**Szegedi Tudományegyetem Informatikai Intézet**

Beltéri hangvezérelt világítás megvalósítása ESP32 mikrovezérlővel

Indoor voice-controlled lighting

using ESP32 microncontroller

Szakdolgozat

|  |  |
| --- | --- |
| Készítette: | Témavezető: |
| **Fodor Teodor** | **Dr. Mingesz Róbert** |
| mérnökinformatikus szakos hallgató | egyetemi adjunktus |

Szeged 2024

Feladatkiírás:

ESP32 mikrovezérlővel számos IOT-projektet elkészíthetünk. A mikrovezérlő sokoldalúságának és könnyen proramozhatóságának köszönhetően a lehetőségek tárháza végtelen. Gyakori jelenség, hogy az épületek beltéri megvilágítását különböző elven működő érzékelővel vezérlik. A diplomamunkám célja, hogy a megszokott vezérléseket figyelmen kívül hagyva, egy sokkal kényelmesebb és képzeletet felülmúló, hangvezérelt megoldással álljak elő, amely lehetőleg gyors, könnyen felhasználható és alacsony hibaárányú. A dolgozat írása során természetesen törekszem megtalálni az arany középutat a felhasznált komponensek árában és minőségében, hiszen a legjobb ötlet sem valósítható meg a megfelelő céleszközök nélkül.

A feladat megvalósításának lépései:

1. Pontos specifikáció meghatározása
2. ESP32 fejlesztőkit és mikrofonok kiválasztása
3. Hardverek tesztelése (oszcilloszkóp, egyéb mérések)
4. Program implementálása
5. Program tesztelése
6. Áramkör elkészítése próbapanelen
7. Program javítása és bővítése

Tartalmi összefoglaló:

A téma megnevezése:

Beltéri hangvezérelt megvilágítás megvalósítása ESP32 mikrovezérlővel

A megadott feladat megfogalmazása:

A feladat követelménye, hogy egy két zónás(két különálló LED-szalagot) világítást tudjunk hanggal vezérelni.

A megoldási mód:

A kiválasztott ESP32 fejlesztőkit és mikrofonok közötti kommunikáció szoftveres megvalósítása, majd a az mesterséges intelligencia modellek beiktatása a hang felismeréséhez.

Alkalmazott eszközök, módszerek:

A programot a hivatalos, Espressif cég által biztosított ESP-IDF keretrendszert felhasználva készítem el, FreeRTOS-s környezetben. A kapcsolást először breadboard-on rakom össze, majd egy próbapanelen összeforrasztom (a két műveletet megelőzi egy áramkörtervezés).

Elért eredmények:

Mind a program, mind az áramkör is elkészült, így a szobámban már megtalálható a működő prototípus.

Kulcsszavak:

ESP-IDF, ESP32, I2S, INMP441, FreeRTOS

Tartalomjegyzék:

[1. Bevezetés: 3](#_Toc180593597)

[2. Hardverkörnyezet bemutatása: 5](#_Toc180593598)

[2.1. A felhasznált mikrovezérlő és fejlesztőkit: 5](#_Toc180593599)

[2.2. A felhasznált mikrofonok: 6](#_Toc180593600)

[2.2.1. A MEMS-technológia: 6](#_Toc180593601)

[2.2.2. INMP441: 8](#_Toc180593602)

[2.2.3. I²S szabvány: 9](#_Toc180593603)

[2.3. A felhasznált LED szalag: 10](#_Toc180593604)

[2.4. Egyéb felhasznált eszközök: 12](#_Toc180593605)

[2.5. A felépített áramkör: 13](#_Toc180593606)

[3. Szoftverkörnyezet bemutatása: 15](#_Toc180593607)

[3.1. Espressif IoT Development Framework: 15](#_Toc180593608)

[3.2. FreeRTOS: 16](#_Toc180593609)

[3.2.1. Feladatok: 16](#_Toc180593610)

[3.2.2. Rendszerórajel: 17](#_Toc180593611)

[3.2.3. Ütemező: 17](#_Toc180593612)

[3.2.4. Feladatok közötti kommunikáció (ITC): 17](#_Toc180593613)

[4. A program bemutatása: 18](#_Toc180593614)

[4.1. A hardware\_driver komponens: 19](#_Toc180593615)

[4.2. A led\_indicator komponens: 23](#_Toc180593616)

[4.3. A pwm\_output komponens: 28](#_Toc180593617)

[4.4. A voice\_assistant komponens: 31](#_Toc180593618)

[4.4.1. ESP-Skainet és Speech Recognition Framework: 32](#_Toc180593619)

[4.4.2. Audio Front End: 32](#_Toc180593620)

[4.4.3. WakeNet: 33](#_Toc180593621)

[4.4.4. MultiNet: 34](#_Toc180593622)

[4.4.5. A feed taszk működése: 40](#_Toc180593623)

[4.4.6. A fetch taszk működése: 41](#_Toc180593624)

[4.5. A program belépési pontja: 43](#_Toc180593625)

[5. A diplomamunka eredménye: 44](#_Toc180593626)

[6. Irodalomjegyzék: 45](#_Toc180593627)

# Bevezetés:

A hangvezérlés egy olyan technológia, amely lehetővé teszi a különböző eszközök és alkalmazások beszédhanggal való vezérlését. A hangvezérlést biztosító rendszerek használata egyre elterjedtebb a mindennapi életben, például okostelefonok, autók és egyéb elektronikus készülékek esetében. Napjainkban egy korszerű okosotthon szinte teljesen elképzelhetetlen hangvezérlés nélkül. A hangvezérlés egy olyan alternatíva, amely egy érintésmentes és kényelmes szolgáltatást biztosít a hagyományos vezérlési módszerekhez képest.

Felmérések alapján a felhasználók nagy része többnyire a következő 4 platform mellett teszi le a voksát: Apple Siri, Samsung Bixby, Amazon Alexa vagy Google asszisztens. A felsorolt szolgáltatások mindegyikére jellemző, hogy online, azaz felhő-alapú hangfelismerő algoritmusokat futtatnak nagy teljesítményű szervereken. A platformokat támogató termékek (Amazon echo dot, Amazon echo studio, Apple Homepod, Samsung televízió, stb.) valójában egy mikrofonként szolgálnak, amelyek továbbítják a beszédhangot az előbb említett távoli számítógépeknek. Ezen rendszerek általában drágábbak illetve a használatukhoz egyszerre több terméket is meg kell vásárolnunk. A diplomamunkámmal azt próbálom bemutatni, hogy van lehetőség jutányos áron beszerezni, kiépíteni egy olyan rendszert, amellyel egyszerű eszközök is hangvezéreltté tehetők. Például egy Amazon Alexát támogató villanykörte körülbelül 30 euróba, a hozzá tartozó legegyszerűbb Amazon Echo Dot 50 euróba kerül - az ellátott feladatot természetesen teljesítik az eszközok, azonban az ár egy kissé magas (az Alexás termékek árai messze elmaradnak az Apple termékeihez képest) egyetlen funkcionalitás megvalósításához. Az elkészített projekttel be szeretném mutatni, hogy sokkal kevesebb anyagi ráfordítással is megteremthető a fenti kényelmi funkció.

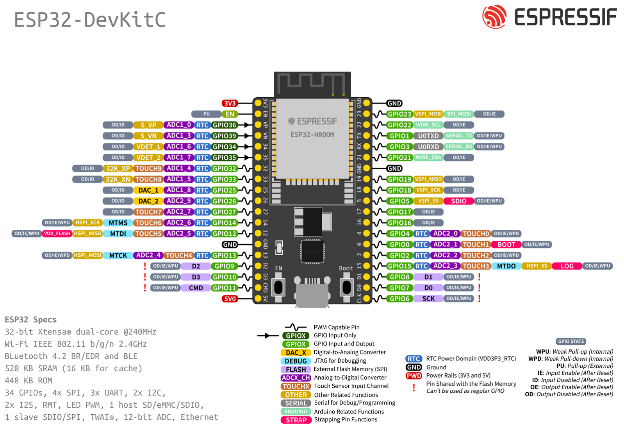
Fontos megjegyezni, hogy ha az ember ESP32 [1] mikrovezérlővel megvalósított hangfelismerés offline hangfelismerés [2], azaz az algoritmus magán az eszközön fut, ezért egy erősebb chip megvásárlása szükséges, amelyben több memória található, illetve az órajele is sokkal magasabb. Az ilyen chipek ára még mindig messzemenőleg elmarad egy felhő-alapú rendszerhez képest. A legegyszerűbb megoldás az, ha az ember vásárol egy olyan fejlesztőkitet, amiben már megtalálható a mikrofon is, így nem kell foglalkozni a hang eljuttatásával a mikrovezérlőbe. Találtam több kitet is, amelybe már integrálva voltak a mikrofonok, azonban a beszerzésük hosszadalmas és drágább folyamat lett volna. Ezért úgy döntöttem, hogy külön-külön beszerzem a fejlesztőkitet és a két mikrofont(lehetőleg olyan típusút, amelyek a hivatalos kitekbe is integrálva vannak) és elkészítem a közöttük lévő kommunikációhoz szükséges illesztőprogramot. A hangfelismerő algoritmust nem én készítetettem, már része az ESP-IDF [3] keretrendszernek, és sajnos nem nyílt forráskodú a framework ezen könyvtára.

Nem titkolt célja a projektnek, hogy egy pénztárcabarát és könnyen használható, illetve konfigurálható eszközt tervezzek, amely egyszerűbb vezérlési feladatok elvégzésére képes.

# Hardverkörnyezet bemutatása:

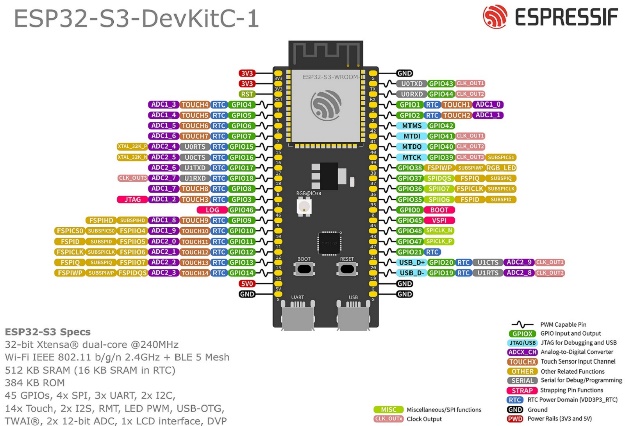
A felhasznált hardverek kiválasztásakor alábbi négy szempont szerint jártam el: legyen költséghatékony, könnyen elérhető, lehetőleg gyors legyen a kiszállítás és ne legyen gyárilag hibás.

## A felhasznált mikrovezérlő és fejlesztőkit:

Az Espressif [4] cég ESP termékcsaládja eddig is népszerű volt a hobbi építők és az olcsó IoT eszköz gyártók körében, révén hogy nagy tudása mellett, olcsó és könnyen programozható. Az elterjedt ESP8266 (illetve ESP8265 és változatai) leváltására érkezett az erősebb ESP32. Az ESP32 az elődjéhez képest sokkal több perifériával, nagyobb memóriával és magasabb órajellel rendelkezik. A programozható pinek száma megnőtt, illetve egyre több hálózati kommunikációs protokollt (Thread, Zigbee, Matter) is képesek támogatni a megszokott WiFi-n és Bluetooth-on kívül az újszériás ESP32 chipek. Az elmúlt évek során rengeteg változat jelent meg, mindegyik széria rendelkezik valamilyen specifikus képességgel, így az adott feladath követelményeinek megfelelően tudjuk kiválasztani a számunkra megfelelő chipet.

Ábra 1 – ESP32 csippel szerelt fejlesztőkit.   
Forrás: [5]

Ábra 2 – ESP32-S3 csippel szerelt fejlesztőkit.  
Forrás: [6]



A diplomamunkámhoz az ESP32-S3[6] AI SoC-t választottam, amely a már jól bevált két processzormagos, akár 240 MHz-es órajel elérésére is képes Xtensa LX7 vezérlőegységet tartalmazza. A mikrovezérlőt kifejezetten AIoT alkalmazásokhoz gyártotta az Espessif Systems cég 2020-ban. Az ilyen chipet tartalmazó modulok különlegessége, hogy tartalmaznak egy külső PSRAM (Pseudostatic RAM) memóriaegységet, 2,4,8 és 16 MB-os kapacitással, verziótól függően. A PSRAM [7] nagy méretének köszönhetően, lehetővé teszi a magas számításigényű mesterséges intelligencia algoritmusok (esetünkben hangfelismerő algoritmusok) futását, a nagy mennyiségű digitalizált hangadat pufferelését – kizárólag ezért használható csak az S3-as szériához a keretrendszer hangfelismerő könyvtára. A jelenlegi projekthez én ESP32-S3-DevKitC-1U-N8R8 chip-el ellátott ESP32-S3-DevKitC-1 v1.1 fejlesztőkitet választottam 8MB flash-memóriával és 8 MB PSRAM-al. [6]

Ábra 3 – Espressif cég által gyártott PSRAM csip. Forrás: [7]

## A felhasznált mikrofonok:

Annak érdekében, hogy a beszédhang digitalizásálával ne kelljen foglalkoznom, mindenképpen egy MEMS-technológiát használó mikrofont választottam, illetve egy olyan kommunikációs protokollt, amellyel könnyen le tudom implementálni a mikrofon és a fejlesztőkit közötti adatátvitelt. Ha körül nézünk a boltok polcain, akkor legtöbb ilyen típusú mikrofon az I²S kommunikációs szabványt támogatja. Az ESP-IDF pedig rendelkezik olyan könyvtárral, amely a fenti kommunikáció lebonyolításáért felelős.

### A MEMS-technológia:

A MEMS mikrofonokat [8] vagy szilíciummikrofonokat már régóta előszeretettel használják okostelefonokban, okostévékben, távirányítókban. Ez nagy részben olyan hangvezérelt személyititkár-alkalmazásoknak köszönhető, amilyen a korábban is más sokszor emlegetett Amazon Alexa, a Google Assistant és az Apple termékek Siri-je. Ezek az alkalmazások olyan hangparancsokra várnak, amelyek ébresztik magát a rendszert, majd a parancs alapján hajtanak végre különböző utasításokat. A tervezők szűrőfunckiókat is integrálnak a mikrofonokba, hogy azok pontosan, jelentős környezeti zajban is messziről érzékeljék a beszédhangot.

