý spustí nástroj `espota.py` [3], ktorý riadi celý proces aktualizácie a nahrávania firmvéru do ESP32 automaticky.

Operačný systém Windows (testovaná verzia 10) vyžaduje vypnutie brány Firewall pre úspešnú aktualizáciu, keďže brána blokuje prichádzajúce pakety zo zdrojového portu 3232, ktorý Basic OTA využíva. Ak ESP32 neodpovedá, vyprší čakacia slučka (timeout) pre vytvorenie spojenia umožňujúceho vykonanie aktualizácie, nahrávanie firmvéru skončí chybou. Na Obr. 6 v prostredí Arduino IDE je dostupný sieťový OTA port, na ktorom počúva ESP32 medzi fyzickými COM portami.



Obr. 6 Sieťový OTA port v prostredí Arduino IDE

Po prvotnom nahratí OTA firmvéru Basic OTA sa v Arduino IDE v časti Nástroje → Porty zobrazí okrem fyzických COM portov aj sieťový OTA port, ktorý je možné vybrať a nahrávať prostredníctvom



neho nový firmvér do mikrokontrolér ESP32. Sieťový port má identifikátor v tvare „esp32-MAC\_ADRESA at IP\_ADRESA“ v LAN sieti, prípadne je možné definovať „Hostname“ zariadenia v sieti prostredníctvom mDNS (Multicast DNS) služby, ktorú je možné taktiež na ESP32 spustiť a sieťový port sa zobrazí v tvare „HOSTNAME at IP\_ADRESA“.

Pre operačný systém MacOS je potrebné využiť službu Bonjour [46], keďže mDNS služba nie je natívne na tomto operačnom systéme podporovaná. Nový program je možné po úspešnej aktualizácii nahráť priamo z vývojového prostredia Arduino IDE. Po kliknutí na možnosť Nahratia programu sa vykoná jeho kompilácia a následne sa cez sieť firmvér odošle do mikrokontroléru, ktorý ho zapíše do dostupnej aplikačnej partície.

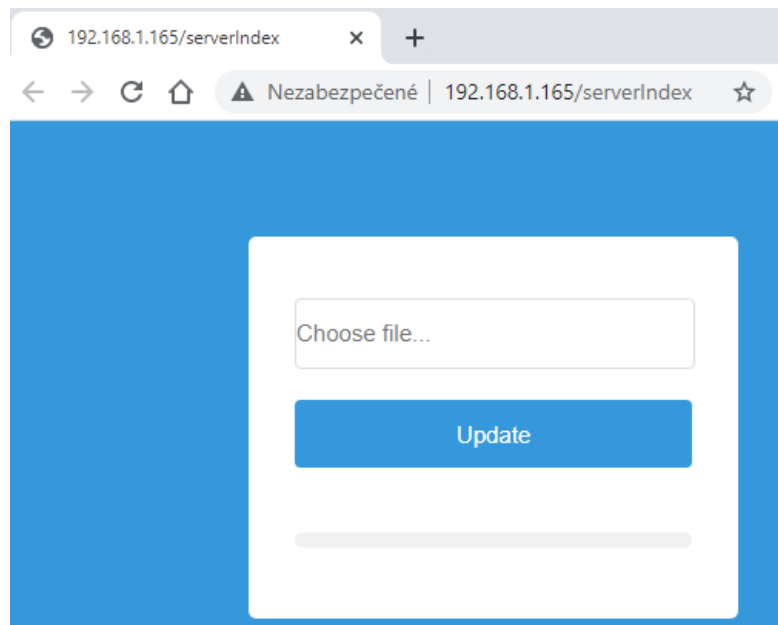
Je však potrebné dodať, že nový program musí taktiež obsahovať OTA časť programu, ak chce používateľ do budúcnosti program opäť aktualizovať vzdialene. V prípade nahratia nového firmvéru cez OTA bez OTA implementácie sa nový firmvér nahrá, spustí bez možnosti ďalšej aktualizácie, vysielanie sieťového OTA portu sa ukončí a nebude naďalej zobrazovaný v Arduino IDE v časti Porty (obnova dostupných portov sa realizuje znovu-otvorením Portov v záložke Nástroje).

Výhodou je ľahká aktualizácia firmvéru priamo z prostredia Arduino IDE. Nevýhodou tejto možnosti OTA aktualizácie je, že ESP32 vždy prijme nový firmvér prostredníctvom OTA aktualizácie aj v prípade nahratia duplicitného (totožného) firmvéru. Programová implementácia Basic OTA umožňuje spustiť aj primitívne funkcionality pre overenie integrity firmvéru a zvolenie špecifického OTA portu pre nahrávanie firmvéru.

Integritu firmvéru je možné garantovať prostredníctvom známeho hesla v otvorenom texte, alebo jeho hašovanej hodnoty v MD5 (algoritmus odtlačku správy 5) formáte [47], ktoré je obsahom firmvéru nastavená v zdrojovom kóde. Hašovacia funkcia MD5 je v súčasnej dobe zastaralá a poskytuje nízku úroveň ochrany, keďže hašované hodnoty MD5 majú vysokú mieru kolízií, resp. sú známe veľké databázy hesiel a textových reťazcov uložených v MD5 formáte, čo by znamenalo prelomenie hesla v rádoch milisekúnd.

Druhým spôsobom možnosti aktualizovať softvér v implementáciách OTA pre Arduino Core je využitie OTA Web Updater projektu [48]. Tento typ OTA aktualizácie využíva ESP32 spustené v režime HTTP webservera (port 80), na ktorý je možné prísť prostredníctvom prehliadača. Vzhľad a grafiku webovej stránky je možné štylizovať cez CSS (kaskádové) štýly. Prístupovať je možné na IP (logický identifikátor zariadenia komunikujúceho v sieti) adresu ESP32 v LAN sieti, alebo cez doménové meno, ktoré je možné nastaviť cez mDNS službu.

Tu však platí, že v prípade HTTP portu je adresa webstránky v tvare doménové\_meno.local (postfix .local je nutný pre použitie a rozpoznanie doménového mena v LAN sieti) pre jej správne preloženie a načítanie HTML webaplikácie v prehliadači. Pre úspešné rozpoznanie zariadenia v sieti je potrebné používať na koncom zariadení DNS server v LAN sieti, nie DNS server poskytovateľa internetového pripojenia. Na Obr. 7 je vizualizované webové rozhranie OTA Web Updater, ktoré slúži pre aktualizáciu firmvéru.



Obr. 7 OTA Web Updater

Súčasťou webového rozhrania je HTML formulár, prostredníctvom ktorého je možné nahráť firmvér do mikrokontroléra ESP32. Binárny súbor je možné vygenerovať v Arduino IDE a uložiť ho pre jeho neskoršie nahranie prostredníctvom HTML formulára. Táto forma aktualizácie je však viac zdĺhavá v porovnaní s Basic OTA. Vývojár musí program skompilovať, uložiť a skompilovaný firmvér nahráť cez formulár do ESP32. Aj tu platí, že pre budúcu aktualizáciu musí nový súbor okrem novej časti programu obsahovať aj OTA Web Updater, aby mohla byť OTA metóda zachovaná.

Prvýkrát sa firmvér nahráva prostredníctvom USB kábla / programátora pre spustenie webservera. Formulár prijme akýkoľvek binárny súbor, aj duplicitný, žiadnym spôsobom neoveruje integritu firmvéru, ani jeho pôvod. Pre vyššiu bezpečnosť je možné využiť login formulár nad pôvodným nahrávacím formulárom, alebo možnosť využitia HTTP autentizácie menom, heslom. Keďže je však komunikácia po HTTP protokole nešifrovaná, útočník dokáže získať odoslané informácie cez HTML formulár rôznymi nástrojmi, napríklad cez Wireshark [49], ktorými môže monitorovať sieť.

V tomto prípade je možné firmvér aktualizovať aj z inej siete ak sa ESP32 nachádza a je dosiahnuteľná na verejnej IP adrese, avšak nezabezpečený HTTP protokol nie je vhodný

pre aktualizáciu firmvéru. Firmvér môže byť kýmkoľvek podvrhnutý, pozmenený, na čo ESP32 nedokáže v tejto implementácii OTA metódy zareagovať ochranným prostriedkom. Potenciálne nebezpečnú aktualizáciu (firmvér) mikrokontrolér spustí.

## 6.2. Aktualizácie firmvéru v ESP-IDF

Framework ESP-IDF obsahuje pokročilejšie implementácie OTA aktualizácii. Nájdeme tu iba projekty pre vzdialené OTA aktualizácie, ktoré sa realizujú spôsobom prebratia aktualizácie z externého zdroja – cloudu, webservera, ktorý je dosiahnuteľný cez internet s využitím zabezpečeného spojenia.

Mikrokontrolér vykonáva požiadavku na preddefinované sieťové umiestnenie, kde je distribuovaný firmvér. Pripojenie je možné realizovať cez technológiu WiFi, alebo Ethernet (riadiaci čip LAN8720 [50] / IP101GRI [51]), ktorý je možné pripojiť k ESP32 cez RMII (Reduced media-independent interface) [52] rozhranie a používať ho s WiFi ovládačom ako PHY (fyzická vrstva) [53] perifériu.

Pre overenie možností aktualizácii som využil technológiu WiFi a projekty „Simple OTA“ a „Native OTA“, ktoré sú v ESP-IDF dostupné. Oba projekty využívajú zabezpečené HTTPS (zabezpečený hypertextový prenosový protokol) spojenie na vzdialený webserver, na ktorý vykonávajú GET požiadavku pre stiahnutie firmvéru. ESP32 má kryptografický hardvérový akcelerátor AES, ktorý sa využíva pre prenos cez zabezpečený prenosový kanál.

Pre realizáciu zabezpečeného spojenia cez HTTPS sa vyžaduje certifikát certifikačnej autority, ktorá vydala certifikát pre webserver. Na rovnakom princípe dnes fungujú aj webové prehliadače, ktoré dôverujú vydavateľovi – certifikačnej autorite, ktorá vydala certifikát pre webserver (doménu) a podpísala ho svojim súkromným kľúčom. Využíva sa princíp reťaze dôvernosti (Chain of Trust).

**Certifikát certifikačnej autority v .pem formáte, ktorý je vložený do programu v procese kompilácie:**

```
-----BEGIN CERTIFICATE-----
MIIDTzCCAjCCFGoeG0BevdCzZX61CSkNsw1yL0geMA0GCSqGSIb3DQEBCwUAMGQx
CzAJBgNVBAYTA1VTMRMwEQYDVQQIDApGYWt1IFN0YXR1MRYwFAYDVQQHDA1GYWt1
IExvY2FsaXR5MRUwEwYDVQQKDAxGYWt1IENvbXBhbnkxETAPBgNVBAMMCGVzCDMy
LnNrMB4XDTE4MTIxNDE4MTcyNVoXDTEzMTIxMzE4MTcyNVoZDELMAkGA1UEBHMx
VVMxEZARBgNVBAGMCKZha2UgU3RhdGUxFTJAUzA1UEAwIIZXNwMzIuc2swggEiMA0G
CSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQAdeima7p/eIaGtXPM2QyD3+tDI+61tB
bPzHUPupBK3Eyi1bcTHGg0Lq8cbCdFVhJJnvRGoIdz/knhTIkMphMvTC5UJLooPx
RnI9KSdxws2n/wXYvPS9TxtHz6zv0Ua9Dp+Ocxvz0wzG1skhd+RPAm9zUJDDJ42P1
FIjcehxH93GgrHzXz6zk1Ft01/8IACbt7g18/qBnjJNA8SS0BoYgfrnNOvn6k1yk
ZZtyfIp68Nw1fdveMd8Ig2pUIuvTh7MyUFjumCDLqAs2J+SenUYHEcfZJK4Vmdfo
bZam4hx8f0BGrxqDhDMuTmh/Jqa8ZYANfr7ctD1DU6u7aceGYmSgCinPAGMBAAEW
DQYJKoZIhvcNAQELBQADggEBACA5n68AsjspvR3N33jF83FZqDXuvQbmY71Bm2nN
Yh2265f9mRgr003z61HKUfs5BJXxxahAKPQ07H9zQuYvIW0y8Jcgs4F1FhAuqgss
```

```
KWuqJicf/nORPq/MC7mTLsGVrCo3R3x8U842j/Jdi0Ygb7AgFmp5F7DoS88QEMuK
2UbJSA0czc24NIimg9Y8Ve6VS4m+1sjs177L15JCuDeZ7UBegmYoDmEIC/MohSD5t
goX0+fCH4EsIkd6Zh8P6GT1RCg1yGrSFgKTV/6HPSptAFvMFUofnFedCYJXrkjKc
psVaEdVBIZ/a4jcB6mkQm06+KusHmWSC/VNYLD6+pjmIE0=
-----END CERTIFICATE-----
```

Pri realizácii mojej bakalárskej práce som túto problematiku riešil a certifikát pre webserver som generoval a podpísal vlastnou certifikačnou autoritou. Certifikačná autorita je stále platná a rovnako tak i certifikát webservera, keďže bol generovaný na obdobie piatich rokov, do roku 2023. Využil som certifikáty generované algoritmom RSA [54] s dĺžkou kľúča 2048 bitov, keďže kratšie kľúče nie sú podporované na strane mikrokontroléru ESP32 a v súčasnosti už ani nie sú bezpečné. Certifikáty je možné generovať aj metódou ECC (kryptografia na báze eliptických kriviek), experimentálne som overil pre eliptickú krivku „prime256v1“ s dĺžkou kľúča 256 bitov.

Obe metódy so spustiteľnými .bat súbormi pre automatizované vygenerovanie certifikátov kryptografickým nástrojom OpenSSL [55] a inštrukciami sú súčasťou prílohy A spoločne so súborom „ssl.conf“, ktorý certifikáty do webservera implementuje. Konfiguračný súbor je pre operačný systém Cent OS [56] a pre serverovú službu HTTPD [57], iné operačné systémy môžu využívať inú službu pre webserver, kde môže byť implementácia podobná.

#### **Príklad generovania certifikátu certifikačnej autority, webservera nástrojom OpenSSL pre metódu RSA:**

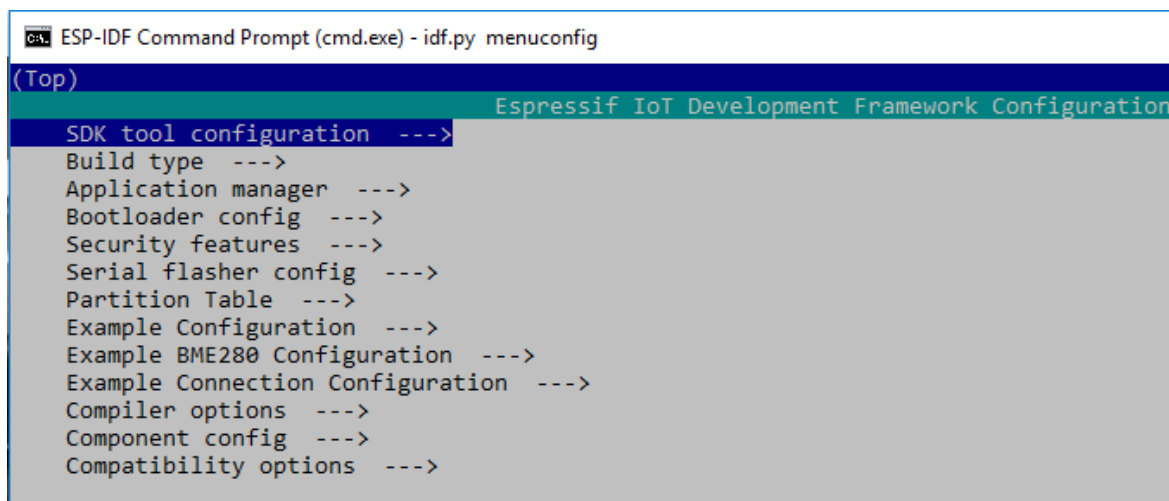
```
rem CA cert
openssl genrsa -out myCA.key 2048
openssl req -x509 -config certificate-authority-options.conf -new -nodes -key myCA.key -sha256 -days 1825 -
out myCA.pem
openssl x509 -outform pem -in myCA.pem -out myCA.crt

rem server cert
openssl genrsa -out server.key 2048
openssl req -config options.conf -new -key server.key -out server.csr
openssl x509 -req -in server.csr -CA myCA.pem -CAkey myCA.key -CAcreateserial -out server.pem -days
1825 -sha256 -extfile server.ext
openssl x509 -outform pem -in server.pem -out server.crt
```

Certifikát certifikačnej autority v .pem (čitateľom) formáte som do oboch projektov vložil do predpripraveného – prázdneho súboru „ca\_cert.pem“ v priečinku server\_certs. Certifikát je vložený do aplikácie v procese kompilácie. Konfigurácia oboch základných projektov s OTA implementáciou sa realizuje cez Menuconfig (konfiguračné menu), ktorý je možné vyvolať priamo z konzolovej aplikácie ESP-IDF.

Na Obr. 8 je Menuconfig otvorený v projekte Native OTA frameworku ESP-IDF. Na obrázku sú viditeľné tri konfiguračné podmenu, ktoré slúžia na konfiguráciu atribútov, ktoré využíva daný projekt. Najčastejšie je to „Example Configuration“ a „Example Connection Configuration“. Vývojár si môže definovať ďalšie menu pre konfiguráciu iných atribútov, ktoré v projekte potrebuje. V tomto prípade je to menu: „Example BME280 Configuration“, ktoré slúži pre konfiguráciu komunikačnej zbernice a vývodov mikrokontroléru so senzorom.

Všetky používateľské menu sú vytvorené cez súbor „Kconfig.projbuild“ [58], ktorý ich definuje a zobrazuje v položkách menu medzi ktorými je možné prechádzať šípkami na klávesnici a konfigurovať atribúty, ktoré sú v nich naprogramované. Pri zatvorení aplikácie Menuconfig sa atribúty uložia do súboru „sdkconfig“.



Obr. 8 Menuconfig – konfiguračné menu projektov v ESP-IDF

Menuconfig ponúka štandardné menu, ktoré slúži pre nastavenie mikrokontroléru, UART rozhrania, jeho rýchlosti, nastavenie módov výpisu, zapnutie doplnkového hardvéru (akcelerátory, podpora so staršími verziami ESP-IDF a iné). Položky menu je možné rozšíriť definovaním vlastného menu, ktoré môže slúžiť pre konfiguráciu daného – používateľského projektu. Ak ide o projekt v prostredí ESP-IDF, ktorý využíva WiFi konektivitu, má štandardne samostatné menu „Example Connection Configuration“ pre zadanie mena a hesla WiFi siete na ktorú chce vývojár ESP32 pripojiť. Tieto údaje sa vložia do programu v procese kompilácie.

Projekty Simple aj Native OTA ponúkajú vytvorené menu vo svojich zložkách projektov s názvom „Example Connection Configuration“ a „Example Configuration“. Prostredníctvom menu je možné zadať meno a heslo WiFi siete štandardu WPA / WPA2-PSK a taktiež aj adresu domény (prípadne IP adresu) webservera a absolútnu cestu k firmvéru odkiaľ ho je možné prevziať. Následne je možné

program v konzolovej aplikácii frameworku ESP-IDF skompilovať a nahráť do mikrokontroléru, ktorý definované atribúty vloží do firmvéru v procese kompilácie.

Identickú kópiu skompilovaného firmvéru som umiestnil cez FTP (File Transfer Protocol) klienta na vlastnú doménu a umiestnenie „<https://esp32.sk/firmware.bin>“, kde je firmvér zapísaný a odkiaľ je ho schopný mikrokontroler ESP32 prevziať a uložiť do svojho flash úložiska. Danú cestu som nastavil aj v konfiguračnom menu projektu „Example Configuration“, aby mikrokontroler vykonal požiadavku na toto sieťové umiestnenie, kde je firmvér dostupný a mohol úspešne dokončiť vzdialenú aktualizáciu firmvéru.

Projekty Simple OTA i Native OTA fungujú po stránke pripojenia a stiahnutia firmvéru identicky. V oboch prípadoch sa vykoná HTTPS GET požiadavka na webserver a jeho sieťové umiestnenie „<https://esp32.sk/firmware.bin>“. Firmvér je stiahnutý a uložený do partície OTA\_0 alebo OTA\_1 (zapisuje sa do partície, ktorá sa aktuálne nepoužíva). Projekt Simple OTA [59] po zápise do partície a zmene OTA\_DATA príznaku pre bootovanie čaká v nekonečnej slučke. V prípade reštartu mikrokontroléru ESP32 cez tlačidlo EN (RESET), bootuje nový – stiahnutý firmvér a opätovne stiahne aj identický firmvér.

