UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE ENGENHARIA ELETRÔNICA

BRUNO TANAKA ADRIANO
GABRIEL CRISTOFOLETTI DIORIO

TRANSMISSÃO DE ÁUDIO SEM FIO EM TEMPO REAL POR BLUETOOTH UTILIZANDO O ESP32

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO
2021

BRUNO TANAKA ADRIANO
GABRIEL CRISTOFOLETTI DIORIO

TRANSMISSÃO DE ÁUDIO SEM FIO EM TEMPO REAL POR BLUETOOTH UTILIZANDO O ESP32

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná - UTFPR Campus Cornélio Procópio, como requisito para a obtenção do título de Bacharel em Engenharia Eletrônica.

Orientador: Prof. Dr. Eduardo Alves Hodgson
Universidade Tecnológica Federal do Paraná

CORNÉLIO PROCÓPIO
2021

Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio
Departamento Acadêmico de Elétrica
Curso de Engenharia Eletrônica

FOLHA DE APROVAÇÃO

Bruno Tanaka Adriano

Transmissão de áudio sem fio em tempo real por Bluetooth utilizando o ESP32

Trabalho de conclusão de curso apresentado às 10:00hs do dia 17/08/2021 como requisito parcial para a obtenção do título de Bacharel em Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná. O candidato foi arguido pela Banca Avaliadora composta pelos professores abaixo assinados. Após deliberação, a Banca Avaliadora considerou o trabalho aprovado.

_____

Prof(a). Dr(a). Eduardo Alves Hodgson - Presidente (Orientador)

_____

Prof(a). Dr(a). Luis Fernando Caparroz Duarte - (Membro)

_____

Prof(a). Dr(a). André Sanches Fonseca Sobrinho - (Membro)

A folha de aprovação assinada encontra-se na coordenação do curso.

Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio
Departamento Acadêmico de Elétrica
Curso de Engenharia Eletrônica

FOLHA DE APROVAÇÃO

Gabriel Cristofoletti Diorio

Transmissão de áudio sem fio em tempo real por Bluetooth utilizando o ESP32

Trabalho de conclusão de curso apresentado às 10:00hs do dia 17/08/2021 como requisito parcial para a obtenção do título de Bacharel em Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná. O candidato foi arguido pela Banca Avaliadora composta pelos professores abaixo assinados. Após deliberação, a Banca Avaliadora considerou o trabalho aprovado.

_____

Prof(a). Dr(a). Eduardo Alves Hodgson - Presidente (Orientador)

_____

Prof(a). Dr(a). André Sanches Fonseca Sobrinho - (Membro)

_____

Prof(a). Dr(a). Luis Fernando Caparroz Duarte - (Membro)

A folha de aprovação assinada encontra-se na coordenação do curso.

Dedicamos esse trabalho a todos que nos auxiliaram e interagiram com ele.

AGRADECIMENTOS

Agradecemos imensamente as nossas famílias, por todo o auxilio e suporte prestados durante nossa formação.

Agradecemos também ao nosso orientador e amigo Prof. Dr. Eduardo Alves Hodgson, por todas as contribuições ao nosso trabalho e também por toda a experiencia profissional transmitida ao longo de nossa convivências.

Além disso, agradecemos os professores avaliadores de banca, Prof. Dr. Luís Fernando Caparroz Duarte e Prof. Dr André Sanches Fonseca Sobrinho, pela correção e avaliação desse nosso trabalho.

Estendemos nosso agradecimento também aos nossos professores, que sempre dedicaram seu tempo e conhecimento durante o curso, além compartilhar suas experiências, para nos ver preparados para os desafios e prontos para o mercado de trabalho. E finalmente agradecemos aos nossos amigo, que direta ou indiretamente, auxiliaram na realização deste projeto.