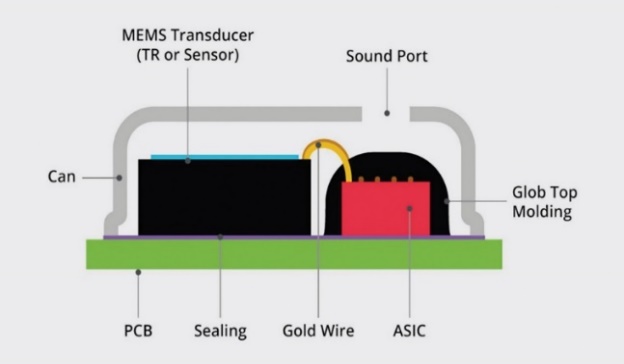
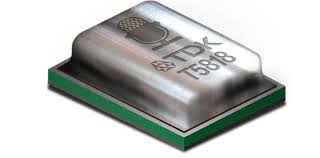
A MEMS mikrofonok jellemzően két, egyazon tokban elhelyezett alkatrészből állnak: egy MEMS membránból, amely átalakítja a hanghullámokat elektromos jellé, valamint egy erősítőből, amely impedanciaátalakítóként működve használható analóg kimenőjelet ad a hangfrekvenciás jelfeldolgozó láncra. A harmadik alkotóelem – ha digitális kimenőjelre van szükség – egy analóg-digitális átalakító, amely szintén elhelyezhető ugyanabban a tokban, számunkra ez a fontos a projekt esetében. [8]

A MEMS-mikrofonok fontosabb paraméterei a következők:

Jel-zaj viszony(signal-noise ratio, azaz SNR): a hasznos és a zavaró jel aránya dB-ben kifejezve. Minnél alacsonyabb, annál több zaj jelenik meg a digitalizált adatokban. Törekedni kell a nagyobb jel-zaj viszonyú mikrofonok kiválasztására, így az eszköz kevesbé „zajosabb” és kevésbé érzékenyebb a zajra.

Frekvenciaátvitel: megadja azt a hangfrekvencia tartományt, ahol a mikrofon képes érzékelni a beszédhangot.

Dinamikatartomány: azon leghangosabb és leghalkabb hangnak megfelelő hangnyomásszint közti különbség, amelyek között a mikrofon jelleggörbéje lineáris.

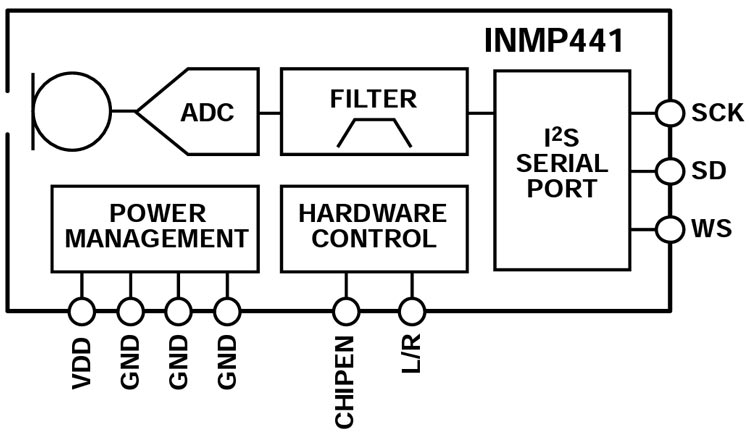
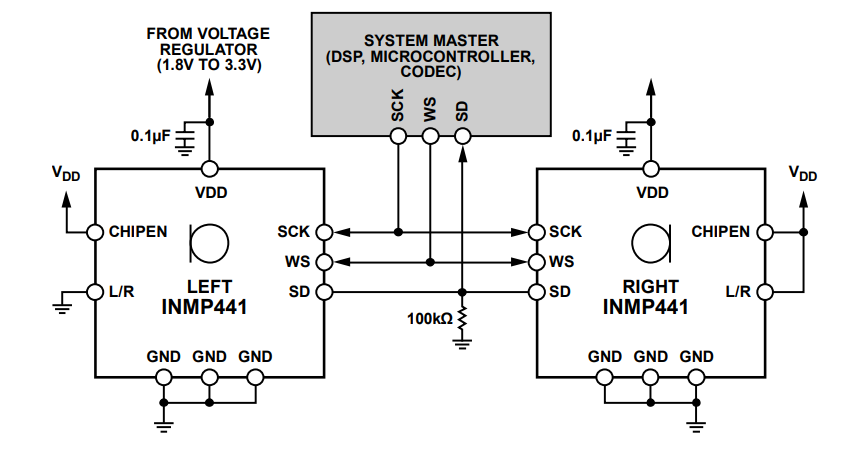
Érzékenység: megadja, hogy a mikrofon egy 1 kHz-es 94 dB hangnyomású hanghullámra adott analóg/digitális kimeneti értékének nagyságát

Ábra 4 – MEMS mikrofon belső felépítése. Forrás: [8].

Ábra 5 – MEMS mikrofon csip. Forrás: [9].

Digitális MEMS mikrofon használata esetén a mikrofon kimenőjele közvetlenül rávezethető digitális áramkörökre, általában egy mikrovezérlőre. Az erősen zajos környezetekre tervezett hangvezérléses felhasználói felületek jellemzően inkább a digitális mikrofonokat részesítik előnyben, mert a digitális kimenőjeleknek nagyobb a zajtűrésük, illetve a kezelésük is könnyebb, mint az analóg társaiknak.

### INMP441:

Az INMP441 [10] egy olcsó és könnyen beszerezhető, alacsony fogyasztású MEMS mikrofon modul. Fontos tulajdonsága, hogy I²S kimeneti interfésszel rendelkezik és omnidirekcionális, ami azt jelenti, hogy minden irányból egyformán érzékeli a hangot. A kimenete 24-bites és könnyen alkalmazható sztereó módban, azaz egyszerre 2 azonos típusú mikrofont is könnyen használhatunk - az architektúrájából kifolyólag szinkronizálva van a mikrofonok közötti adatátvitel.

Ábra 6 – az INMP441 mikrofoncsip belső felépítése. Forrás: [10].

Ábra 8 – INMP441 mikrofonnal szerelt mikrofon. Forrás: [11].

Ábra 7 – 2 INMP411 mikrofon sztereó módban használva.  
Forrás: [10].

A fenti ábrán jól látható a „L/R” pin a mikrofon blokkdiagramján – ennek segítségével konfigurálható, hogy az eszköz a „jobb” vagy a „bal” oldali csatornán küldje a

kimeneti adatot. Az 7-es ábra szemlélteti a sztereó-kimenet kapcsolását.

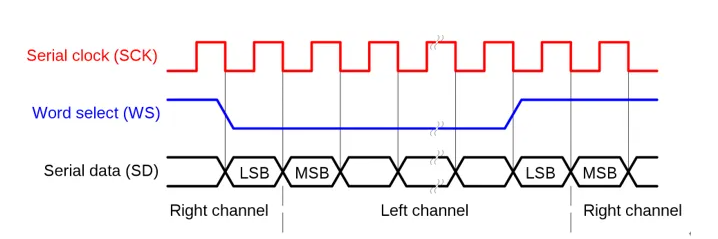
Jól látható, hogy az egyik mikrofon csip L/R pinje tápfeszültségre, míg a másik eszköz L/R pinje GND-re van csatlakoztatva. Az ábrán feltüntetett szűrőkondenzátorok és ellenállások már alapból megtalálhatóak a mikrofonos lapocskán, ezért használatuk elhanyagolható. Az ábra utolsó fontos eleme, a „SYSTEM MASTER” blokk esetünkben nem más, mint maga az ESP32 S3-as fejlesztőkit.

### I²S szabvány:

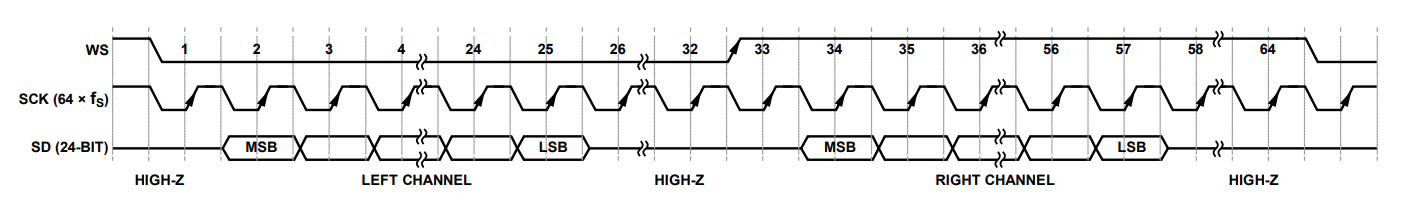
Az I²S [12] (Inter-IC Sound) egy elektronikus soros busz interfész szabvány, amelyet digitális audióeszközök összekapcsolására használnak. Ezt a szabványt először a Philips Semiconductor vezette be 1986-ban. Digitális audió adatok átvitelére használják az elektronikus eszközök integrált áramkörei között. A kimenet PCM-formátumú.

Összesen 3 jelalak jellemzi:

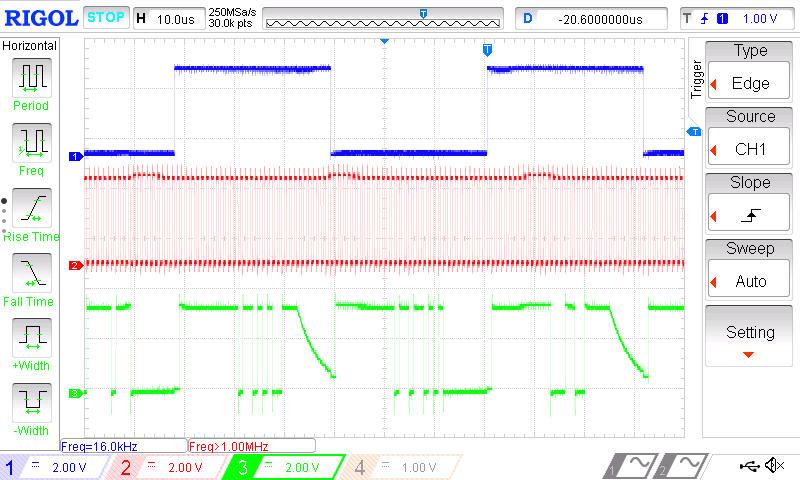
* Bit órajel – az általunk használt mikrofon esetében ez az SCK-pinen (serial clock) jelenik meg. Feladata a bitek átvitelének az ütemezése
* Szó óravonal – formálisan ez a mikrofon WS-pinjének (word select) a jele. Feladata, hogy sztereó kimenet esetén kiválasztja, hogy a jobb vagy bal oldali csatornáról szeretnénk fogadni az adatot. Logikai alacsony szint esetén a bal-, míg logikai magas jelszint esetén a jobb csatornáról történik az adat olvasása. Leegyszerűsítve ez a mintavételi frekvencia egy szóra vonatkozóan, ami esetünkben 24-bit (32 bit).
* Soros adatvonal – ezen az adatvonalon jelennek meg a mikrofon által digitalizált adatok bájtjai. Az INMP441 mikrofon esetén ez az SD-pin(serial data).



Ábra 9 – Az I2S idődiagramja. Forrás: [12].



Ábra 10 – Az INMP441 mikrofon I2S idődiagramja sztereó-kimenet esetén.  
Forrás: [10].

Adatlap alapján a mikrofon kimenetén 24-bites adatok jelennek meg, amelyek a legnagyobb helyiértékű bittel kezdődnek. A dolog érdekessége, hogy mégis 32 jelenik meg egy fél-WS-ciklus (1 darab csatorna olvasása ) alatt. Ez azzal magyarázható, hogy a mikrofon ilyenkor egy magas impedanciás állapotba kerül – ezek a bitek számunkra közömbösek és a szoftveres lekezelésénél is figyelmen kívül hagyjuk őket. Kicsit nehezebben észrevehető, de még mielőtt kiolvasnánk a digitalizált adat legnagyobb helyiértékű bitjét a WS-ciklus elején, szükségünk van egy bit-órajel impulzusra (eltolás), ez a formátum a „Philips formátum”. Az ESP-IDF I2S könyvtárában [13] ez szerencsére gond nélkül konfigurálható. Fontos még kiemelni, hogy egy teljes WS-ciklus alatt 64 SCK impulzusnak kell megjelennie annak ellenére, hogy csak 24 bit a számunkra értékes információ.[8]

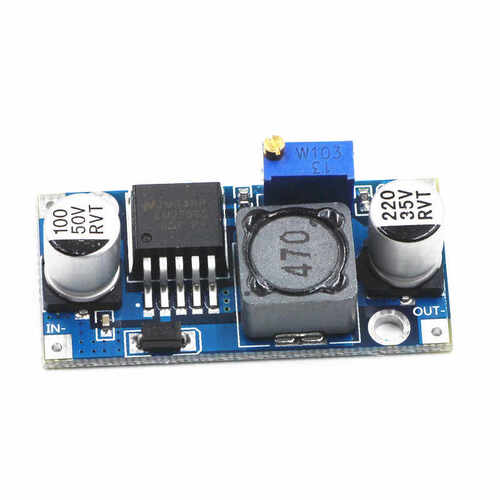
Ábra 11 – az INMP441 vizsgálata oszcilloszkóppal. Kékkel a WS-, pirossal az SCK- és zölddel az SD- jel látható.

## A felhasznált LED szalag:

Végül, de nem utolsó sorban a megvilágításért felelős LED-szalagról teszek említést. A két szalagot a szobámban található két szekrény hátoldalán helyeztem. Miután lemértem a szekrények hosszait, méretre vágtam a szalagokat, végül felragasztottam őket. Ehhez a szalagon található ragasztón kívül még univerzális ragasztót is felhasználtam a biztos tapadás érdekében. Összesen 3 méter szalagra volt szükségem, amelyekről a gyári csatlakozót le kellett cserélnem csupasz vezetékekre.

A műszaki paramétereket áttekintve egy 12 V-on működő, méterenként 120 LED diódával szerelt szalagról van szó, 3000K színmélységgel (ún. “meleg fehér” a szalag által emittált fénysugár). Számomra fontos paraméter volt a dimmelhetőség, mivel egyszere 5-6 különböző fényerősséget szerettem volna felkínálni, amelyeket a hangparancsokkal lehet változtatni. A különböző fényerősségek eléréséhez természetesen PWM-jelet alkalmaztam.

