**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 (osszcilloszkóp, 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 lényege, hogy egy két zónás led-világítás tudjunk hanggal vezérelni.

## A megoldási mód:

A kiválasztott ESP32 fejlesztőkit és mikrofonok közötti kommunikáció szoftveres megvalósítása.

## 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

# Bevezetés:

A hangvezérlés egy olyan technológia, amely lehetővé teszi a különböző eszközök és alkalmazások vezérlését beszédhanggal. A hangvezérlést biztosító rendszerek használata egyre elterjedtebb a mindennapi életben, például okostelefonok, autók és egyéb elektronikus készülékek esetében, illetve ma már 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 lehetőség van 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 mikrovezérlővel megvalósított hangfelismerés offline, 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 keretrendszernek, és sajnos nem nyílt forráskodú a framework ezen könyvtára.

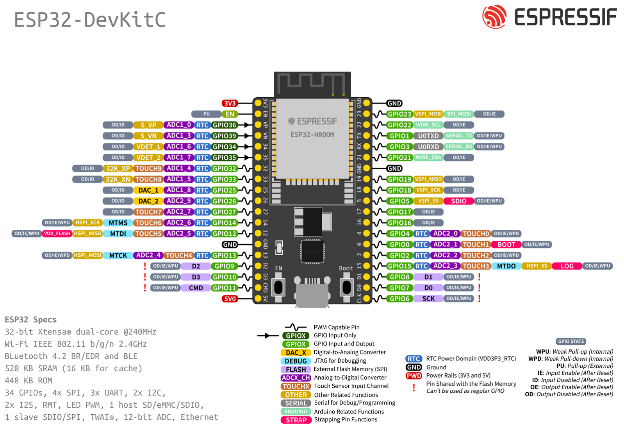
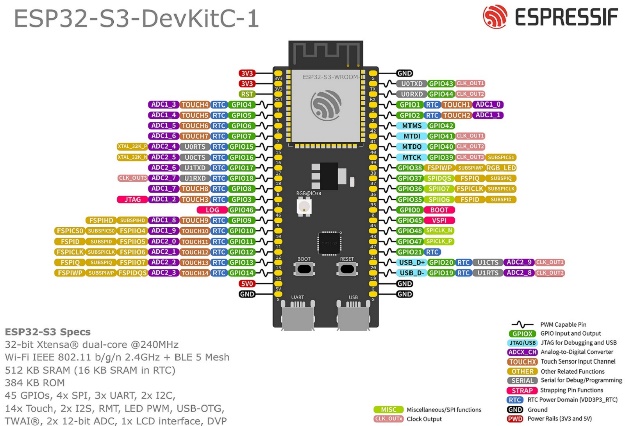
Nem titkolt célja a projektnek, hogy egy pénztárcabarát és könnyen használható, illetve konfigurálható eszközt tervezzek, amely egyszerűbb vezérlési feladatok elvégzésére képes.

# 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 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 feladathoz megfelelően tudjuk kiválasztani a számunkra megfelelő chipet.

Ábra 1

Ábra 2

A diplomamunkámhoz az ESP32-S3 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 gyártott le 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 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.

Ábra 3 – Espressif cég által gyártott külső PSRAM csip

## 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 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.

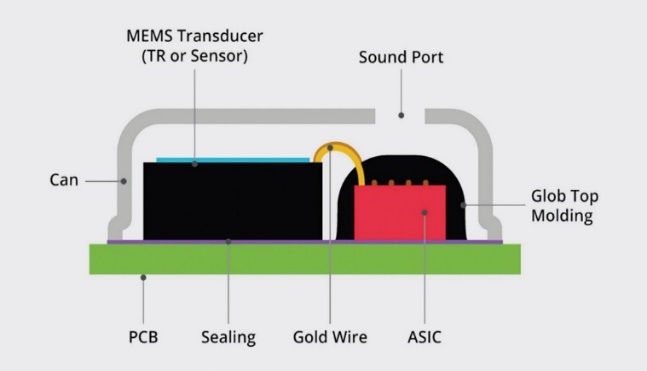
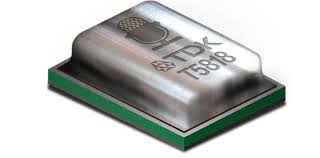
A MEMS mikrofonok jellemzően két, egyazon tokban elhelyezett alkatrészből állnak: egy MEMS membránból, amely átalakítja a hanghullámokat elektromos jellé, valamint egy erősítőből, amely impedanciaátalakító-ként működve használható analóg kimenőjelet ad a hangfrekvenciás jelfeldolgozó láncra. A harmadik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

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”.

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.

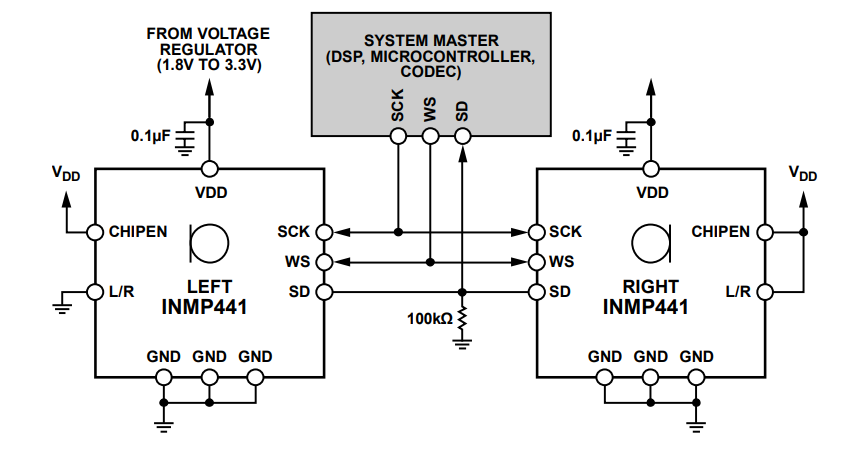
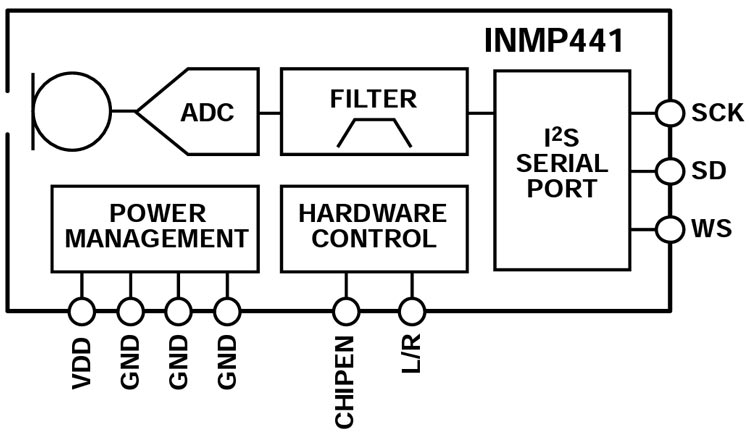
Érzékenység: megadja, hogy a mikrofon egy 1 kHz-es 94 dB hangnyomású hanghullámra adott analóg/digitális kimeneti értéket

Ábra 4

Digitális MEMS mikrofon használata esetén a mikrofon kimenőjele közvetlenül rávezethető digitális áramkörökre, általában egy mikrovezérlőre Az erős villamos zajú környezetekre tervezett hangvezérléses felhasználói felületek jellemzően inkább a digitális mikrofonokat részesítik előnyben, mert a digitális kimenőjeleknek nagyobb a zajtűrésük, mint az analógoknak.