Nossa tecnologia passou a frente de nosso entendimento, e a nossa inteligência desenvolveu-se mais do que a nossa sabedoria.(REVELLE, Roger)

RESUMO

Este trabalho propõe a utilização de um microcontrolador ESP32 para a criação de um protótipo eletrônico capaz de executar a recepção de áudio sem fio proveniente de um computador ou celular transmitindo áudio pelo protocolo Bluetooth. Além disso, este documento comenta sobre a concepção de um receptor de radiofrequência que converte o sinal recebido em um sinal de áudio e após processamento digital, reproduz em uma saída de fone de ouvido (conector TRS de 3.5mm). Por último, foram apresentados os protocolos, interfaces de programação e softwares necessárias para a realização e concepção do protótipo.

Palavras-chave: Microcontrolador. Bluetooth. Transmissão de áudio.

ABSTRACT

This work presents the use of a micro-controller ESP32 in the development of an electronic prototype capable to perform the wireless audio reception where a computer or a cell phone transmits audio through Bluetooth protocol. Furthermore, this document comments the conception of a audio radio-frequency receptor that converts the acquired signal and digitally process and play in an audio connector (TRS 3.5mm connector). Finally, this work presents the used materials to perform the data transmission and the digital processing, including the protocols, coding and all needed software.

Keywords: Microcontroller. Bluetooth. Wireless transmission.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

A2DP	*Advanced Audio Distribuition Profile*
ABNT	Associação Brasileira de Normas Técnicas
ADC	Conversor Analógico Digital
AVRCP	*Audio Video Remote Control Profile*
DAC	Conversor Digital Analógico
DAELT	Departamento Acadêmico da Eletrônica
ESP32	Microcontrolador da ESPRESSIF
FSK	*Frequency-Shift Key*
FTP	*File Transfer Profile*
GPIO	*General Purpose Input Output*
I²C	*Inter-Integrated Circuit*
I²S	*Inter IC Sound*
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
PCM	*Pulse Code Modulation*
PWM	*Pulse Width Modulation*
SAR	*Successive Approximation Register*
SIG	*Special Interest Group*
SNK	*Sink*
SoC	*System on Chip*
SPI	*Serial Peripherals Interface*
SRC	*Source*
TRS	*Tip-Ring-Sleeve*
UART	*Universal Asynchronous Receiver / Transmitter*
USB-C	*Universal Serial Bus Type C*

SUMÁRIO

1 INTRODUÇÃO

Em 2016, a empresa fabricante de celular, Apple, introduziu ao mercado sua nova linha de fones sem fio, os AirPods. Estes dispositivos foram lançados com o Bluetooth 4.0 além de outras funcionalidades exclusivas do Icloud. Outro lançamento no mesmo evento foi o Iphone 7, um dos primeiros celulares *smatphones* produzidos até então sem o conector de saída TRRS (*Tip-Ring-Ring-Sleeve*) de 3.5mm para áudio. No começo de 2020, outras empresas como a Samsung, acompanharam esse movimento e a partir do Samsung Galaxy 20, os dispositivos de alta performance da empresa passaram a serem produzidos sem a entrada de áudio para fones convencionais.

Esse movimento das empresas de transformar o celular em um ambiente sem fios faz com que antigos dispositivos de áudio não funcionem diretamente nos *smartphones*, tendo assim que adaptá-los através do conector Universal Serial Bus tipo C (USB-C) ou através de um dispositivo capaz de receber áudio Bluetooth e reproduzi-los para o fone.

Neste trabalho, propõe-se a criação de uma solução para que fones com fios possam ser utilizados não somente nesses celulares que já não possuem mais saída TRS, mas também em outros dispositivos com saídas de áudio, trazendo assim a praticidade e mobilidade dos fones sem fio, aos cabeados.

Com isso definido, para adequar-se a essa realidade, métodos de transmissão de dados tornam-se requisitos e que devem ser estudados. Segundo (ROTTA MáRIO DANTAS, 2020), por conta da grande diversidade de protocolos, cada qual com suas características e peculiaridades, é necessário a investigação dos diferentes protocolos disponíveis, para a melhor escolha.

Dentre os vários microcontroladores disponíveis no mercado, foi definido utilizar o ESP32,da Espressif, empresa chinesa desenvolvedora de placas de circuito impresso com microcontroladores e componentes focados na conectabilidade. O ESP32 é uma série popular no ramo de processamento de dados, um SoC *(System on Chip)*, capaz de atender os requisitos de processamento e conectividade. (ESPRESSIF, 2020).