Ábra 12 – COB LED-szalag. Forrás: [14]

Kihagyhatatlan volt a felhasznált teljesítmény és áramerősség meghatározása is, hiszen a rendszerhez egy megfelelő méretű tápegységet is be kellett szereznem. . Mérések és kísérletezés alapján a két LED szalag egyenként 750 mA-t fogyasztott (maximális fényerősség esetén), illetve a mikrovezérlő elenyésző fogyasztását is belaszámoltam, aminek eredménye egy 3A áramerősségű 12 voltos tápegység [15] lett. (230V hálózati feszültségről alakít). A fejlesztőkit tápfeszültséggel való ellátásáért egy LM2596 [16] DC-DC kapcsolüzemű konvertert használtam, melynek kimeneti feszültségét 3.3V-ra állítottam.

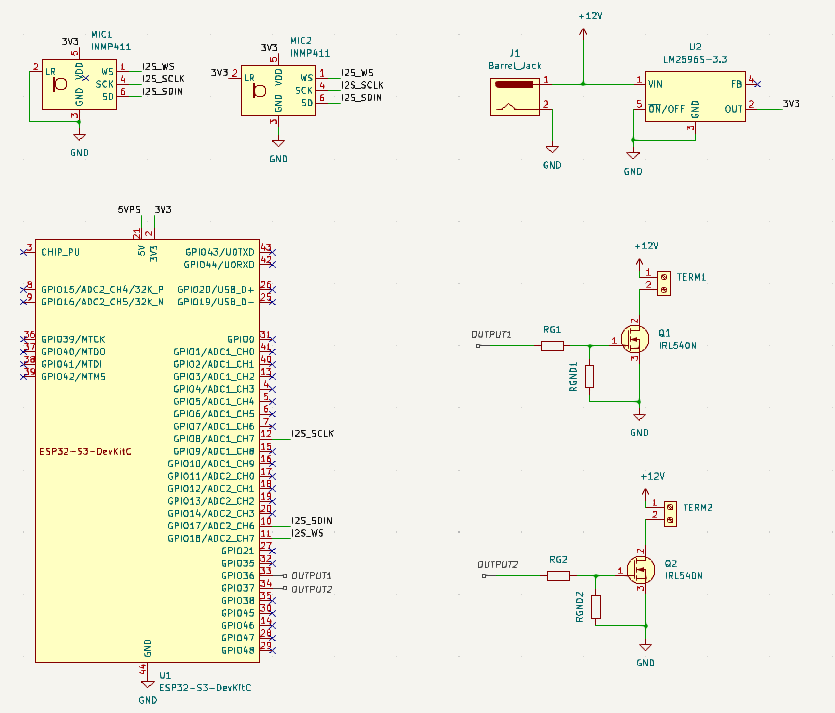
Ábra 13 – a projekt megvalósításához felhasznált tápegységek. Bal oldalon a 12V-os AC/DC átalakító, jobb oldalon pedig az LM2595-os DC/DC átalakító. Forrás: [15, 16].

## Egyéb felhasznált eszközök:

* Műanyag doboz – 95 x 60 mm méretű, az áramkör tárolására
* Próbapanel – a dobozhoz mérten méretre vágtam, erre forrasztottam az áramkört
* IRL540N [17] logikai MOSFET – a két LED szalag meghajtására használtam. Önmagában az ESP32 fejlesztőkit GPIO lábai nem képesek 12 voltos eszközöket meghajtani. A tranzisztor threshhold feszültszége 1.8-1.9 V, ami könnyen kapcsolható a kit 3.3 V-os pinjeivel. [14]
* Ellenállások – 100 Ω és 10 kΩ értékkel
* 2.54mm lábtávolságú aljzat: a fejlesztőkit és a mikrofonok könnyebb rögzítéséhez és esetleges cseréjéhez
* DC-csatlakozó – egy tápegység, egy csatlakozó. Az egyszerűség kedveért barrel-jack-t használtam
* Kétutas csavaros sorkapocs – ezzel kötöttem a LED szalagokat az áramkör kimeneteire

## A felépített áramkör:

Ábra 14 – a projekthez használt áramkör kapcsolási rajza

A fenti ábra alaposan bemutatja az általam tervezett áramkört. A program, amelyet használtam a kapcsolási rajz elkészítéséhez a nyílt-forráskódú és ingyenesen letölthető KiCad. A könnyebb átláthatóság érdekében a vezetékek helyett címkéket használtam (pl. OUTPUT1, I2S\_SDIN, stb.). Szeretném kiemelni, hogy az áramkör 3 dimenziós modelljét, amely a nyomtatott áramköri lapot tartalmazza, nem készítettem el, ezért is érdemesebb a rajzra egy funkcionális diagramra tekinteni, amely a forrasztás során segített létrehozni a megfelelő kapcsolásokat(erre jó példa a LM2596 DC-DC átalakítónak csak a integrált áramköri komponense van feltüntetve az ábrán). A sokszor felhasznált részeket (+12v, GND) egységesítettem.

Jobb felső sarokban található az áramkör tápellátásért felelős részegység. A J1-es jelzésű BarrelJack-en keresztül tápláljuk meg 12 volttal a rendszert a 13-as ábrán található AC-DC tápegységgel. Ezt a 12 volt feszültséget használjuk fel a LED szalagok meghajtására illetve közvetett módon a fejlesztőkit áramellátására is: az LM2596 DC-DC átalakító a 12 voltot 3.3 voltra alakítja át, amely a fejlesztőkit 3V3 kivezetésére csatlakozik.

Az ábra bal alsó sarkában látható a ESP32 S3 fejlesztőkit. Az „I2S” előtaggal rendelkező kivezetéseket szabadon választottam, ügyelve arra, hogy az eszköz működéséhez szükséges pineket (BOOT, GND, RST, stb.) ne használjam. Hasonló módon jártam el az OUTPUT1 és OUTPUT2 pinek esetén is, általános célú lábakat választottam, amelyek a későbbiekben a váltakozó kitöltési tényezőjű PWM-jeleket juttatják el a logikai MOSFET tranzisztorokra.

Az ábra bal felső sarkában található a két darab INMP441-es omnidirekcionális mikrofonmodul. Ezeket is LM2596-os feszültségkonverter látja el tápfeszültséggel, adatlapi értékek alapján maximálisan 3.6V-al táplálhatóak. A korábban említett sztereó konfiguráció érdekében az egyik L/R kivezetését a rendszer földpontjára, míg a másikat 3.3 V feszültségre kötöttem. A további három vezeték az I2S által szüséges jelek továbbítására szolgál, amelyek az I2S\_WS (word select), I2S\_SD(serial data) és I2S\_SCK (serial clock).

Az áramkör utolsó részegysége a LED szalagok meghajtásáért felelős aláramkör a kapcsolási rajz jobb alsó sarkában. Az RG1 és RG2 ellenállások 100 Ω-sak, míg az RGND1 és RGND2 10kΩ-sak. Az RGND1 és RGND2 ellenállások lehúzó ellenállásként funkcionálnak, így biztosítva azt, hogy a mikrovezérlő bootolásakor a tranzisztorok kimenetei nem magas impedanciás-, hanem biztosan kikapcsolt állapotban vannak. Az RG1 és RG2 ellenállás megléte nem kötelező, hiszen a logikai térvezérlésű tranzisztorokat elméleti szinten kizárólag feszültséggel vezéreljük, azonban tudjuk, hogy a gyakorlatban némi áram is megjelenik a mosfet gatején – a két ellenállás a túláramok ellen biztosít védelmet. [18] Noha a egy feszültségosztó áramkört kapunk, az RG1 és RG2 ellenállásértékei elenyészőek a 10kΩ-hoz képest, ezért a jelenlétük nem fogja zavarni a működést:

A kapott „V” feszültség érték messze meghaladja az IRL540 fet threshold feszültéségét(~ 1.8-1.9 V). A tranzisztor drain kivezetése a csavaros sorkapocs egyik lábára csatlakozik, természetesen arra, amelyre a LED szalag negatív polarítású vezetékét kötöttem, míg másikra, amelyen a 12V-os tápfeszültség van, a szalag pozitív vezetéke került bekötésre. A tranzisztorokat meghajtó pwm-jel biztosítja azt, hogy azokat kapcsolóüzemben használjuk így tövábbi munkapont beállításra nincs szükség. A tranzisztor N-csatornás, így a bemeneten megjelenő négyszögjel a szalag földpontját nagyfrekvencián szaggatja” (dimmeli), a váltakozó kitöltési tényező pedig a jel effektív értékét, azaz a fényerőt szabályozza.

# Szoftverkörnyezet bemutatása:

## Espressif IoT Development Framework:

Az ESP-IDF a kínai Espressif cég által fejlesztett keretrendszer, amelyet az ESP8266- és ESP32 sorozatú mikrovezérlők programozására használunk. Az szoftvercsomag rengeteg API-t kínál, amellyel kényelmesen programozható a mikrovezérlő minden egyes perifériája és felkínált szolgáltatása, C és C++ programozási nyelveken egyaránt. Fontos kiemelni, hogy maga a keretrendszer és annak szoftverkomponensei a FreeRTOS [19] valós-idejű operációs rendszer alapjaira vannak lefektetve, amely kivétel nélkül mindegy egyes Espressif-es csipet támogat-

A keretrendszer nagy előnye, hogy teljesen nyílt-forráskodú, szabadon megtekinthetőek a forrásfájlok a hivatalos GitHub repository-ban, emellett jól dokumentált, minden apró részletre kitér és, ami a legfontosabb, hogy az egyes könyvtárakhoz példaprogramokat is bemutat, amelyek a dokumentáció olvasásával kombinálva megkönnyítik és felgyorsítják a fejlesztést. [3]

## FreeRTOS:

Ábra 15 – az idf.py menuconfig parancs futtatása után megjelenik a konfigurátor interfész, jelen esetben a FreeRTOS-ra vonatkozó opciókat láthatjuk

A valós idejű operációs rendszer, mint olyan, egy erőforráskímélő operációs rendszer, amely lehetővé teszi a többfeladatos programvégrehajtást. Maga a „valós idő” az adott folyamat elvégzéséhez szükséges idő determinisztikusságára és (vagy) mennyiségének megjósolhatóságára utal. A beágyazott rendszerek többnyire valós idejű (Real-Time) követelményeket támasztanak, ami azt jelenti, hogy egy-egy esemény bekövetkezésére adott időkorláton belül reagálni kell. [19]

Ábra 16 – FreeRTOS logó. Forrás: [19].

### Feladatok:

A FreeRTOS az adott feladatok (taszk) között osztja meg az hardver erőforrásait. A taszkok egymástól független végtelen ciklusban futó függvények, amelyek az adott mikrovezérlő heap-memóriájából foglalnak le a stack-ük nagyságával megegyező memóriát. Bár a taszkok logikailag el vannak szigetelve egymástól, az MPU (memória védelmi egység) a különböző memóriaterületek védelmét szolgálja, megakadályozva, hogy egy taszk egy másik taszk memóriaterületére írjon vagy olvasson. [19]

A feladatok különböző állapotúak lehetnek:

* Blokkolt (blocked/suspended) – a feladat eseményre vár (pl. késleltetés leteltére, adatok/erőforrások elérhetővé válására)
* Felfüggesztett (suspended) – a feladat nem kap erőforrást és nem kerül ütemezésre
* Készenléti (ready) – a feladat a CPU-n történő futásra kész, de nem fut, mert a CPU erőforrásait egy másik feladat használja
* Futó (running) – a CPU erőforrást biztosít a feladat futásához

### Rendszerórajel:

A FreeRTOS-nak szüksége van egy időalapra az idő méréséhez, illetve az ütemező működéséhez. Ez általában egy számláló változó segítségével valósul meg, amelyet a hardver egy periódikusan generált megszakításkezelő függvényben növel. A rendszerórajelnek köszönhetően egy alkalmazás több időalapú szolgáltatást (feladat-végrehajtási intervallumot, várakozási időtúllépést, időszeletelést) képes fenntartani egyetlen hardveres időzítő használatával. A gyorsabb rendszerórajel azonban csak az RTOS időalap felbontását növeli, a szoftver futási ideje nem javul tőle. [19]

### Ütemező:

A FreeRTOS-ban futó feladatok ütemezését az ütemező végzi. Típusa szerint ez egy preemptív ütemező, ami azt jelenti, hogy a taszkokhoz a létrehozásukor egy prioritás értéket rendelünk, így az ütemező ezen érték alapján osztja meg az erőforrásokat és a processzoridő a taszkok között. A különböző értékek megléte miatt a magasabb prioritással rendelkező feladatok megszakíthatják az alacsonyabb prioritással rendelkező társaikat.[19]

### Feladatok közötti kommunikáció (ITC):

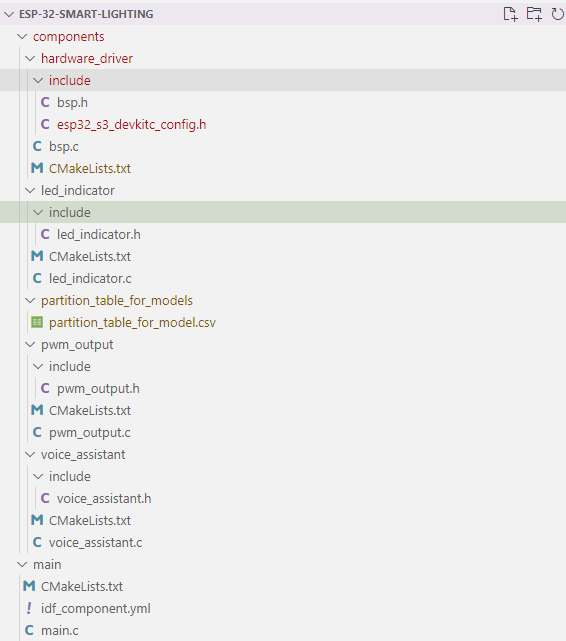
Ha több feladatot futtatunk párhuzamosan, gyakori jelenség a köztük lévő információcsere, illetve egy globális változó írása/olvasása. Az ilyen „közös” entitások megléte gyakran idézhet elő versenyhelyzetet. Ekkor kettő (vagy több taszk) egyszerre próbálja meg elérni a globálist változót, ami a benne lévő adat sérülését okozhatja. Az információk megosztása mellett a FreeRTOS-objektumok képesek szinkronizálni a feladatok végrehajtását is. A feladatok blokkolhatóak, várhatnak egymásra – ez esetben a jelenhet a meg holtpont (dead-lock), amikor két vagy több taszk egymást blokkolja, mivel mindegyik vár valamilyen erőforrásra, amit egy másik birtokol. Ennek eredményeként egyik folyamat sem tud előrehaladni, és a rendszer megreked. [19]

A FreeRTOS a következő objektumokkal rendelkezik:

* Üzenetsor (queue vagy message queue) - Adatok küldésére szolgál a feladatok vagy a megszakítás és a feladat között, FIFO technikát alkalmaz
* Mutex –egyetlen erőforrás használatának védelmére használatos
* Szemafor - Az erőforrások használatának védelmére vagy a feladat-végrehajtás szinkronizálására szolgál. Lehet bináris- és számláló szemafor.
* Eseménycsoport (Event group): valójában egy 8- vagy 16 bites változó, melynek a meghatározott bit-kombinációi egy adott álapotot vagy esemény bekövetkezését jelzik.