Programová implementácia projektu Native OTA [60] je obsahovo robustnejšia a obsahuje rôzne mechanizmy pracujúce s aktualizáciou, ktoré rozhodujú o jej bootovaní a reštarte mikrokontroléru. Po zápise firmvéru do partície OTA\_0 alebo OTA\_1 sa overuje verzia aktuálne bežiaceho a stiahnutého firmvéru. V prípade, že sú verzie rôzne, vykoná sa zápis do OTA\_DATA partície, ktorá definuje príznak pre bootloader, ktorý volí firmvér, ktorý je bootovaný. Nastáva reštart a ESP32 nabojuje nový firmvér. Na Obr. 9 je záznam UART monitoru pri úspešnej aktualizácii firmvéru.

Spustený firmvér z partície FACTORY z offsetu 0x20000 projektu Native OTA vo verzii 9 deteguje nový firmvér s verziou 10 pri prevzatí aktualizácie, ktorý je zapísaný na offset 0x120000 – partícia OTA\_0, nastáva prepis OTA\_DATA partície pre prioritné bootovanie nového firmvéru a vykoná sa softvérový reštart pre zavedenie nového firmvéru do RAM pamäte bootloaderom.

```

Starting OTA example
Running partition type 0 subtype 0 (offset 0x00020000)
Writing to partition subtype 16 at offset 0x120000
New firmware version: 10
Running firmware version: 9
esp_ota_begin succeeded
Written image length 1024
Written image length 2048
Written image length 3072
Written image length 4096
Written image length 5120
Written image length 6144
Written image length 7168
Written image length 8192
Written image length 9216
Written image length 849920
Written image length 850944
Written image length 851956
Connection closed,all data received
Total Write binary data length : 851956
I (26020) esp_image: segment 6: paddr=0x001ed2f4 vaddr=0x00000000 size=0x02c8c ( 11404)
I (26020) esp_image: Verifying image signature...
Prepare to restart system!
I (27610) wifi:state: run -> init (0)
I (27610) wifi:pm stop, total sleep time: 5056577 us / 24236673 us

I (27610) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:1
W (27620) wifi:hmac tx: stop, discard
W (27620) wifi:hmac tx: stop, discard
I (27650) wifi:flush txq
I (27650) wifi:stop sw txq
I (27650) wifi:lmac stop hw txq
I (27650) wifi:Deinit lldesc rx mblock:10
ets Jun  8 2016 00:22:57

```

Obr. 9 Úspešná aktualizácia firmvéru, reštart systému

Vždy po reštarte mikrokontroléru je stiahnutý nový firmvér z preddefinovaného sieťového umiestnenia, pričom sa porovnáva jeho verzia s aktuálne bežiacim firmvérom. Verzia je definovaná v textovom súbore projektu Native OTA s názvom „version.txt“, kde je možné číselne aktuálnu verziu vyjadriť, tá je vložená do firmvéru v procese kompilácie.

V prípade, že sú verzie stiahnutého a aktuálne bežiaceho firmvéru identické, ESP32 čaká v nekonečnej slučke s inkrementačným počítadlom a výzvou na reštart mikrokontroléru cez EN (RESET) tlačidlo každú sekundu. Tento stav je na Obr. 10 – výstup z UART monitora.

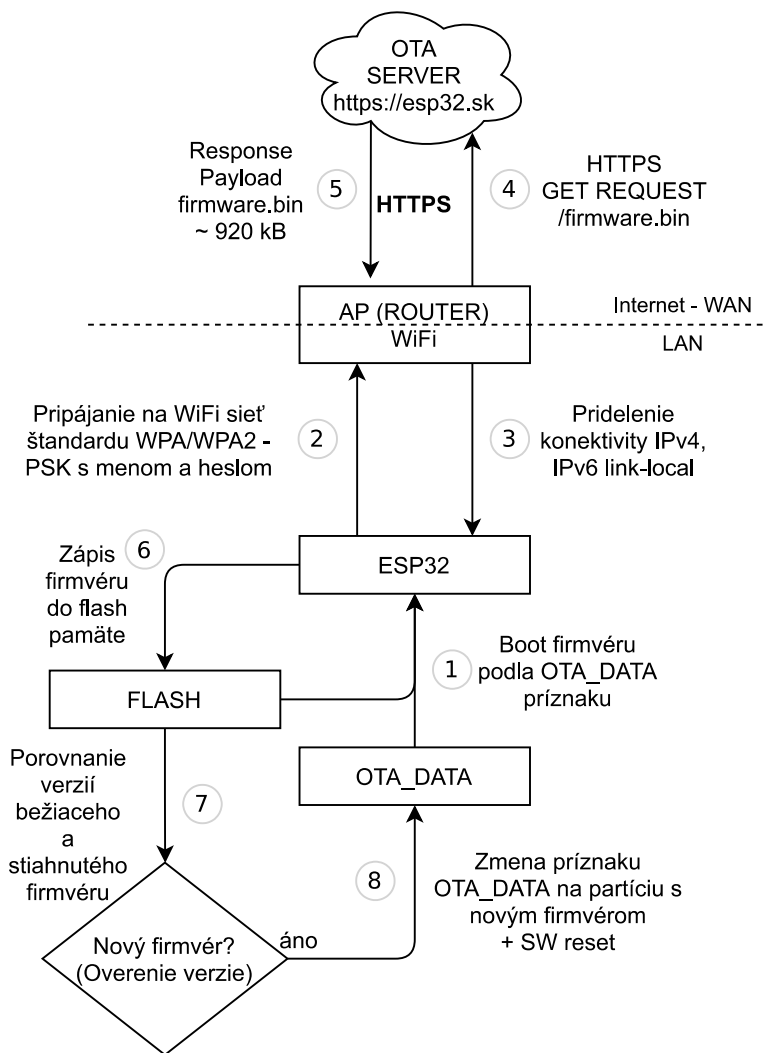
```

Starting OTA example
Running partition type 0 subtype 0 (offset 0x00020000)
Writing to partition subtype 16 at offset 0x120000
New firmware version: 9
Running firmware version: 9
Current running version is the same as a new. We will not continue the update.
When a new firmware is available on the server, press the reset button to download it
Waiting for a new firmware ... 1
Waiting for a new firmware ... 2
Waiting for a new firmware ... 3

```

Obr. 10 Stiahnutie identického firmvéru, čakanie v slučke na reštart

Samotná aktualizácia pozostáva z viacerých krokov, ktoré sú na blokovej schéme Obr. 11. Schéma obsahuje aj opisy jednotlivých krokov aktualizácie s opisom logiky, ktorá sa pri rozhodovaní o prepnutí príznaku pre OTA\_DATA partíciu vykonáva.



Obr. 11 ESP32 – bloková schéma Native OTA – proces aktualizácie

Ani jeden z projektov pre OTA aktualizáciu nerieši integritu firmvéru a jeho autenticitu. Z toho dôvodu musia byť tieto funkcionality implementované dodatočne. Framework ESP-IDF má k dispozícii rôzne vývojárske nástroje, ktorými je možné spustiť zabezpečovacie mechanizmy firmvéru pre OTA aktualizácie, i pre firmvér nahratý do zariadenia prostredníctvom USB-UART rozhrania.

Hlavnou výhodou nástrojov je, že ich nie je potrebné do aplikačného programu vkladať prostredníctvom fragmentov zdrojových kódov, ale sú dynamicky kompilované a linkované do výstupného firmvéru bez nutnosti akýchkoľvek zmien v používateľskej aplikácii napísanej v jazyku C.

Mikrokontrolér ESP32 je možné zabezpečiť rôznymi spôsobmi, ktoré prostredie ESP-IDF umožňuje na úrovni hardvéru ESP32. Jednotlivé metódy sú opísané v Tab. 2 a v nasledujúcich kapitolách sú jednotlivé metódy bližšie opísané aj s príkladom implementácie v prostredí ESP-IDF.



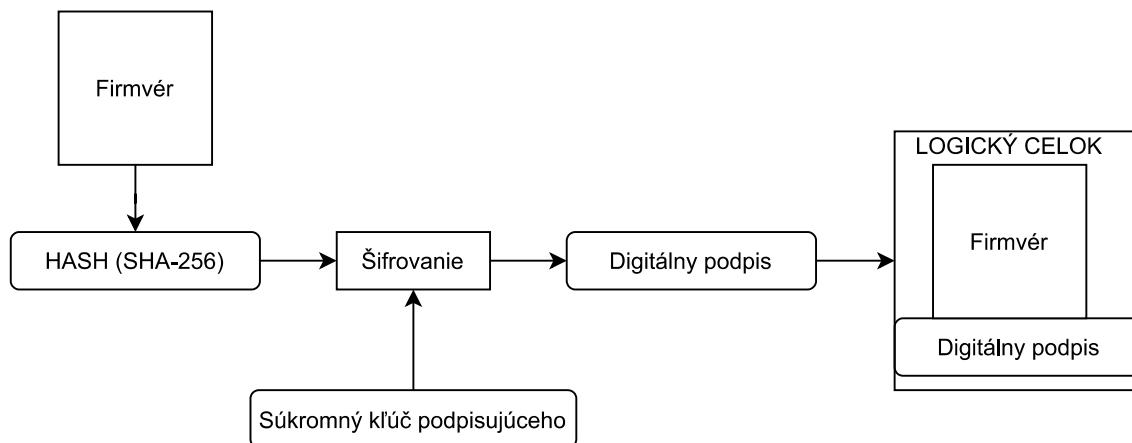
Tab. 2 Metódy zabezpečenia mikrokontroléru

Metóda	Funkcia
Digitálny podpis	Umožňuje overiť integritu (nepozmenenosť) podpísaného firmvéru. Využíva súkromný kľúč pre podpísanie firmvéru a verejný kľúč pre overenie digitálneho podpisu.
Bezpečný bootovací proces (Secure Boot)	Hardvérová funkcionálna schopnosť overiť softvérový bootloader a zamedziť jeho spustenie.  Implementuje digitálny podpis a jeho overenie vo fáze bootovania firmvéru, aj v procese aktualizácie.  Pre výpočet obrazu bootloadera využíva AES symetrický kľúč, ktorý sa používa iba pre operáciu šifrovania.
Šifrovanie flash pamäte (Flash Encryption)	Umožňuje šifrovať dôležité partície a miesta vo flash pamäti mikrokontroléru ESP32 a v reálnom čase ich pri zavedení do RAM pamäte dešifrovať.  Využíva symetrický AES kľúč pre obe operácie.

### 6.3. Metóda digitálneho podpisu

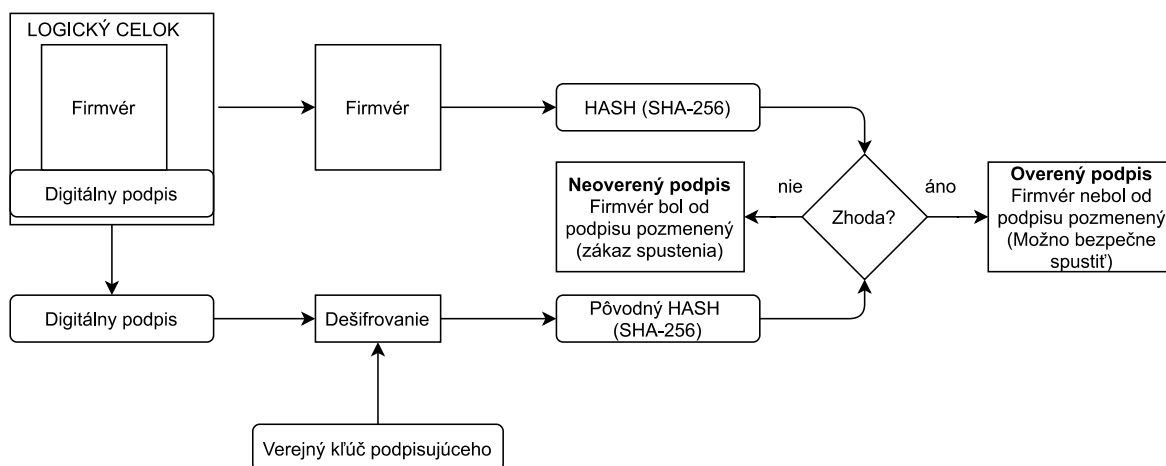
Digitálny podpis je metóda používaná pre overenie autenticity firmvéru, čím garantuje, že nebol pozmenený treťou osobou (útočníkom). Overený firmvér je bezpečný pre spustenie na vývojovej platforme ESP32 [61]. Digitálny podpis vychádza z asymetrickej kryptografie, ktorá používa dvojicu kľúčov – súkromný a verejný.

Súkromný kľúč je použitý pre podpísanie firmvéru, ktorý je distribuovaný cez OTA službu – server. Pre prenos firmvéru sa využíva bezpečný prenosový kanál – HTTPS protokol. Proces podpísania firmvéru súkromným kľúčom vydavateľa (podpisujúceho) je blokovou schémou znázornený na Obr. 12.



Obr. 12 Podpísanie firmvéru súkromným kľúčom dôveryhodného vydavateľa

Verejný kľúč je použitý pre overenie digitálneho podpisu. V procese kompilácie je vložený do softvérového bootloadera, ktorý realizuje toto overenie pri bootovaní firmvéru s možnosťou nastavenia tohto overenia aj v procese samotnej aktualizácie, kedy je firmvér zapísaný na dostupnú bootovateľnú (APP) partíciu. Proces overenia digitálneho podpisu je opísaný blokovou schémou na Obr. 13.



Obr. 13 Overenie digitálneho podpisu verejným kľúčom podpisujúceho

Najčastejšie sa pre generovanie dvojice kľúčov využíva algoritmus RSA (Rivest–Shamir–Adleman) [], alebo pre vstavané (embedded) systémy s obmedzeným výpočtovým a pamäťovým vybavením sa využíva kryptografia na báze ECC [62], ktorá umožňuje využiť porovnateľne kratší kľúč s rovnakou, alebo vyššou kryptografickou bezpečnosťou.

Spôsob generovania súkromného kľúča v ECC je v porovnaní s RSA, ktoré využíva násobenie veľkých prvočísel iný. Metóda ECC je založená na zložitosti hľadania diskretného logaritmu pre náhodný bod eliptickej krivky nad konečným poľom. Tento typ kryptografie využíva pre generovanie dvojice

klúčov aj framework ESP-IDF. Pre samotný digitálny podpis sa využíva podpisová schéma ECDSA [63].

### 6.3.1. Digitálny podpis – implementácia v prostredí ESP-IDF

Pre kryptografické operácie je možné využiť nástroj `espsecure.py` [64], ktorý je možné spustiť v prostredí ESP-IDF cez konzolovú aplikáciu frameworku. Príkazom `espsecure.py generate_signing_key private.pem` je možné vygenerovať súkromný kľúč pre eliptickú krivku „NIST256p“ [65], ekvivalentné označenie krivky je aj „prime256v1“ v nástroji OpenSSL. Generovanie sa vykonáva funkciami z Python knižnice ECDSA [66], výstupom je kľúč s dĺžkou 256 bitov.

#### Fragment .py skriptu pre generovanie súkromného kľúča s dĺžkou 256-bit:

```
def generate_signing_key(args):
    if os.path.exists(args.keyfile):
        raise esptool.FatalError("ERROR: Key file %s already exists" % args.keyfile)
    if args.version == "1":
        sk = ecdsa.SigningKey.generate(curve=ecdsa.NIST256p)
        with open(args.keyfile, "wb") as f:
            f.write(sk.to_pem())
        print("ECDSA NIST256p private key in PEM format written to %s" % args.keyfile)
```

#### Výstup .py skriptu → private.pem – súkromný kľúč s dĺžkou 256-bit:

```
-----BEGIN EC PRIVATE KEY-----
MHCCAQEElARiX9L3giT0FjCKy746uZlF1iCt4kweOXTzyT4gJVXE0AoGCCqGSM49
AWEHouQDQgAEYE6lFHaI0/ncxx4MXqDTIhoyVG0Lae+1N2FQhXdKnU5Y8VVOzNuS
I/c1xRf3q4ZTYL4fHXBggZWVrzJQjUKKNw==
-----END EC PRIVATE KEY-----
```

Eliptická krivka má parametre  $a$ ,  $b$  [67], ktoré ju definujú nad konečným (Galoisovým) poľom  $E(Gp)$ . Môžeme ju definovať súradnicami  $[x, y]$ , ktoré vyhovujú Weierstrassovej rovnici:

$$E(Gp): y^2 = x^3 + ax + b \pmod{p}$$

Verejný kľúč je možné kryptografickým nástrojom `espsecure.py` vygenerovať zo súkromného kľúča príkazom `espsecure.py extract_public_key --keyfile private.pem public.bin`. Aby bolo možné metódu digitálneho podpisu pre aplikáciu založenú na ESP32 použiť, je potrebné v prostredí ESP-IDF v konfiguračnom menu – Menuconfig v časti „Security Features“ zapnúť možnosti „Require signed apps“, „Verify via Bootloader on startup“ sa zvolia zakliknutím, pričom je možné digitálny podpis overiť aj priamo v procese stiahnutia firmvéru, pri akcii „ON\_UPDATE“. Táto akcia je tiež dostupná v menu „Security Features“ s možnosťou zapnutia.

### Fragment .py scriptu pre vygenerovanie verejného kľúča (public.bin) zo súkromného kľúča (private.pem):

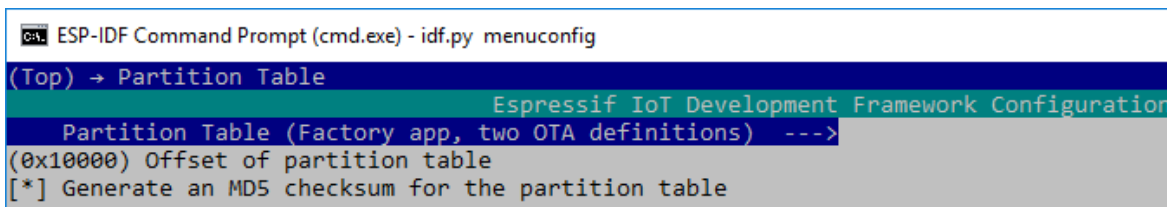
```
def extract_public_key(args):
    _check_output_is_not_input(args.keyfile, args.public_keyfile)
    if args.version == "1":
        """ Load an ECDSA private key and extract the embedded public key as raw binary data. """
        sk = _load_ecdsa_signing_key(args.keyfile)
        vk = sk.get_verifying_key()
        args.public_keyfile.write(vk.to_string())
        print("%s public key extracted to %s" % (args.keyfile.name, args.public_keyfile.name))
```

Ďalším parametrom, ktorý je potrebné nastaviť je relatívna cesta (vzhľadom na koreňový priečinok projektu) k verejnému kľúču – „public.bin“, ktorý sa v procese kompilácie automatizovane vloží do softvérového bootloadera. Keďže sa pridaním verejného kľúča zväčší veľkosť bootloadera a ten zasahuje veľkosťou do začiatočného offsetu tabuľky partícií, je potrebné posunúť jej začiatočný offset vo flash pamäti.

Cez Menuconfig v časti „Partition Table“ je možné zmeniť parameter – začiatočný offset z 0x8000 na 0x10000, čo je pre potrebu výslednej aplikácie dostatočné. Po posunutí začiatočného offsetu tabuľky partícií sa posunú aj všetky ostatné partície definované tabuľkou partícií. V Tab. 3 je možné vidieť posun offsetu u všetkých aplikačných partícií. Na Obr. 14 je viditeľná konfigurácia offsetu tabuľky partícií z pôvodnej hodnoty 0x8000 na 0x10000.

Tab. 3 Rozdiel offsetu aplikačných partícií po a pred posunutím

Názov partície	Typ	Podtyp	Aktuálny offset	Pôvodný offset	Veľkosť
FACTORY	APP	FACTORY	0x20000	0x10000	1MB
OTA_0	APP	OTA_0	0x120000	0x110000	1MB
OTA_1	APP	OTA_1	0x220000	0x210000	1MB



Obr. 14 Nastavenie offsetu tabuľky partícií v Menuconfig

Podpísanie firmvéru je potrebné realizovať manuálne po kompilácii firmvéru pred nahrávaním do dosky cez USB-UART rozhranie, alebo pred vloženíu firmvéru na OTA webserver. Pre podpísanie

firmvéru [61] je možné opätovne využiť kryptografický nástroj `espsecure.py`, ktorý je v ESP-IDF dostupný a príkazom `espsecure.py sign_data --version 1 --keyfile private.pem --output native_ota.bin native_ota.bin` sa vykoná podpísanie firmvéru „`native_ota.bin`“ súkromným kľúčom „`private.pem`“ a uloží ho do rovnakého súboru, pričom je možné využiť aj zápis podpísanej aplikácie do samostatného `.bin` súboru so zachovaním pôvodného (nepodpísaného) firmvéru.