Ábra 5

### INMP441:

Az INMP441 egy olcsó és könnyen beszerezhető, alacsony fogyasztású MEMS mikrofon modul. Fontos tulajdonsága, hogy I²S kimeneti interfésszel rendelkezik és omnidirekcionális, amely 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 az adatátvitel.

Ábra 6 – INMP441-el szerelt mikrofon modul

Ábra 7 az INMP441 típusú MEMS-mikrofon blokkadiagrammja

Ábra 8 – 2 INMP441 típusú mikrofon kapcsolása sztereó módban

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 szolgáltassa a

kimeneti adatot. Az alábbi ábra szemlélteti a sztereó-kimenet kapcsolását.

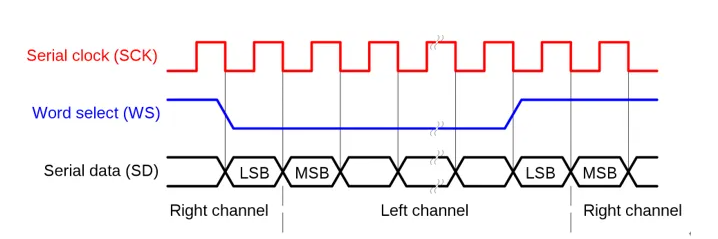
Jól látható, hogy az egyik mikrofon csip L/R pinje tápfeszültségre, míg a másik eszköz L/R pinje GND-re van csatlakoztatva. Az ábrán feltüntetett szűrőkondenzátorok és ellenállások már alapból megtalálhatóak a mikrofonos lapocskán, ezért használatuk elhanyagolható. Az ábra utolsó fontos eleme, a „SYSTEM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

### I²S szabvány:

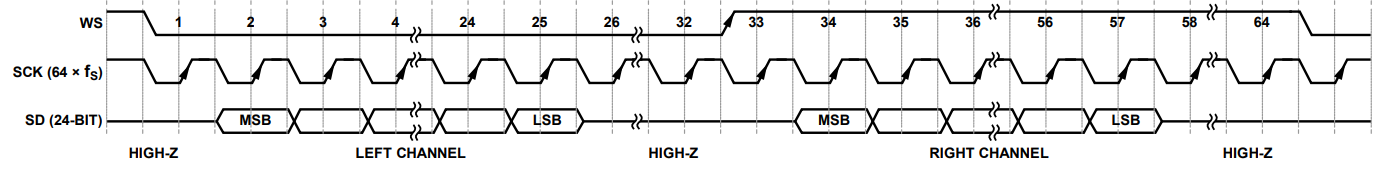
Az I²S (Inter-IC Sound) egy elektronikus soros busz interfész szabvány, amelyet digitális audioeszközök összekapcsolására használnak. Ezt a szabványt először a Philips Semiconductor vezette be 1986-ban. Digitális audió adatok átvitelére használják az elektronikus eszközök integrált áramkörei között.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

Összesen 3 jelalak jellemzi:

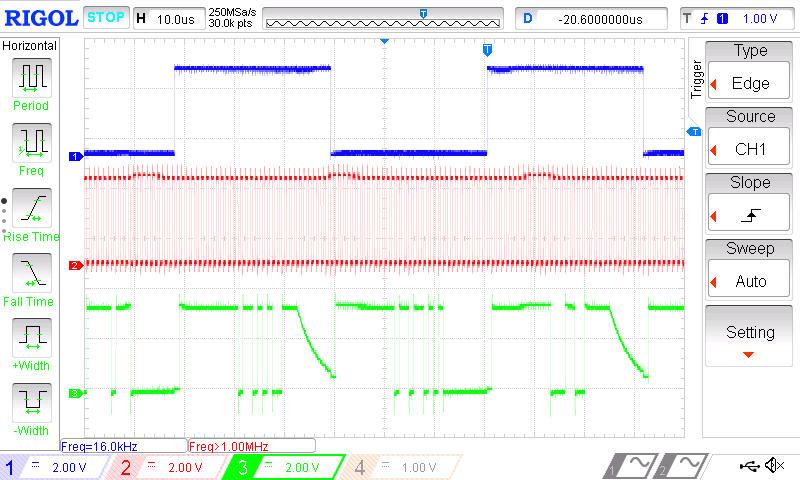
* Bit órajel – az általunk használt mikrofon esetében ez az SCK-pinen (serial clock) jelenik meg. Feladata az átviteli adat bitjeinek az ütemezése
* Szó óravonal – formálisan ez a mikrofon WS-pinjének (word select) a jele. Feladata, hogy sztereó kimenet esetén kiválasztja, hogy a jobb vagy bal oldali 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



Ábra 10 – Az INMP441 mikrofon I2S idődiagramja sztereó-kimenet esetén

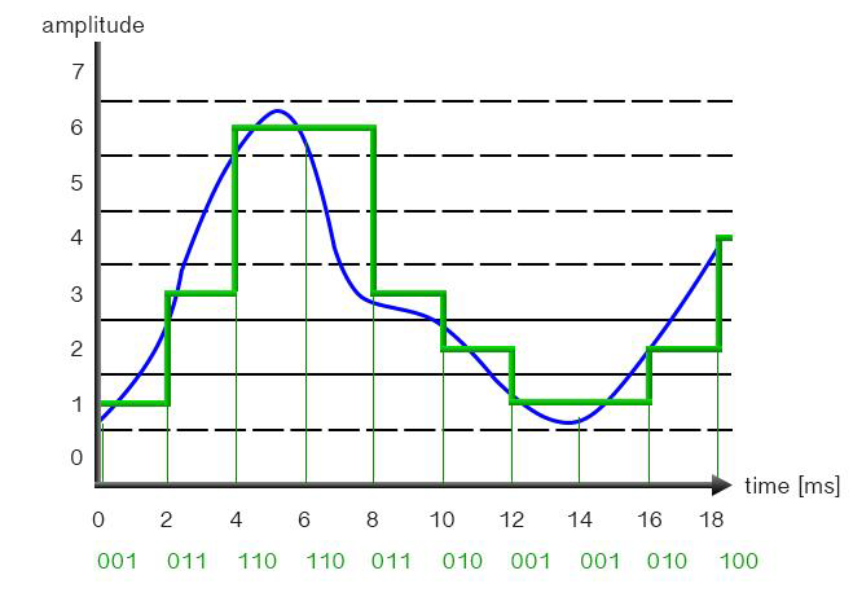
Adatlap alapján a mikrofon kimenetén 24-bites adatok jelennek meg, amelyek a legnagyobb helyiértékű bittel kezdődnek. A dolog érdekessége, hogy mégis 32 impulzus van egy fél-WS-ciklus(1 darab csatorna olvasása ) folyamán. Ez azzal magyarázható, hogy a mikrofon ilyenkor egy magas impedanciás állapotba 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 legnagyobhelyiértékű bitjét a WS-ciklus elején szükségünk van egy bit-órajel impulzusra, ez a formátum a „Philips formátum”. Az ESP-IDF I2S könyvtárában 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.