# A program bemutatása:

A projekt megvalósításához összesen 4 komponenst készítettem el. Ezek a *hardware\_driver, led\_indicator, pwm\_output és voice\_assistant*. A projekt architektúráját tekintve a voice\_assistant a program fő része, hiszen megteremti a kapcsolatot a fennmaradó 3 komponens között, amelyek egymástól teljesen függetlenül is működőképesek, azonban az elvárt követelmény a képességeik kombinálásával teljesíthető. A program belépési pontja a szokásos *app\_main* függvényben található, amely elindítja a *voice\_assistant* komponensben található taszkokat. Minden inicializációs- és konfigurációs lépés a taszkok indulása előtt fut le, hiba esetén jelez a program. A következő fejezetben az elkészített komponenseket mutatom be lépésről-lépésre haladva – minden egyes részegységnél ügyeltem arra, hogy a „publikus header” fájlokban csak az olyan függvényeket helyezzem el, amelyeket később a főprogramban hívok meg és próbáltam a lehető legjobban elrejteni a valós implementációt (fekete doboz). Az alábbi ábrával szemléltetem a teljes projekt logika felépítését, amely útmutatóként szolgál.



Ábra 17 – az idf-projekt strultúrája

## A hardware\_driver komponens:

A *hardware\_driver* komponens felelős a fejlesztőkit és a két INMP441 típusú mikrofon közötti I2S kommunikáció megvalósításáért. A program talán második legfontosabb része, hiszen egy alapból nem beépített funkcionalitással/képességgel látja el a fejlesztőkitet – jelen esetben a beszédhang digitalizásról, majd a digitalizált értékek mikrovzérlőbe való továbbításáról beszélünk.

Header fájl(ok):

esp\_err\_t **bsp\_board\_init**(uint32\_t sample\_rate, int

channel\_format, int bits\_per\_chan);

esp\_err\_t **bsp\_read\_i2s\_data**(int16\_t \*buffer, int buffer\_size);

esp\_err\_t **bsp\_board\_deinit**(void);  
  
int **bsp\_board\_channel\_fmt**(void);

#define **GPIO\_I2S\_WS** (GPIO\_NUM\_11)

#define **GPIO\_I2S\_MCLK** (GPIO\_NUM\_NC)

#define **GPIO\_I2S\_SCLK** (GPIO\_NUM\_12)

#define **GPIO\_I2S\_DIN** (GPIO\_NUM\_10)

#define **GPIO\_I2S\_DOUT** (GPIO\_NUM\_NC)

#define **I2S\_CONFIG\_DEFAULT**(sample\_rate, channel\_fmt, bits\_per\_chan)

{

.clk\_cfg = **I2S\_STD\_CLK\_DEFAULT\_CONFIG**(sample\_rate),

.slot\_cfg = **I2S\_STD\_PHILIPS\_SLOT\_DEFAULT\_CONFIG**(bits\_per\_chan, channel\_fmt),

.gpio\_cfg = {

.mclk = GPIO\_I2S\_MCLK,

.bclk = GPIO\_I2S\_SCLK,

.ws = GPIO\_I2S\_WS,

.dout = GPIO\_I2S\_DOUT,

.din = GPIO\_I2S\_DIN,

.invert\_flags = {

.mclk\_inv = false,

.bclk\_inv = false }}

};

A *bsp.h* (felül) tartalmazza a komponens által publikusan felkínált függvények deklarációját, míg az *esp32\_s3\_devkitc\_config.h* (alul) a közvetlen módon definiált pineket, amelyeket később az I2S a komponens implementációjánál használtam fel. A *GPIO\_NUM\_NC* jelzi, hogy az adott pineket nem használjuk majd – az *DOUT* pinre azért nincs szükségünk, mert nem küldünk adatot a fejlesztőkitből, illetve felesleges az *MCLK* is, mivel nem biztosítok referencia órajelet a mikrofonoknak.

Source fájl(ok):

#include "freertos/FreeRTOS.h"

#include "freertos/semphr.h"

#include "esp\_log.h"

#include "esp32\_s3\_devkitc\_config.h"

#include "driver/i2s\_std.h"

#include "driver/i2s\_common.h"

#include "bsp.h"

#define **TAG** "bsp"

static SemaphoreHandle\_t i2s\_mutex = NULL;

static i2s\_chan\_handle\_t rx\_handle = NULL;

static esp\_err\_t **init\_i2s\_mutex**(void)

{

    i2s\_mutex = **xSemaphoreCreateMutex**();

    return (i2s\_mutex != NULL) ? ESP\_OK : ESP\_FAIL;

}

A mutex-k használatára azért volt szükség, mert a program egyszerre több taszkot fog párhuzamosan futtatni: könnyen kialakulhatnak olyan helyzetek, amikor egy adott entitást már az előtt szeretnénk használni, hogy befejeződött volna az alap állapotba való helyezése. Mutex használatával könnyen és elegánsan tudjuk elérni, hogy egyszerre több taszk ne tudja elérni az adott memóriaterületet, így elkerülve az esetleges felülírásokat és befejezetlen inicializációs lépéseket.

static esp\_err\_t **i2s\_init**(i2s\_port\_t i2s\_port\_num, uint32\_t sample\_rate, int channel\_format, int bits\_per\_chan)

{

    esp\_err\_t ret = ESP\_OK;

**ESP\_ERROR\_CHECK**(**init\_i2s\_mutex**());

**xSemaphoreTake**(i2s\_mutex, **pdMS\_TO\_TICKS**(0));

    i2s\_chan\_config\_t channel\_cfg = **I2S\_CHANNEL\_DEFAULT\_CONFIG**(i2s\_port\_num,

I2S\_ROLE\_MASTER);

    ret |= **i2s\_new\_channel**(&channel\_cfg, NULL, &rx\_handle);

    i2s\_std\_config\_t std\_cfg = **I2S\_CONFIG\_DEFAULT**(sample\_rate, channel\_format,

bits\_per\_chan);

    ret |= **i2s\_channel\_init\_std\_mode**(rx\_handle, &std\_cfg);

    ret |= **i2s\_channel\_enable**(rx\_handle);

**xSemaphoreGive**(i2s\_mutex);

    return ret;

}

Az implementáció elrejtéséhez a C-programozási nyelven static függvényeket használunk, ez a kulcsszó jelzi a fordítóprogramnak, hogy az adott függvényt csak az őt tartalmazó fordítási egységből tudjuk elérni és hívni (ebben az esetben az őt tartalmazó fájlból). A szálbiztos inicializáció abban merül ki, hogy a mutex-t zároljuk, majd az inicialiációt követően. „visszaadjuk”.

esp\_err\_t **bsp\_board\_init**(uint32\_t sample\_rate, int channel\_format, int bits\_per\_chan)

{

esp\_err\_t ret = ESP\_OK;

    ret |= **i2s\_init**(I2S\_NUM\_1, sample\_rate, channel\_format, bits\_per\_chan);

    if(ret == ESP\_OK){

**ESP\_LOGI**(**TAG**, "I2S peripherial intialized successfully.");

    }

    else {

**ESP\_LOGI**(**TAG**, "Error while initializing I2S peripherial.);

}

    return ret;

}

A *bsp\_board\_init* publikus függvény valójában csak meghívja a felhasználó elől elrejtett, statikus *i2s\_init függvényt* – callback pattern. Amennyiben i2s periféria inicializálása sikertelen, hibaüzenetet ír ki és visszatér valamelyik esp-idf hibát jelző értékkel, például ESP\_FAIL.

static esp\_err\_t **i2s\_read\_data**(int16\_t \*buffer, int buffer\_size)

{

    esp\_err\_t ret = ESP\_OK;

    size\_t bytes\_read;

    int sample\_num = (buffer\_size / sizeof(int16\_t));

    int32\_t \*temp\_buffer = (int32\_t \*)**calloc**(sample\_num, sizeof(int32\_t));

    if (temp\_buffer == NULL) {

        ret = ESP\_ERR\_NO\_MEM;

        return ret;

    }

**i2s\_channel\_read**(rx\_handle, temp\_buffer, sample\_num \* sizeof(int32\_t), &bytes\_read, portMAX\_DELAY);

    for (int i = 0; i < sample\_num; i++) {

        buffer[i] = (temp\_buffer[i] >> 14);

    }

**free**(temp\_buffer);

    return ret;}

esp\_err\_t **bsp\_read\_i2s\_data**(int16\_t \*buffer, int buffer\_size)

{

    esp\_err\_t ret = ESP\_OK;

    ret |= **i2s\_read\_data**(buffer, buffer\_size);

    return ret;

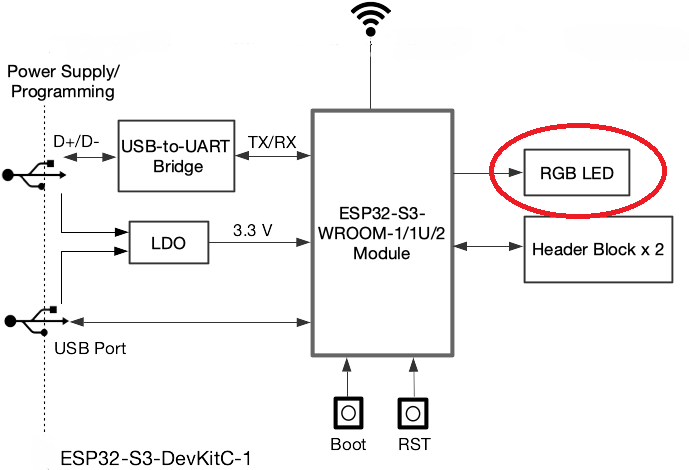
}

Az *i2s\_read\_data* függvény under-the-hood módon végzi el a konfigurált I2S-csatornán az adatok beolvasását a paraméterként kapott buffer-be, melynek a mérete *buffer\_size* (bájtokban mérve). A függvény törzsében dinamikusan allokáltam egy 32-bites egészeket (tudjuk, hogy a mikrofonok kimenete 24-bites) tároló tömböt *sample\_num* méretben. A *sample\_num* valójában a később bemutatott hangfelismerő algoritmus által meghatározott hangminta (16-bites érték) mennyisége. Ahhoz, hogy a 32-biten tárolt 24-bites értékek a megfelelő módon kerüljenek a paraméterként megadott pufferbe, el kell shiftelnünk őket a megfelelő mértékben – ügyelve arra is, hogy a felső 8-bit a 32-ből magas impedanciás (jelen esetben határozatlan vagy nulla). Utolsó lépésként szigorúan felszabadítjuk az allokált memóriát.

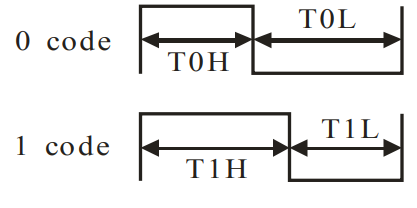
## A led\_indicator komponens:

Az alábbi részegység a felhasználóval való interakciót valósítja meg: lényege, hogy a konfigurált ébresztő hangparancsokra (wakeword) bekapcsolja a fejlesztőkiten található

RGB-LED izzót. A fehér fényű dióda mindaddig világít, amíg a hangfelismerő algoritmus aktív, ez konfigurációtól függően 5-7 másodperc.

A beépített RGB LED-dióda egy WS2812 [20] modellszámú intelligens, és címezhető dióda. Gyakran alkalmazzák különböző fényeffektusok és egyedi világítást megvalósító projektekhez. Ezekben a LEDek-ben egyetlen integrált áramkör található, amely tartalmazza a piros, zöld és kék diódákat, valamint egy csipet, amely az általuk kibocsátott fény erősségéért, színéért és vezérléséért felelős.

Ábra 18 – az ESP32-S3 fejlesztőkit blokkdiagrammja, pirossal megjelölve a beépített RGB LED-dióda. Forrás: [6].

A diódát még az teszi vonzóvá, hogy egyetlen vezeték elegendő a kommunikáció lebonyolításához: szinte bármilyen mikrovezérlő GPIO-pinjét használva megvalósítható a kommunikáció. A kommunikáció lényege, hogy a mikrovezérlő 24 bites adatcsomagokat küld: az adatcsomagban három 3 8-bites (0-255 skálán terjedő) érték található, amelyek rendre a piros, zöld és kék „fénykomponens” intenzitását reprezentálják. Fontos megjegyezni, hogy a logikai 0 és 1-es érték is kódolva van az ábrán látható módon:

Ábra 19 – a WS2812-es típusú LED-dióda kommunikációs protokolljában felhasznált kódolás. Forrás: [20].

Ahhoz, hogy a diódában levő vezérlő csip a kívánt színkombinációkat tudja előállítani, a logikai nullák és egyesek kódolásnál szigorúan be kell tartani az időzítéseket: a protokoll esetében mind a 4 időtartam (T0H, T0L, T1H és T1l) mikroszekundumos skálán mozog. Külön-külön meghatározott ideig kell, hogy tartson – egy illesztőprogram megírásánál ez nehézkes és meglehetősen időigényes lehet.

Szerencsére az ESP-IDF nem hivatalos komponensei (ESP Component Registry weboldal [21]) között található a WS2812 (és a hozzá hasonló) diódatípus programozásához szükséges könyvtár – az espressif\_ledstrip. Az ilyen és ehhez hasonló könyvtárak az alábbi paranccsal adhatók hozzá a projekthez:

**idf.py add-dependency "espressif/led\_strip^2.5.4"**

A parancs lényege, hogy létrehozza a projekt main mappájában az *idf\_component.yaml* fájlt az alábbi módon:

dependencies:

  espressif/led\_strip: "^2.5.3"

  idf:

    version: ">=4.1.0"

A fájl tartalmának köszönhetően mostmár komponensként hozzáadhatjuk az általunk definiálthoz (jelen esetben *led\_strip*) és az #include paranccsal tudjuk beilleszteni az adott forrásfájlba.