**Fragment .py scriptu, ktorý realizuje podpísanie firmvéru súkromným kľúčom:**

```
def sign_data(args): _
    check_output_is_not_input(args.keyfile, args.output) _
    check_output_is_not_input(args.datafile, args.output)
    if args.version == '1': return sign_secure_boot_v1(args)

def sign_secure_boot_v1(args):
    """ Sign a data file with a ECDSA private key, append binary signature to file contents """
    if len(args.keyfile) > 1:
        raise esptool.FatalError("Secure Boot V1 only supports one signing key")
    sk = _load_ecdsa_signing_key(args.keyfile[0])

    # calculate signature of binary data
    binary_content = args.datafile.read()
    signature = sk.sign_deterministic(binary_content, hashlib.sha256)

    # back-verify signature
    vk = sk.get_verifying_key()
    vk.verify(signature, binary_content, hashlib.sha256)

    # throws exception on failure
    if args.output is None or os.path.abspath(args.output) == os.path.abspath(args.datafile.name):
        # append signature to input file
        args.datafile.close()
        outfile = open(args.datafile.name, "ab")
    else:
        # write file & signature to new file
        outfile = open(args.output, "wb")
        outfile.write(binary_content)
        outfile.write(struct.pack("I", 0))

    # Version indicator, allow for different curves/formats later
    outfile.write(signature)
    outfile.close()
    print("Signed %d bytes of data from %s with key %s" % (len(binary_content), args.datafile.name,
        args.keyfile[0].name))
```

V procese podpisovania firmvéru sa z pôvodného súboru „native\_ota.bin“ vypočíta hašovaná hodnota funkciou SHA256 (bezpečný hašovací algoritmus, dĺžka hašu 256-bit). Odtlačok je zašifrovaný súkromným kľúčom a jeho výsledkom je digitálny podpis, ktorý je vložený do firmvéru za jeho pôvodný obsah.

Podpis má dĺžku 68 B, pričom prvé 4 B sú slovo tvorené nulami a nasledujúcich 64 B je samotný digitálny podpis. Na Obr. 15 je dostupná výstup konzolovej aplikácie ESP-IDF po podpísaní firmvéru súkromným kľúčom s kryptografickým nástrojom `espsecure.py`.

```
C:\Users\Martin\esp-idf\examples\system\ota\native_ota_example\build>
    espsecure.py sign_data --version 1 --keyfile private.pem --output native_ota.bin native_ota.bin
espsecure.py v3.0
Signed 851956 bytes of data from native_ota.bin with key private.pem
```

Obr. 15 Podpísanie firmvéru súkromným kľúčom v konzolovej aplikácii ESP-IDF

Tento firmvér je možné nahráť do mikrokontroléru cez USB-UART rozhranie, alebo ho umiestniť na OTA webserver, ktorý bude túto aktualizáciu distribuovať klientom. Pri nahratí firmvéru cez USB-UART rozhranie je firmvér zapísaný do partície FACTORY a je z nej aj štandardne bootovaný, ak nie je partíciou OTA\_DATA s príznakom pre bootloader nastavená iná partícia, ktorá má byť prioritne bootovaná.

Firmvér je prenesený zabezpečeným prenosným kanálom ku prijímateľovi. Využíva sa zabezpečený HTTPS protokol s end-to-end šifrovaním medzi serverom a klientom. Prijímateľ (Klient) po prevzatí firmvéru rozdelí stiahnutý súbor na časť firmvéru a digitálny podpis. Verejným kľúčom podpisujúceho, ktorý pozná dešifruje digitálny podpis a získa pôvodnú hašovanú hodnotu obsahu súboru. Hašovacou funkciou SHA256 vytvorí odtlačok firmvéru a hardvérová funkcionálna porovná oba haše. V prípade, že sa zhodujú, firmvér je autentický a nebol pozmenený treťou osobou.

Firmvér je dôveryhodný a je ho možné spustiť bezpečne na mikrokontroléri ESP32. Úspešnosť samotného overenia spustí následne funkciu z programu Native OTA, ktorá overuje, či je verzia bežiacieho firmvéru iná, ako verzia stiahnutého. V prípade, že je digitálny podpis pri stiahnutom firmvéri overený, API implementácie Native OTA pozmení OTA\_DATA príznak, ktorý definuje pre bootloader miesto, odkiaľ chceme bootovať nový firmvér a vykoná softvérový reštart mikrokontroléru ESP32 pre spustenie procesu bootovania nového firmvéru. Úspešné overenie digitálneho podpisu firmvéru v procese bootovania je na Obr. 16 – výstup UART monitora.

```

I (1095) esp_image: segment 6: paddr=0x001ed2f4 vaddr=0x00000000 size=0x02c8c ( 11404)
I (1100) esp_image: Verifying image signature...
I (1452) boot: Loaded app from partition at offset 0x120000
I (1476) cpu_start: Pro cpu up.
I (1479) cpu_start: Application information:
I (1484) cpu_start: Project name:      native_ota
I (1490) cpu_start: App version:      10
I (1494) cpu_start: Compile time:     Apr 13 2021 23:38:20
I (1500) cpu_start: ELF file SHA256:  008244dc44754ef5...
I (1507) cpu_start: ESP-IDF:         v4.2-dirty
I (1512) cpu_start: Starting app cpu, entry point is 0x400819e0
0x400819e0: call_start_cpu1 at C:/Users/Martin/esp-idf/components/esp32/cpu_start.c:287

I (0) cpu_start: App cpu up.
I (1522) heap_init: Initializing. RAM available for dynamic allocation:
I (1529) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (1535) heap_init: At 3FFB8648 len 000279B8 (158 KiB): DRAM
I (1542) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (1548) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (1555) heap_init: At 40096F9C len 00009064 (36 KiB): IRAM
I (1561) cpu_start: Pro cpu start user code

```

Obr. 16 Úspešné overenie digitálneho podpisu firmvéru pri bootovaní firmvéru

Ak firmvér neobsahuje digitálny podpis, alebo nie je platný, alebo hašovaná hodnota firmvéru nie je zhodná s dešifrovanou hašovanou hodnotou, Native OTA sa k overeniu verzie firmvéru nedostane a aktualizácia je ignorovaná. Tento typ overenia sa realizuje pri každom bootovaní firmvéru, ale aj pri prevzatí a uložení firmvéru do dostupnej OTA partície.

Ak v procese bootovania nie je digitálny podpis firmvéru overený, bootloader skúša spustiť firmvér z inej dostupnej partície – OTA / FACTORY. Ak sa nepodarí spustiť firmvér ani z jednej z dostupných partícií, bootovací proces sa ukončí a mikrokontrolér nenabootuje firmvér, vykonáva cyklický reštart a opakuje pokus o bootovanie. Na Obr. 17 je prípad nenabootovania firmvéru ani z jednej dostupnej aplikačnej partície.

```

W (831) esp_image: image valid, signature bad
E (832) boot: Factory app partition is not bootable
E (832) esp_image: image at 0x120000 has invalid magic byte
W (837) esp_image: image at 0x120000 has invalid SPI mode 194
W (843) esp_image: image at 0x120000 has invalid SPI size 11
E (850) boot: OTA app partition slot 0 is not bootable
E (856) esp_image: image at 0x220000 has invalid magic byte
W (862) esp_image: image at 0x220000 has invalid SPI mode 241
E (868) boot: OTA app partition slot 1 is not bootable
E (874) boot: No bootable app partitions in the partition table
ets Jun  8 2016 00:22:57

rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2

```

Obr. 17 Neúspešné bootovanie firmvéru zo všetkých aplikačných partícií

Prípad, kedy nie je overený digitálny podpis firmvéru z príznakovej partície – FACTORY (ofset 0x20000) je ukázaný na výpise UART monitora na Obr. 18. bootloader následne bootuje firmvér z OTA\_0 partície (ofset 0x120000), kde digitálny podpis overí a zavedie firmvér do RAM pamäte. Existujúci príznak v OTA\_DATA partícii nie je prepísaný. Po reštarte mikrokontroléru ESP32 bude

v procese bootovania prioritne bootovaný ako prvý firmvér znova z partície FACTORY, kde digitálny podpis overený nebol.

```
I (480) esp_image: segment 6: paddr=0x000ed2f4 vaddr=0x00000000 size=0x02c8c ( 11404)
I (485) esp_image: Verifying image signature...
E (485) secure_boot: image has invalid signature version field 0xffffffff
E (490) esp_image: Secure boot signature verification failed
I (496) esp_image: Calculating simple hash to check for corruption...
W (745) esp_image: image valid, signature bad
E (745) boot: Factory app partition is not bootable
I (745) boot_comm: chip revision: 1, min. application chip revision: 0
I (751) esp_image: segment 0: paddr=0x00120020 vaddr=0x3f400020 size=0x21194 (135572) map
I (812) esp_image: segment 1: paddr=0x001411bc vaddr=0x3ffb0000 size=0x03ba4 ( 15268) load
I (818) esp_image: segment 2: paddr=0x00144d68 vaddr=0x40080000 size=0x00404 ( 1028) load
0x40080000: _WindowOverflow4 at C:/Users/Martin/esp-idf/components/freertos/xtensa/xtensa_vectors.S:1730

I (819) esp_image: segment 3: paddr=0x00145174 vaddr=0x40080404 size=0x0aea4 ( 44708) load
I (847) esp_image: segment 4: paddr=0x00150020 vaddr=0x400d0020 size=0x915d0 (595408) map
0x400d0020: _stext at ???:?

I (1074) esp_image: segment 5: paddr=0x001e15f8 vaddr=0x4008b2a8 size=0x0bcf4 ( 48372) load
0x4008b2a8: xEventGroupSetBits at C:/Users/Martin/esp-idf/components/freertos/event_groups.c:626

I (1095) esp_image: segment 6: paddr=0x001ed2f4 vaddr=0x00000000 size=0x02c8c ( 11404)
I (1100) esp_image: Verifying image signature...
I (1452) boot: Loaded app from partition at offset 0x120000
```

Obr. 18 Neúspešné overenie dig. podpisu pri jednej z partícií

#### 6.4. Secure Boot v ESP-IDF

Metóda zabezpečenia bootovacieho procesu (Secure Boot) [68], skrátene MZBP. Kombinuje metódu digitálneho podpisu firmvéru a jeho overenia softvérovým bootloaderom obsiahnutom vo flash pamäti v procese bootovania a priamo v procese aktualizácie pre akciu „ON\_UPDATE“. Zároveň je MZBP rozšírená o overenie softvérového bootloadera na začiatku bootovacieho procesu, ktorú obsluhuje hardvérový bootloader (First-Stage bootloader) uložený v ROM pamäti.

V princípe ide o výpočet a porovnanie referenčného odtlačku a aktuálne vypočítaného odtlačku obrazu softvérového bootloadera, ktorý je zapísaný vo flash pamäti na preddefinovaný ofset 0x1000 s využitím AES šifrovacieho kľúča s dĺžkou 256 bitov. Kľúč je z pohľadu charakteristiky možno označiť za symetrický (použitie pre šifrovanie i dešifrovanie), využíva sa však iba na proces šifrovania.

AES kľúč je uložený v jednorazovo programovateľnej pamäti eFuse BLK2 s veľkosťou 256 bitov, odkiaľ ho nie je možné softvérovo prečítať / prepísať. Cieľom výpočtu odtlačku a následného overenia je verifikácia, že je softvérový bootloader vo flash pamäti dôveryhodný a zhoduje sa s odtlačkom, ktorý je ako referenčný uložený vo flash pamäti na preddefinovanom ofsete 0x0.

Zaručuje tak, že softvérový bootloader je od dôveryhodného vydavateľa, ktorý dokázal pre tento bootloader vytvoriť aj odtlačok s využitím referenčného AES šifrovacieho kľúča (rovnaký je zapísaný v eFuse BLK2), ktorý má k dispozícii. Ak by MZBP nebola implementovaná, mohlo by mať



za následok spúšťanie neovereného firmvéru, ktorý by mohol podvrhnúť útočník, spoločne s upraveným bootloaderom, ktorý by ignoroval digitálny podpis firmvéru.

MZBP je možné zapnúť cez Menuconfig v podmenu „Security Features“. Neúspešné overenie bootloadera môže byť spôsobené taktiež aj chýbajúcim odtlačkom na očakávanom ofsete 0x0 vo flash pamäti. Celý proces tejto metódy využíva princíp reťaze dôvernosti. Pokiaľ nie je overený softvérový bootloader, nevykoná sa bootovanie firmvéru, ani overenie jeho digitálneho podpisu.

Neúspešné overenie odtlačku bootloadera je ukázané výstupe UART monitora na Obr. 19. Mikrokontrolér ESP32 sa reštartuje v nekonečnej slučke a vypisuje cyklický výpis o neúspešnom overení bootloadera pri každom spustení ESP32. Z dôvodu, že nebol bootloader overený, nevykoná sa pokus o bootovanie firmvéru z dostupných partícií. Nie je splnený predpoklad pre možnosť prístupu k ďalšej fáze bootovacieho procesu pre bootovanie firmvéru z aplikačných partícií.

```
secure boot check fail
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
0x40080400: _init at ??:?
```

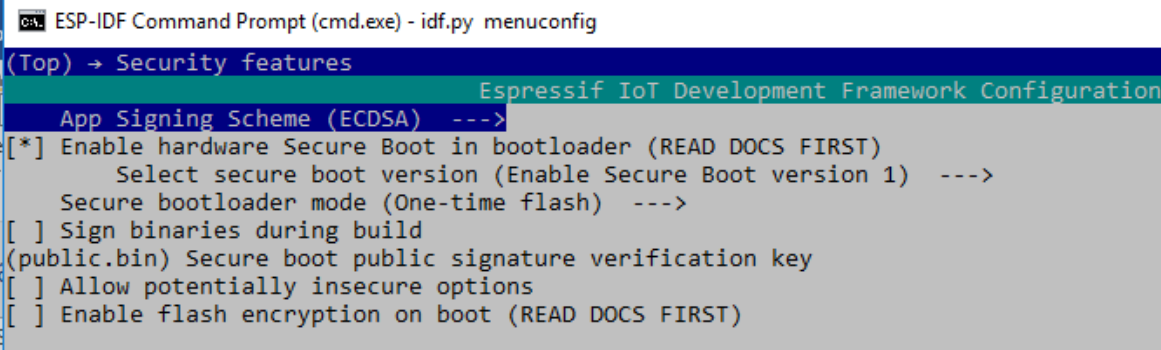
Obr. 19 MZBP – neoverený softvérový bootloader, zakázaná fáza bootovania

Z pohľadu prevádzky MZBP je možné zvoliť si jednu z jeho dvoch implementácií – „Re-flashable“, ktorá je vhodná pre vývoj aplikácií a umožňuje nahrávať AES šifrovací kľúč viackrát do určitej časti flash pamäte, ktorú hardvérová metóda zabezpečenia používa. Tento režim nie je vhodný pre produkčné (finálne) aplikácie, keďže útočník dokáže získať AES kľúč po prečítaní flash pamäte z dôvodu, že nie je chránený v eFuse, kde môže ku kľúču pristupovať iba hardvérový bootloader uložený v ROM pamäti.

Druhou implementáciou metódy, ktorú je možné použiť je „One-Time Flash“. Tento typ je vhodný pre produkčné aplikácie s jednorazovým zápisom AES šifrovacieho kľúča do eFuse BLK2. Zároveň je použitie tejto funkcionality permanentné bez možnosti jej vypnutia v budúcnosti. V nasledujúcej podkapitole je vysvetlená implementácia v prostredí ESP-IDF práve pre „One-Time Flash“ implementáciu MZBP.

Na Obr. 20 je základné nastavenie podmenu „Security Features“ v Menuconfigu pre produkčnú verziu MZBP s využitím „One-Time Flash“. Ako je z konfiguračného menu zrejmé, firmvér

nie je v procese kompilácie podpísaný, keďže je možné zvoliť možnosť automatizovaného vloženia verejného kľúča do softvérového bootloadera v procese kompilácie, alebo podpísanie firmvéru. Z toho dôvodu sa podpísanie firmvéru realizuje manuálne v konzolovej aplikácii prostredia ESP-IDF s využitím Python nástroja `espsecure.py`.



```

ESP-IDF Command Prompt (cmd.exe) - idf.py menuconfig
(Top) → Security Features
Espressif IoT Development Framework Configuration
App Signing Scheme (ECDSA) --->
[*] Enable hardware Secure Boot in bootloader (READ DOCS FIRST)
    Select secure boot version (Enable Secure Boot version 1) --->
    Secure bootloader mode (One-time flash) --->
[ ] Sign binaries during build
(public.bin) Secure boot public signature verification key
[ ] Allow potentially insecure options
[ ] Enable flash encryption on boot (READ DOCS FIRST)

```

Obr. 20 Nastavenie MZBP a verejného kľúča pre bootloader v Menuconfigu

Zapnutím tejto funkcionality cez Menuconfig v časti „Security Features“ však ešte nie je MZBP plne aktívna, pretože sa musí zapnúť manuálne zápisom AES šifrovacieho kľúča do eFuse BLK2 a zápisom bitu do potvrdzovacej 1-bitovej eFuse `ABS_DONE_0`, ktorá túto metódu permanentne zapne.

Využil som MZBP vo verzii V1, ktorého funkčnosť je opísaná v tejto kapitole. Táto hardvérová funkcionality existuje aj vo verzii V2 [69], ktorá je však viazaná na modernejšie ESP32 platformy modelov z produkcie posledných rokov (napr. ESP32-S2 [70], ESP32-C6 [71]), ktoré využívajú eFuse `ABS_DONE_1` pre spustenie MZBP V2. Použitý mikrokontroler ESP32-WROOM-32 túto metódu druhej verzie nepodporuje, aj keď má eFuse `ABS_DONE_1` k dispozícii.

#### 6.4.1. Metóda zabezpečenia bootovacieho procesu – implementácia v ESP-IDF

Šifrovací kľúč AES pre MZBP je potrebné najprv vygenerovať. Pre generovanie kľúča je možné využiť viackrát používaný vstavaný kryptografický nástroj `espsecure.py`. Príkazom `espsecure.py generate_ generate_flash_encryption_key secure-bootloader-key-256.bin` som vygeneroval 256-bitový kľúč s využitím náhodnosti operačného systému. Využíva sa funkcia `os.random()` v Pythone, ktoré zvyšuje entropiu – náhodnosť kľúča.

##### Fragment .py skriptu pre generovanie kľúča pre MZBP:

```

def generate_flash_encryption_key(args):
    print("Writing %d random bits to key file %s" % (args.keylen, args.key_file.name))
    args.key_file.write(os.urandom(args.keylen // 8))

```

Takto vygenerovaný kľúč je potrebné zapísať do jednorazovo programovateľnej pamäte eFuse BLK2 s veľkosťou 256 bitov, ktorá je pre tento kľúč určená. Pre prácu s eFuses existuje v ESP-IDF nástroj `espefuse.py` [72], ktorý sa označuje ako aj eFuse manager.

Umožňuje prácu so všetkými eFuses, ktoré sú v ESP32 dostupné. Spomenutá jednorazovo programovateľná eFuse BLK2 je špeciálna systémová eFuse s veľkosťou 256 bitov do ktorej sa zapisuje AES šifrovací kľúč pre funkcionality MZBP. Na Obr. 21 je sumár eFuses mikrokontroléru ESP32 zobrazených cez nástroj `espefuse.py` s využitím príkazu `espefuse.py summary`. Príkaz môže vyžadovať doplnenie COM portu, kde je mikrokontrolér ESP32 detegovaný operačným systémom. Do existujúceho príkazu sa doplní napríklad `-p COM20` pre cieľový COM port 20.