Ábra 11

### PCM – Impulzus-kódmoduláció:

Az impulzus-kódmoduláció a távközlésben régóta széleskörben használt módszer analóg jelek digitálissá alakítására és átvitelére egy digitális csatornán. A módszer két fő komponense az időbeli diszkretizálás, a mintavételezés, és a minták értékeinek diszkretizálása, azaz kvantálása.

* Mintavételezés: Az analóg jelből a mintavételezési frekvenciának megfelelő gyakorisággal mintát vesznek. A mintavételezés szigorúan a Shannon-mintavételezési törvénynek megfelelően kell, hogy történjen: az eredeti jel rekonstruálásának érdekében a mintavételezési frekvencia legalább kétszerese kell legyen a mintavételezett jel frekvenciájának.
* Kvantálás: A mintavételezett értékeket diszkrét szintekre (kvantumszintekre) kerekítik. Ez a folyamat során minden mintát a legközelebbi kvantumszinthez rendelnek hozzá, ami bizonyos fokú pontatlanságot, az úgynevezett kvantálási zajt eredményez.
* Kódolás: Utolsó mozzanatként a kvantált értékeket binárissá alakítjuk impulzusok reprezentálásával.

Az ábrán kék színnel van jelölve az analóg, mintavételezett jelünk. Zölddel vannak jelölve az egyenlő időközönként kvantált értékek, a vízszintes tengely alatt található zöld értékek pedig a bináris kódok. Például a 4-es és 8-as időpillanat között a kvantált érték 6, ezért ezen az időtartományon belül „110” bináris értékkel kódoljuk a jelet. A 16-os és 18-as időpillanatban a kvantált érték 2, amit a bináris „010” értékkel reprezentálunk.

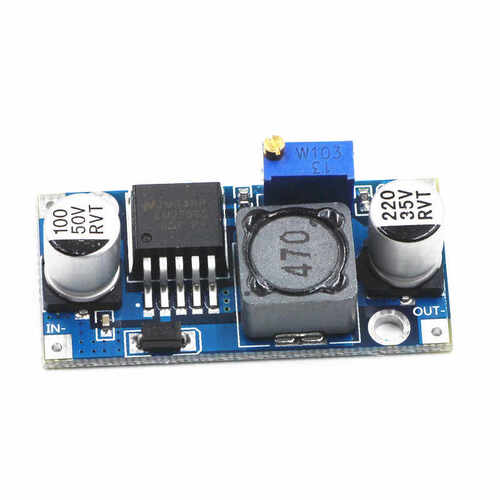
Ábra 12-PCM-moduláció

## A felhasznált LED szalag:

Végül, de nem utolsó sorban a megvilágításért felelős LED szalagról teszek említést. A tervek szerint a szalagot a szobámban található két szekrény hátoldalán szeretném elhelyezni. Miután lemértem a szekrények hosszait és felragasztottam őket, ehhez a szalagon található ragasztón kívül még univerzális ragasztót is felhasználtam a biztos tapadás érdekében. Összesen 3 méter szalagra volt szükségsem, 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 13

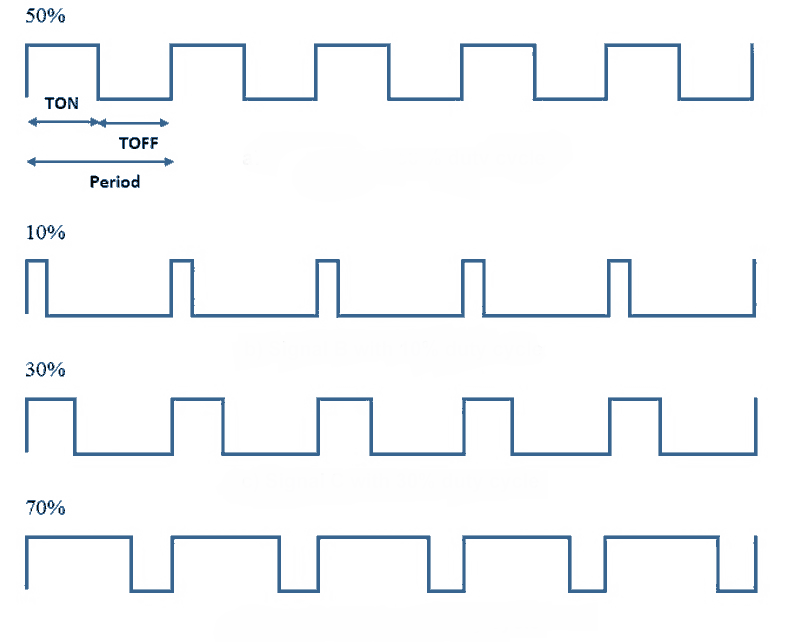
Kihagyhatatlan volt a felhasznált teljesítmény és áramerősség meghatározása is, hiszen a rendszerhez egy megfelelő méretű tápegységet is be kellett szereznem. . Mérések és kísérletezés alapján a két LED szalag egyenként 750 mA-t fogyasztott(maximális fényerősség esetén), illetve a mikrovezérlő elenyésző fogyasztását is belaszámoltam, aminek eredménye egy 3A áramerősségű 12 voltos tápegység lett. (230V hálózati feszültségről alakít). A fejlesztőkit tápfeszültséggel való ellátásáért egy LM2595 DC-DC konvertert használtam, melynek kimeneti feszültségét 3.3V-ra állítottam.

Ábra 15

Ábra 14

### PWM, a digitális analóg jel:

Azok az idők, ahol a fényt csak arra használták, hogy egy teret megvilágítsanak, rég elmúltak. A modern otthonokban már elengedhetetlenek a dimmelhető, azaz fényerejüket szabályozható LED égők és szalagok. Elsősorban a költséghatékonyság érdekében hozták létre a technológiát, ezzel is csökkentve az áramfogyasztást.