Header fájl:

typedef struct **led\_indicator** \***led\_indicator\_t**;

**led\_indicator\_t** **led\_indicator\_init**();

esp\_err\_t **led\_indicator\_start**(**led\_indicator\_t** led\_indicator);

esp\_err\_t **led\_indicator\_turn\_on**(**led\_indicator\_t** led\_indicator);

esp\_err\_t **led\_indicator\_turn\_off**(**led\_indicator\_t** led\_indicator);

A header fájlban definiáltam egy pointert a kód írásának megkönnyítéséhez, amely az általam létrehozott struktúrára mutat(*led\_indicator\_t*) – ez a struktúra szintén el van rejtve a header fájlt használó komponensek elől, csak a forrásfájlban található meg.

Forrásfájl

#include "freertos/FreeRTOS.h"

#include "freertos/semphr.h"

#include "led\_indicator.h"

#include "led\_strip.h"

#define **LED\_INDICATOR\_PIN** 38

static SemaphoreHandle\_t led\_indicator\_mutex = NULL;

struct **led\_indicator** {

    led\_strip\_handle\_t led\_strip;

    bool run;

    bool led\_state;

};

static led\_strip\_handle\_t **configure\_led**(void)

{

    led\_strip\_config\_t strip\_config = {

        .strip\_gpio\_num = **LED\_INDICATOR\_PIN**,

        .max\_leds = **NUM\_OF\_LEDS**,

        .led\_pixel\_format = LED\_PIXEL\_FORMAT\_GRB,

        .led\_model = LED\_MODEL\_WS2812,

        .flags.invert\_out = false,

    };

    led\_strip\_spi\_config\_t spi\_config = {

        .clk\_src = SPI\_CLK\_SRC\_DEFAULT,

        .flags.with\_dma = true,

        .spi\_bus = SPI2\_HOST,

    };

    led\_strip\_handle\_t led\_strip;

**ESP\_ERROR\_CHECK**(**led\_strip\_new\_spi\_device**(&strip\_config, &spi\_config, &led\_strip));

**ESP\_LOGI**(TAG, "Created LED strip object with SPI backend");

    return led\_strip;

}

**led\_indicator\_t** **led\_indicator\_init**()

{

**led\_indicator\_t** led\_indicator = (**led\_indicator\_t**)**calloc**(1, sizeof(struct **led\_indicator**));

    if(led\_indicator == NULL){

        return NULL;

    }

    led\_indicator->led\_strip = **configure\_led**();

**ESP\_ERROR\_CHECK**(**init\_led\_indicator\_mutex**());

**ESP\_ERROR\_CHECK**(**turn\_off**(led\_indicator));

    led\_indicator->run = false;

    return led\_indicator;

}

A LED\_INDICATOR\_PIN konstans a fejlesztőkit 38-as GPIO pinjét jelzi, amely össze van kötve a beépített LED diódával. A *led\_strip\_handle\_t*, *led\_strip\_config\_t* és a *led\_strip\_spi\_config\_t* a *led\_stip.h* header-ben definiált struktúrák, amelyet a konfiguráláshoz használunk. A *led\_indicator\_init* függvényben előszőr létrehozok egy led\_indicator entitást, melynek ezután az adattagjait is inicializálom a statikus *configure\_led* függvény visszatérési értékével. A *configure\_led* függvény egyik érdekéssége még, hogy a könyvtár lehetőséget ad arra, hogy a mikrovezérlőbe épített SPI periféria segítségével végezzük el a korábban említett időzítéseket a logikai nullák és egyesek kódolásához.

static void **led\_indicator\_task**(void \*arg)

{

**led\_indicator\_t** led\_indicator = (**led\_indicator\_t**) arg;

    led\_strip\_handle\_t led\_strip = led\_indicator->led\_strip;

    led\_indicator->run = true;

    while (led\_indicator->run) {

        if(led\_indicator->led\_state){

*/\*Setting the LED's color to white\*/*

**ESP\_ERROR\_CHECK**(**led\_strip\_set\_pixel**(led\_strip, 0, 32, 32, 32));

**ESP\_ERROR\_CHECK**(**led\_strip\_refresh**(led\_strip));

        }

        else{

**ESP\_ERROR\_CHECK**(**led\_strip\_clear**(led\_strip));

        }

**vTaskDelay**(**pdMS\_TO\_TICKS**(100));

    }

**vTaskDelete**(NULL);

}

esp\_err\_t **led\_indicator\_start**(**led\_indicator\_t** led\_indicator)

{

    if(led\_indicator == NULL) {

**ESP\_LOGE**(TAG, "led\_indicator instance is NULL");

        return ESP\_FAIL;

    }

if(**xTaskCreate**(&**led\_indicator\_task**, "led\_indicator\_task", 2\*1024, led\_indicator,5,NULL) != pdTRUE) {

**ESP\_LOGE**(TAG, "error starting led\_indicator task");

        return ESP\_FAIL;

    }

    return ESP\_OK;

}

A statikus *led\_indicator\_task* egy FreeRTOS feleadatot valósít meg, amely a paraméterként átadott *led\_indicator\_t* típusú entitás adattagjainak állapotával fogja elvégezni a LED dióda ki- és bekapcsolását. A taszk egy végtelen ciklusban fut, amely ellenőrzi, hogy az indicator objektum boolean típusú *led\_state* adattagjának értéke igaz vagy hamis. Ezen logikai érték alapján fogja fehér színűre kapcsolni a diódát. A *led\_indicator\_start* függvény pedig a taszk létrehozásáért felelős, amely még azt is ellenőrzi, hogy megtörtént-e minden szükséges beállítás.

esp\_err\_t **led\_indicator\_turn\_on**(**led\_indicator\_t** led\_indicator){

    return **turn\_on**(led\_indicator);}

esp\_err\_t **led\_indicator\_turn\_off**(**led\_indicator\_t** led\_indicator){

    return **turn\_off**(led\_indicator);}

Az alábbi két függvényt pedig az indicator objektum *led\_state* változójának billentésére használom, amely szálbiztos módon történik a *turn\_on* és *turn\_off* függvények segítségével:

static esp\_err\_t **turn\_on**(struct **led\_indicator** \*led\_indicator)

{

    if(**xSemaphoreTake**(led\_indicator\_mutex, **pdMS\_TO\_TICKS**(5)) != pdTRUE) {

        return ESP\_FAIL;

    };

    led\_indicator->led\_state = true;

**xSemaphoreGive**(led\_indicator\_mutex);

    return ESP\_OK;

}

static esp\_err\_t **turn\_off**(struct **led\_indicator** \*led\_indicator)

{

    if (**xSemaphoreTake**(led\_indicator\_mutex, **pdMS\_TO\_TICKS**(5)) != pdTRUE) {

        return ESP\_FAIL;

    };

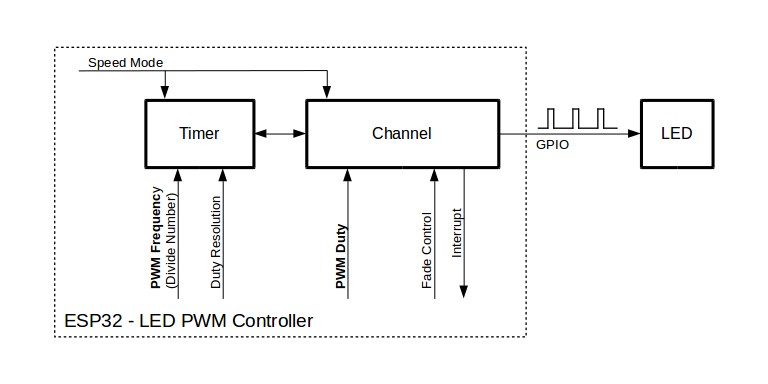
    led\_indicator->led\_state = false;

**xSemaphoreGive**(led\_indicator\_mutex);

    return ESP\_OK;

}

## A pwm\_output komponens:

A *pwm\_output* komponens a különböző fényintenzitások előállításáért felelős: a működéséhez felhasznált periféria és az ehhez készített könyvtár a LEDC [22], azaz LED Controller. A periféria összesen 8 csatornán képes különböző frekvenciájú és változtatható kitöltési tényezőjű PWM-jelet generálni, amellyel a projekthez megépített kimeneti áramkörben található logikai tranzisztorokat vezérlem.

Ábra 20 – a LEDC periféria belső felépítésének blokkdiagramja, egy csatornára vonatkozóan. Esetünkben a LED blokk helyett egy IRL540N tranzisztorra van kapcsolva a kimenete. Forrás: [22].

Header fájl:

esp\_err\_t **init\_pwm\_outputs**(void);

esp\_err\_t **update\_pwm\_outputs**(int command\_id);

esp\_err\_t **stop\_pwm\_outputs**(void);

A header fájl összesen 3 függvényt tartalmaz, amelyek közül az első kettő a számunkra igazán fontos. Az *init\_pwm\_outputs* csak egy wrapper-függvény a belső inicializációs függvény köré, az *update\_pwm\_outputs* függvény pedig a betanított hangparancs paraméterként kapott azonosítója alapján frissíti az általunk használt 2 kimenetet.

Forrásfájl:

#include "esp\_log.h"

#include "freertos/FreeRTOS.h"

#include "freertos/semphr.h"

#include "driver/ledc.h"

#include "pwm\_output.h"

#define **DUTY\_CYCLE\_100** (uint32\_t) 256U

#define **DUTY\_CYCLE\_75**   (uint32\_t) 192U

#define **DUTY\_CYCLE\_50**   (uint32\_t) 128U

#define **DUTY\_CYCLE\_25**   (uint32\_t) 64U

#define **DUTY\_CYCLE\_MOOD** (uint32\_t) 25U

#define **DUTY\_CYCLE\_0**    (uint32\_t)   0U

#define **DUTY\_CYCLE\_NUM**               6U

#define **OUTPUT\_CHANNEL\_1** (int)       4

#define **OUTPUT\_CHANNEL\_2** (int)       5

static SemaphoreHandle\_t pwm\_output\_mutex = NULL;

typedef enum {

    DUTYC\_100\_IDX = 0,

    DUTYC\_0\_IDX,

    DUTYC\_25\_IDX,

    DUTYC\_50\_IDX,

    DUTYC\_75\_IDX,

    DUTYC\_MOOD\_IDX,

} **duty\_cycle\_idx\_t**;

static const uint32\_t duty\_cycle\_values[**DUTY\_CYCLE\_NUM**] =

{

**DUTY\_CYCLE\_100**,

**DUTY\_CYCLE\_0**,

**DUTY\_CYCLE\_25**,

**DUTY\_CYCLE\_50**,

**DUTY\_CYCLE\_75**,

**DUTY\_CYCLE\_MOOD**

};

static esp\_err\_t **init\_pwm\_output\_mutex**(void) {

    pwm\_output\_mutex = **xSemaphoreCreateMutex**();

    return (pwm\_output\_mutex != NULL) ? ESP\_OK : ESP\_FAIL;

}

static uint32\_t **select\_duty\_cycle\_value**(**duty\_cycle\_idx\_t** dutyc\_idx) {

    return duty\_cycle\_values[dutyc\_idx % **DUTY\_CYCLE\_NUM**];

}

static esp\_err\_t **update\_outputs**(int command\_id)

{

    esp\_err\_t ret = ESP\_OK;

    uint32\_t duty\_cycle\_value = **select\_duty\_cycle\_value**((**duty\_cycle\_idx\_t**)command\_id);

    if(command\_id >= 0 && command\_id <= 5) {

        ret|= **update\_duty\_cycle**(LEDC\_CHANNEL\_0, &duty\_cycle\_value);

    }

    else if(command\_id >= 6 && command\_id <= 11) {

        ret |= **update\_duty\_cycle**(LEDC\_CHANNEL\_1, &duty\_cycle\_value);

    }

    else{

        ret |= **update\_duty\_cycle**(LEDC\_CHANNEL\_0, &duty\_cycle\_value);

        ret |= **update\_duty\_cycle**(LEDC\_CHANNEL\_1, &duty\_cycle\_value);

    }

    return ret;

}

esp\_err\_t **update\_pwm\_outputs**(int command\_id) {

    return **update\_outputs**(command\_id);

}

A DUTY\_CYCLE\_\* előtagú konstansok reprezetálják a kitöltési tényezőkhöz rendelt számértéket. Mivel a komponens összesen 6 különböző kitöltési tényezőt tud előállítani, ezért 8-bites rezolúció bőven elegendő. A 0 érték tartozik a 0 %-os, míg a 255 érték a 100 %-os kitöltési tényezőhöz. A *duty\_cycle\_idx\_t* egy enumeráció, amely a *duty\_cycle\_values* konstans tömb elemeinek (az előbb említett konstansok) elérésére szolgál. A *select\_duty\_cycle* függvény a hangparancs azonosítója alapján választja ki maradékos osztással a fenti tömb elemeit – valójában egy „map” függvényként működik, ami mindig nullától hatig terjedő skálán választ egy indexet.

static esp\_err\_t **update\_duty\_cycle**(ledc\_channel\_t led\_channel, uint32\_t \*new\_duty\_cycle\_value)

{

    esp\_err\_t ret = ESP\_OK;

    ret |= **ledc\_set\_duty**(LEDC\_LOW\_SPEED\_MODE, led\_channel, \*new\_duty\_cycle\_value);

    ret |= **ledc\_update\_duty**(LEDC\_LOW\_SPEED\_MODE, led\_channel);

    return ret;

}

Az *update\_duty\_cycle* függvény 2 paramétert vár: az egyik az a kimeneti csatorna, a második pedig az 8-biten kódolt kitöltési tényezőt tároló érték, amely a csatornára kell, hogy kerüljön.