Após definido o microcontrolador, tem-se a variável que será estudada e trabalhada: o som, uma combinação de várias amplitudes distribuídas em inúmeras frequências e para que ele possa ser transmitido de um ponto a outro por meios digitais, segundo (BYCAST, 2018), o mesmo precisa ser amostrado, lido, codificado e após isso pode ser transmitido. Essa é a definição do conceito de *Audio Streaming* ou transmissão de áudio.

Assim, nesse trabalho, será apresentado o desenvolvimento de um protótipo utilizando um ESP32 para realizar o processamento da transmissão e recepção de áudio, sem fio, entre dois dispositivos, sendo que essa transmissão será feita através do protocolo Bluetooth. Com a transmissão sem fio estabelecida, o áudio transmitido será interpretado por um DAC *(Conversor Digital-Analógico)* que converterá os dados digitais em sinais elétricos, sendo então interpretados pelo fone de ouvido.

Além disso, será discutido outra proposta de um segundo protótipo para realizar a transmissão e recepção de áudio sem fio, sendo utilizado um par de ESP32 em conjunto com o transmissor e receptor de Rádio Frequência denominado RFM69HW. Contudo, a implementação física desse sistema não foi obtida em sua totalidade até a realização deste documento, na qual somente as discussões e informações sobre o sistema serão apresentadas.

2 OBJETIVOS GERAIS

Este trabalho visa realizar um estudo das principais características necessárias para se realizar a recepção de dados de áudio através do protocolo Bluetooth e assim criar um protótipo aplicando os conhecimentos obtidos. Em seguida, é proposto um sistema para realizar a transmissão e recepção de um dado (bit) através de um transceptor RF, buscando estuda-lo para que, caso atenda as especificações, seja aplicado na transmissão e recepção de áudio sem fio juntamente com protótipo que contém a transmissão via Bluetooth.

2.1 OBJETIVOS ESPECÍFICOS

Os tópicos a seguir descrevem com mais detalhes os assuntos estudados nesse trabalho e aplicações que serão desenvolvidas:

- Implementar um dispositivo com o propósito de realizar a recepção de áudio por protocolos sem fios (*Wireless*);
- Desenvolver um protótipo utilizando o ESP-32 em conjunto com o DAC PCM5102, que seja capaz de se conectar com um dispositivo que possua conectividade Bluetooth(celular, notebook ...), receber o áudio digital transmitidos por esses dispositivos e em seguida reproduzi-los em uma saída TRS 3.5mm, compatível com fones de ouvido e caixas de som;
- Desenvolver um protótipo capaz de realizar a transmissão de um bit de dado, para então comprovar comportamento de transmissão RF do chip RFM69HW;
- Propor um sistema capaz de realizar a amostragem de um sinal através do conector TRS 3.5mm ou de uma entrada USB e transmiti-lo através do chip RFM69HW;
- Apresentar o desenho de dois circuitos impressos, o primeiro com um sistema focado em realizar a amostragem do sinal de áudio e transmiti-lo. Já o segundo, focado na recepção de áudio via RF e Bluetooth e convertê-lo para reproduzir no fone.

3 JUSTIFICATIVA

Em 2020, como um dos efeitos da pandemia do Covid-19, muitas pessoas passaram a trabalhar de casa. Com isso, as industrias de tecnologia (Computadores e periféricos) cresceu expressivamente (MELLO, 2020).

Além disso, alguns fones de ouvido sem fio começaram a ser produzidos nos últimos anos. Contudo muitas pessoas ainda preferem fones com fios. Ou então, investiram um quantia considerável em um fone de qualidade com fio. Outro ponto importante a ser considerado seria o de que um fone sem fio, com a mesma qualidade, custaria três vezes mais.