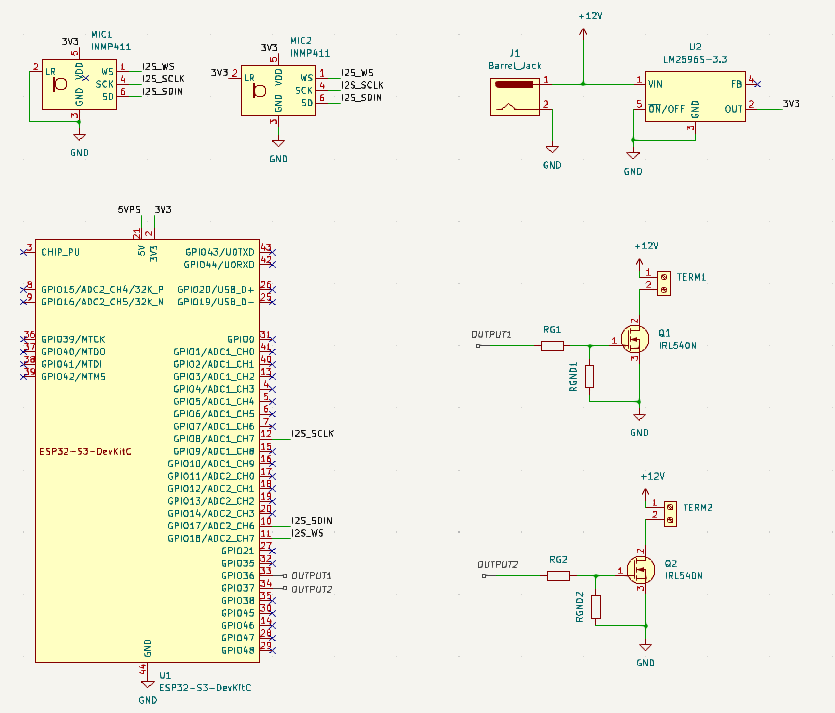
A dimmelhetőség mögött rejlő technológia a PWM moduláció. Ennek elve az, hogy az analóg kimenő feszültségjelek helyettesíthetők digitális impulzussorozat-jelekkel, amelyek hosszabb időtartamra vonatkoztatott átlagfeszültsége egyenértékű az analóg feszültségjellel: fényerő intenzitását azzal szabályozzák, hogy a PWM-jel egy periódusa alatt milyen hosszú ideig van bekapcsolt és kikapcsolt állapotban a világítótest, ezt nevezzünk kitöltési tényezőnek. Ezen egyszerű eljárás az emberi szem „lassú” reakciódejejét használja ki: szemünk a 100 Hz-es villogást már képtelen érzékelni, ezért látjuk folytonosnak ha világít a LED szalag.

Ábra 16 – Különböző kitoltési tényezőjű PWM-jelek

## Egyéb felhasznált eszközök:

* Műanyag doboz – 95 x 60 mm méretű, az áramkör tárolására
* Próbapanel – a dobozhoz mérten méretre vágtam, erre forrasztottam az áramkört
* IRL540N 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 kit 3.3 V-os pinjeivel.
* 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:

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 tartalmazz, 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.

Ábra 17

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 ne az eszköz működéséhez szükséges pineket (BOOT, GND, RST, stb.) 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 kötjük. 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 ellenállásoknak lehúzó ellenállásként funkcionálnak, így biztosítva azt, hogy a térvezérlésű tranzisztor gate-jén vagy logikai magas(3.3V) vagy logikai alacsony (0V) feszültség jelenik meg – a PWM jel bináris értékeinek és a kiválasztott tranisztor karakterisztikájának alapján kapcsolóüzemű a meghajtó áramkör Továbbá a mikrovezérlő pinjeinek a védelmére is szolgál, hiszen sohasem lehetünk biztosak benne, hogy a program inicializációs szakaszát megelőzően milyen állapotban lesznek. 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 ellenállások ezeknek a töltéseknek az elszivárogtatásáért felelős. 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 tápfeszültsg van, a szalag pozitív vezetéke került bekötésre. A tranzisztor N-csatornás, így a PWM-jel a szalag földpontját „szaggatja” nagyfrekvencián (dimmeli), a váltakozó kitöltési tényező pedig a jel effektív értékét, azaz a fényerőt szabályozza.

# Szoftverkörnyezet bemutatása:

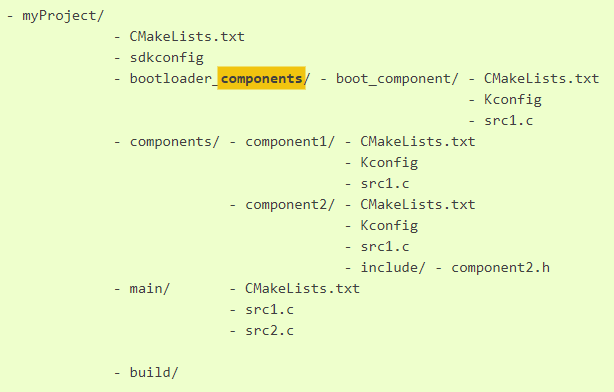
## Espressif IoT Development Framework:

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 valós-idejű alapjaira vannak lefektetve, kivétel nélkül mindegy egyes Espressif-es csipet támogat.

A keretrendszer, hogy nagy előnye, hogy teljese nyílt-forráskodú, teljesen 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.

### Build System:

Az IDF projekt struktúrája a már korábban említett komponensekre (components) épül: például a Wifi-vagy Bluetooth kommunikációért felelős API és ezzel együtt minden felkínált vagy a szoftverfejlesztő által készített component az alábbi struktúrát kötelezően követi:

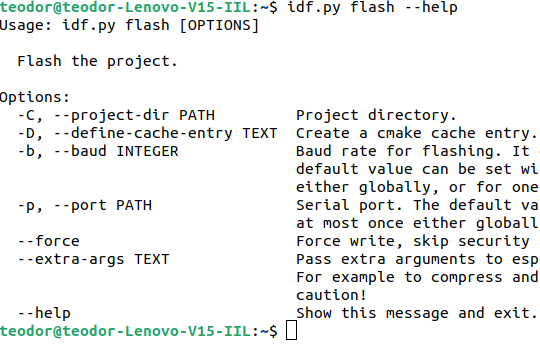
A „components” egy kitüntetett direktórium, ide került minden egyes építőkocka, amelyek szükséges a program futásához. Az IDF a CMake build rendszert használja, amely a C/C++ programokódot tartalmazó forrásfájlok buildelési folyamat konfigurálásáért és az automatizálásáért felelős.