Na obrázku je viditeľné, že v bloku eFuse BLK1 a BLK2 je zapísaný kľúč, ktorý nie je možné softvérovým prepísať / prečítať (nie je možnosť R/W pre dané bloky – viditeľné v pravej časti atribútov), teda sa hodnota týchto blokov nezobrazí v tomto softvérovom nástroji pre hodnoty, ktoré do eFuses už boli zapísané.

```
Security fuses:
FLASH_CRYPT_CNT (BLOCK0):           Flash encryption mode counter          = 1 R/- (0b0000001)
UART_DOWNLOAD_DIS (BLOCK0):        Disable UART download mode (ESP32 rev3 only) = False R/- (0b0)
FLASH_CRYPT_CONFIG (BLOCK0):      Flash encryption config (key tweak bits) = 15 R/W (0xf)
CONSOLE_DEBUG_DISABLE (BLOCK0):   Disable ROM BASIC interpreter fallback = True R/- (0b1)
ABS_DONE_0 (BLOCK0):               Secure boot V1 is enabled for bootloader image = True R/- (0b1)
ABS_DONE_1 (BLOCK0):               Secure boot V2 is enabled for bootloader image = False R/W (0b0)
JTAG_DISABLE (BLOCK0):             Disable JTAG                            = False R/W (0b0)
DISABLE_DL_ENCRYPT (BLOCK0):        Disable flash encryption in UART bootloader = False R/- (0b0)
DISABLE_DL_DECRYPT (BLOCK0):       Disable flash decryption in UART bootloader = True R/- (0b1)
DISABLE_DL_CACHE (BLOCK0):        Disable flash cache in UART bootloader  = False R/- (0b0)
BLOCK1 (BLOCK1):                   Flash encryption key
= ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? -/-
BLOCK2 (BLOCK2):                   Secure boot key
= ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? -/-
BLOCK3 (BLOCK3):                   Variable Block 3
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W

Flash voltage (VDD_SDI0) determined by GPIO12 on reset (High for 1.8V, Low/NC for 3.3V).
```

Obr. 21 Sumár eFuses zobrazených cez nástroj `espefuse.py`

Nástrojom `espefuse.py` som AES šifrovací kľúč zapísal do eFuse BLK2 s využitím príkazu `espefuse.py burn_key secure_boot secure-bootloader-key-256.bin`. Každý príkaz zápisu do eFuse je potrebné potvrdiť zápisom hodnoty (textu) „BURN“ do konzolovej aplikácie prostredia ESP-IDF pre vykonanie zápisu. Od tohto momentu je eFuse BLK2 chránená pred softvérovým zápisom a čítaním.

Pre permanentné zapnutie MZBP, je potrebné zapísať aj potvrdzovací bit do eFuse `ABS_DONE_0`. Zápis som realizoval príkazom `espefuse.py burn_efuse ABS_DONE_0`.

#### Fragment .py scriptu pre funkciu `burn_key`, ktorá umožňuje zápis kľúča do eFuse:

```
def burn_key(esp, efuses, args):
    datafile_list = args.keyfile[0:len([keyfile for keyfile in args.keyfile if keyfile is not None]):]
```

```

block_name_list = args.block[0:len([block for block in args.block if block is not None]:)]
efuses.force_write_always = args.force_write_always
no_protect_key = args.no_protect_key
util.check_duplicate_name_in_list(block_name_list)
if len(block_name_list) != len(datafile_list):
    raise esptool.FatalError("The number of blocks (%d) and datafile (%d) should be the same." %
    (len(block_name_list), len(datafile_list)))
print("Burn keys to blocks:")
for block_name, datafile in zip(block_name_list, datafile_list):
    efuse = None for block in efuses.blocks:
        if block_name == block.name or block_name in block.alias:
            efuse = efuses[block.name]

    if efuse is None:
        raise esptool.FatalError("Unknown block name - %s" % (block_name))
    num_bytes = efuse.bit_len // 8
    data = datafile.read()
    revers_msg = None

    if block_name in ("flash_encryption", "secure_boot_v1"):
        revers_msg = "\tReversing the byte order"
        data = data[::-1]
        print(" - %s -> [%s]" % (efuse.name, util.hexify(data, " ")))

    if revers_msg:
        print(revers_msg)
    if len(data) != num_bytes:
        raise esptool.FatalError("Incorrect key file size %d.
        Key file must be %d bytes (%d bits) of raw binary key data." % (len(data), num_bytes, num_bytes * 8))
    efuse.save(data)

    if block_name in ("flash_encryption", "secure_boot_v1"):
        if not no_protect_key:
            print("\tDisabling read to key block")
            efuse.disable_read()
        if not no_protect_key:
            print("\tDisabling write to key block")
            efuse.disable_write()

    if args.no_protect_key:
        print("Key is left unprotected as per --no-protect-key argument.")
        msg = "Burn keys in efuse blocks"

    if no_protect_key:
        msg += "The key block will left readable and writeable (due to --no-protect-key)"

```

```
else: msg += "The key block will be read and write protected (no further changes or readback)"
print(msg)
efuses.burn_all()
print("Successful")
```

Potencionálny útočník nedokáže získať kľúč uložený v tejto eFuse, keďže k nej nemá prístup žiadnym softvérovým nástrojom. K predmetnej eFuse BLK2 môže pristupovať už iba hardvér, respektíve hardvérová funkcionálna MZBP prostredníctvom hardvérového bootloadera (First-stage bootloader) uloženého v ROM pamäti. Funkcionálna zabezpečuje verifikáciu softvérového bootloadera (Second-stage bootloader) mikrokontroléru ESP32 zapísaného vo flash pamäti na preddefinovanom ofsete 0x1000.

Aby bolo možné verifikovať softvérový bootloader a zabezpečiť tak proces bootovania firmvéru, je potrebné vygenerovať odtlačok (digest), ktorý je výstupom algoritmu SBDA (Secure Bootloader Digest Algorithm) a zapísať ho do flash pamäte na ofset 0x0, kde ho očakáva MZBP a používa ho ako referenčný pre overenie.

#### 6.4.2. SBDA algoritmus

Algoritmus SBDA [68] je spúšťaný hardvérovým bootloaderom. Vykoná načítanie AES šifrovacieho kľúča z eFuse BLK2 v reverznej bitovej reprezentácii a obraz softvérového bootloadera z flash pamäte z ofsetu 0x1000. Pred obraz bootloadera dosadí 128-bajtový vygenerovaný inicializačný vektor. Algoritmus vykoná zarovnanie obrazu bootloadera modulo 128 a doplní 0xFF (hexadecimálna hodnota) do reprezentácie obrazu pre dosiahnutie dĺžky 128 bajtov. Na každých 16 bajtov otvoreného textu obrazu bootloadera sa aplikuje bloková šifra AES256 v ECB móde s využitím AES šifrovacieho kľúča z eFuse BLK2 s dĺžkou 256 bitov.

Výsledný šifrovaný text má reverznú bitovú reprezentáciu. Algoritmus vymení bajt každého 4-bajtového slova šifrovaného textu, vypočíta hašovanú hodnotu SHA-512 výsledného šifrovaného textu. Výstupom je 192-bajtový reťazec, ktorý je tvorený 128-bajtovým inicializačným vektorom a 64-bajtovou hašovanou hodnotou SHA-512 zo šifrovaného textu.

Odtlačok je možné algoritmom SBDA vygenerovať aj lokálne použitím nástroja `espsecure.py`. Princíp algoritmu SBDA je totožný, využijeme lokálne dostupný vygenerovaný AES šifrovací kľúč „secure-bootloader-key-256.bin“, ktorý sme do eFuse v predchádzajúcom kroku zapísali. Príkazom `espsecure.py digest_secure_bootloader --keyfile secure-bootloader-key-256.bin --output ./bootloader-digest.bin build/bootloader/bootloader.bin` sa vygeneruje odtlačok „bootloader-digest.bin“, ktorý je možné zapísať do flash pamäte na preddefinovaný ofset 0x0.

**Fragment .py scriptu pre generovanie odťažku bootloadera: bootloader-digest.bin:**

```
def digest_secure_bootloader(args):
    """ Calculate the digest of a bootloader image, in the same way the hardware secure boot engine would do so.
    Can be used with a pre-loaded key to update a secure bootloader. """
    _check_output_is_not_input(args.keyfile, args.output)
    _check_output_is_not_input(args.image, args.output)
    _check_output_is_not_input(args.iv, args.output)
    if args.iv is not None:
        print("WARNING: --iv argument is for TESTING PURPOSES ONLY")
        iv = args.iv.read(128)
    else: iv = os.urandom(128)
    plaintext_image = args.image.read()
    args.image.seek(0)

    # secure boot engine reads in 128 byte blocks (ie SHA512 block # size), but also doesn't look for any appended
    # SHA-256 digest
    fw_image = esptool.ESP32FirmwareImage(args.image)
    if fw_image.append_digest:
        if len(plaintext_image) % 128 <= 32:

            # ROM bootloader will read to the end of the 128 byte block, but not
            # to the end of the SHA-256 digest at the end
            new_len = len(plaintext_image) - (len(plaintext_image) % 128)
            plaintext_image = plaintext_image[:new_len]

            # if image isn't 128 byte multiple then pad with 0xFF (ie unwritten flash)
            # as this is what the secure boot engine will see

            if len(plaintext_image) % 128 != 0:
                plaintext_image += b"\xFF" * (128 - (len(plaintext_image) % 128))

            plaintext = iv + plaintext_image

            # Secure Boot digest algorithm in hardware uses AES256 ECB to
            # produce a ciphertext, then feeds output through SHA-512 to
            # produce the digest. Each block in/out of ECB is reordered
            # (due to hardware quirks not for security.)

            key = _load_hardware_key(args.keyfile)
            backend = default_backend()
```

```

cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
encryptor = cipher.encryptor()
digest = hashlib.sha512() for block in get_chunks(plaintext, 16): block = block[::-1]

# reverse each input block
cipher_block = encryptor.update(block)

# reverse and then byte swap each word in the output block cipher_
block = cipher_block[::-1] for block in get_chunks(cipher_block, 4):

# Python hashlib can build each SHA block internally
digest.update(block[::-1])
if args.output is None: args.output = os.path.splitext(args.image.name)[0] + "-digest-0x0000.bin"
with open(args.output, "wb") as f:
f.write(iv) digest = digest.digest()
for word in get_chunks(digest, 4):
f.write(word[::-1])

# swap word order in the result
f.write(b'\xFF' * (0x1000 - f.tell()))
# pad to 0x1000
f.write(plaintext_image) print("digest+image written to %s" % args.output)

```

Príkaz má vstup – šifrovací AES kľúč „secure-bootloader-key-256.bin“ s dĺžkou 256 bitov v binárnom formáte (rovnaký bol zapísaný do eFuse BLK2) a obrazu bootloadera „bootloader.bin“ z priečinku build v projekte Native OTA. Pre generovanie odtlačku musí mať vývojár k dispozícii šifrovací kľúč a zodpovedá za jeho archiváciu a ochranu. Pri zápise odtlačku do flash pamäte je potrebné zvoliť cieľový ofset na 0x0, kde ho očakáva hardvérový bootloader ako referenciu pre porovnanie vypočítaného odtlačku softvérového bootloadera.

Zápis odtlačku do flash pamäte som realizoval použitím príkazu: `esptool.py write_flash 0x0 bootloader-digest.bin`. Po zapísaní odtlačku do flash pamäte a úspešnom overení odtlačku bootloadera je možné pristúpiť k ďalšej fáze bootovacieho procesu – k bootovaniu firmvéru. Pri spustení mikrokontroléru sa vykoná hardvérové vypočítanie odtlačku, kedy hardvér (ROM bootloader) pristúpi k prečítaniu obsahu eFuse BLK2 a zo známeho ofsetu, kde je zapísaný softvérový bootloader (0x1000) sa prevezme jeho obraz.

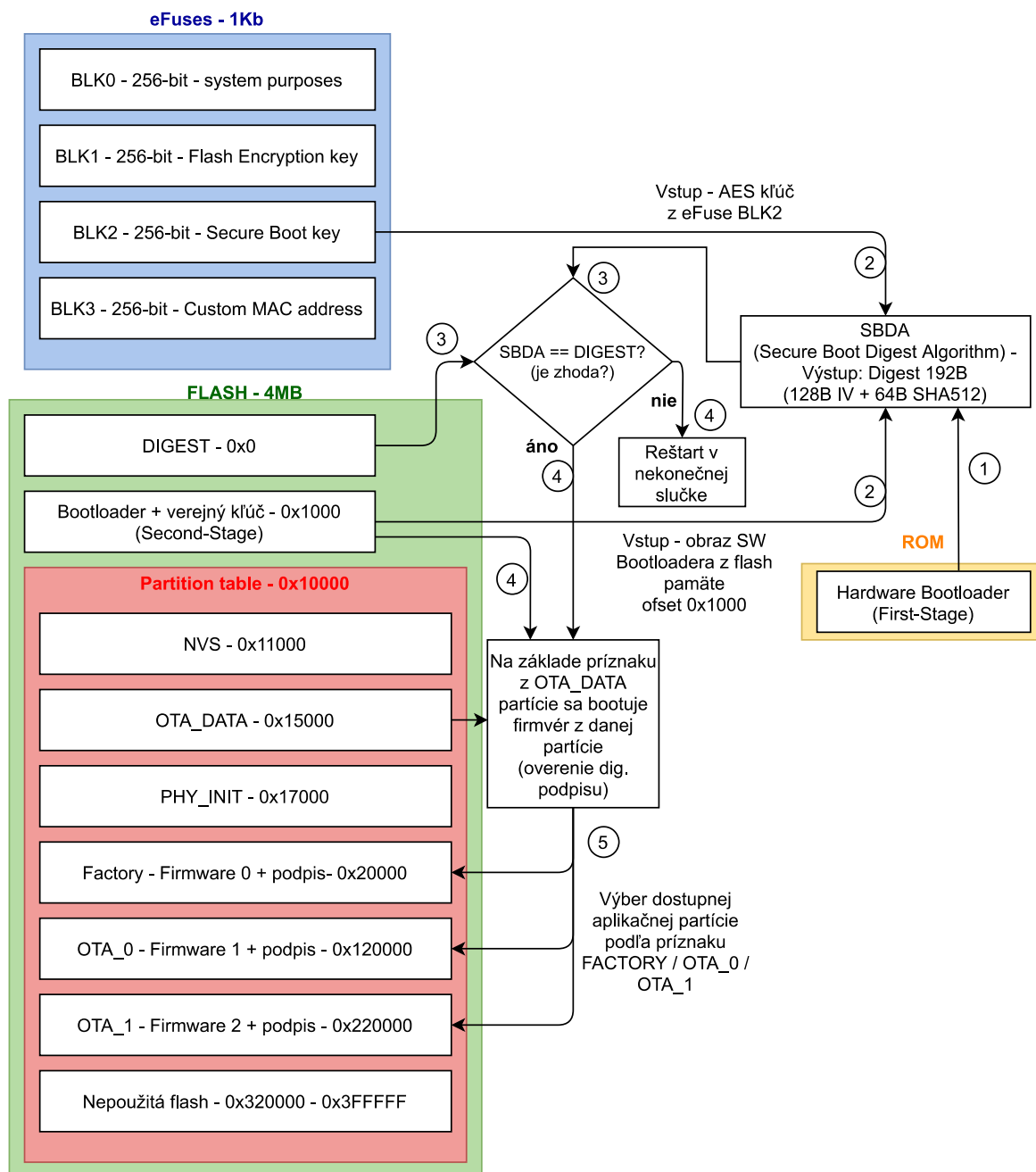
Vykoná sa algoritmus SBDA a výsledný odtlačok porovná hardvérový bootloader s odtlačkom uloženým vo flash pamäti na ofsete 0x0, ktorý slúži ako referenčný. V prípade, že sú oba odtlačky identické, hardvérový bootloader umožní spustiť softvérový bootloader a ten pristúpi k bootovaniu

firmvéru (s overením jeho digitálneho podpisu pri bootovaní, ale aj pri vzdialenej aktualizácii firmvéru...).

Ak sú odtlačky rozdielne, bootovanie je zakázané hardvérovou funkcionalitou MZBP. Tento spôsob ochrany je efektívny v prípade, ak by útočník získal fyzický prístup ku mikrokontroléru a pokúsil by sa spustiť na mikrokontroléri svoj program. V procese nahrávania firmvéru cez USB-UART rozhranie sa prepíše aj softvérový bootloader, tabuľka partícií na základe konfigurácie prostredia z ktorého sa nahrávanie firmvéru realizuje.

Vypočítaný odtlačok sa tak nebude zhodovať s odtlačkom zapísaným na ofsete 0x0 (za predpokladu, že nebude prepísaný) pri nahrávaní firmvéru. V prípade, ak by útočník dokázal zapísať iba firmvér na konkrétny ofset a prepísal by aj príznak v OTA\_DATA partícií, firmvér by naboťovaný nebol, keďže nie je digitálne podpísaný kľúčom, ktorý útočník k dispozícii nemá. Na Obr. 22 je v blokovej schéme zakreslený priebeh spúšťania systému mikrokontroléra ESP32 s využitím funkcionality MZBP po fázu pred bootovaním firmvéru.



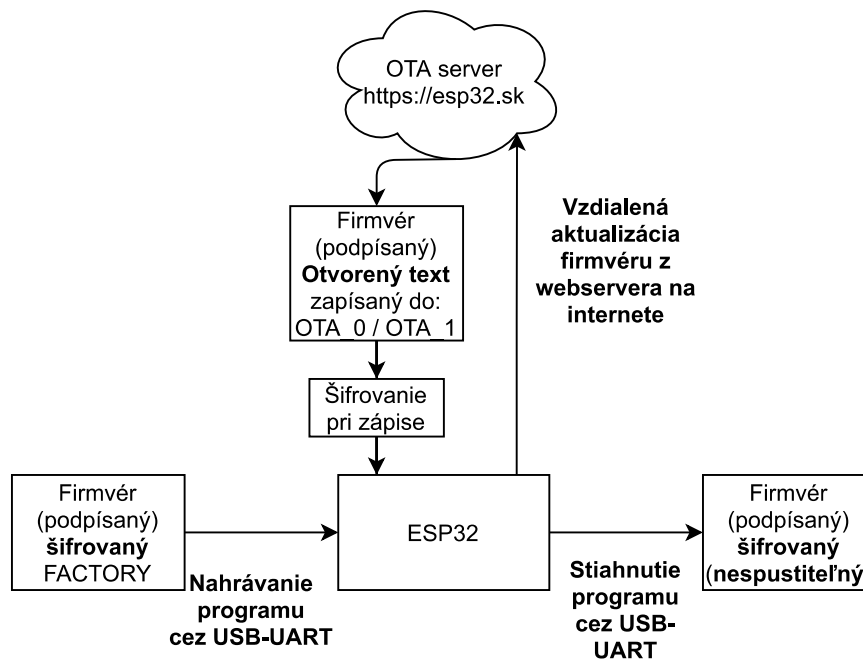


Obr. 22 Bloková schéma procesu MZBP po bootovanie firmvéru

## 6.5. Šifrovanie flash pamäte

Keďže funkcionálnosť MZBP nerieši ochranu firmvéru a softvérového bootloadera, ale iba bootovací proces, odporúča sa na základe dokumentácie ESP-IDF MZBP doplniť o metódu šifrovania flash pamäte (Flash Encryption) [73], skrátene MŠFP a využívať tieto funkcionality spolu pre vysokú úroveň bezpečnosti. Šifrovať je možné obsah celej flash pamäte, alebo jej časti, ktoré sa definujú v tabuľke partícií príznakom.

Šifrovaná časť flash pamäte nie je potencionálnym útočníkom spustiteľná, keďže dokáže prevziať cez USB-UART rozhranie iba zašifrovaný firmvér (ak sa využíva produkčná verzia MŠFP). Bloková schéma MŠFP je na Obr. 23. OTA firmvér môže byť distribuovaný štandardne v otvorenom texte, alebo po úprave v programovej implementácii v šifrovanom texte, ktorý nie je pri zápise do flash pamäte už šifrovaný.



Obr. 23 Metóda šifrovania flash pamäte – Release režim

Táto funkcia dokáže zároveň obsah flash pamäte dešifrovať v reálnom čase, ak sa daná partícia používa. Používa symetrický AES kľúč s dĺžkou 256 bitov pre šifrovanie a dešifrovanie obsahu flash pamäte. Kľúč je uložený v jednorazovo programovateľnej pamäti eFuse BLK1 s veľkosťou 256 bitov, ktorá je pre tento kľúč vyhradená.

V základnom nastavení MŠFP sú šifrované sektory:

- softvérový bootloader,
- tabuľka partícií,
- všetky firmvérové partície – APP (aplikačné),
- odtlačok bootloadera na ofsete 0x0 – (ak sa využíva MZBP).

Iné typy partícií môžu byť šifrované dodatočne pridaním príznaku encrypted do tabuľky partícií.

**Tabuľka partícií je zapísaná v .csv formáte s príznakom šifrovania pre aplikačné partície:**

```
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
```

```
nvs, data, nvs, 0x9000, 0x4000,  
otadata, data, ota, 0xd000, 0x2000,  
phy_init, data, phy, 0xf000, 0x1000,  
factory, app, factory, 0x10000, 1M, encrypted,  
ota_0, app, ota_0, 0x110000, 1M, encrypted,  
ota_1, app, ota_1, 0x210000, 1M, encrypted,
```

### 6.5.1. Proces šifrovania flash pamäte

Po vytvorení buildu (firmvér, bootloader, odtlačok, tabuľka partícií) a jeho nahratí do mikrokontroléra ESP32 sú všetky dáta vo flash pamäti v otvorenom texte – nešifrované. Za predpokladu, že je spustená aj funkcionálna MZBP, hardvérový bootloader overí softvérový bootloader pre možnosť spustenia ďalších fáz bootovacieho procesu.