E é nesse ponto que os dispositivos propostos entram para poder proporcionar mobilidade e a utilização dos fones que o usuário já possui, sem todos os empecilhos que um fio traz. Portanto, as principais justificativas para o desenvolvimento do projeto se mostram as seguintes:

- Fones com fios possuem uma vida útil reduzida e tendem a apresentar falhas de contato ou quebra dos fios;
- Aplicar conhecimentos de transmissão de dados e sistemas embarcados. Esta área está em crescente expansão e também possuí um possível mercado aquecido e em busca de novos dispositivos móveis;
- Trabalhar com microcontroladores diferentes dos utilizadas em sala de aula;
- Entre outros como: desenvolvimento tecnológico, mobilidade, e ter a possibilidade de trabalhar com dispositivos capazes de operar futuramente com novas funções e aplicações.

4 DISPOSITIVO BLUETOOTH

Neste capitulo, estão descritos todos os materiais utilizados na construção do protótipo do receptor Bluetooth. Além disso, após a apresentação dos componentes, tem-se os métodos utilizados durante a programação e teste.

4.1 MATERIAIS UTILIZADOS

Primeiramente, neste capítulo, será apresentados os materiais utilizados e suas particularidades.

- Microcontrolador ESP32;
- Protocolo de comunicação Bluetooth;
- Protocolo de transmissão de áudio A2DP;
- Protocolo de transmissão de áudio I2S;
- Conversor Digital Analógico (DAC);
- Bateria (Célula);
- Conector TRS 3,5mm;

4.1.1 ESP32

No mundo contemporâneo, vivemos sempre conectados a aparelhos eletrônicos, seja ele o *smartphone* que carregamos no bolso, computador que utilizamos para trabalhar, as televisões nos entretêm e fones sem fio para ouvirmos nossas musicas favoritas. Essa automação que vivemos atualmente vem do avançado manuseio com os semicondutores e microcontroladores.

Um dos *chipsets* produzidos pela Espressif é o popular ESP32 (ESPRESSIF, 2020), um microprocessador SoC *(System on Chip)* que possui dois núcleos Xtensa® 32-bit LX6 operando em 240MHz, com 520kB de SRAM, e 34 GPIO (Pinos de entrada ou saída), quatro canais de SPI, dois canais I²S e I²C, dois DAC (Conversor Digital-Analógico) de oito bits, 18 canais de 12 bits SAR ADC (Conversor Analógico-Digital), dez sensores *touch* (toque), três canais de UART, saída PWM, Comunicações sem fio Bluetooth 4.2 e WIFI 802.11 b/g/n 150Mbps. O ESP32 é recomendado para aplicações que trabalha com aplicações de IoT ou precisam de processamento de dados tais como *Headphones* (Fones de Ouvidos), *Smart Ligths* (Lâmpadas Inteligentes), robótica industrial, *Smart watches* (Relógios inteligentes) entre outros.

4.1.2 PLACAS E MÓDULOS DE DESENVOLVIMENTO DO ESP32

Por ser uma opção barata e versátil, o chip ESP32 é utilizado por várias empresas para a criação de placas de desenvolvimento tanto para engenheiros quanto entusiastas. A própria Espressif conta com uma gama de placas de desenvolvimento e módulos contendo o básico (Chip ESP32, Cristal oscilador e *Flash*) para o seu funcionamento (ESPRESSIF, 2020).

Os módulos ESP32-WROOM-32, ESP32-WROOM-32D e ESP32-WROOM-32U possuem a mesma pinagem, entretanto diferem no modelo do seu microcontrolador ESP32, no qual o ESP32-WROOM-32 possui o ESP32-D0WDQ6 e os módulos ESP32-WROOM-32D e ESP32-WROOM-32U utilizam o ESP32-D0WD. A maior diferença entre os modelos citados estão em suas diferentes capacidades de armazenamento para a memória *flash*. O ESP32-D0WDQ6 possui somente 4MB de *flash*, enquanto o ESP32-D0WD, além de poder operar com 4MB, tem versão com 8MB ou 16MB dependendo do modelo escolhido (ESPRESSIF, 2020).

No caso do ESP32-WROOM-32D e ESP32-WROOM-32U, eles são praticamente iguais, diferindo somente na antena e no tamanho consequentemente. O ESP32-WROOM-32D apresenta uma antena interna no PCB, já no caso do ESP32-WROOM-32U, a antena no PCB é retirada, diminuindo seu tamanho e utilizando um conector para uma antena externa IPEX.