Ábra 18 – egy esp-idf példaprojekt struktúrája

Megjegyzem, hogy nem csak „hagyományos” CMake parancsokat használhatóak, az idf ún. „wrapper” parancsokat támogat. A „CMakelists.txt” fájlok alapján a Ninja elnevezésű build tool ezután buildeli a projektet, nagy előnye, hogy sokkal gyorsabb, mint egy hasonló, hagyományos tool, mivel kihasználja a többmagos processzorok erejét és párhuzamosítási eljárásokat alkalmaz. A ninja által készített fájlok a „build” mappában tárolódnak: itt találhatóak meg az object fájlok (a fordított program bináris alakban), a linkelt, azaz végrehajtható és a csipre feltölthető program és egyéb CMake fájlok is.

Ábra 19 – a projekt egyik komponensének CMake fájlja

### idf.py:

Ez egy parancssori-eszköz (command-line tool), amely egy egyszerű kis frontend felületet biztosít a projekt buildeléséhez, konfigurálásához, monitorozásához illetve az elkészített firmware a mikrovezérlőre való feltöltéséhez is.

Ábra 20 – idf.py parancs használata Linux terminálból

### A projekt konfigurálása:

Az IDF-projekteket az idf.py eszközzel konfigurálhatjuk, melynek eredményeket áll elő az „sdkconfig” fájl. Ezen fájl segítségével az általános beállításoktól a komponensekre vonatkozó specifikus beállításokig szinte minden szabadon változtatható.

Ha például az ESP32-modul flash memóriájának a partíciónálását szeretnénk testreszabni, akkor készítenünk kell egy .csv kiterjesztésű fájl a projektben belül majd az elérési útvonalát megadni a konfigurátor interfészében. Ezen kívül a FreeRTOS órajelét, wifi-pufferek számát, méretét a mikrovezérlő frekvenciáját változtathatjuk és. egyedi konfigurációs beállításokat is létrehozhatunk (ezt egy Kconfig fájlban kell elkészíteni). A konfigurátor segítségével a programozott ESP32 csip szériája is, így az adott sorozatra vonatkozó(és nem vonatkozó) specifikus beállítások automatikus kerülnek felkínálásra.

## FreeRTOS:

Ábra 21 – 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 valósidejű (Real-Time) követelményeket támasztanak, ami azt jelenti, hogy egy-egy esemény bekövetkezésére adott időkorláton belül reagálni kell.

Ábra 22 – FreeRTOS logó

### Feladatok:

A FreeRTOS az adott feladatok (task) között osztja meg az hardver erőforrásait. A task-ok egymástól független végtelen ciklusban futó függvények, amelyek az adott mikrovezérlő heap-memóriájából foglalnak le a stack-ük nagyságával megegyező memóriát. Az előző állítás alapján kijelenthető, hogy a feladatok mind időben és térben (a memória szintjén) el vannak szigetelve egymástól, amelyről az MPU(memória védelmi egység) gondoskodik. Az MPU a memóriában előforduló hibák miatt, például veremtúlcsordulás, elmaradt deallokáció, a szabad memória elfogyása stb. kivételes eseményt fog elindítani, amely nem engedi, hogy a program hibásan tovább fusson.

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-t egy másik feladat használja
* Futó(running) – a feladat hozzá van rendelve a CPU-hoz futtatásra

### 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, nem fogja gyorsabban futtatni a szoftvert.

### Ütemező:

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

### Feladatok közötti kommunikáció (ITC):

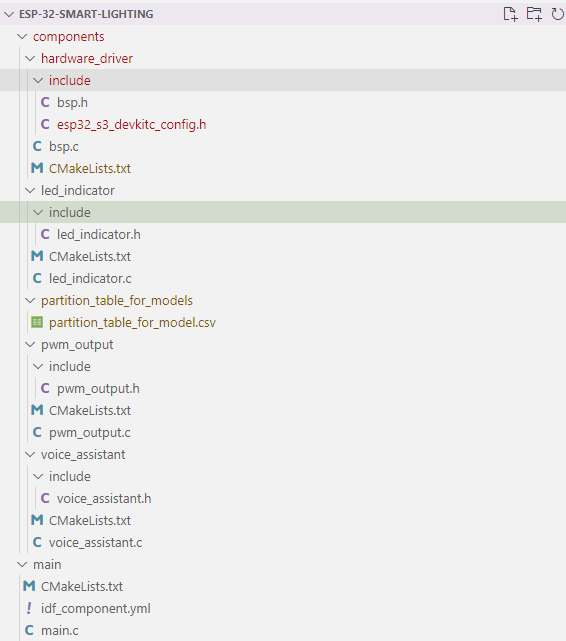
Ha több feladatunk fut párhuzamosan, gyakori jelenség az köztük lévő információcsere, illetve egy globális változó írása/olvasása. Az ilyen „közös” entitások megléte gyakran idézhet elő versenyhelyzetet. Ekkor kettő (vagy több task) egyszerre próbálja meg elérni a globálist, ami a benn lévő adat sérülését okozhatja. Az információk megosztása mellett a FreeRTOS-objektumok képesek szinkronizálni a feladatok végrehajtását is, mivel a feladatok blokkolhatók, hogy várják meg a másik objektumok elérhetőségét – ez esetben a jelenhet a meg holtpont (dead-lock), amikor két vagy több task egymást blokkolja, mivel mindegyik vár valamilyen erőforrásra, amit egy másik tart foglalva. Ennek eredményeként egyik folyamat sem tud előrehaladni, és a rendszer megreked.

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 bitets 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 architectúráját tekintve a voice\_assistant a program fő része, hiszen megteremti a kapcsolatot a fennmaradó 3 komponens között, amelyek egymástól teljesen függetlenül is működőképesek, azonban a kívánt követelmény a képességeik kombinálásával teljesíthető. A program belépési pontja a szokásos app\_main függvényben található, amely elindítja a voice\_assist komponensben található taszkokat. Minden egy inicializációs- és konfigurációs lépés a taszkok indulása előtt fut le, hiba esetén jelez a program. Az következő fejezetben az elkészített komponenseket mutatom be lépésről-lépésre haladva – minden egyes részegységnél ügyeltem arra, hogy a „publikus header” fájlokban csak az olyan függvényeket helyeztem 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 23 – 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.

CMake fájl:

**set**(requires log driver )

idf\_component\_register(SRCS "bsp.c"

INCLUDE\_DIRS "include"

REQUIRES ${requires})

A komponenshez felhasznált idf-komponensek a driver és a log. A driver mappán találhatók meg a beépített I2S API függvényei, konstansai. A log segítségével hívhatóak az IDF képernyőre írást megvalósító eljárásai, amelyekkel általános-, figyelmeztető- és hibaüzeneteket is logolhatunk a monitorozást során.