Az inicializálást végző eljárás az alábbi kódrészletben található:

static esp\_err\_t **pwm\_output\_init**(void)

{

    esp\_err\_t ret = ESP\_OK;

**ESP\_ERROR\_CHECK**(**init\_pwm\_output\_init\_mutex**());

**xSemaphoreTake**(pwm\_output\_mutex, **pdMS\_TO\_TICKS**(1));

    ledc\_timer\_config\_t ledc\_timer\_0 = {

        .speed\_mode = LEDC\_LOW\_SPEED\_MODE,

        .duty\_resolution = LEDC\_TIMER\_8\_BIT,

        .timer\_num = LEDC\_TIMER\_0,

        .freq\_hz = 5000,

        .clk\_cfg = LEDC\_AUTO\_CLK

    };

    ret |= **ledc\_timer\_config**(&ledc\_timer\_0);

    if(ret != ESP\_OK) {

**ESP\_LOGE**(**TAG**, "unable to configure ledc timer 0");

        goto end;

    }

    ledc\_channel\_config\_t ledc\_channel\_0 = {

        .speed\_mode = LEDC\_LOW\_SPEED\_MODE,

        .channel = LEDC\_CHANNEL\_0,

        .timer\_sel = LEDC\_TIMER\_0,

        .intr\_type = LEDC\_INTR\_DISABLE,

        .gpio\_num = **OUTPUT\_CHANNEL\_1**,

        .duty = 0,

        .hpoint = 0

    };

    ret |= **ledc\_channel\_config**(&ledc\_channel\_0);

    if (ret != ESP\_OK) {

**ESP\_LOGE**(**TAG**, "unable to configure ledc channel 0");

        goto end;

    }

    ledc\_timer\_config\_t ledc\_timer\_1 = {

        .speed\_mode = LEDC\_LOW\_SPEED\_MODE,

        .duty\_resolution = LEDC\_TIMER\_8\_BIT,

        .timer\_num = LEDC\_TIMER\_1,

        .freq\_hz = 5000,

        .clk\_cfg = LEDC\_AUTO\_CLK};

    ret |= **ledc\_timer\_config**(&ledc\_timer\_1);

    if (ret != ESP\_OK) {

**ESP\_LOGE**(**TAG**, "unable to configure ledc timer 1");

        goto end;

    }

    ledc\_channel\_config\_t ledc\_channel\_1 = {

        .speed\_mode = LEDC\_LOW\_SPEED\_MODE,

        .channel = LEDC\_CHANNEL\_1,

        .timer\_sel = LEDC\_TIMER\_1,

        .intr\_type = LEDC\_INTR\_DISABLE,

        .gpio\_num = **OUTPUT\_CHANNEL\_2**,

        .duty = 0,

        .hpoint = 0};

    ret |= **ledc\_channel\_config**(&ledc\_channel\_1);

    if (ret != ESP\_OK) {

**ESP\_LOGE**(**TAG**, "unable to configure ledc channel 1");

        goto end;

    }

**ESP\_LOGI**(**TAG**, "pwm outputs initialized successfully.");

    end:

**xSemaphoreGive**(pwm\_output\_mutex);

        return ret;

}

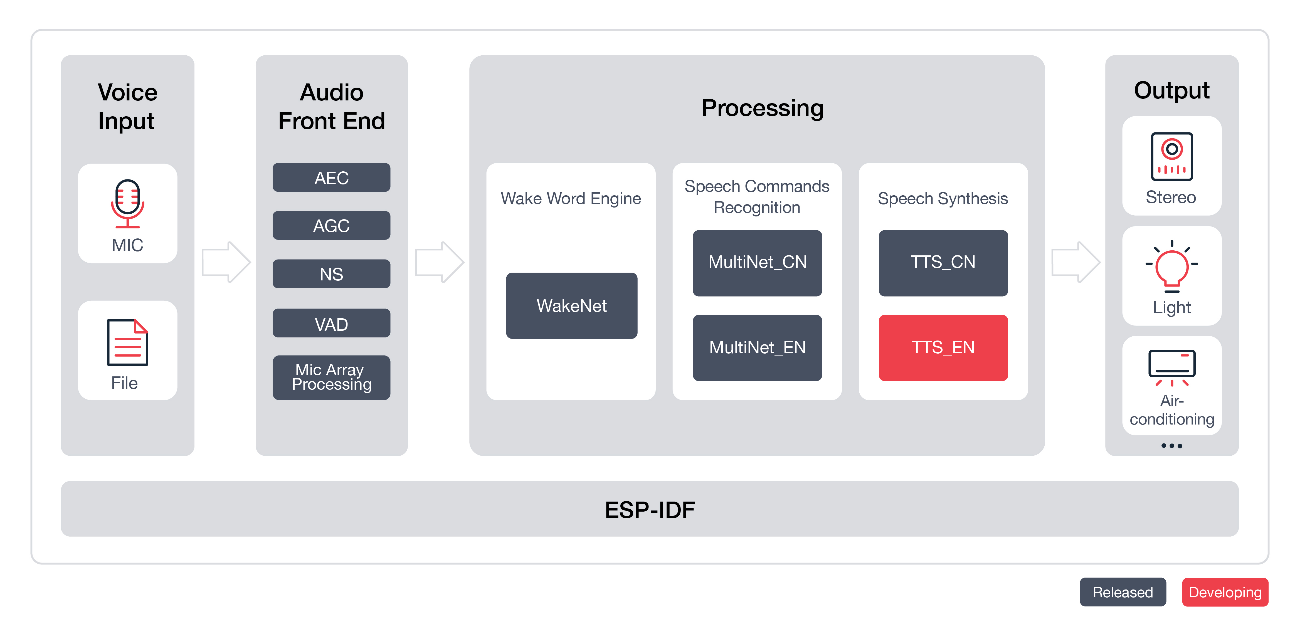
A *pwm\_output\_init* függvény 2 lépésben készíti elő a LEDC periféria kimenetei csatornáit: előszőr a jel előállításhoz szükséges időzítő áramkört konfigurálja, amely 5 kHz-es pwm-jelet fog előllítani, 8-bites felbontással. Második lépésként pedig magát a csatornát helyezi az konfigurációs struktúrában megadott gpio-pinre. Ez a műveletet kétszer ismételi meg. [22]

## A voice\_assistant komponens:

A *voice\_assistant* a projekt legfontosabb része, hiszen egyszerre hangolja az előző három komponens funkcionalitását, illetve futtatja a hangfelismerő algoritmusokat végző taszkokat.

### ESP-Skainet és Speech Recognition Framework:

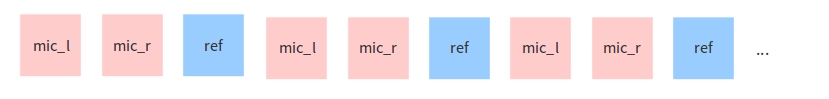
Ábra 21 – az ESP-Skainet felépítésének blokkdiagramja. Forrás: [23].

A *voice\_assistant* működéséhez felhasznált legfontosabb esp-idf komponens az ESP-SR (speech recognition framework) [23], amely az ESP-Skainet intelligens hangasszistens alkalmazásnak a magja. Az ESP-SR az első olyan keretrendszer, amely lehetővé tette az ESP32 mikrovezérlők számára IoT-alkalmazások mellett az AIoT megoldások létrehozását is, amelyek egyszerre használják a mesterséges intelligencia és a dolgok iternete által nyújtott lehetőségeket, létrehozva ezek stratégiai integrációját. A fenti ábrán szereplő részegységek nem mindegyike alkotóeleme a Skainet-nek. A „Voice Input” esetünkben a kettő INMP441-es mikrofon, melynek a kimenetén megjelenő adat a Skainet első részegységébe jut, ez az Audio Front End. A digitalizált hang ezután vagy a WakeNet-, vagy közvetlenül a Multinet motorba kerül további feldolgozásra. A projekt nem használja fel TTS\_\* előtaggal jelölt komponenseket, az “Output” oldalon valójában a világításvezérlés valósul meg.

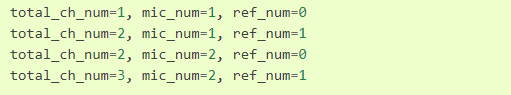
### Audio Front End:

A hangvezérlést támogató termékek esetén alapvető követelmény, hogy azok zajos környezetben is hatékonyan, alacsony hibaránnyal működjenek. Az Audio Front End [24] (vagy röviden AFE) olyan (hang)előfeldolgozó algoritmuskészletet tartalmaz, amelyek képesek a korábban említett elvárásokat biztosítani. A szoftvercsomag lehetővé teszi, hogy a felhasználók az ESP32-S3 sorozatú csipekre egy olyan hangfelismerő „felületet” készítsenek el, amely egyszerre nagy teljesítményű és alacsony memóriaigényű. Ezen egység felelős a következő feladatok elvégzésért: accoustic echo cancellation, blind source separation, noise suppresion, voice activity detection és automatic gain control. Az első három a bemeneti oldalra kerülő hangminták zajtalanítása és „tisztítása”, míg az utolsó kettő a bemenet kifinomultabb érzékeléséért és szükséges erősítésésért felelős.

Az AFE dokumentáció egyik, ha nem a legfontosabb eleme az AFE-ba kerülő hangminták formátuma. Két mikrofonos megoldás esetén: 16 kHz-es és 16-bites adatot vár a következő sorrendben és formátumban (a Skainet kizárólag PCM formátumot támogat):



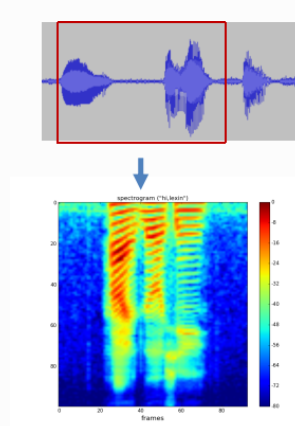
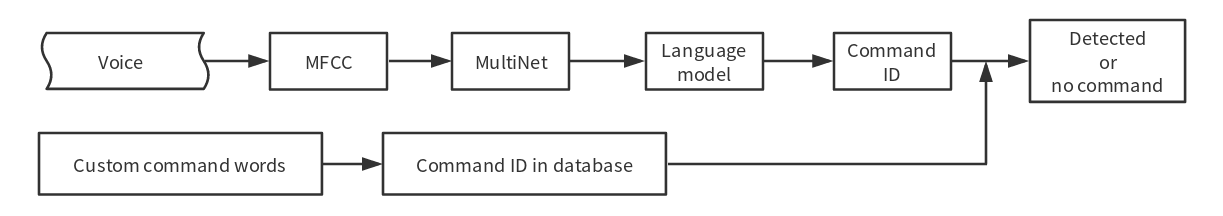
Ábra 22 – az AFE által elvárt adatformátum. Forrás: [24].

Jól látható, hogy először a baloldali mikrofonból, majd a jobb oldali mikrofonból, végül a referencia mikrofonból érkező adatra várakozik. A projekt megvalósítása során én csak kettőt használtam, viszont egyiket sem referenciaként, hiszen egy beltéri környezetre készítettem a projektet, ahol nincs annyi háttérzaj, mint egy kültéri helyiségben. Az Audio Front End a következő mikrofonkiosztásokat támogatja:

Ábra 23 – az AFE által támogatott mikrofonkonfigurációk. Forrás: [24].

Számunkra a harmadik konfiguráció kiválasztása szükséges, illetve az AEC funkció kikapcsolása, hiszen nincs referenciamikrofunk, amely viszonyítási hangszintet biztosítani a visszhang szűréséshez.

### WakeNet:

Ez egy mesterséges neuronhálót használó szoftveregység, amely az ébresztő hangparancsok felismerését végzi. Nagy teljesítménye , memóriatakarékossága és az általa támogatott ébresztőparancsok (“Hi, ESP!”, “Alexa”, “Jarvis”, “Hi, Lexin!”, stb.) hosszú sora teszi igazán vonzóvá. A projektben az „Alexa” wakeword-t használtam fel. A WakeNet [25] az ébresztőparancsok detektálásához az MFCC (Mel-Frekvencia Cepstrum Koeficiens) módszert használja, melynek lényege, hogy a beszédhangot alkotó jelet komponensekre bontja aszerint, hogy a hangzása milyen formában hat az emberi fülre. Az alábbi ábrán jól látható, hogy „ablakozással” az beszédből kiragadnak egy adott hosszúságú keretet, amelyet később a spektrális komponensek alapján az eljárás által használt koefficiensekre bontanak. Az MFCC módszere természetesen több lépést is magába foglal, de ezekkel a jelenlegi dokumentumban nem foglakozom részletesen. A WakeNet-ben használt neurális modellek száma mára már eléri a kilencet, legfőképpen a támogatott ESP32 csipek típusában és a betanítható ébresztőparancsok számában különböznek. A jelenlegi projekt a WakeNet 9. kiadását használja.

Ábra 24– MFCC eljárás. Forrás: [25].

Ábra 25 – A Multinet működésének folyamatábrája.   
Forrás: [26].

### MultiNet:

A MultiNet [26] egy könnyed hangfelismerő mesterséges intelligencia modell, amely az offline hangfelismerő algoritmusok futtaja az ESP32 mikrovezérlőkön. Jelenleg akár 200 parancsot is képes megtanulni, angol és kínai nyelven egyaránt (a modell egyszerre csak egy nyelv használatát támogatja). A MultiNet (és a WakeNet) mesterséges intelligencia modellek használatához, elöszőr a a fejlesztőkit flash memóriájából el kell különítenünk egy partíciót, amely a modellek tárolására szolgál. Az idf egy .csv kiterjesztésű fájl segítségével lehetőve teszi a flash memória egyéni partícionálását: A fájl létrehozása után, a menuconfig segítségével a fájl eléris útvonalát kell megadni. Az általam kiakalított partíciókat a következő fájl szemléltei:

partitions.csv fájl:

# ESP-IDF Partition Table

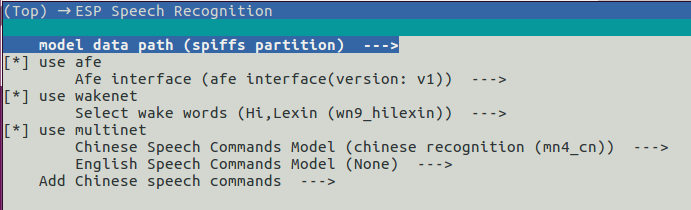
# Using the default Single factory app, no OTA partition, **extending with a spiffs partition,** *used by the wakenet models*

# Name,   Type, **SubType,** *Offset,* Size, Flags

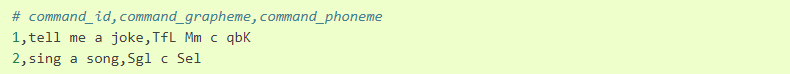
nvs,      data, **nvs,** *,* 0x6000,

phy\_init, data, **phy,** *,* 0x1000,

factory,  app, **factory,** *,* 3000K,

****model,   data, **spiffs,** *,* 3800K,

Ábra 26 – a menuconfig „Speech Recognition” almenüje

A MultiNet-nek összesen három kiadása ismert: különböznek, mind a felhasznált erőforrások mennyiségében, illetve a betanított hangparancsok formátumában: a MultiNet 5 és 7 a parancsokat fonémaként, míg MultiNet 6 esetén grafémaként tároljuk a konfigurációs fájlban. A kettő tárolási formátum közötti különbséget az odal alján lévő ábrák szemlétetik. A projekt megvalósításához a Multinet 5-ös verzióját használtam. Ahhoz, hogy a szövegesen beírt hangparacsokat fonémákká alakítsam, a *g2p.py* Python-nyelven megírt szkriptet használtam.