A placa de desenvolvimento ESP32 DevKitC V4 utiliza o módulo ESP32-WROOM-32 como pode ser observado na Figura 1. Essa placa possui o conversor USB serial CP2102N, responsável pela transmissão do *firmware* do computador para o micro, um LED que é ativado quando a placa é alimentada pelo USB ou por uma fonte de energia externa, pinos de saída para facilitar os testes com a placa e botões nas portas IO0 e EN para o download do *firmware* e *reset* respectivamente.

Figura 1 – Placas e Módulos da Espressif

Fonte: ESPRESSIF, 2020

A Shenzhen Xin Yuan Electronic Technology Co.Ltd é uma das empresas focadas na produção de placas de desenvolvimento utilizando o chip ESP32. Sua serie de placas chamada TTGO abrange várias áreas da eletrônica, unindo sensores para o desenvolvimento de hardware e software para os chips da Espressif. Placas da serie TTGO T-Camera (LILYGO, 2020), uma placa de desenvolvimento que já possuí uma câmera integrada, TTGO LCD OLED (LILYGO, 2020), placas com um *display* junto ao modulo e TTGO LORA (LILYGO, 2020), com capacidades de transmissão de dados a longa distância são exemplos de placas de desenvolvimento criadas pela Shenzhen.

Para o nosso protótipo, foi escolhido a placa TTGO T-Display como a placa de desenvolvimento, por conta de suas GPIO, conector USB-C, apresentar um *display* com preço acessível. Como visto na figura 2.

Figura 2 – Placas TTGO

Fonte: (LILYGO, 2020)

4.1.3 BLUETOOTH

No ano de 1998, cinco empresas em conjunto, criaram o Bluetooth, um protocolo de comunicação que visa unificar a conexão entre dispositivos (BLUETOOTH, 2020a). Operando em uma frequência de 2.4 GHz (ISM), muito usada na indústria em outros protocolos como Wi-Fi e ZigBee (ZIGBEE, 2020), pelo seu alcance e confiabilidade. Atualmente trabalhando com mais de 34 mil empresas ao redor do mundo, a tecnologia sem fio vem ganhando mais espaço e sendo atribuída à aparelhos do dia-a-dia como celulares, caixas de som, computadores entre outros eletroeletrônicos.

Cada versão da plataforma atualiza e acrescenta mais melhorias para a transferência de arquivos sem fio. A versão 4.0 trouxe consigo as funcionalidades *Low Energy* (LE) para a otimização do uso de energia dos dispositivos e a versão 5.0 aprimorou o LE para operar em longos alcances (BLUETOOTH, 2020c). Atualmente o Bluetooh está em sua versão 5.2 (BLUETOOTH, 2020b).

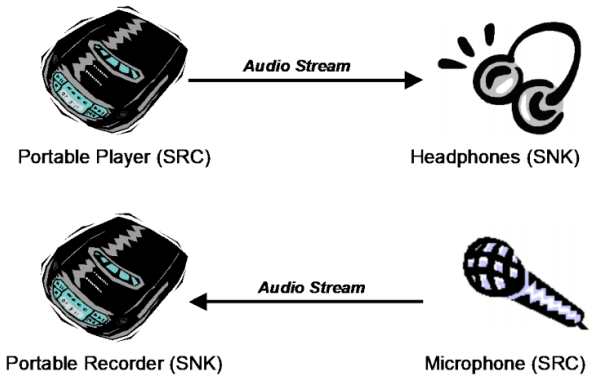
Ao unificar a comunicação sem fio em único protocolo, o *Bluetooth Special Interest Group* (SIG) criou perfis para facilitar o desenvolvimento de áreas especificas utilizando a comunicação Bluetooth. Perfis como *Advanced Audio Distribution Profile* A2DP para comunicação

Bluetooth de áudio, *Audio Video Remote Control Profile* AVRCP, encarregado de transferir comandos de controle de áudio e vídeo entre dispositivos (*pause*, *play*,*volume* +-), *File Transfer Profile* (FTP) usado em aplicações de transferência de arquivos, são alguns exemplos.