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 periféria 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 mikrofonoknak, 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 párhuzamos futtatni: könnyen kialakulhatnak olyan helyzetek, amikor egy adott entitást már az előtt szeretnénk használni, hogy befejeződött volna az alap állapotba való tétele. Mutex használatával könnyen és elegánsan tudjuk elérni, hogy egyszerre több folyamat 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. A szálbiztos inicializáció abban merül ki, hogy a mutex-t zároljuk, majd a folyamat végén „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 értékével, 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;

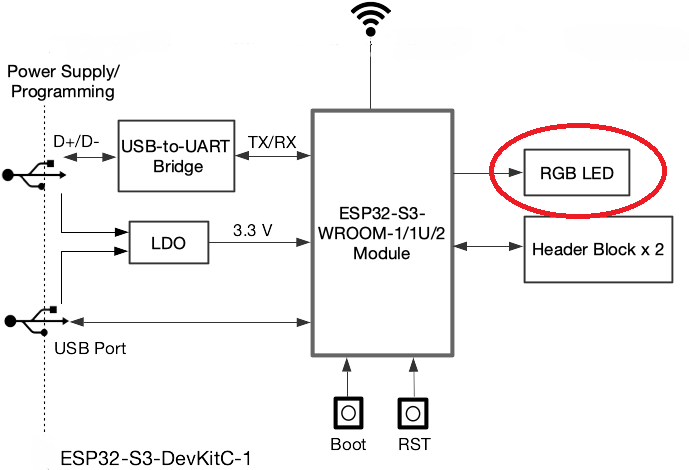
    ret |= **i2s\_read\_data**(buffer, buffer\_size);

    return ret;

}

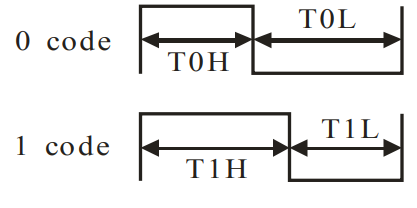
Az i2s\_read\_data() függvény under-the-hood módon végzi el az I2S konfigurélt I2S-csatornán az adatok beolvasását a paraméterként kapott buffer-be, melynek a mérete buffer\_size(bájtokban mérve). A függvény törzsében dinamikusan allokáltam egy 32-bites egészeket (tudjuk, hogy a mikrofonok kimenete 24-bites) tároló tömböt sample\_num méretben. A sample\_num valójában a később bemutatot hangfelismerő algoritmus által meghatározott hangminta (16-bites érték) mennyisége. Ahhoz, hogy a 32-biten tárolt 24-bites értékek megfelelően legyen kerüljenek a paraméterben 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.

Ábra 24 – az ESP32-S3 fejlesztőkit blokkdiagrammja, pirossal megjelölve a beépített RGB LED-dióda

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

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 25 – a WS2812-es típusú LED-dióda kommunikációs protokolljámak kódolása

Ahhoz, hogy a diódát vezérlő csip valóban 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 meghatározott ideig kell, hogy tartson – egy illesztőprogram megírásánál ez nehézkés és meglehetősen időigényes lehet.

Szerencsére az ESP-IDF nem hivatalos komponensei (ESP Component Registry weboldal) 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"

*## Required IDF version*

  idf:

    version: ">=4.1.0"

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

CMake fájl:

**set**(requires log freertos)

**set**(priv\_requires **led\_strip** )

idf\_component\_register(SRCS "led\_indicator.c"

                        INCLUDE\_DIRS "include"

                        REQUIRES ${requires}

                        PRIV\_REQUIRES ${priv\_requires})

A PRIV\_REQUIRES makróval jelzem, hogy a led\_strip komponens egy belső részegysége a led\_indicator-nak, melyet a rejtett implementációban használunk.

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 – ez a struktúra szintén a 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){

**ESP\_LOGE**(TAG, "memory exhausted");

        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 struktúra, 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 fog megvalósítani, amely a paraméterként átaddot led\_indicator entitás adattagjainak állapotával fogja elvégezni a LED dióda ki- és bekapcsolását. A taszk egy végtelen ciklusban fut, itt ellenőrzi, hogy az indicator entitást 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 a minden szükséges beállítás el lett-e végezve.

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 entitás led\_state változójának billentésére használjuk, amely szálbiztos kódon 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;

    };

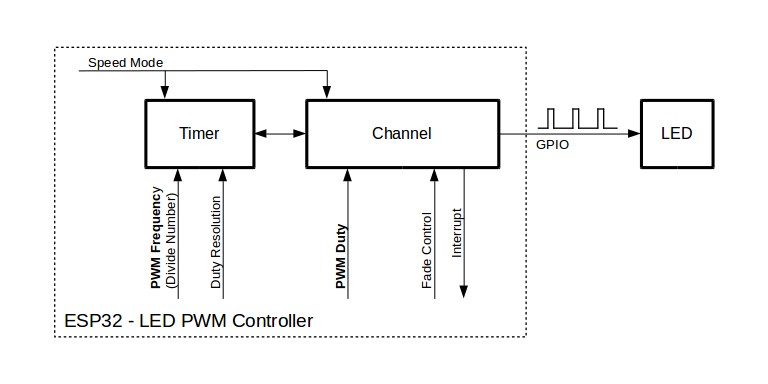
    led\_indicator->led\_state = false;

**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, 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 vezéreljük.

Ábra 26 – a LEDC periféria belső felépítésének blokkdiagramja, egy csatornára vonatkozóan. Esetünkben a LED helyett egy IRL540 tranzisztorra van kapcsolva a kimenete

CMake fájl:

**set**(requires log freertos driver)

idf\_component\_register(SRCS "pwm\_output.c"

INCLUDE\_DIRS "include" REQUIRES ${requires})

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 mindössze 3 függvényt tartalmaz, amelyek közül az első 2 számunkra az igazán fontos. Az init\_pwm\_outputs csak egy wrapper-függvény a belső inicializációs függvény köré, az update\_pwm\_outputs(int command\_id) 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\_mtuex**(void) {

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

    return (pwm\_output\_mtuex != 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 rendel 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\_mtuex, **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\_mtuex);

        return ret;