Po úspešnom overení softvérový bootloader na základe príznaku zo súboru „sdkconfig“ [74] pre príznak `y` (yes) pre makro prislúchajúce k zapnutiu MŠFP načíta hodnotu eFuse `FLASH_CRYPT_CNT`. Ak je jej hodnota 0 (ešte nešifrovaná flash pamäť), nastaví a aktivuje sa blok šifrovania obsahu flash pamäte. bootloader nastaví 4-bitovú eFuse `FLASH_CRYPT_CONFIG` na hodnotu 0xF (hexadecimálna hodnota).

Operácie samotného šifrovania flash pamäte už vykonáva hardvérový bootloader, keďže softvér nemá prístup pre zápis a čítanie do už zapísaných systémových eFuses (BLK1), ktoré sú od momentu zapísania kľúča chránené voči softvérovému prepisu a čítaniu. Šifrujú sa partície a sektory vo flash pamäti označené príznakom `encrypted`. Pre šifrovanie aj dešifrovanie sa využíva (symetrický) AES kľúč zapísaný do eFuse BLK1 s veľkosťou 256-bitov, ktorá je na tento účel určená.

Pri väčších partíciách môže trvať šifrovanie až minútu. Softvérový bootloader po ukončení šifrovania nastaví eFuse `FLASH_CRYPT_CNT` na hodnotu 0x01, čo znamená, že je obsah flash pamäte šifrovaný a opätovné šifrovanie sa pri reštarte a spúšťaní systému nevykoná. Ak je použitý režim `Development`, od tohto momentu je eFuse chránená aj proti opätovnému zápisu.

MŠFP je možné prevádzkovať v dvoch režimoch:

- `development` (vývojársky) režim,
- `release` (produkčný) režim.

Pre režim `Development` nastavuje softvérový bootloader bity eFuse `DISABLE_DL_DECRYPT` a `DISABLE_DL_CACHE` na hodnotu 1, čím umožní bootloaderu cez UART opätovne nahrávať už iba šifrovaný firmvér. Pri stiahnutí firmvéru – obsahu flash pamäte je obsah dešifrovaný a vývojár tak získava výstup v otvorenom texte, čo je pre produkčné aplikácie nebezpečné, keďže sa k firmvéru môže dostať útočník a spustiť ho na svojej mikrokontrolérovej platforme.

Zároveň však v tomto režime nie je eFuse FLASH\_CRYPT\_CNT chránená proti opätovnému zápisu. Prepis tejto pamäte je podľa dokumentácie ESP-IDF možné vykonať maximálne tri krát, čím je možné šifrovanie flash pamäte po testovacej fáze projektu vypnúť a používať mikrokontrolér ESP32 bez tejto funkcie v ďalšom vývoji / produkčnej aplikácie.

V produkčnom režime Release nastavuje bootloader eFuse DISABLE\_DL\_ENCRYPT, DISABLE\_DL\_DECRYPT a DISABLE\_DL\_CACHE na hodnotu 1, čím zabraňuje dešifrovaniu obsahu flash pamäte pri jej čítaní cez UART. Výstupom v tomto prípade je tak zašifrovaný text, ktorý nie je spustiteľný na inej platforme ESP32, keďže kľúč pre šifrovanie / dešifrovanie je uložený v eFuse, ktorý nie je softvérovo prečítateľný. Útočník tak bez znalosti kľúča nedokáže firmvér spustiť.

Režim Release MŠFP taktiež zavádza ochranu proti zápisu do eFuse FLASH\_CRYPT\_CNT, čo znamená, že MŠFP už nie je možné v budúcnosti vypnúť, ani využiť 3 dostupné prepisy tejto eFuse ako v prípade Development režimu. Po prvotnom nastavení eFuses a zašifrovaní obsahu flash pamäte sa následne mikrokontrolér ESP32 reštartuje a bootuje už partície v šifrovanom texte, ktoré v reálnom čase dešifruje s využitím AES symetrického kľúča v eFuse BLK1. Dešifrovaný firmvér sa tak zavedie do RAM pamäte a spustí.

### 6.5.2. Algoritmus šifrovania flash pamäte

Algoritmus šifrovania flash pamäte využíva blokovú šifru AES256, ktorá pracuje na 16 B blokoch dát [73]. Pri tomto algoritme pracuje bloková šifra na 32 B blokoch dát, teda sa používajú dva bloky AES v sérii. AES256 používa symetrický kľúč z eFuse BLK1 s dĺžkou 256 bitov v reverznej bitovej reprezentácii.

Pre proces šifrovania a dešifrovania flash pamäte sa používajú kryptografické operácie AES opačne. Proces šifrovania je tvorený funkciou AES decrypt a proces dešifrovania flash pamäte funkciou AES encrypt. Pri šifrovaní sa na každý 32 B blok (2 bloky v sérii) otvoreného textu aplikuje unikátny šifrovací kľúč, ktorý je odvodený od hlavného vykonaním operácie XOR (exkluzívny súčet) medzi AES kľúčom a ofsetom bloku otvoreného textu vo flash pamäti.

XOR-ovanie konkrétnych bitov kľúča závisí od hodnoty v 4-bitovej eFuse FLASH\_CRYPT\_CONFIG, ktorá je štandardne nastavená na hodnotu 0xF. To zaručuje, že sú XOR-ované všetky bity odvodeného AES kľúča s ofsetom bloku dát. Zmena hodnoty zapísanej v eFuse FLASH\_CRYPT\_CONFIG by mohla znížiť kryptografickú bezpečnosť šifrovania flash pamäte, keďže by sa operácia XOR nevykonala pre všetky bity AES kľúča.

Nastavenie bitov eFuse FLASH\_CRYPT\_CONFIG a operácia XOR pre rozsahy bitov kľúča:

- ak je nastavený 1. bit, bity 0 až 66 sú XOR-ované,
- ak je nastavený 2. bit, bity 67 až 131 sú XOR-ované,
- ak je nastavený 3. bit, bity 132 až 194 sú XOR-ované,
- ak je nastavený 4. bit, bity 195 až 256 sú XOR-ované.

**Fragment .py skriptu vykonávajúci operáciu šifrovania súboru pre nahratím do flash pamäte:**

```
def _flash_encryption_operation(output_file, input_file, flash_address, keyfile, flash_crypt_conf, do_decrypt):
    key = _load_hardware_key(keyfile)
    if flash_address % 16 != 0:
        raise esptool.FatalError("Starting flash address 0x%x must be a multiple of 16" % flash_address)
    if flash_crypt_conf == 0:
        print("WARNING: Setting FLASH_CRYPT_CONF to zero is not recommended")
    if esptool.PYTHON2:
        tweak_range = _flash_encryption_tweak_range(flash_crypt_conf)
    else: tweak_range = _flash_encryption_tweak_range_bits(flash_crypt_conf)
    key = int.from_bytes(key, byteorder='big', signed=False)
    aes = None
    block_offs = flash_address
    while True:
        block = input_file.read(16)
        if len(block) == 0: break
        elif len(block) < 16:
            if do_decrypt: raise esptool.FatalError("Data length is not a multiple of 16 bytes")
            pad = 16 - len(block)
            block = block + os.urandom(pad)
        print("Note: Padding with %d bytes of random data (encrypted data must be multiple of 16 bytes long)" % pad)
        if (block_offs % 32 == 0) or aes is None:
            # each bit of the flash encryption key is XORed with tweak bits derived from the offset of 32 byte block of
            # flash
            block_key = _flash_encryption_tweak_key(key, block_offs, tweak_range)
            aes = ECB(block_key)
            block = block[::-1] # reverse input block byte order
            # note AES is used inverted for flash encryption, so
            # "decrypting" flash uses AES encrypt algorithm and vice
            # versa. (This does not weaken AES.)
            if do_decrypt: block = aes.encrypt(block)
            else: block = aes.decrypt(block)
        block = block[::-1]
        # reverse output block byte order
        output_file.write(block)
        block_offs += len(block)
```

### 6.5.3. Šifrovanie flash pamäte – implementácia v prostredí ESP-IDF

Prvým krokom pre spustenie šifrovania flash pamäte je vygenerovanie a zapísanie symetrického kľúča do príslušnej jednorazovo programovateľnej pamäte eFuse BLK1. S využitím nástroja

espsecure.py a príkazom `espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin` je možné kľúč vygenerovať. Funkcia pre generovanie AES kľúča je totožná ako v prípade kľúča pre MZBP.

Vygenerovaný kľúč „my\_flash\_encryption\_key.bin“ je následne potrebné zapísať do vyhradenej jednorazovo programovateľnej pamäte eFuse BLK1, ktoré je pre neho určená. Príkazom `espefuse.py -port PORT burn_key flash_encryption my_flash_encryption_key.bin` sa kľúč zapíše do eFuse BLK1. Aby MŠFP mohlo byť spustené pre Release režim (použitý pre hlavnú aplikáciu) bolo potrebné vyhotoviť build na základe zvolenej konfigurácie z Menuconfig.

Makrá pre šifrovanie flash pamäte sú obsiahnuté v súbore „sdkconfig“ a definujú makrá pre zvolený režim šifrovanie flash pamäte. Makrá sú vložené do bootloadera v procese vytvárania buildu. Pri prvom spustení vykoná bootloader úpravy v eFuses, ktoré sú ovplyvnené navoleným režimom šifrovanie flash pamäte.

Na Obr. 24 sú definované makrá v súbore „sdkconfig“ pre funkcionality MZBP a MŠFP (obe) v Release režime, ktoré sú v procese kompilácie vložené do softvérového bootloadera. Na základe makier a nastavení sa vykoná prvotné zašifrovanie flash pamäte na požadovaných partičiách. Keďže ide o produkčný (Release) režim, nie je možné neskôr funkcionality šifrovanie flash pamäte vypnúť.

```
# Security features
#
CONFIG_SECURE_SIGNED_ON_BOOT=y
CONFIG_SECURE_SIGNED_ON_UPDATE=y
CONFIG_SECURE_SIGNED_APPS=y
CONFIG_SECURE_SIGNED_APPS_ECDSA_SCHEME=y
CONFIG_SECURE_BOOT=y
CONFIG_SECURE_BOOT_V1_ENABLED=y
CONFIG_SECURE_BOOTLOADER_ONE_TIME_FLASH=y
# CONFIG_SECURE_BOOTLOADER_REFLASHABLE is not set
# CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES is not set
CONFIG_SECURE_BOOT_VERIFICATION_KEY="public.bin"
# CONFIG_SECURE_BOOT_INSECURE is not set
CONFIG_SECURE_FLASH_ENC_ENABLED=y
# CONFIG_SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT is not set
CONFIG_SECURE_FLASH_ENCRYPTION_MODE_RELEASE=y
CONFIG_SECURE_ENABLE_SECURE_ROM_DL_MODE=y
# end of Security features
```

Obr. 24 Konfigurácia bezpečnostných makier pre MZBP a MŠFP

Nastavením eFuses v produkčnom režime šifrovanie flash pamäte je zároveň zabránené aj dešifrovaniu obsahu flash pamäte, ktorá môže byť stále prečítaná cez USB-UART rozhranie. To by malo za následok, že by potencióálny útočník dokázal získať pôvodný firmvér, alebo celý obsah flash pamäte v otvorenom texte a dokázal by ho spustiť na svojom hardvéri.

Po prvotnom spustení funkcionality šifrovania flash pamäte vykoná hardvérový bootloader šifrovanie obsahu flash pamäte na preddefinovaných oddieloch a partíciách vo flash pamäti s označením encrypted. Od tohto momentu je možné zapísať cez UART iba šifrovaný firmvér.

V prípade, že je firmvér nahratý do mikrokontroléru ako otvorený text, nedôjde k jeho spusteniu, keďže po jeho dešifrovaní nie je v spustiteľnej podobe, tento stav je na Obr. 25, ktorý je výpisom z UART monitoru po nahratí firmvéru v otvorenom texte, mikrokontrolér sa reštartuje v nekonečnom cykle a indikuje v chybovom hlásení nemožnosť načítania obsahu flash pamäte, ktorá nie je šifrovaná.

```
rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57
```

Obr. 25 Problém s načítaním obsahu flash pamäte

Keďže využívam MZBP a MŠFP súčasne, proces nahrávania firmvéru do mikrokontroléru ESP32 cez USB-UART rozhranie je zložitejší a vyžaduje viacero krokov s presnou postupnosťou. V prvom rade je potrebné skompilovať program do binárnej podoby pre vytvorenie firmvéru aplikácie v otvorenom texte.

Následne je potrebné podpísať súkromným kľúčom firmvér v otvorenom texte a zašifrovať ho s použitím symetrického AES kľúča. Ďalším krokom je nahranie firmvéru do flash pamäte mikrokontroléru na konkrétny ofset. V tomto prípade je efektívne vytvorenie jednoduchého spustiteľného súboru .bat, ktorý dokáže všetky kroky vykonať automatizovane. Tento súbor je súčasťou prílohy A pri zdrojových kódach aplikácie pre senzorový uzol na ESP32 v prostredí ESP-IDF. Použitie príkazy, ktoré súbor spúšťa sú platné iba pre ESP-IDF vo verzii 4.2, respektíve pre nástroj espsecure.py verzie 3 (ktorý je v aktuálnej verzii ESP-IDF).

Šifrovanie firmvéru (obdobne pre bootloader, jeho odtlačok a iné súbory zapísané vo flash pamäti, ktoré majú byť šifrované), sa realizuje príkazom: `espsecure.py encrypt_flash_data --keyfile my_flash_encryption_key.bin --address 0x20000 -o ./build/app-encrypted.bin ./build/native_ota.bin`. Príkaz má viacero parametrov – AES symetrický kľúč „my\_flash\_encryption\_key.bin“ pre šifrovanie, adresu (začiatkový ofset), kde bude firmvér zapísaný, keďže algoritmus využíva operáciu XOR s ofsetom blokov dát.

Ďalšími parametrami je zašifrovaný súbor a pôvodný súbor. Do parametrov nie je možné dosadiť rovnaký názov súboru pre zašifrovaný a pôvodný. Výstupom algoritmu pri takomto vstupe súborov je prázdny súbor firmvéru s veľkosťou 0 B. Príkazom `esptool.py write_flash 0x20000 ./build/app-encrypted.bin`

je možné zapísať šifrovaný firmvér na konkrétne umiestnenie do flash pamäte, aby bol korektne dešifrovaný a spustiteľný v RAM pamäti.

OTA aktualizácie je však možné naďalej distribuovať v otvorenom texte a k ich zašifrovaniu dôjde v procese uloženia do flash pamäte mikrokontroléru ESP32 automaticky. Pri zmene funkcie pre zápis OTA aktualizácie na `esp_partition_write()` do flash pamäte je možné zapísať už zašifrovanú aktualizáciu, ktorú musí vydavateľ zašifrovať ešte pred umiestnením na OTA úložisko.

Rozhodujúcim parametrom a obmedzením tejto implementácie je však možnosť voľby iba jedného offsetu, kam bude zašifrovaný firmvér zapísaný, nakoľko algoritmus MŠFP vyžaduje zadanie offsetu pre vykonanie XOR operácie AES kľúča a blokov dát otvoreného textu. Stiahnutý firmvér tak v tomto prípade bude možné zapísať iba na jednu OTA partíciu a prepisovať ho na nej, čo zníži počet možných aktualizácií na polovicu z pohľadu životnosti flash pamäte, keďže v prípade firmvéru v otvorenom texte sa do flash pamäte zapisuje striedavo do oboch OTA partícií.

Aktualizovaný firmvér v tomto prípade nebude v procese aktualizácie šifrovaný. Ak vydavateľ pri takejto zmene nahrá firmvér v otvorenom texte, mikrokontrolér ho nedokáže spustiť, keďže ho proces dešifrovania prevedie do nespustiteľnej podoby, bootovanie zlyhá.



## 7. Experimentálny program pre aktualizáciu senzového uzla

Pri návrhu a následnej realizácii programu pre senzový uzol na ESP32 bola prioritou predovšetkým implementovať a demonštrovať čo najviac bezpečnostných mechanizmov z nástrojov prostredia ESP-IDF pre overenie integrity firmvéru s možnosťou jeho bootovania, ktorý mohol byť do flash pamäte mikrokontroléra zapísaný aj cez fyzické USB-UART rozhranie.

Pre vzdialenú aktualizáciu firmvéru som použil projekt Native OTA vo frameworku ESP-IDF, ktorý zaručuje prevzatie aktualizácie (firmvéru) z externého webservera s overením verzie firmvéru. Okrem toho zabezpečuje pripojenie na webserver cez šifrovaný protokol HTTPS, čím zaručuje prenos cez bezpečný prenosový kanál. Základný projekt som následne rozšíril o digitálny podpis pre overenie integrity firmvéru v procese bootovania a v procese aktualizácie „ON\_UPDATE“. V začiatku realizácie diplomovej práce a aplikácie pre senzový uzol som využíval ESP-IDF vo verzii release 3.3, ktorá bola v roku 2019 najvyššou release verziou.

Táto verzia ESP-IDF ešte nemala integrovanú podporu pre implementáciu digitálneho podpisu. Digitálny podpis bolo možné cez Menuconfig nastaviť, avšak problém nastal v procese kompilácie, kedy sa mal verejný kľúč vložiť do obrazu bootloadera. Následnou aktualizáciou frameworku na novšiu release verziu 4.0 sa mi podarilo digitálny podpis implementovať s plnou podporou. Firmvér som umiestnil cez FTP klienta na vlastný webserver dostupný na doméne „https://esp32.sk“ na preddefinované umiestnenie, kde ho očakáva mikrokontrolér.

V poslednej fáze testovania finálnej aplikácie pre senzový uzol som využil ESP-IDF v najnovšej verzii 4.2. Native OTA projekt sa v programovej implementácii zmenil, na čo ma upozornil kompilátor, ktorý detegoval nepodporované funkcie. Z toho dôvodu som musel vykonať korekciu hlavičkových súborov a podľa aktuálnej verzie upraviť niekoľko volaní funkcií v hlavnej úlohe `app_main()`.

Pomocou postupov v prechádzajúcich kapitolách sa mi podarilo implementovať aktualizáciu firmvéru s overením integrity v plnom rozsahu. Digitálny podpis overuje mikrokontrolér ESP32 pri každom bootovaní firmvéru z flash pamäte a tiež aj priamo v procese aktualizácie pre akciu „ON\_UPDATE“, kedy sa overuje verzia práve stiahnutého – uloženého firmvéru do flash pamäte.

Následnou implementáciou funkcionality MZBP v Release režime som zabezpečil bootovací proces a spúšťanie firmvéru od dôveryhodného vydavateľa, keďže šifrovací kľúč nie je softvérovo prečítateľný z eFuse a má ho k dispozícii iba vývojár (vydavateľ) pre vygenerovanie odtlačku, ktorý je zapísaný do flash pamäte na offset 0x0. Pretože môže byť firmvér a obsah celej flash pamäte

stiahnutý cez UART linku, bolo potrebné zamedziť prečítaniu a zároveň zamedziť možnosti spustenia firmvéru na iných mikrokontroléroch.

Z toho dôvodu som využil MŠFP – funkcionality mikrokontroléru ESP32, ktorá štandardne obsah najdôležitejších oblastí – odtlačok bootloadera, obraz bootloadera a aplikačné partície šifruje a pri zavádzaní danej oblasti do RAM pamäte dešifruje. Symetrický kľúč pre šifrovanie / dešifrovanie flash pamäte je chránený v eFuse a nie je softvérovo čitateľný ani prepisovateľný.

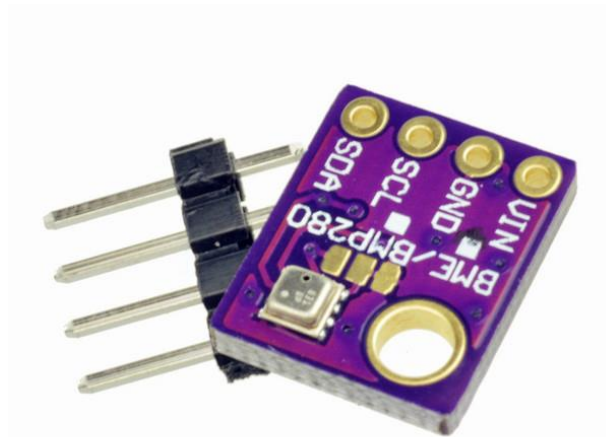
Potencionálny útočník tak dokáže obsah flash pamäte prečítať, avšak nedokáže firmvér spustiť, keďže štandardný mikrokontroler ESP32 bez potrebného kľúča v eFuse pamäti nedokáže spustiť šifrovaný firmvér. Využil som Release režim funkcionality šifrovania flash pamäte. Mikrokontroléry ESP32 prechádzajú revíziami a v súčasnosti sú produkované už v revízii 3, ktorá umožňuje zakázať prostredníctvom eFuse UART\_DOWNLOAD\_DIS sťahovanie firmvéru cez UART.