Ábra 27 – Multinet 5 és 7 tárolási formátuma fent, Multinet 6 tárolási formátuma lent.   
Forrás: [26].

sdkconfig fájl

*#*

*# ESP Speech Recognition*

*#*

CONFIG\_USE\_AFE=y

CONFIG\_USE\_WAKENET=y

CONFIG\_SR\_WN\_WN9\_ALEXA=y

CONFIG\_USE\_MULTINET=y

CONFIG\_SR\_MN\_EN\_MULTINET5\_SINGLE\_RECOGNITION\_QUANT8=y

*#*

*# Add English speech commands*

*#*

CONFIG\_EN\_SPEECH\_COMMAND\_ID0="TkN nN ZbN WcN"

CONFIG\_EN\_SPEECH\_COMMAND\_ID1="TkN eF ZbN WcN"

CONFIG\_EN\_SPEECH\_COMMAND\_ID2="ZbN WcN BRiTNcS TWfNTgViF PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID3="ZbN WcN BRiTNcS FgFTm PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID4="ZbN WcN BRiTNcS ScVfNTcFiT PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID5="ZbN WcN MoD LiT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID6="TkN nN ZbN To"

CONFIG\_EN\_SPEECH\_COMMAND\_ID7="TkN eF ZbN To"

CONFIG\_EN\_SPEECH\_COMMAND\_ID8="ZbN To BRiTNcS TWfNTgViF PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID9="ZbN To BRiTNcS FgFTm PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID10="ZbN To BRiTNcS ScVfNTcFiT PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID11="ZbN To MoD LiT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID12="TkN nN jc LiTS"

CONFIG\_EN\_SPEECH\_COMMAND\_ID13="TkN eF jc LiTS"

CONFIG\_EN\_SPEECH\_COMMAND\_ID14="BRiTNcS TWfNTgViF PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID15="BRiTNcS FgFTm PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID16="BRiTNcS ScVfNTcFiT PkSfNT"

CONFIG\_EN\_SPEECH\_COMMAND\_ID17="MoD LiT"

Az alábbi konfigurációs fájlbeli részlet az Skainet részegységeiek konfigurációs beállításait és a projektben betanított hangparancsokat mutatja be fonemás alakban. Például a CONFIG\_EN\_SPEECH\_COMMAND\_ID0-ban található fonéma a “Turn on the lights” parancsot enkódolja. Az hozzájuk rendelt ID segítségével azonosítjuk be a paracsot, majd hajtjuk végre a megfelelő szalag ki-, be- vagy dimmelt kapcsolását.

voice\_assistant.h fájl

#pragma once

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "esp\_log.h"

void **start\_voice\_assistant**(void);

#define **AFE\_CUSTOM\_CONFIG**()

{

.aec\_init = **false**,

.se\_init = **true**,

.vad\_init = **true**,

.wakenet\_init = **true**,

.vad\_mode = VAD\_MODE\_4,

.wakenet\_model\_name = "wn9\_alexa",

.wakenet\_mode = DET\_MODE\_2CH\_95,

.afe\_mode = SR\_MODE\_HIGH\_PERF,

.afe\_perferred\_core = 0,

.afe\_perferred\_priority = 5,

.afe\_linear\_gain = 1.0,

.memory\_alloc\_mode = AFE\_MEMORY\_ALLOC\_MORE\_PSRAM,

.agc\_mode = AFE\_MN\_PEAK\_AGC\_MODE\_2,

.pcm\_config.total\_ch\_num = 2,

.pcm\_config.mic\_num = 2,

.pcm\_config.ref\_num = 0,

.pcm\_config.sample\_rate = 16000,

.debug\_init = **false**,

.debug\_hook = {{AFE\_DEBUG\_HOOK\_MASE\_TASK\_IN, NULL},

}

A hangfelismerő komponens header-jében egyetlen függvény és egy makró található. A makró az Audio Front End-et repezentáló struktúra mezőit inicializálja a következő értékkel. A visszhang csillapítása (*aec\_init*) nincs engedélyezve tekintettel a *pcm\_config*\* előtagú beállítások értékeire: összesen 2 hangforrásunk (mikrofonok) van, amelyek 16 kHz-en mintavételeznek és egyik sem szolgál referenciaként. A WakeNet modell neve és az általa támogatott ébresztőparancs az “Alexa!”, amely a szükséges memóriát a PSRAM-ból igyekszik allokálni.

voice\_assisstant.c fájl

#include <stdbool.h>

#include "esp\_wn\_iface.h"

#include "esp\_wn\_models.h"

#include "esp\_afe\_sr\_models.h"

#include "esp\_afe\_config.h"

#include "model\_path.h"

#include "esp\_mn\_iface.h"

#include "esp\_mn\_models.h"

#include "model\_path.h"

#include "esp\_process\_sdkconfig.h"

#include "esp\_log.h"

#include "voice\_assistant.h"

#include "bsp.h"

#include "led\_indicator.h"

#include "pwm\_output.h"

*/\* Private mutex used for initialization \*/*

static SemaphoreHandle\_t sr\_init\_mutex = NULL;

static esp\_afe\_sr\_iface\_t \*\_afe\_handle = NULL;

*/\* Private AFE data container \*/*

static esp\_afe\_sr\_data\_t \*\_afe\_data = NULL;

*/\* Private wakenet interface \*/*

const esp\_wn\_iface\_t \*wakenet = NULL;

*/\* Private multinet interface \*/*

const esp\_mn\_iface\_t \*multinet = NULL;

model\_iface\_data\_t \*mn\_model\_data = NULL;

Az alábbi NULL értékre inicializált mutatók olyan entitásokat jelölnek, melyek kötelezően jelen kell lenniük a hangfelismerést végző komponens inicializáló- és feldolgozó eljárásaiban. Az inicializálizációs folyamat szintén védett az *sr\_init\_mutex* által.

static esp\_err\_t **sr\_flash\_models**(void)

{

**ESP\_ERROR\_CHECK**(**init\_sr\_mutex**());

**xSemaphoreTake**(sr\_init\_mutex, **pdMS\_TO\_TICKS**(0));

    srmodel\_list\_t \*models = **esp\_srmodel\_init**("model");

    if (models == NULL) {

**ESP\_LOGE**(**TAG**, "Unable to load model(s)");

        return ESP\_FAIL;

    }

    char \*wn\_name = **esp\_srmodel\_filter**(models, ESP\_WN\_PREFIX, "wn9\_alexa");

    wakenet = **esp\_wn\_handle\_from\_name**(wn\_name);

    model\_iface\_data\_t \*wn\_model\_data = wakenet->**create**("wn9\_alexa", DET\_MODE\_2CH\_95);

    char \*mn\_name = **esp\_srmodel\_filter**(models, ESP\_MN\_PREFIX, ESP\_MN\_ENGLISH);

    multinet = **esp\_mn\_handle\_from\_name**(mn\_name);

    mn\_model\_data = multinet->**create**(mn\_name, 5000);

**esp\_mn\_commands\_update\_from\_sdkconfig**(multinet, mn\_model\_data);

*/\* Initialize the afe\_handle \*/*

    \_afe\_handle = (esp\_afe\_sr\_iface\_t \*)&ESP\_AFE\_SR\_HANDLE;

    afe\_config\_t afe\_cfg = **AFE\_CUSTOM\_CONFIG**();

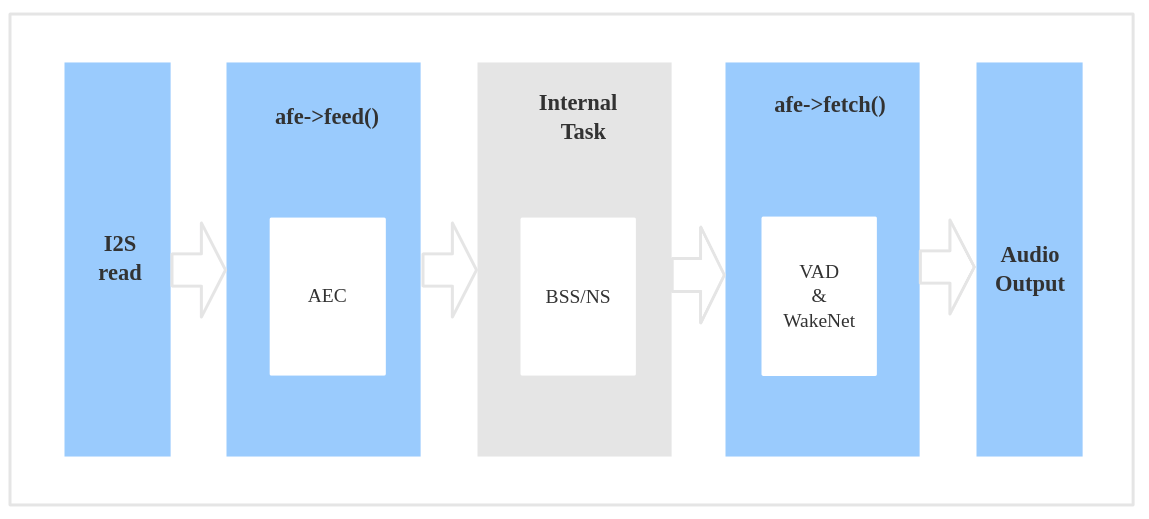
    \_afe\_data = \_afe\_handle->**create\_from\_config**(&afe\_cfg);

**xSemaphoreGive**(sr\_init\_mutex);

     return ESP\_OK;

}

Az *sr\_flash\_models* függvény kiolvassa és előkészíti a program feltöltésekor a flash memóriába kerülő mesterséges intelligencia modelleket és a betanított parancsokat. Miután végzett, az Audio Front End-t kezelő entitást inicializálja a headerben található makró segítségével.



Ábra 28 – Az Audio Front End működési vázlata hangfelismerő üzemmódban.   
Forrás: [24].

static void **feed\_task**(void \*arg)

{

    esp\_afe\_sr\_data\_t \*afe\_data = arg;

    int audio\_chunksize = \_afe\_handle->**get\_feed\_chunksize**(afe\_data);

    int nch = \_afe\_handle->**get\_channel\_num**(afe\_data);

    int feed\_channel = **bsp\_board\_channel\_fmt**();

int16\_t \*i2s\_buff = (int16\_t \*)**calloc**(audio\_chunksize \* feed\_channel,

sizeof(int16\_t));

**ESP\_LOGI**(**TAG**, "feed\_task started");

    while (1) {

**ESP\_ERROR\_CHECK**(**bsp\_read\_i2s\_data**( i2s\_buff,

audio\_chunksize\*feed\_channel, sizeof(int16\_t)));

       \_afe\_handle->**feed**(afe\_data, i2s\_buff);

    }

}

static void **fetch\_task**(void \*arg)

{

    esp\_afe\_sr\_data\_t \*afe\_data = arg;

    int afe\_chunksize = \_afe\_handle->**get\_fetch\_chunksize**(afe\_data);

    int mu\_chunksize = multinet->**get\_samp\_chunksize**(mn\_model\_data);

**assert**(mu\_chunksize == afe\_chunksize);

**bool** detect\_flag = **false**;

**led\_indicator\_t** led\_indicator = **led\_indicator\_init**();

**ESP\_ERROR\_CHECK**(**led\_indicator\_start**(led\_indicator));

**ESP\_LOGI**(**TAG**, "fetch\_task started");

    while (1) {

        afe\_fetch\_result\_t \*res = \_afe\_handle->**fetch**(afe\_data);

        if (res->wakeup\_state == WAKENET\_DETECTED) {

**printf**("wakeword detected\n");

            multinet->**clean**(mn\_model\_data);

            \_afe\_handle->**disable\_wakenet**(afe\_data);

**led\_indicator\_turn\_on**(led\_indicator);

            detect\_flag = **true**;

        }

        if(detect\_flag == **true**)

        {

            esp\_mn\_state\_t mn\_state = multinet->**detect**(mn\_model\_data, res->data);

            if (mn\_state == ESP\_MN\_STATE\_DETECTING) {

                continue;

            }

            if (mn\_state == ESP\_MN\_STATE\_DETECTED) {

*/\* Check the results calculated by Multinet \*/*

                esp\_mn\_results\_t \*mn\_result = multinet->**get\_results**(mn\_model\_data);

                int command\_id = (mn\_result->command\_id[0]);

                (void) **update\_pwm\_outputs**(command\_id);

**ESP\_LOGI**(**TAG**, "-----------listening-----------\n");

            }

            if (mn\_state == ESP\_MN\_STATE\_TIMEOUT) {

                esp\_mn\_results\_t \*mn\_result = multinet->**get\_results**(mn\_model\_data);

                \_afe\_handle->**enable\_wakenet**(afe\_data);

                detect\_flag = **false**;

**led\_indicator\_turn\_off**(led\_indicator);

**ESP\_LOGI**(**TAG**, "-----------awaits to be waken up-----------\n");

            }

        }

    }

}

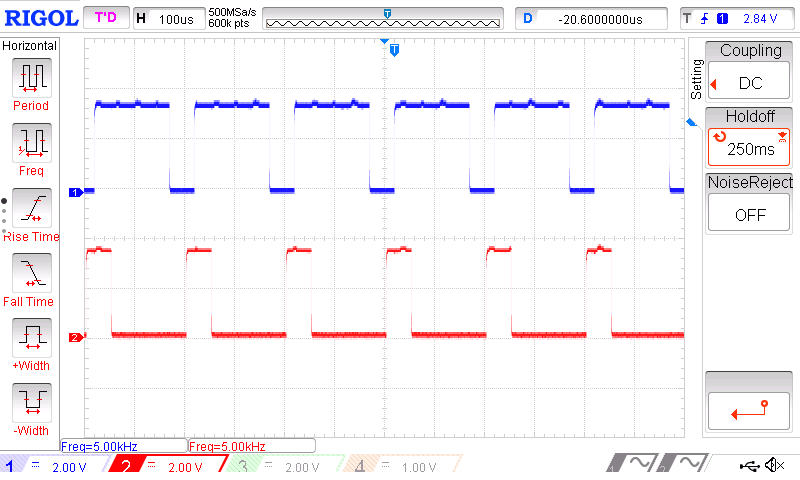
Ahogy a fenti ábrán látható, a Skainet működéséhez 2 taszkot kell definiálnunk, amelyek a *feed* és a *fetch*. A *feed\_task* először egy adott mennyiségű hangmintát előfeldolgoz, majd megtisztítja a felesleges zajoktól – ebben a programegységben sem a WakeNet, sem a MultiNet nem akív. A *fetch\_task* ezután kiolvassa egy belső pufferből az előkészített hangmintákat és a mesterséges intelligencia modelleknek továbbítja. A két taszk részletes működéséről a következő fejezetben teszek említést.