4.1.4 A2DP - ADVANCED AUDIO DISTRIBUTION PROFILE

O perfil A2DP - *Advanced Audio Distribution Profile* (Perfil de Distribuição avançado de áudio) é um dos perfis do protocolo Bluetooth especializado na transmissão de áudio com alta qualidade, seja ele em mono ou estéreo (um ou dois canais de áudio), com ênfase no *streaming* (transmissão em tempo real) de áudio para fones ou caixas de som (BLUETOOTH, 2019) como podemos ver na figura 3.

Figura 3 – Exemplo de comunicação A2DP



Fonte: (BLUETOOTH, 2019)

Para estabelecer uma transmissão utilizando o protocolo A2DP, é necessário ter dois dispositivos com funções diferentes, um deles sendo o *Source* (SRC) e o outro será o *Sink* (SNK). O SRC será o dispositivo que transmitirá o sinal digital de áudio, já o SNK será o dispositivo que receberá o sinal de áudio digital entregado pelo SRC. A escolha da função que cada dispositivo irá realizar depende da aplicação de cada projeto desenvolvido. Neste trabalho, para efeito de comparação, será utilizado um dispositivo reprodutor de áudio como um computador ou celular com capacidade Bluetooth como SRC, e o ESP32 como SNK, recebendo áudio digital para ser convertido em um sinal analógico.

4.1.5 A2DP NO ESP32

Para utilizar o perfil A2DP no ESP32, primeiramente é necessário ativar as funcionalidades de *Classic Bluetooth* e *Bluedroid* presentes no arquivo *sdkconfig* do código. Os comandos para utilizar as funcionalidades do perfil A2DP estão na biblioteca disponibilizada pela EspressIf, fabricante do *chipset* ESP32 (ESPRESSIF, 2020).

O sinal de áudio recebido pela transmissão Bluetooth pode ser convertido pelo ESP32 em dois tipos de sinais, um sinal de áudio gerado pelo DAC interno de 8 bits de resolução ou via protocolo I²S para um DAC externo.

4.1.6 I²S - INTER IC SOUND

Para realizar interface de comunicação entre o microcontrolador e o conversor digital, será utilizado o I²S, *Inter-IC Sound*, um protocolo desenvolvido para atuar com o *streaming* de dados, sendo a transmissão de áudio sua principal aplicação (NXP, 2020).

O protocolo utiliza três fios para sua comunicação, um fio é responsável pela transmissão de dados, outro pelo sinal de sincronismo *Clock* e o último seleciona qual lado do canal do fone será escrito o dado *Channel Select*. São eles:

- Serial Data;
- Clock Signal;
- Channel Select;

Os dados de áudio que serão transmitidos pelo *Serial Data*, são de 16 bits. O *Channel Select* indicará para qual canal o dado será transmitido, seja ele o lado esquerdo ou direito atribuindo nível lógico baixo ou alto respectivamente.

Ao estabelecer o protocolo é necessário que cada dispositivo de transmissão ou recepção tenha um perfil de operação, sendo ele mestre (*master*) ou escravo (*slave*). O fluxo de dados é definido segundo o perfil, caso o transmissor seja escolhido como mestre, os dados partirão dele para o receptor, caso o transmissor seja definido como escravo, os sinais de *Clock* e *Channel Select* serão gerados pelo receptor e o *Serial Data* pelo transmissor como pode ser visto na figura 4 e 5.

Figura 4 – Exemplo de comunicação I2S - Transmissor Mestre

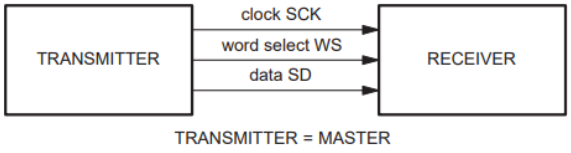
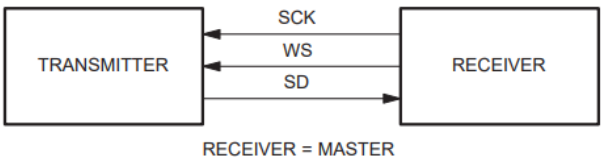