}

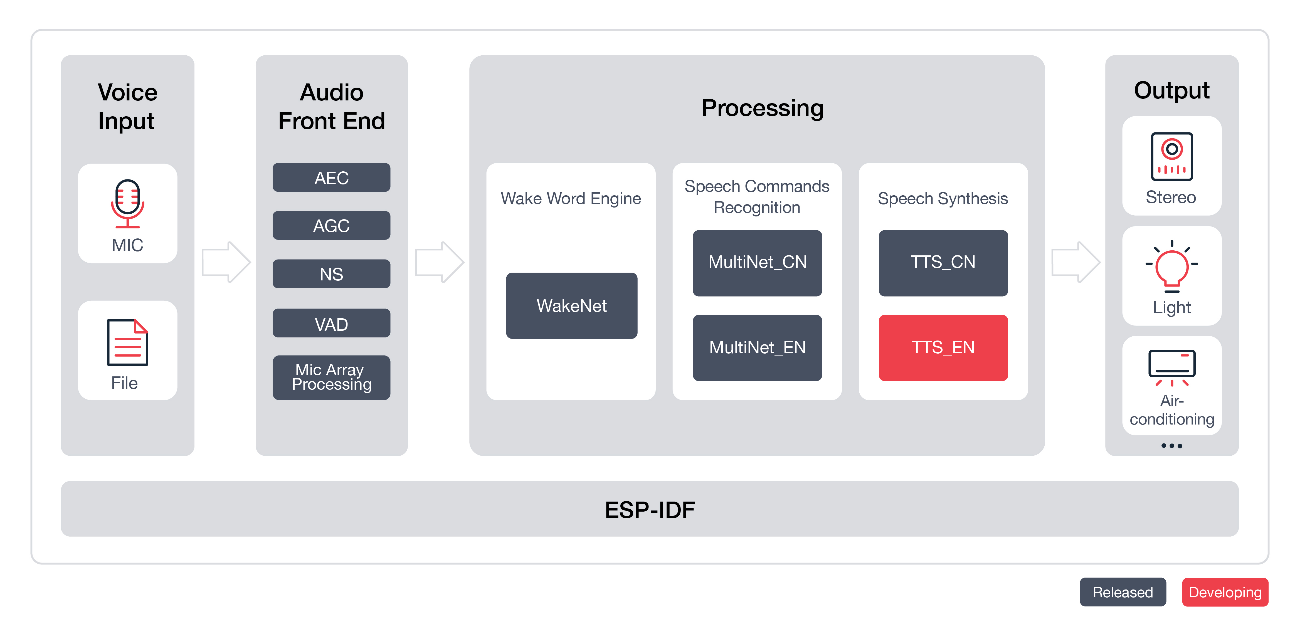
A pwm\_output\_init függvény 2 lépésben készíti elő a LEDC periféria kimenetei csatornáit: előszőr a jel előállításhoz szükséges időzítő áramkört konfigurálja, amely 5 kHz-es pwm-jelet fog előllítani, 8-bites felbontással. Második lépésként pedig magát a csatornát helyezi az konfigurációs struktúrában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

### A voice\_assistant komponens:

A voice\_assistant komponens a projekt legfontosabb része, hiszen egyszerre összehangolja az előző 3 funkcionalitását, illetve futtatja a hangfelismerő algoritmusokat végző taszkokat.

### ESP-Skainet és Speech Recognition Framework:

Az voice\_assistant működéséhez felhasznált legfontosabb esp-idf komponens az ESP-SR (speech recognition framework), 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.

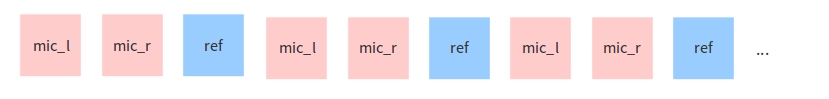
   
A fenti ábrán szereplő részegységek nem mindegyike alkotóeleme a Skainet-nek. A „Voice Input” esetünkben a kettő INMP441-es mikrofon, melynek a kimenetén megjelenő adat a Skainet első részegységébe jut, ez az Audio Front End. A digitalizált hang ezután vagy a WakeNet-, vagy közvetlenül a Multinet motorba kerül további feldolgozásra. A projekt nem használja fel TTS\_\* előtaggal jelölt komponenseket, az “Output” oldalon valójában a világításvezérlés valósul meg.

Ábra 27 – az ESP-Skainet blokkdiagramja

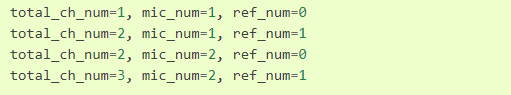
### Audio Front End:

A hangvezérlést támogató rendszerek termékek esetén alapvető követelmány, hogy azok zajos környezetben is hatékonyan, alacsony hibaránnyal működjenek. Az Audio Front End (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 memória költségű. 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 bemenete kifinomultabb érzékeléséért és szükséges erősítésésért felelős.

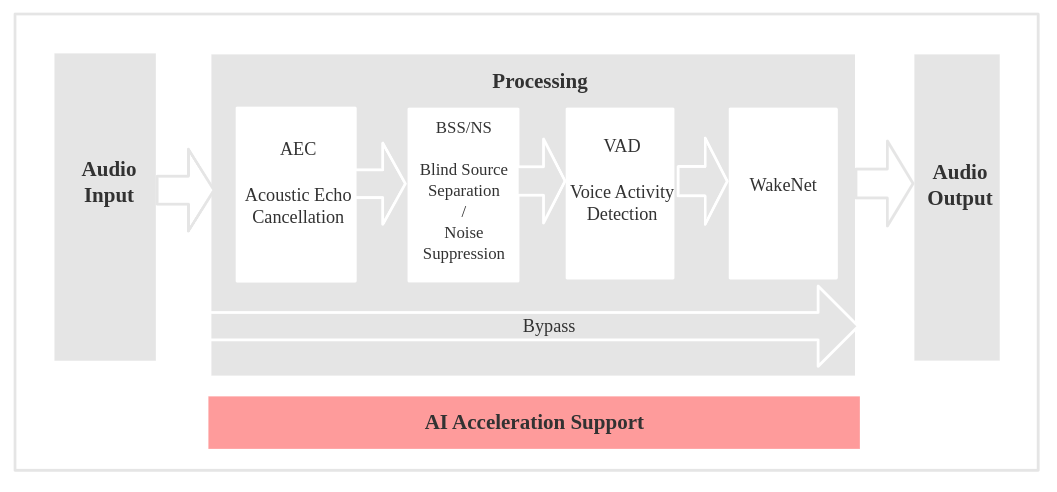
Az AFE dokumentáció egyik, ha nem a legfontosabb eleme az AFE-ba kerülő hangminták formátuma. Két mikrofonos megoldás esetén: 16 kHz-es és 16-bites adatot vár a következő sorrendben és formátumban(minden Skainet komponens ilyen adatot vár):



Ábra 28

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:

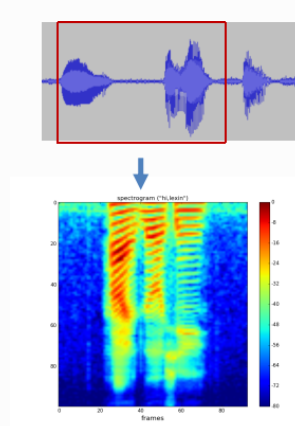
Ábra 29 – az AFE által támogatott mikrofonkonfigurációk