### A feed taszk működése:

A feed taszk első lépésként a paraméterben kapott mutatót konvertálja *afe\_data* típusúra, ezáltal elérthetővé vállik az Audio Front End által támogatott adatformátum. Az *i2s\_buff* az a 16-bites egészeket tartalmazó tömb lesz, amelybe a feldolgozandó hang digitalizált értékei kerülnek – ennek mérete az AFE által lekért darabszám szorozva kettővel (*feed\_channel*), hiszen sztereó módban olvassuk a pcm formátumú adatokat. Ezután a végtelen while-ciklusban a *hardware\_driver* komponens *bsp\_read\_i2s\_data* függvényét használjuk, amivel az I2S csatornákon érkező adatokat olvassuk be a pufferbe, amelyet ezután feldolgoz az *\_afe\_handle* objektum *feed* függvénye.

### A fetch taszk működése:

A *fetch\_taszk* kezdő lépései szinte ugyanazok, mint a *feed\_taszk* esetén, annyi különbséggel, hogy itt hozom létre a *led\_indicator* objektumot, majd indítom el – alap állapotban a fehér jelző LED dióda nem világít. A *detect\_flag* változót arra használom, hogy jelezni tudjam az algoritmus számára, hogy jelenleg milyen állapotban van a program. Az *afe\_fetch\_result\_t* típusú *res* változó a feed taszkban feldolgozott hangmintát és a hozzá tartozó egyéb információkat tartalmazza (fontos megemlíteni, hogy a *feed-* és *fetch* taszkok az idf-keretrendszer szinjtén egymás között szinkronizálva vannak). A végtelen while-ciklusban a következő folyamat zajlik: folyamatosan olvassuk és feed taszk által feldolgozott hangmintákat és a hozzájuk tartozó metainformációkat alapján lépünk be az algoritmus különböző ágaiba. Amennyibe egy ébresztő parancsot észlelünk(teljesül *res->wakeup\_state* == WAKENET\_DETECTED feltétel), kikapcsoljuk a WakeNetet, felkapcsoljuk a fehéren világító LED diódát, majd billentjük a *detect\_flag* változót. Miután a detektálást jelző flag aktív, tovább olvassuk a feldolgozott hangot és amennyiben egy előre betanított hangparacsont észlel a Multinet (teljesül az *mn\_state* == ESP\_MN\_STATE\_DETECTED feltétel), az *mn\_result* változóban a Multinet által felismert parancsok találhatóak meg valószínűség alapján, csökkennő sorrendben (ezért mentjük el a tömb első elemét a *command\_id* nevű változóba). A *command\_id* alapján ezután frissítjük a 2 pwm-kimenetet, így a LED-szalagok az adott erősségű fénnyel fognak világítani. A MultiNet ezután még 5 másodpercig aktív állapotban marad, miután ez az időtartam lejárt (az *mn\_state* == ESP\_MN\_STATE\_TIMEOUT feltétel teljesül), engedélyezzük a WakeNetet, negáljuk a *detect\_flag* változót és kikapcsoljuk a LED diódát. Az *update\_pwm\_outputs* függvény lehetővé teszi a két zóna egymástól független kapcsolását.



Ábra 29 – az 1-es zóna 75%-os, míg a 2-es 25%-os kitöltési tényezővel világít

static esp\_err\_t **init\_sr\_tasks**(void)

{

**ESP\_ERROR\_CHECK**(**sr\_flash\_models**());

    esp\_err\_t ret = ESP\_OK;

**xSemaphoreTake**(sr\_init\_mutex, **pdMS\_TO\_TICKS**(10));

    ret = (**xTaskCreatePinnedToCore**(&**feed\_task**, "feed", 8 \* 1024, (void \*)\_afe\_data, 10, NULL, 0) == pdTRUE) ? ESP\_OK : ESP\_FAIL;

    ret = (**xTaskCreatePinnedToCore**(&**detect\_task**, "detect", 4 \* 1024, (void \*)\_afe\_data, 10, NULL, 1) == pdTRUE) ? ESP\_OK : ESP\_FAIL;

**xSemaphoreGive**(sr\_init\_mutex);

    return ret;

}

Az *init\_sr\_tasks* függvény első lépésként elvégzi a Skainet 3 komponensének konfigurálását, majd definiál kettő FreeRTOS taszkot, a megfelelő paraméterrel, veremmérettel és prioritással – fontos, hogy a két taszk fizikailag és a mikrovezérlő két különálló processzormagján fusson a számítási teljesítmény maximalizálásának érdekében (xTaskCreatePinnedToCore).

void **start\_voice\_assistant**(void)

{

**ESP\_ERROR\_CHECK**(**init\_pwm\_outputs**());

**ESP\_ERROR\_CHECK**(**bsp\_board\_init**(16000, 2, 32));

**ESP\_ERROR\_CHECK**(**init\_sr\_tasks**());

}

A *voice\_assistant* utolsó függvénye az előző eljárásokat hívja meg egymás után, az ESP\_ERROR\_CHECK makró pedig megállítja a program futását és jelzi, ha bármelyik függvény hibával tér vissza.

## A program belépési pontja:

main.c fájl:

#include <stdio.h>

#include <stdlib.h>

#include "esp\_log.h"

#include "voice\_assistant.h"

#define **TAG** "main"

void **app\_main**(void)

{

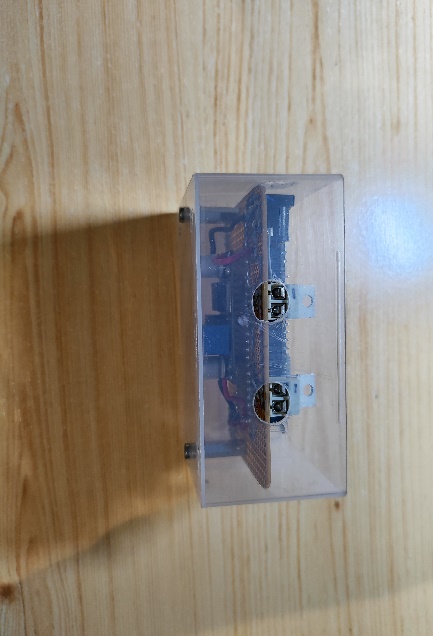
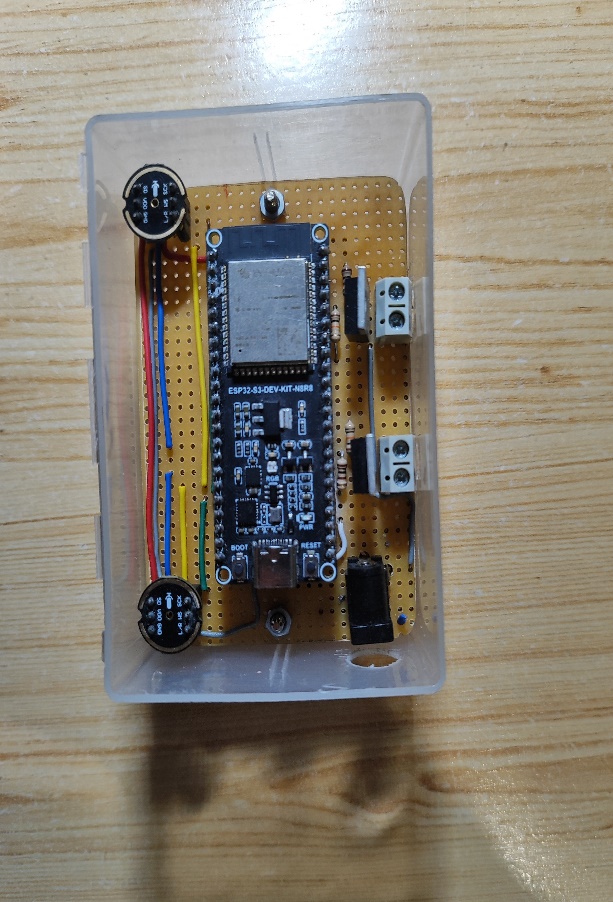
**start\_voice\_assistant**();

}

A program indítása egyszerű: az *app\_main* függvényben meghívjuk a voice\_assisstant-beli *start\_voice\_assistant* függvényt.

# A diplomamunka eredménye:

Az alábbi képeken megtekinthető az elkészült prototípus, melynek a fejlesztésénél törekedtem a tömör kialakításra és a könnyen szerlhetőségre. Az áramkört tartalmazó lap, egy átlátszó műanyag dobozba került, amely a szükséges furatok is megtalálhatóak. A doboz bárhova elhelyezhető, ahol könnyen éri a hang (akár falra is), illetve közel vannak LED szalagok vezetékei.

Az áramkör elkészítéséhez egy eredeti Alexa-t támogató termék árának töredékét használtam fel. A projekt a jövőben megalapozhatja egy komolyabb áramkör/eszköz megépítését is, amely sokkal több kimenettel és vezérlési móddal (MQTT, WIFI, Zigbee) képes működni. Nem titkolt célom az, hogy megtanuljak nyomtatott áramköri lapokat készíteni, megalkotva egy sokkal elegánsabb és szakszerűbb eszközt.

Ábra 30 – az általam elkészített áramköri lap és a tárolásához szükséges doboz.

# Irodalomjegyzék:

[1]: ESP32 mikrovezérlő ismertető

Megtekintés dátuma: 2024.07.01.

<https://www.espressif.com/en/products/socs/esp32>

[2]: Hangfelismerés fogalma

Megtekintés dátuma: 2024.07.01.

<https://hu.shaip.com/blog/voice-recognition-overview-and-applications/>

[3]: ESP-IDF keretrendszer:

Megtekintés dátuma: 2024.07.01.

<https://github.com/espressif/esp-idf>

[4]: Espressif cég ismertető

Megtekintés dátuma: 2024.07.01.

<https://www.espressif.com/>

[5]: ESP32 DevkitC ismertető és lábkiosztás:

Megtekintés dátuma: 2024.07.01.

<https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32/esp32-devkitc/user_guide.html>

[6]: ESP32-S3 DevkitC ismertető és lábkiosztás:

Megtekintés dátuma: 2024.07.01.

<https://docs.espressif.com/projects/esp-idf/en/v5.2.3/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html>

[7]: PSRAM csip ábra:

Megtekintés dátuma: 2024.07.01.

<https://www.gridconnect.com/products/esp-psram-64-for-wroom-32-and-esp32>

[8]: MEMS-technológia és MEMS-mikrofonok:

Megtekintés dátuma: 2024.07.04.

<https://www.magyar-elektronika.hu/tartalom/mems-mikrofonok-segitik-a-hangeszlelest-es-a-kulcsszofelismerest-a-hangvezerelt-rendszerekben>

[9]: MEMS-mikrofon csip ábra:

Megtekintés dátuma: 2024.07.04.

<https://invensense.tdk.com/products/digital/t5818/>

[10]: INMP441 adatlap és ábrák:

Megtekintés dátuma: 2024.07.04.

<https://invensense.tdk.com/wp-content/uploads/2015/02/INMP441.pdf>

[11]: INMP441 csippel szerelt mikrofonmodul

Megtekintés dátuma: 2024.07.04.

<https://techiesms.com/product/inmp441-microphone-module-i2s/>

[12]: I²S szabvány ismertető:

Megtekintés dátuma: 2024.07.06.

<https://www.feasycom.com/hu/what-is-audio-i2s-interface.html>

[13]: ESP-IDF I²S könyvtár:

Megtekintés dátuma: 2024.07.06.

<https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/i2s.html>

[14]: Felhasznált LED szalag ismertető:

Megtekintés dátuma: 2024.07.07.

<https://www.elementa.rs/proizvod/61062/led-traka-toplo-bela-120-led-1m>

[15]: AC-DC átalakító:

Megtekintés dátuma: 2024.07.08.

<https://www.elementa.rs/proizvod/65129/ispravljac-12v-3a>

[16]: LM2595 tápegység:

Megtekintés dátuma: 2024.07.08.

<https://www.addicore.com/products/lm2596-step-down-adjustable-dc-dc-switching-buck-converter>

[17]: IRL-540N tranzisztor:

Megtekintés dátuma: 2024.07.11.

<https://www.elementa.rs/proizvod/15979/fet-tranzistor-n-logl-to220>

[18]: Laboratory practical with the C8051Fxxx microcontroller family

Megtekintés dátuma: 2024.07.12.

<https://eta.bibl.u-szeged.hu/901/>

[19]: FreeRTOS ismertető és ábra

Megtekintés dátuma: 2024.07.18.

<https://www.freertos.org/>

[20]: WS2812 címezhető RGB-LED IC

Megtekintés dátuma: 2024.07.21.

<https://cdn-shop.adafruit.com/datasheets/WS2812.pdf>

[21]: IDF Component Manager:

Megtekintés dátuma: 2024.07.21.

<https://docs.espressif.com/projects/idf-component-manager/en/latest/>

[22]: ESP-IDF LEDC könyvtár:

Megtekintés dátuma: 2024.07.22.

<https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/ledc.html>

[23]: ESP-IDF Skainet könyvtár

Megtekintés dátuma: 2024.07.23.

<https://github.com/espressif/esp-skainet>

[24]: ESP-IDF Audio Front-end keretrendszer

Megtekintés dátuma: 2024.07.24.

<https://docs.espressif.com/projects/esp-sr/en/latest/esp32s3/audio_front_end/README.html#audio-front-end-framework>

[25]: ESP-IDF WakeNet keretrendszer:

Megtekintés dátuma: 2024.07.25.

<https://docs.espressif.com/projects/esp-sr/en/latest/esp32s3/wake_word_engine/README.html>

[26]: ESP-IDF MultiNet keretrendszer:

Megtekintés dátuma: 2024.07.25.

<https://docs.espressif.com/projects/esp-sr/en/latest/esp32s3/speech_command_recognition/README.html>

# Nyilatkozat:

Alulírott Fodor Teodor mérnökinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Műszaki Informatika Tanszékén készítettem, mérnökinformatikus BSc diploma megszerzése érdekében.  
  
Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.   
  
Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

2024. december 12.

# Mellékletek:

Az alábbi GitHub repository-ban tekinthető meg a forráskód:

<https://github.com/Teodor12/ESP-32-SMART-LIGHTING/tree/development>