Keďže som pre vývoj práce využil vlastný, niekoľko-ročný mikrokontroler ESP32 ešte v revízii 0, na ktorom táto funkcionality nie je podporovaná, nedokázal by som zamedziť stiahnutiu firmvéru bez fyzického zásahu do mikrokontroléru (odstránenie USB portu, prerušenie vývodov hardvérovej UART zbernice).

### 7.1. Realizácia programu pre senzorový uzol

Program senzorového uzla som sa rozhodol demonštrovať na aplikácii senzorového uzla pre ESP32 navrhnutú v Bakalárskej práci v prostredí Arduino IDE [75]. Aplikáciu som musel prepracovať do prostredia ESP-IDF, kde som sa snažil o zachovanie pôvodnej funkcionality – záznam dát z meteorologického senzora.

Pôvodne som pre senzorový uzol využíval senzor Sensirion SHT21 [76], keďže som ho však nemal k dispozícii, využil som senzor Bosch BME280 [77] znázornený na Obr. 26, ktorý umožňuje záznam trojkombinácie – teplota, tlak, vlhkosť vzduchu. Využil som verziu senzora s operačnou logikou 3,3V pre plnú kompatibilitu s operačnou logikou mikrokontroléru ESP32 bez nutnosti používať napäťový delič, ako v prípade BME280 s 5V operačnou logikou, pre ktorú vývody ESP32 nie sú tolerantné.



Obr. 26 Bosch BME280 - senzor teploty, tlaku, vlhkosti vzduchu

Pre ovládanie a komunikáciu so senzorom Bosch BME280 som využil implementáciu ovládača (Drivera) v jazyku C od Bosch Sensortec [78]. Senzor komunikuje s mikrokontrolérom po I2C (Intel-Integrated-Circuit) zbernici [79] so štandardnou rýchlosťou I2C zbernice s hodinovým signálom 100 kHz.

Na základe známeho Bosch vzorca z dokumentácie BME a BMP (senzor tlaku a teploty vzduchu) senzorov je možné prepočítať absolútny tlak vzduchu na relatívny (na hladinu mora), pričom musíme do vzorca dosadiť aktuálnu nadmorskú výšku, prípadne ju môžeme vypočítať odhadom s priemernou hodnotou tlaku vzduchu na hladine mora (t.j. 1013,25 hPa), prípadne pre presnejší odhad nadmorskej výšky použijeme absolútny tlak vzduchu lokálnej meteorologickej stanice.

V mojom prípade bola odhadnutá nadmorská výška oproti skutočnej (Šuňava, okres Poprad – 854 m.n.m) nižšia o približne 20 metrov, keďže som senzor prevádzkoval v interiéri z dôvodu logiky webovej aplikácie pre riadenie výstupu na základe interiérovej teploty. Nevyužil som kompenzovaný – referenčný tlak vzduchu z lokálnej meteostanice s certifikovanými meracími prístrojmi pre aktuálnu nadmorskú výšku, ale využil som „strednú“ hodnotu 1013,25 hPa, čo je atmosférický tlak na hladine mora.

**Fragment zdrojového kódu v jazyku C – vzorec pre prepočet absolútneho tlaku vzduchu na relatívny:**

```
altitude = 44330 * (1.0 - pow(bme280_compensate_pressure_double(v_uncomp_pressure_s32)/100 / 1013.25, 0.1903));
```

**Fragment zdrojového kódu v jazyku C – vzorec pre prepočet absolútneho tlaku vzduchu na relatívny:**

```
pressure_sea = (bme280_compensate_pressure_double(v_uncomp_pressure_s32)/100) / pow(1 - ((0.0065 * altitude) / (bme280_compensate_temperature_double(v_uncomp_temperature_s32) + (0.0065 * altitude) + 273.15)), 5.257);
```

Vzorec pre prepočet absolútneho tlaku vzduchu na relatívny využíva aj teplotu vzduchu. Keďže bol senzor prevádzkovaný v interiéri, teplota bola voči vonkajšej vyššia aj o niekoľko desiatok stupňov vzhľadom na zimné a jarné mesiace, čo do prepočtu vnieslo istú nepresnosť, rádovo  $\pm 1$  hPa. V programovej implementácii ovládača od Bosch Sensortec bolo potrebné vykonať úpravu existujúcej programovej implementácie pre platformu ESP32 vo frameworku ESP-IDF.

Samotný ovládač od Bosch Sensortec obsahuje inicializáciu komunikácie medzi mikrokontrolérom ESP32 a senzorom Bosch BME280 na I2C zbernici, umožňuje nastaviť dátové GPIO vývody pre signály SCL (Synchronizačné hodiny – hodinový signál) a SDA (Synchronizované dáta), ktoré komunikácia vyžaduje, definuje rýchlosť zbernice – hodinový signál v Hz.

Ovládač od Bosch Sensortec implementuje dva prevádzkové režimy senzora Bosch BME280, ktoré je možné použiť v programovej implementácii:

- normálny,
- vynútený (nazýva sa aj FORCED).

Normálny prevádzkový režim umožňuje využiť vyššie vzorkovanie nameraných údajov pre dosiahnutie presných meraní spoločne s nastavením koeficienta IIR (filter s nekonečnou impulznou odozvou) filtra, ktorý kompenzuje krátkodobé výkyvy tlaku vzduchu a teploty. Vzorkovanie je možné nastaviť v rozsahu 1 až 16 krát [80]. Koeficient IIR filtra je možné nastaviť v rozsahu 0 až 16. Medzi jednotlivými vzorkami je možné nastavovať pauzu – tzv. StandBy čas, ktorý definuje pauzu medzi meraniami.

Hodnota sa nastavuje makrami, ktoré sú preddefinované v ovládači v rozsahu 1 až 4000 ms (možno nastaviť aj hodnoty 63, 125, 250, 500, 1000, 2000). Energeticky úspornejší je prevádzkový režim FORCED (tzv. vynútený). Nevyužíva koeficient pre IIR filter, nekompensuje žiadnym spôsobom krátkodobé výkyvy tlaku a teploty vzduchu. Využíva vzorkovanie 1 krát pre každú meranú hodnotu, teda prvá nameraná hodnota je výsledkom danej meteorologickej veličiny.

Pri výpise a porovnaní oboch režimov je normálny prevádzkový mód stabilný a ponúka konzistentné merania pri kontrole výpisu na tri desiatinné miesta (na tisíciny). Na Obr. 27 je záznam z UART monitora, ktorý poukazuje na konzistentnosť nameraných dát v režime NORMAL s frekvenciou záznamov 25Hz, teda 40 záznamov za sekundu. Teplota využíva 16-násobné vzorkovanie, tlak vzduchu (absolútny) a vlhkosť vzduchu 2-násobné vzorkovanie.

Výpis na z UART monitora obsahuje záznam veličín zľava doprava:

- teplota vzduchu,

- absolútny tlak vzduchu pre aktuálnu nadmorskú výšku,
- vlhkosť vzduchu,
- odhadnutá nadmorská výška,
- relatívny tlak vzduchu prepočítaný na hladinu mora.

```

21.88 degC / 916.938 hPa / 57.154 % / 834.634 m / 1009.112 hPa
21.88 degC / 916.939 hPa / 57.149 % / 834.621 m / 1009.112 hPa
21.88 degC / 916.940 hPa / 57.144 % / 834.608 m / 1009.112 hPa
21.89 degC / 916.940 hPa / 57.159 % / 834.615 m / 1009.112 hPa
21.89 degC / 916.939 hPa / 57.144 % / 834.618 m / 1009.111 hPa
21.89 degC / 916.941 hPa / 57.160 % / 834.607 m / 1009.111 hPa
21.89 degC / 916.940 hPa / 57.144 % / 834.610 m / 1009.111 hPa
21.89 degC / 916.940 hPa / 57.149 % / 834.612 m / 1009.111 hPa
21.89 degC / 916.940 hPa / 57.150 % / 834.615 m / 1009.110 hPa
21.89 degC / 916.941 hPa / 57.139 % / 834.602 m / 1009.110 hPa
21.89 degC / 916.939 hPa / 57.150 % / 834.620 m / 1009.110 hPa
21.89 degC / 916.938 hPa / 57.155 % / 834.622 m / 1009.109 hPa
21.89 degC / 916.937 hPa / 57.134 % / 834.630 m / 1009.109 hPa

```

Obr. 27 Konzistentnosť meraní senzora BME280 v NORMAL móde

FORCED režim nebol tak stabilný a neponúkal konzistentné merania, ako je možné vidieť na Obr. 28 z výstupu UART monitora. Namerané údaje sa v čase menili skokovito pri každom meracom cykle nad i pod strednú hodnotu (známu z merania normálnym prevádzkovým módom). Ku skokovitému priebehu nameraných veličín prispel aj fakt, že v tomto režime nie sú krátkodobé výkyvy kompenzované IIR filtrom a taktiež sa nevykonáva viacero priemerovaných meraní.

```

22.69 degC / 916.949 hPa / 53.800 % / 834.513 m / 1008.852 hPa
22.69 degC / 916.894 hPa / 53.810 % / 835.016 m / 1008.848 hPa
22.69 degC / 916.894 hPa / 53.795 % / 835.016 m / 1008.848 hPa
22.68 degC / 916.879 hPa / 53.810 % / 835.016 m / 1008.851 hPa
22.69 degC / 916.886 hPa / 53.825 % / 835.153 m / 1008.849 hPa
22.69 degC / 916.922 hPa / 53.805 % / 834.833 m / 1008.850 hPa
22.68 degC / 916.879 hPa / 53.815 % / 835.016 m / 1008.851 hPa
22.68 degC / 916.907 hPa / 53.810 % / 834.902 m / 1008.852 hPa
22.68 degC / 916.899 hPa / 53.820 % / 834.902 m / 1008.854 hPa
22.68 degC / 916.843 hPa / 53.820 % / 835.472 m / 1008.850 hPa
22.68 degC / 916.907 hPa / 53.825 % / 834.970 m / 1008.852 hPa
22.68 degC / 916.823 hPa / 53.825 % / 835.655 m / 1008.847 hPa
22.68 degC / 916.871 hPa / 53.825 % / 835.153 m / 1008.852 hPa

```

Obr. 28 Konzistentnosť meraní senzora BME280 vo FORCED móde

Mikrokontrolér ESP32 využíva z pôvodného konceptu sensorového uzla ovládanie digitálneho výstupu GPIO 23, ktorý riadi SSR (Solid-state relay) relé [81]. Tento typ relé neobsahuje v porovnaní s elektromagnetickým relé cievku, ani mechanickú časť. Z pohľadu opotrebenia jeho častí má tak prakticky neobmedzenú životnosť prevádzky, pretože jeho výkonovým prvkom je polovodičová súčiastka – triak, ktorý je ovládaný tranzistorom.

Bližší opis SSR relé OMRON G3MB-202P [82] je dostupný v prílohe A na CD nosiči. V Tab. 4 je tabuľkové zapojenie vývodov mikrokontroléru ESP32 s perifériami, ktoré boli pre senzorový uzol použité.

Tab. 4 Pripojenie vývodov ESP32 k vývodom použitých periférii

<b>ESP32</b>	<b>Bosch BME280 (senzor teploty, tlaku a vlhkosti vzduchu)</b>
3,3V	Vcc
GND	GND
GPIO22 (HW SCL)	SCL
GPIO21 (HW SDA)	SDA
<b>ESP32</b>	<b>SSR relé OMRON G3MB-202P – externé 5V napájanie</b>
GPIO23	CH1
GND	GND

Súčasťou systému je webové rozhranie, ktoré umožňuje zber nameraných údajov zo senzorového uzla a ovládanie jeho výstupu v automatickom, alebo manuálnom režime na spôsob izbového termostatu na Obr. 29 s konfiguráciou cieľovej (referenčnej) teploty a hysterézy [83]. Základné rozhranie spoločne s vygenerovaním a nakonfigurovaním certifikátov certifikačnej autority a webservera pre realizáciu zabezpečeného spojenia bolo vytvorené v mojej bakalárskej práci.

## Ovládanie relé

---

**Referenčná teplota**

**Hysteréza +-**

**ULOŽIŤ**

**Zmeniť na manual**

**Aktuálny stav: ZAP**

Obr. 29 Webové rozhranie termostatu pre ovládanie relé

Aplikácia samotného sensorového uzla využíva prenos údajov na rovnaké webové rozhranie, ktoré poskytuje OTA aktualizáciu firmvéru. Pre realizáciu bezpečného spojenia pre možnosť prenosu údajov na server som využil projekt HTTPS\_request, ktorý je dostupný v ESP-IDF a skombinoval som ho s projektom Native OTA.

Vykonal som niekoľko úprav v konfiguračných súboroch aj v samotnom programe, keďže pôvodne projekt HTTPS\_request využíva samostatný súbor pre certifikát certifikačnej autority s iným názvom ako Native OTA a v inom umiestnení (hĺbka priečinka).

Úpravou som dosiahol, že projekt Native OTA aj HTTPS request využíva rovnaký súbor s certifikátom certifikačnej autority. Programová implementácia pre spojenie oboch projektov bola priamočiara, keďže obe implementácie využívajú úlohy a tak je ich možné spúšťať nezávisle na sebe. V prípade chyby v programe a pretečení buffra sa ukončí iba konkrétna úloha a ostatné úlohy pokračujú v behu. Celkovo som využil tri HTTPS úlohy pre tri rôzne druhy požiadaviek, ktoré sa na server cyklicky vykonávajú.

Základná implementácia projektu HTTPS\_request využíva GET požiadavku. Keďže pre prenos údajov mikrokontroler vykonáva POST požiadavku, musel som do programovej implementácie vykonať úpravu vykonávanej HTTP metódy, vypočítať dĺžku payloadu (dát) a nastaviť kódovanie (application/x-www-form-urlencoded) payloadu a následne odoslať payload (na základe normy RFC 1867, ktorá ten typ prenosu a formátu dát definuje [84]).

Doba opakovania úlohy sa realizuje nastavením parametra funkcie *vTaskDelay()* v programovej implementácii s časom čakacej slučky v milisekundách. Pre lepšie rozlíšenie hlásení na UART rozhraní od jednotlivých úloh som využil systém tagovania, ktorý vypíše informáciu o názve úlohy, ktorá dané hlásenie na UART rozhranie posiela. Táto metóda bola efektívna aj pri ladení programu, nakoľko som vedel efektívne získať informáciu, ktorej úlohe je priradený daný výpis.

Na základe dôležitosti správy je možné využiť rôzne typy príznakov, ktoré dokážu výpis na UART rozhranie sformátovať do bielej, zelenej, červenej, žltej farby prostredníctvom funkcie *ESP\_LOGI* (informácia), *ESP\_LOGW* (varovanie), *ESP\_LOGE* (chyba) [85]... V Tab. 5 je zoznam úloh vykonávajúcich HTTPS požiadavku na webserver s opisom konkrétnej funkcie a logiky, ktorú vykonávajú.

Tab. 5 Úlohy sensorového uzla pre odosielanie a načítanie dát z webového rozhrania

Názov úlohy	Funkcia
https_get_task	Vykonáva pravidelný HTTPS POST požiadavku s dátami obsiahnutými v tele správy. Pri každom cyklickom behu úlohy sa vyskladá dynamická požiadavka

	<p>z nameraných údajov s kľúčom a priradením hodnoty, ktoré sú obsiahnuté v globálnych premenných.</p> <p>Vykonaním požiadavky na server a spustením .php scriptu sa vykoná zápis nameraných údajov do textových súborov a spustí sa logika termostatu, ktorá zmení stav ZAP / VYP pre výstup, ktorý načítava iná úloha. Požiadavka sa opakuje každých 15 sekúnd.</p>
https_get_task2	<p>Vykonáva pravidelný HTTPS GET požiadavku na textový súbor, načítava stav ZAP / VYP pre výstup – relé, aplikuje na GPIO23 (D23). Požiadavka sa opakuje každých 15 sekúnd.</p>
https_get_task3	<p>Vykonáva pravidelný HTTPS GET požiadavku na textový súbor, načítava payload, obsah súboru. Môže načítať jeden z dostupných stavov: OK, alebo RST. Ak načíta RST, vykoná softvérový reštart ESP, predtým opätovnou HTTPS požiadavkou potvrdí reštart, čím prepíše obsah textového súboru na OK. Požiadavka sa opakuje každých 15 sekúnd.</p>

Súčasťou práce s úlohou `https_get_task2` bolo aj využitie digitálneho GPIO vývodu 23, ktorý je využitý pre riadenie SSR relé. Inicializácia vývodu, nastavenie vývodu ako výstupu sa realizovala vo funkcii `main` ešte pred spustením jednotlivých úloh. Úloha `https_get_task` až `https_get_task3` implementovali funkcionality, ktorá existovala už z bakalárskej práce v Arduino Core.

Programová implementácia pre meteorologický senzor Bosch BME280 využíva Driver od Bosch Sensortec je navrhnutá pre jazyk C. Niekoľkými úpravami v spolupráci vedúcim práce sa nám podarilo implementovať pôvodnú programovú implementáciu v jazyku C do frameworku ESP-IDF pre možnosť inicializácie senzora Bosch BME280 v dvoch prevádzkových režimoch.

Pre každý z režimov som vytvoril samostatnú úlohu, ktorá vykonáva inicializáciu a komunikáciu so senzorom v konkrétnom režime. Aby bola iba jedna z úloh podmienená zapnutá, vytvoril som v konfiguračnom súbore „`Kconfig.projbuild`“ [58], ktorý je dostupný v priečinku projektu samostatné druhé menu (prvé menu je existujúca konfigurácia Native OTA projektu). Menu je rozvetvené na dve podmenu – „I2C Master“ a „BME280 Sensor“. Prvá podmenu „I2C Master“ ponúka zvolenie hodinového signálu v Hz a čísla GPIO pre SCL a SDA signál, ktoré sú štandardne nastavené na hardvérové I2C vývody.

Druhé menu – „BME280 Sensor“ umožňuje výber komunikačnej I2C adresy senzora Bosch BME280 (Obr. 30 opisuje hlavné podmenu a jednotlivé vetvenia menu pre „I2C Master“, „BME280 Sensor“).