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

Ábra 30

### WakeNet:

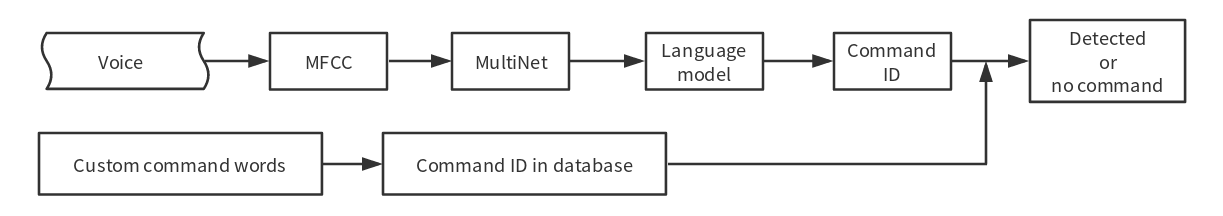
Ez egy mesterséges neuronhálót használó szoftveregység, amely az ébresztő hangparancsok felismerését végzi. Nagy teljesítménye , memóriatakarékossága és az általa támogatott ébresztőparancsok (“Hi, ESP!”, “Alexa”, “Jarvis”, “Hi, Lexin!”, stb.) hosszú sora teszi igazán vonzóvá. A projektben az „Alexa” wakeword-t használtam fel.

A WakeNet 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, amely egyben a sorozat legfrisebb verziója.

Ábra 31

### MultiNet:

A MultiNet 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:

Ábra 32 – A Multinet működésének folyamatábrája

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*

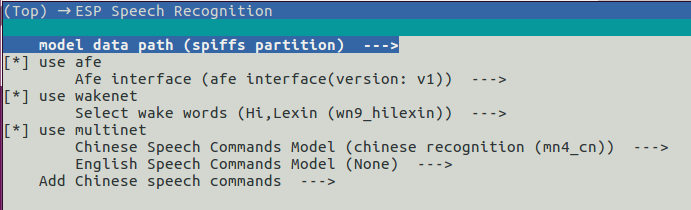
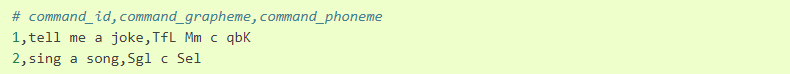
# Name,   Type, **SubType,** *Offset,*  Size, Flags

nvs,      data, **nvs,** *,* 0x6000,

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

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

model,   data, **spiffs,** *,* 3800K,

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 tároljuk a konfigurációs fájlban. A kettő tárolási formátum közötti különbséget az következő 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 33

Ábra 34 – Multinet 5 és 7 tárolási formátuma fent, Multinet 6 tárolási formátuma lent

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.

CMake fájl:

**set**(requires log esp-sr)

**set**(priv\_requires hardware\_driver led\_indicator pwm\_output)

idf\_component\_register(SRCS "voice\_assistant.c" INCLUDE\_DIRS "include"

REQUIRES ${requires}

PRIV\_REQUIRES ${priv\_requires})

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;

*/\* Private AFE-interface to enable functions called by the sr-library \*/*

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;

*/\* Private multinet interface data \*/*

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**();

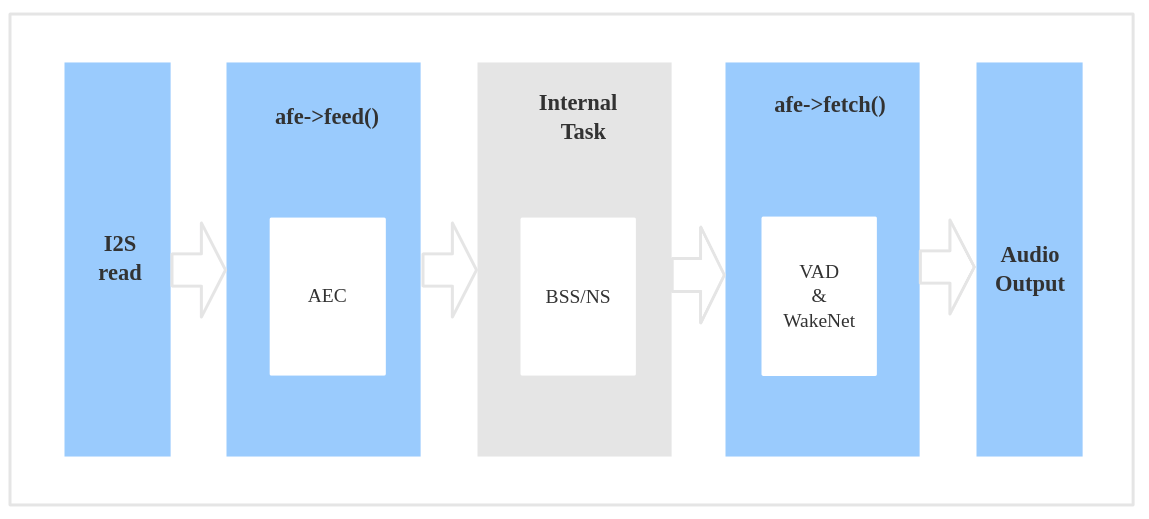
    \_afe\_data = \_afe\_handle->**create\_from\_config**(&afe\_cfg);

**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 35 – Az Audio Front End működési vázlata hangfelismerő üzemmódban.

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");

            }

        }

    }

}

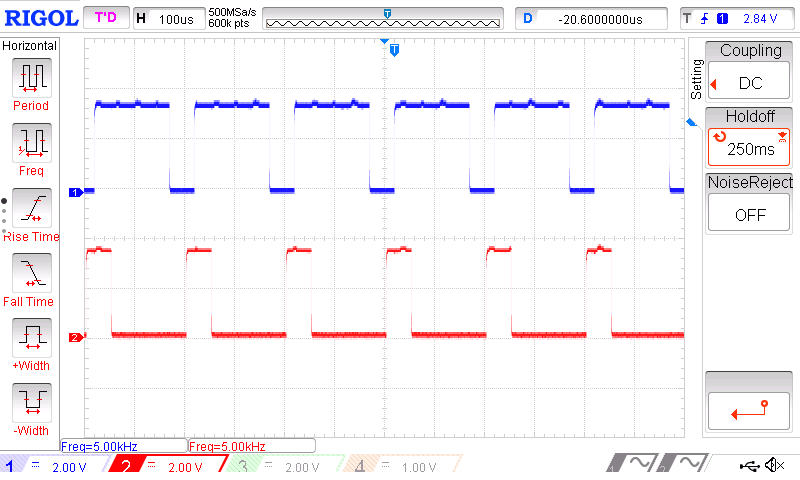
Ahogy a fenti ábrán látható, a Skainet működéséhez 2 feladatot kell definiálnunk, amelyek a feed és a fetch. A feed taszk először egy adott mennyiségű hangmintát előfeldolgoz, majd megtisztítja a felesleges zajoktól – ebben a programegységben sem a WakeNet, sem a MultiNet nem akív. A fetch\_task ezután 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 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 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 36 – 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).