Na základe logickej úrovne vývodu SD0 je jeho komunikačná adresa na I2C zbernici 0x76 (PULLDOWN), alebo 0x77 (PULLUP), obe adresy sú v hexadecimálnom tvare.

```

ESP-IDF Command Prompt (cmd.exe) - idf.py menuconfig
(Top) → Example BME280 Configuration
Espressif IoT Development Framework Configuration
I2C Master --->
BME280 Sensor --->

ESP-IDF Command Prompt (cmd.exe) - idf.py menuconfig
(Top) → Example BME280 Configuration → I2C Master
Espressif IoT Development Framework Configuration
(22) SCL GPIO Num
(21) SDA GPIO Num
(100000) Master Frequency

ESP-IDF Command Prompt (cmd.exe) - idf.py menuconfig
(Top) → Example BME280 Configuration → BME280 Sensor
Espressif IoT Development Framework Configuration
BME280 I2C Address (BME280 (0x76) - 4 PIN - ORIG. ADDR) --->
BME280 Operation Mode (Normal Mode) --->

ESP-IDF Command Prompt (cmd.exe) - idf.py menuconfig
(Top) → Example BME280 Configuration → BME280 Sensor → BME280 I2C Address
Espressif IoT Development Framework Configuration
( ) BME280 (0x77) - 6 PIN + GND SD0
(X) BME280 (0x76) - 4 PIN - ORIG. ADDR

```

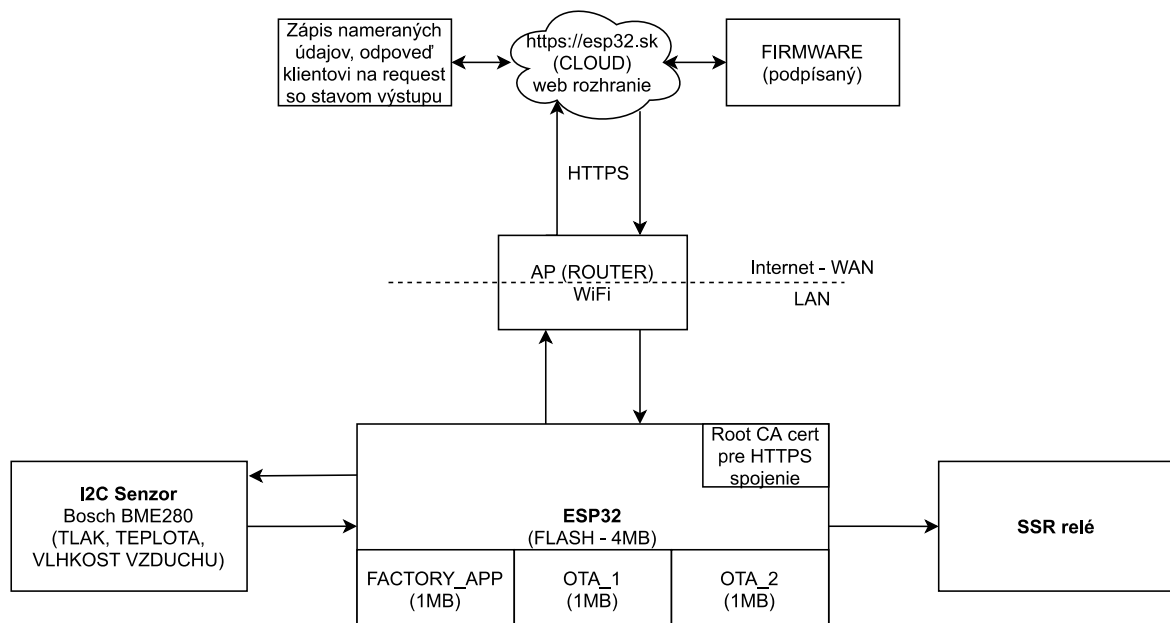
Obr. 30 Vlastné konfiguračné menu pre voľbu I2C parametrov a adresy, režimu BME280

Logická úroveň vývodu SD0 mení LSB (Najmenej významný bit) komunikačnej I2C adresy. Taktiež je možné vybrať operačný mód senzora Bosch BME280 NORMAL / FORCED v ktorom bude pracovať. Premenné obsiahnuté v menu sú uložené do makier, ktoré môžu nadobúdať príznak y (yes), n (no), označujú, či je makro definované, prípadne nadobúdajú číselnú, textovú hodnotu. Všetky konfiguračné makrá obsahujú predponu „CONFIG\_“ a sú uložené do súboru „sdkconfig“ po úpravách v Menuconfigu, kde sú jednotlivé menu vyobrazené.

Pre prístup aplikácie napísanej v jazyku C k súboru „sdkconfig“ v projekte je potrebné v programe definovať makro „COMBINED\_INIT\_CODE“. Následne už môže aplikácia používať jednotlivé makrá zo súboru „sdkconfig“. Makro zapísané v súbore „sdkconfig“ môže mať dátový typ boolean (binárny yes/no), integer (celé číslo), string (otvorený text). Na základe dátového typu, hodnoty je možné v programe vykonávať prostredníctvom direktív podmienenú kompiláciu. V mojom prípade som overoval, či je definované makro „CONFIG\_BME280\_OPMODE“ a akú hodnotu nadobúda. Hodnota 0x01 (hexadecimálna) je pre FORCED operačný režim, alebo 0x03 pre NORMAL operačný režim.

Daná hodnota je hodnotou registra, ktorým sa daný operačný režim senzora Bosch BME280 definuje pri inicializácii komunikácie mikrokontroléru ESP32 so sensorom. Konfiguračné menu vždy zdefinuje iba jeden režim, čím zabraňuje voľbe zvoliť oba režimy súčasne. Zaručuje tak aj štart iba jednej úlohy pre komunikáciu so sensorom Bosch BME280 na základe zvoleného režimu.

Každá z BME úloh zapisuje namerané údaje do globálnych premenných, ktoré používa HTTPS úloha pre prenos údajov do webového rozhrania. Táto úloha je univerzálna a funguje nezávisle na zvolenom režime senzora Bosch BME280. Celkovo tak v aktuálnej verzii aplikácie funguje v reálnom čase nezávisle na sebe päť úloh (OTA , 3x HTTPS, BME280). Bloková schéma riešenia kompletného sensorového uzla komunikujúceho so serverom zabezpečeným prenosovým kanálom je na Obr. 31.



Obr. 31 Bloková schéma riešenia sensorového uzla na báze ESP32

## 7.2. Úpravy webového rozhrania

Realizácia diplomovej práce na strane webservera spočívala v úpravách existujúceho webového rozhrania po grafickej stránke, počtu a následnej vizualizácii zaznamenaných údajov zo sensorového uzla. Zaznamenané údaje vo webovom rozhraní sú vizualizované na hlavnej stránke – ukážka zaznamenaných dát na Obr. 32.

## Miesto záznamu: Šuňava (okr. Poprad)

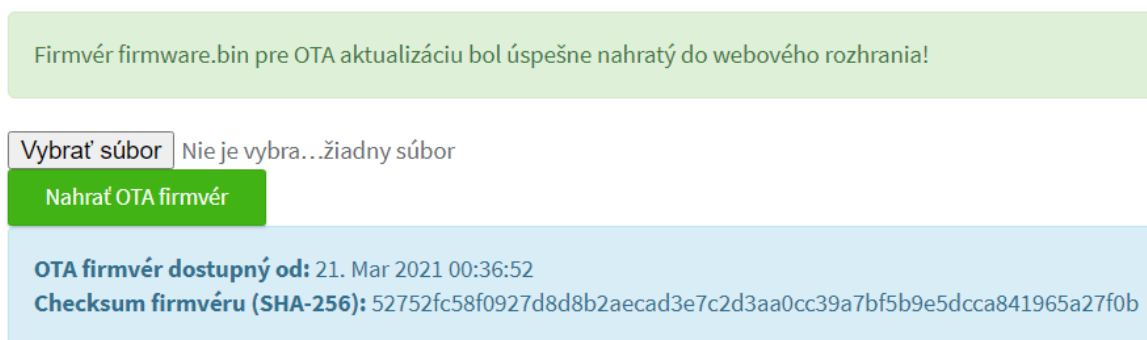
Entita	Hodnota
Teplota (BME280)	20.89 °C
Vlhkosť (BME280)	60.17 % RH
Nadmorská výška	832.23 m.n.m.
Tlak (BME280) - RAW	917.27 hPa
Tlak (BME280) - prepočítaný	1009.45 hPa
Teplota - CPU	57 °C
Hall - CPU	65443 (Analog)

13. Apr 17:54:44

Obr. 32 Vizualizácia nameraných údajov vo webovom rozhraní

Úpravy backendu (funkčnosti na serverovej strane) sa týkali PHP [86] súboru „zapisdata.php“, ktorý dokáže prevziať dáta, ktoré odosiela mikrokontrolér ESP32 a na základe ktorých vie vykonať logiku termostatu. V tomto PHP súbore som zmenil HTTP metódu pre príjem dát, pôvodná metóda HTTP GET bola nahradená za HTTP POST, ktorá je pre prenos údajov zo senzového uzla bezpečnejšia, keďže sú dáta obsiahnuté v tele správy a nie v URL adrese ako parametre kľúčov s hodnotami.

Do webového rozhrania bola implementovaná samostatná – nová PHP podstránka, ktorá umožňuje nahráť firmvér pre mikrokontrolér ESP32 v binárnom formáte cez HTML formulár. Grafické rozhranie webaplikácie s možnosťou nahratia firmvéru do úložiska webservera je na Obr. 33).



Obr. 33 Grafické rozhranie webstránky pre nahratie OTA firmvéru cez HTML formulár

Serverový jazyk PHP pri spracovaní overí formát súboru (očakáva .bin), následne ho premenuje na „firmware.bin“ (uložený firmvér na disku počítača po kompilácii projektu Native OTA je „native\_ota.bin“) a uloží ho do preddefinovaného – koreňového umiestnenia priečinku

webaplikácie, kde ho očakáva mikrokontrolér vykonávajúci GET požiadavku pre stiahnutie jeho obsahu a vykonanie aktualizácie firmvéru.

Používateľ má možnosť po nahratí firmvéru vidieť jeho kontrolný súčet – checksum (SHA-256) a tiež dátum a čas jeho publikácie. Obdobne je možné pre nahranie firmvéru do webového rozhrania použiť aj FTP klienta (napríklad WinSCP, Total Commander, Filezilla). Toto riešenie som využíval v čase, keď som ešte formulár pre nahranie firmvéru nemal vytvorený a implementovaný do webového rozhrania.

Keďže jazyk PHP nemá implementáciu podpory pre kontrolu podpísaného firmvéru (resp. súboru), nepodarilo sa mi do serverovej časti doplniť kontrolu pre akceptáciu iba podpísaného firmvéru, prípadne ho overiť dodatočne po zápise do zložky. Z dôvodu, že som chcel autorizovať používateľský prístup pre nahranie firmvéru do úložiska webservera, kde ho očakáva mikrokontrolér, implementoval som v PHP autorizáciu jednoduchého overenia prístupu (HTTP Basic Auth) [87]. Prihlasovací formulár jednoduchého overenia prístupu je na Obr. 34, slúži pre zadanie informácie pre používateľské meno a heslo.

Prihlásiť sa  
https://esp32.sk

Meno používateľa

Heslo

Prihlásiť sa Zrušiť

Obr. 34 Jednoduché overenie prístupu menom a heslom

#### Fragment kódu v jazyku PHP pre implementáciu jednoduchého overenia prístupu

```
<?php
$pouzivatelske_meno = "Username"; //Meno používateľa
$pouzivatelske_heslo = "Password"; //Heslo

$valid_passwords = array ($pouzivatelske_meno => $pouzivatelske_heslo);
$valid_users = array_keys($valid_passwords);
$user = $_SERVER['PHP_AUTH_USER'];
$pass = $_SERVER['PHP_AUTH_PW'];

//Overenie, ci bol používateľ autorizovaný
```

```
$validated = (in_array($user, $valid_users)) && ($pass == $valid_passwords[$user]);  
if (!$validated) {  
    header('WWW-Authenticate: Basic realm="My Realm");  
    header('HTTP/1.0 401 Unauthorized');  
    die ("Not authorized"); //ukončenie PHP skriptu, nenačíta sa stránka za autorizáciu  
}  
?>
```

Tento typ HTTP autorizácie umožňuje používateľovi zadať autentizačné údaje po vyzvaní serverom pri vstupe na stránku, ktorá vyžaduje autorizáciu. Autentizačné údaje sú potrebné pre úspešnú autorizáciu a následné sprístupnenie webovej stránky. V prípade neúspešnej autorizácie môže klient opakovať požiadavku a zadať znova meno a heslo. V prípade, že klient okno prihlasovacieho formulára zatvorí, webserver mu odpovie návratový kódom HTTP 401 – Unauthorized.

Výhodou tejto autorizácie je jednoduchá implementácia na strane webservera, ktorá sa môže vzťahovať na celú webovú stránku, alebo jej časť. Autorizáciu je možné umiestniť napríklad do backendu – spracovania HTML formulára, ktorým sa nový firmvér pre mikrokontrolér do úložiska webservera nahráva. Klient tak dokáže prezerať celú stránku, zadať cieľový firmvér pre nahranie a po spustení spracovania HTML formulára ho webserver vyzve k zadaniu autentizačných informácií pre dokončenie PHP skriptu.

Klient je po úspešnom overení autorizovaný (má platnú reláciu) štandardne 30 minút, ak nie je v konfigurácii webservera nastavený iný interval. Medzi výhody autorizácie určite patrí aj širokospektrálna podpora v prehliadačoch, keďže ho podporujú takmer všetky webové prehliadače. Meno a heslo sa pri odoslaní autentizačných informácií prevedie na jeden textový reťazec otvoreného textu, ktorý je tvorený menom, dvojbodkou a heslom.

Výsledný reťazec sa zakóduje do BASE64 [88] formátu a je poslaný serveru. Tento reťazec je veľmi jednoduché dekodovať a tak sa HTTP autentizácia využíva iba pri zabezpečenom prenosovom kanáli – HTTPS, ktorý vytvára šifrované spojenie priamo medzi serverom a klientom. Pri HTTP spojení cez internet sa tento typ autentizácie nevyužíva, keďže ktokoľvek dokáže obsah správy s autentizačnými dátami prečítať a dekodovať.

Pri HTTP spojení sa autentizácia využíva iba v LAN sieti, kde poznáme všetky sieťové zariadenia a sieť je dôveryhodná. Existuje aj množstvo variantov autorizácii, napríklad Digest, či Bearer token, ktorý je vhodný pre autorizáciu požiadaviek, využíva sa napríklad v cloudových riešeniach pre API s obmedzeným počtom požiadaviek za. Zdrojové kódy celej webaplikácie pre senzorový uzol sú dostupné v prílohe A na CD médiu.

### 7.3. Minimálna schéma zapojenia

Súčasťou diplomovej práce je aj návrh minimálnej schémy zapojenia, ktorá je navrhnutá pre samostatný čip ESP32-WROOM-32 a je plným ekvivalentom súčasného zapojenia s vývojovou doskou DevKit. Schéma obsahuje aj návrh napájacej časti s regulátormi napätových úrovní pre možnosť napájania externým zdrojom napájania v rozsahu 6V až 18V.

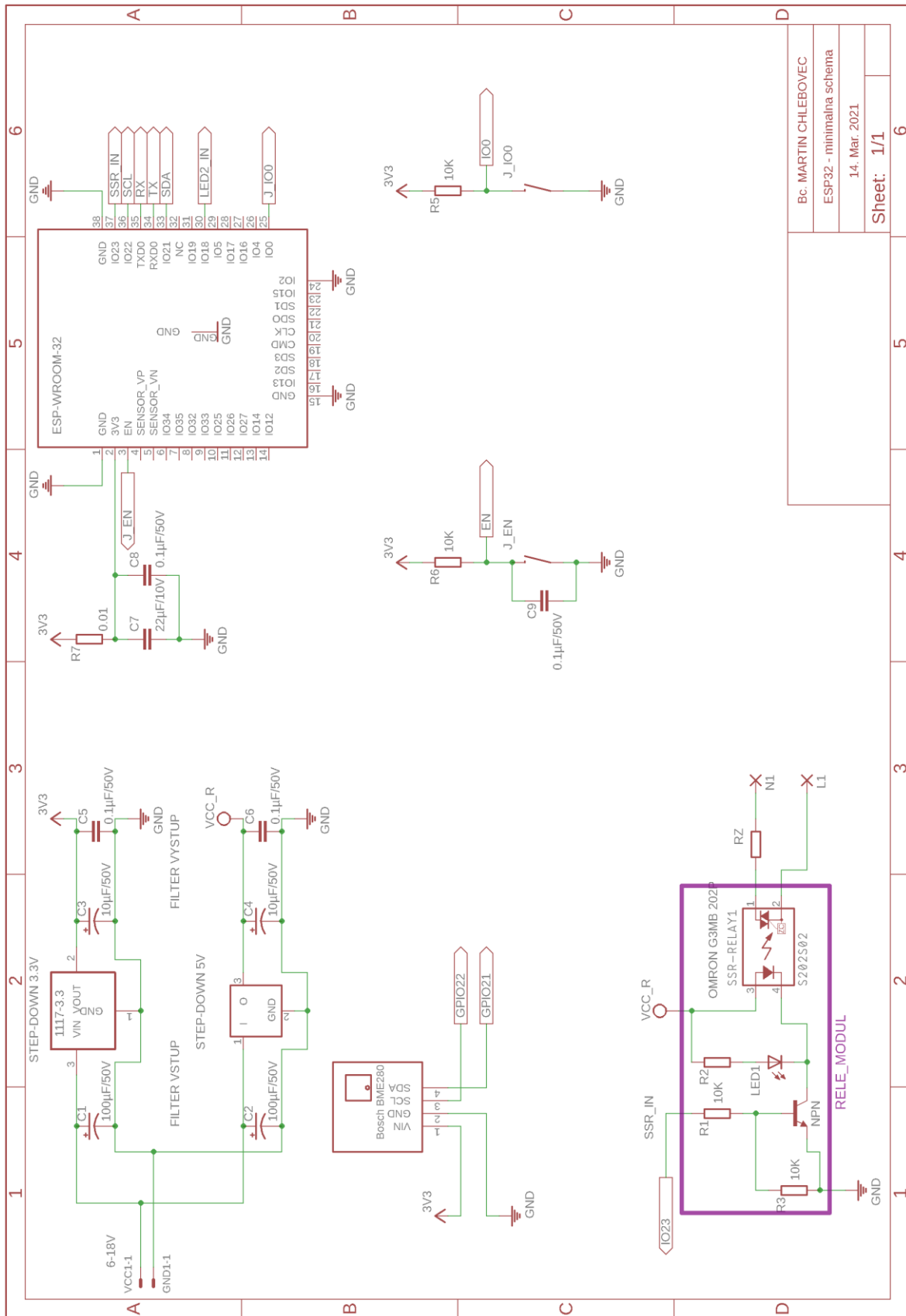
V prípade použitia samostatného čipu sú do schémy zakomponované tlačidlá pre signály EN a BOOT, ktorými je možné nastavovať špecifický režim čipu ESP32 na základe kombinácie ich stlačenia. Špecifickou sekvenciou stlačení je možné nastaviť hlavný čip ESP32 do režimu nahrávania programu (tzv. programovací režim), prípadne režim pre jeho stiahnutie, prípadne je možné vykonať reštart a spustenie bootovacieho procesu, aplikácie.

Navrhnutá časť signálov pre EN a BOOT vývod mikrokontroléru ESP32 prostredníctvom tlačidiel je navrhnutá na základe známeho problému DevKitov od Espressifu, ktorý má pre oba signály použitý kondenzátor s totožnou kapacitou, čo má za následok rovnaký napätový nábeh signálov v čase. Keďže sú oba signály pri spúšťaní mikrokontroléru v logickej 0, doska sa prepne do režimu prevzatia firmvéru, kde čaká v nekonečnej slučke a pre beh firmvéru a proces bootovania sa vyžaduje fyzický reštart tlačidlom.

Tento problém je nebezpečný najmä v prípade, ak ESP32 riadi kritickú aplikáciu na ktorú pri power-on cykle (odpojení a pripojení napájania) nedokáže reagovať až do fyzického reštartu tlačidlom, kedy sa spustí firmvér z bootovateľnej partície. Tento typ hardvérového problému je ľahko riešiteľný, keďže stačí použiť jeden z kondenzátorov s inou kapacitou, alebo jeden nepoužiť ako v prípade mojej minimálnej schémy zapojenia, aby bola na jednom zo signálov logická úroveň 1 okamžite.

V schéme zapojenia (na Obr. 35) je zakreslený relé modul OMRON G3MB-202P ku ktorému však neexistuje dostupná elektrotechnická schéma (existuje iba pre relé samostatne, nie pre relé modul), jeho zapojenie je tak iba približné. Schéma zapojenia periférie s mikrokontrolérom ESP32 je plne ekvivalentná s tabuľkovým zapojením vývodov z tabuľky (Tab. 4) v kapitole 7.1.

Na výstup relé je možné pripojiť záťaž (v minimálnej schéme zapojenia na Obr. 35 označené ako RZ) – pre demonštratívne účely som využil LED žiarovku na 230V s výkonom 5W. Záťaž môže mať maximálne prúdové zaťaženie 2A pri 230V vzhľadom na charakteristiku odporúčanú v katalógovom liste predmetného SSR relé OMRON G3MB-202P. Schéma je exportovaná z programu pre návrh elektrotechnických schém Autodesk Eagle [89].



Bc. MARTIN CHLEBOVEC  
 ESP32 - minimálna schéma  
 14. Mar. 2021  
 Sheet: 1/1

Obr. 35 Minimálna schéma senzového uzla

## Záver

Diplomová práca opisuje možnosti IoT mikrokontrolérovej platformy ESP32 v rôznych aplikáciách s možnosťou aktualizácie vzdialenej aktualizácie firmvéru lokálnymi, alebo externými OTA metódami. Zameriava sa predovšetkým na bezpečnosť, ktorú je možné implementovať vývojárskymi nástrojmi v prostredí frameworku ESP-IDF. Zároveň demonštruje aplikáciu senzorového uzla s využitím funkcionalít pre vzdialenú aktualizáciu firmvéru cez internet.

Metóda digitálneho podpisu má široké využitie nielen v mikrokontrolerovej technike. Digitálny podpis je spoľahlivá metóda pre overenie integrity akéhokoľvek súboru (dokumenty, spustiteľné súbory, firmvér). Pre túto metódu moja implementácia využíva súkromný kľúč generovaný eliptickou krivkou NIST256p s dĺžkou kľúča 256 bitov, ktorá má porovnateľnú kryptografickú bezpečnosť s kľúčom generovaným cez algoritmus RSA s dĺžkou kľúča 3072 bitov.

Použitie kryptografie na báze ECC sa výrazne redukuje dĺžku kľúča a pamäťovú náročnosť na jeho uloženie. Pôvodná implementácia Native OTA po overení integrity firmvéru dokáže na základe overenia verzie vykonať zmenu príznaku v OTA\_DATA partícií a vykonať softvérový reštart mikrokontroléru pre bootovanie nového firmvéru.

Pre zabezpečenie samotného bootovacieho procesu som implementoval hardvérovú metódu MZBP prostredníctvom zápisu šifrovacieho kľúča do jednorazovo programovateľnej pamäte eFuse BLK2 a následne som funkcionalitu permanentne zapol zápisom bitu do 1-bitovej eFuse ABS\_DONE\_0. Táto implementácia zabezpečuje, že systém nabootuje, iba ak bol do mikrokontroléru nahratý dôveryhodný bootloader a správny odtlačok zo SBDA algoritmu.

Keďže eFuse nie je softvérovo druhý krát prepisovateľná, alebo čitateľná, nedokáže potenciálny útočník získať pôvodný šifrovací kľúč, respektíve ho nedokáže získať metódou totálnych skúšok (brute force) s aktuálnymi výpočtovými prostriedkami v prijateľnom čase (čas útoku prevyšuje plánované použitie aplikácie).

Podarilo sa mi demonštrovať funkciu senzorového uzla s bezpečnou aktualizáciou firmvéru cez internet s využitím bezpečného prenosového kanála a vývojárskych nástrojov v prostredí ESP-IDF, ktorými som implementoval vysokú mieru bezpečnosti pre bezpečnú prevádzku IoT platformy ESP32 v aplikácii s prenosom dát cez internet.

Platforma je zároveň chránená metódou digitálneho podpisu aj pred útokom podvrhnutia firmvéru cez fyzické USB-UART rozhranie útočníkom. Taktiež je chránený aj firmvér pred prečítaním a spustením útočníkom v prípade jeho prevzatia z flash pamäte mikrokontroléru ESP32, čomu sa pri aktuálnej revízii použitého čipu ESP32 nedá zabrániť systémovou funkcionalitou. Firmvér



je pri stiahnutí cez USB-UART rozhranie vo formáte šifrovaného textu, ktorý nie je spustiteľný bez dešifrovania s využitím symetrického kľúča bezpečne archivovaného v eFuse BLK1.

Pre mikrokontroler ESP32 je v prílohách dostupný zdrojový kód aplikácie sensorového uzla, README súbor s príkazmi pre generovanie kľúčov v prostredí ESP-IDF 4.2 a implementáciu metód pre zabezpečený bootovací proces a šifrovanie flash pamäti. Experimentálny návrh firmvéru pre prototypovacu KEMT ESP32 dosku s meraním napätia batérie a nízkonápríkonovým režimom je súčasťou prílohy A, vrátane zdrojových kódov pre prostredie ESP-IDF.

## Zoznam použitej literatúry

- [1]. ESP32 Technical Reference Manual [online]. Espressif Systems [cit. 2020-01-01]. Dostupné z:  
[https://www.espressif.com/sites/default/files/documentation/esp32\\_technical\\_reference\\_manual\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf)
- [2]. IoT Development Framework - Overview [online]. Espressif Systems [cit. 2021-01-20]. Dostupné z:  
<https://www.espressif.com/en/products/sdks/esp-idf>
- [3]. ESPTOOL (bundle) [online]. Github [cit. 2021-02-22]. Dostupné z:  
<https://github.com/espressif/esptool/>
- [4]. Arduino IDE – Documentation [online]. Arduino Foundation [cit. 2021-01-19]. Dostupné z:  
<https://www.arduino.cc/reference/en/>
- [5]. ESP-NOW [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z:  
[https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp\\_now.html](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html)
- [6]. USING THE BLE FUNCTIONALITY OF THE ESP32 [online]. Electronics-Lab.com [cit. 2020-03-14]. Dostupné z: <https://www.electronics-lab.com/project/using-the-ble-functionality-of-the-esp32/>
- [7]. Bluetooth Low Energy (BLE) Beacon Technology Made Simple: A Complete Guide to Bluetooth Beacons [online]. Beaconstac [cit. 2021-03-22]. Dostupné z:  
<https://www.electronics-lab.com/project/using-the-ble-functionality-of-the-esp32/>
- [8]. ESP32 Series Datasheet [online]. Espressif Systems [cit. 2021-02-27]. Dostupné z:  
[https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)
- [9]. ESP32 SPI Flash API [online]. ESP-IDF Programming Guide [cit. 2020-12-13]. Dostupné z:  
[https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spi\\_flash.html](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spi_flash.html)
- [10]. Tensilica Xtensa LX6 Datasheet [online]. Cadence [cit. 2016-01-01]. Dostupné z:  
[https://mirrobo.ru/wp-content/uploads/2016/11/Cadence\\_Tensilica\\_Xtensa\\_LX6\\_ds.pdf](https://mirrobo.ru/wp-content/uploads/2016/11/Cadence_Tensilica_Xtensa_LX6_ds.pdf)
- [11]. Getting started with MicroPython on the ESP32 [online]. MicroPython [cit. 2021-04-15]. Dostupné z: <https://docs.micropython.org/en/latest/esp32/tutorial/intro>
- [12]. Getting started with Lua Language on ESP32 [online]. IoT Design Pro [cit. 2019-01-23]. <https://iotdesignpro.com/projects/getting-started-with-lua-programming-on-esp32>
- [13]. low.js | Node.js for microcontrollers [online]. Neonious [cit. 2021-04-15]. <https://iotdesignpro.com/projects/getting-started-with-lua-programming-on-esp32>

- 
- [14]. ESP8266EX Datasheet [online]. Espressif Systems [cit. 2020-01-01]. Dostupné z: [https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf)
- [15]. Arduino Core – ESP8266 [online]. Github [cit. 2021-04-19]. Dostupné z: <https://github.com/esp8266/Arduino>
- [16]. Application startup flow [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/general-notes.html#application-startup-flow>
- [17]. ESP32 ROM Console [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/romconsole.html>
- [18]. The FreeRTOS™ Reference Manual [online]. Amazon.com, Inc. [cit. 2018-07-01]. Dostupné z: [https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS\\_Reference\\_Manual\\_V10.0.0.pdf](https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf)
- [19]. ESP32 [online]. Wikipedia [cit. 2021-03-31]. Dostupné z: <https://en.wikipedia.org/wiki/ESP32>
- [20]. Katalógový list CP2102 [online]. Silicon Labs [cit. 2017-01-17]. <https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf>
- [21]. ESP32-DevKitC V4 - schéma [online]. Espressif Systems [cit. 2017-12-07]. [https://dl.espressif.com/dl/schematics/esp32\\_devkitc\\_v4-sch.pdf](https://dl.espressif.com/dl/schematics/esp32_devkitc_v4-sch.pdf)
- [22]. ESP32 DevKitC V4 [obrázok]. [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/v3.3.4/get-started-cmake/get-started-devkitc.html>
- [23]. Esptool.py [online]. Github [cit. 2021-02-22]. Dostupné z: <https://github.com/espressif/esptool/blob/master/esptool.py>
- [24]. Katalógový list ESP32-WROOM-32 [online]. Espressif Systems [cit. 2021-01-01]. [https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf)
- [25]. Katalógový list ESP32-S [online]. Shenzhen Anxinke Technology [cit. 2016-10-03]. <https://www.es.co.th/Schemetic/PDF/ESP32.PDF>
- [26]. Product Specification – ESP-CAM [online]. AI-Thinker [cit. 2017-01-01]. <https://loboris.eu/ESP32/ESP32-CAM%20Product%20Specification.pdf>
-

- 
- [27]. Olimex – produkty ESP32 - IoT [online]. Olimex [cit. 2021-03-20].  
<https://www.olimex.com/Products/IoT/ESP32/>
- [28]. ESP32 Modules and Boards [online]. ESP-IDF Programming Guide [cit. 2021-02-27].  
Dostupné z:  
<https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/hw-reference/modules-and-boards.html>
- [29]. Katalógový list ESP32-PICO-D4 [online]. Espressif Systems [cit. 2021-01-01].  
[https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf)
- [30]. Deep Sleep Wake Stubs [online]. ESP-IDF Programming Guide [cit. 2021-04-19].  
Dostupné z:  
<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/deep-sleep-stub.html>
- [31]. Katalógový list ESP32-PSRAM64 [online]. Espressif Systems [cit. 2020-01-01].  
[https://www.espressif.com/sites/default/files/documentation/esp-psram64\\_esp-psram64h\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp-psram64_esp-psram64h_datasheet_en.pdf)
- [32]. Partition Tables [online]. ESP-IDF Programming Guide [cit. 2021-02-27].  
Dostupné z:  
<https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-guides/partition-tables.html>
- [33]. eFuse Manager [online]. ESP-IDF Programming Guide [cit. 2021-01-28]. Dostupné z:  
<https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/system/efuse.html>
- [34]. Sleep Modes [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z:  
[https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/system/sleep\\_modes.html](https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/system/sleep_modes.html)
- [35]. ESP32 ULP coprocessor instruction set [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: [https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-guides/ulp\\_instruction\\_set.html](https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-guides/ulp_instruction_set.html)
- [36]. Assembler - Dokumentácia [online]. Oracle [cit. 2010-03-01].  
<https://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf>
- [37]. Idf.py [online]. Github [cit. 2021-03-09]. Dostupné z:  
<https://github.com/espressif/esp-idf/blob/master/tools/idf.py>
- [38]. ESP-IDF Release v4.2 [online]. Github [cit. 2020-12-07]. Dostupné z:  
<https://github.com/espressif/esp-idf/releases/tag/v4.2>
-

- 
- [39]. ESP32: Inter task communication using FreeRTOS Queues [online]. iCircuit [cit 2017-08-26]. Dostupné z: <https://icircuit.net/esp32-inter-task-communication-using-freertos-queues/1946>
- [40]. Wiring (development platform) [online]. Wikipedia [cit 2021-03-29]. Dostupné z: [https://en.wikipedia.org/wiki/Wiring\\_\(development\\_platform\)](https://en.wikipedia.org/wiki/Wiring_(development_platform))
- [41]. Arduino Core for the ESP32[online]. Github [cit. 2021-02-24]. Dostupné z: <https://github.com/espressif/arduino-esp32>
- [42]. ULPTOOL [online]. Github [cit. 2021-02-08]. Dostupné z: <https://github.com/duff2013/ulptool>
- [43]. Over-the-air programming [online]. Wikipedia [cit 2021-02-04]. Dostupné z: [https://en.wikipedia.org/wiki/Over-the-air\\_programming](https://en.wikipedia.org/wiki/Over-the-air_programming)
- [44]. How to Approach OTA Updates for IoT [online]. DZone IoT [cit 2018-01-19]. Dostupné z: <https://dzone.com/articles/how-to-approach-ota-updates-for-iot>
- [45]. Basic OTA [online]. Github [cit. 2018-03-04]. Dostupné z: <https://github.com/espressif/arduino-esp32/blob/master/libraries/ArduinoOTA/examples/BasicOTA/BasicOTA.ino>
- [46]. Bonjour (software) [online]. Wikipedia [cit. 2021-04-18]. Dostupné z: [https://en.wikipedia.org/wiki/Bonjour\\_\(software\)](https://en.wikipedia.org/wiki/Bonjour_(software))
- [47]. Message-Digest algorithm [online]. Wikipedia [cit 2013-03-07]. Dostupné z: [https://sk.wikipedia.org/wiki/Message-Digest\\_algorithm](https://sk.wikipedia.org/wiki/Message-Digest_algorithm)
- [48]. OTA Web Updater [online]. Github [cit. 2020-11-02]. Dostupné z: <https://github.com/espressif/arduino-esp32/blob/master/libraries/ArduinoOTA/examples/OTAWebUpdater/OTAWebUpdater.ino>
- [49]. WireShark - Dokumentácia [online]. Wireshark Foundation [cit. 2021-04-08]. Dostupné z: <https://www.wireshark.org/docs/>
- [50]. Katalógový list LAN8720 [online]. MicroChip [cit. 2016-01-01]. <https://ww1.microchip.com/downloads/en/DeviceDoc/00002165B.pdf>
- [51]. Katalógový list IP101GRI [online]. IC PLUS Corporation Manufacturer [cit. 2011-01-01]. <https://datasheetspdf.com/pdf-file/1469951/ICPlus/IP101GRI/1>
- [52]. Reduced media-independent interface [online]. Wikipedia [cit 2021-03-14]. Dostupné z: [https://en.wikipedia.org/wiki/Media-independent\\_interface#Reduced\\_media-independent\\_interface](https://en.wikipedia.org/wiki/Media-independent_interface#Reduced_media-independent_interface)

- [53]. PHY [online]. Wikipedia [cit. 2021-02-25]. Dostupné z: <https://en.wikipedia.org/wiki/PHY>
- [54]. RSA (cryptosystem) [online]. Wikipedia [cit. 2021-04-05]. Dostupné z: [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- [55]. OpenSSL - Dokumentácia [online]. Cryptography toolkit [cit. 2021-04-20]. Dostupné z: <https://www.openssl.org/docs/>
- [56]. CentOS - Dokumentácia [online]. Cent OS Docs Site [cit. 2021-04-21]. Dostupné z: <https://docs.centos.org/en-US/docs/>
- [57]. HTTPD – Dokumentácia v2.4 [online]. Apache server HTTP project [cit. 2021-03-13]. Dostupné z: <http://httpd.apache.org/docs/2.4/>
- [58]. Project Configuration [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html>
- [59]. Simple OTA [online]. Github [cit. 2021-01-15]. Dostupné z: [https://github.com/espressif/esp-idf/blob/master/examples/system/ota/simple\\_ota\\_example/main/simple\\_ota\\_example.c](https://github.com/espressif/esp-idf/blob/master/examples/system/ota/simple_ota_example/main/simple_ota_example.c)
- [60]. Native OTA [online]. Github [cit. 2021-01-15]. Dostupné z: [https://github.com/espressif/esp-idf/blob/master/examples/system/ota/native\\_ota\\_example/main/native\\_ota\\_example.c](https://github.com/espressif/esp-idf/blob/master/examples/system/ota/native_ota_example/main/native_ota_example.c)
- [61]. Signed App Verification Without Hardware Secure Boot [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/security/secure-boot-v1.html#signed-app-verification-without-hardware-secure-boot>
- [62]. Kryptografia na báze eliptických kriviek [online]. Wikipédia [cit. 2015-10-16]. Dostupné z: [https://sk.wikipedia.org/wiki/Kryptografia\\_na\\_b%C3%A1ze\\_eliptick%C3%BDch\\_kriviek](https://sk.wikipedia.org/wiki/Kryptografia_na_b%C3%A1ze_eliptick%C3%BDch_kriviek)
- [63]. Elliptic Curve Digital Signature Algorithm [online]. Wikipédia [cit. 2015-10-19]. Dostupné z: [https://sk.wikipedia.org/wiki/Elliptic\\_Curve\\_Digital\\_Signature\\_Algorithm](https://sk.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm)
- [64]. espsecure.py [online]. Github [cit. 2021-01-19]. Dostupné z: <https://github.com/espressif/esptool/blob/master/espsecure.py>
- [65]. ADALIER, M. a iní, "Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256", s. 7-8
- [66]. Python ECDSA [online]. Github [cit. 2021-01-25]. Dostupné z: <https://github.com/tlsfuzzer/python-ecdsa>
- [67]. HANKERSON, D. a iní, "Guide to Elliptic Curve Cryptography", Springer 2003, s. 13

- 
- [68]. Secure Boot [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/security/secure-boot-v1.html>
- [69]. Secure Boot V2 [online]. ESP-IDF Programming Guide [cit. 2021-03-10]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v2.html>
- [70]. Katalógový list ESP32-S2 Family [online]. Espressif Systems [cit. 2020-01-01]. [https://www.espressif.com/sites/default/files/documentation/esp32-s2\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf)
- [71]. Predstavenie ESP32-C6 [online]. Espressif Systems [cit. 2021-04-09]. [https://www.espressif.com/en/news/ESP32\\_C6](https://www.espressif.com/en/news/ESP32_C6)
- [72]. espefuse.py [online]. Github [cit. 2020-12-15]. Dostupné z: <https://github.com/espressif/esptool/blob/master/espefuse.py>
- [73]. Flash Encryption [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html>
- [74]. Project Configuration Menu [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html#project-configuration-menu>
- [75]. CHLEBOVEC, M., " Inteligentné relé s WiFi konektivitou do siete eduroam", Košice 2019
- [76]. Katalógový list Sensirion SHT21 [online]. Farnell [cit. 2011-12-01]. <http://www.farnell.com/datasheets/1780639.pdf>
- [77]. Katalógový list Bosch BME 280 [online]. Bosch Sensortec [cit. 2018-11-04]. <https://readthedocs.org/projects/bme280/downloads/pdf/latest/>
- [78]. BME280 Driver - Bosch Sensortec [online]. Github [cit. 2020-07-06]. Dostupné z: [https://github.com/BoschSensortec/BME280\\_driver](https://github.com/BoschSensortec/BME280_driver)
- [79]. I2C [online]. Wikipedia [cit. 2021-02-07]. Dostupné z: <https://en.wikipedia.org/wiki/I%C2%B2C>
- [80]. Dokumentácia BME280 Bosch Sensortec Driver [online]. Texas Instruments [cit. 2018-01-01]. [http://software-dl.ti.com/simplelink/esd/simplelink\\_cc13x2\\_sdk/2.20.00.71/exports/docs/thread/doxygen/thread/html/bme280\\_8h.html](http://software-dl.ti.com/simplelink/esd/simplelink_cc13x2_sdk/2.20.00.71/exports/docs/thread/doxygen/thread/html/bme280_8h.html)
- [81]. Solid-state relay [online]. Wikipedia [cit. 2021-03-02]. Dostupné z: [https://en.wikipedia.org/wiki/Solid-state\\_relay](https://en.wikipedia.org/wiki/Solid-state_relay)
-

- 
- [82]. Katalógový list OMRON G3MB-202P [online]. Omron Electronics [cit. 2021-02-27].  
<http://www.omron-russia.com/doc/relay/ssr/g3mb.pdf>
- [83]. Termostat [online]. Wikipédia [cit. 2019-10-06]. Dostupné z:  
<https://sk.wikipedia.org/wiki/Termostat>
- [84]. RFC 1867 [online]. Internet Engineering Task Force [cit. 1995-11-01]. Dostupné z:  
<https://tools.ietf.org/html/rfc1867>
- [85]. Logging library [online]. ESP-IDF Programming Guide [cit. 2021-02-27]. Dostupné z:  
<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/log.html>
- [86]. PHP – Dokumentácia [online]. PHP Group [cit. 2021-03-01]. Dostupné z:  
<https://www.php.net/docs.php>
- [87]. HTTP Basic authentication [online]. Axway Documentation Portal [cit. 2020-08-21].  
[https://docs.axway.com/bundle/APIGateway\\_762\\_PolicyDevFilterReference\\_allOS\\_en\\_HTML5/page/Content/PolicyDevTopics/authn\\_http\\_basic.htm](https://docs.axway.com/bundle/APIGateway_762_PolicyDevFilterReference_allOS_en_HTML5/page/Content/PolicyDevTopics/authn_http_basic.htm)
- [88]. Base64 [online]. Wikipédia [cit. 2016-11-27]. Dostupné z:  
<https://sk.wikipedia.org/wiki/Base64>
- [89]. Autodesk Eagle – Overview [online]. Autodesk [cit. 2021-04-10]. Dostupné z:  
<https://www.autodesk.com/products/eagle/overview?plc=F360&term=1-YEAR&support=ADVANCED&quantity=1>



## Prílohy

Príloha A: CD médium – obsahuje:

- diplomová práca v elektronickej podobe,
- opis použitého SSR relé pre senzorový uzol v elektronickej podobe,
- zdrojové kódy testovaných OTA metód v prostredí Arduino IDE v elektronickej podobe,
- hlavný projekt senzorového uzla, zdrojový kód hlavnej aplikácie senzorového uzla v jazyku C (ESP-IDF), použité symetrické kľúče, README v elektronickej podobe,
- zdrojové kódy PHP webaplikácie v elektronickej podobe,
- spustiteľné .bat súbory pre generovanie certifikátov metódou ECC, RSA pre certifikačnú autoritu, webserver kryptografickým nástrojom OpenSSL s konfiguračným súborom webservera ssl.conf v elektronickej podobe,
- opis experimentálnej implementácie ULP – nízkopríkonového režimu na platforme ESP32 pre KEMT prototypovaciu dosku s využitím senzora Bosch BME680, zdrojové kódy pre ESP-IDF v elektronickej podobe,
- opis čerpania informácií z literárnych zdrojov v elektronickej podobe.