Figura 5 – Exemplo de comunicação I2S - Receptor Mestre



Fonte: (NXP, 2020)

4.1.7 CONVERSÃO DIGITAL ANALÓGICA (DAC)

A conversão dos dados recebidos através do protocolo Bluetooth A2DP em um sinal elétrico é feita através de um circuito integrado DAC. Esse chip recebe os dados digitais pelo protocolo I²S e transforma esse sinal em analógico. O livro *The Audio Programming Book* (BOULANGER; LAZZARINI, 2011), descreve que áudios digitais possuem tipicamente 16 bits de resolução, e visando atingir essa resolução ou uma maior, foram pesquisados DAC que atendem ou superem essa característica.

A Texas Instruments é uma das muitas empresas que fabricam, em larga escala, circuitos integrados, sensores e microcontroladores. Sua área de conversores DAC é repleta de CIs, como por exemplo, o PCM1795 e as famílias PCM510xA e PCM270xC. A tabela 1 a seguir mostra uma comparação entre três desses CIs que podem atender as necessidades desse projeto.

Tabela 1 – Conversores Digital-Analógico (DAC).

PCM270xC	**PCM1795**	**PCM510xA**
• Interface USB compatível com USB 2.0, podendo ser utilizado sem programação; • 32kHZ, 44.1kHz, 48kHZ de Sample rate; • Gerador de clock interno; • Resolução de 16bits; • 5V bus power e 3.3V self power; • Configurado por SPI;	• 10kHz até 200kHz de Sample rate; • Necessita de um Clock na entrada SCK; • Resolução de 16, 24 ou 32bits; • 5V Analógico e 3.3V Digital; • Configurado por SPI ou I2C;	• 8kHz até 384kHz de Sample rate; • Utiliza o sinal de clock do I2S para gerar o system clock; • Resolução de 16, 24 ou 32bits; • 3.3V Analógico e 1.8 ou 3.3V Digital; • Configurado pelos pinos no hardware;

Fonte: Autoria Própria (2020)

4.1.8 BATERIA (CÉLULA)

Um dos principais pontos necessários quando desenvolve-se dispositivos móveis, é o dimensionamento de uma boa bateria (chamada de célula). Neste ponto, a célula deve ser capaz de atender os parâmetros de alimentação do circuito, além de ter um número considerável de quantidade de ciclos de carga e descarga, garantindo uma vida prolongada do aparelho a ser desenvolvido.

Estas células podem ser dos tipos recarregáveis ou de uso único, como as pilhas comuns. Contudo, visto que o dispositivo desenvolvido é focado na mobilidade e praticidade, será dimensionada uma célula recarregável.

Com essa característica em mente, será necessário saber o consumo médio do dispositivo, pois, com isso, tem-se o tempo de operação até o esgotamento da carga gerada pela célula,

sendo necessário a recarga da mesma.

Primeiramente, a célula deverá ter a tensão máxima de 4,2V e a mínima tensão de trabalho será de 3,2V. A sua capacidade de carga será de no mínimo 1000mA/h, ou seja, se o circuito consumir uma corrente de 200 mA, a célula conseguirá manter a alimentação dentro de um intervalo de máximo e mínimo por pelo menos cinco horas.

Outra característica importante que é levado em conta, é relacionada as dimensões físicas, pois quanto maior o formato do encapsulamento da célula mais carga ela pode armazenar, contudo deve manter a relação tamanho/peso/potência capaz de ser instalada dentro do protótipo para manter seu propósito de mobilidade.

Após esses pontos apresentados, outra característica extremamente relevante é o tipo de química utilizada na construção das células. Algumas apresentam certas vantagens em determinados pontos, como, por exemplo, quantidade de ciclos maiores, já outras químicas apresentam uma relação potência/peso maior entre outras individualidades. As células mais comuns e analisadas para a execução desse trabalho foram as de Niquel-Cádmio (NiCd), Hidreto Metálico/Óxido de Níquel (NiOH), Lithium-Polímero(LiPo), Lithium-Ion(Li-ion) entre outras.

As baterias de NiCd, vem sendo substituídas por apresentarem muitos impactos negativos ao meio ambiente. As células de NiOH apresentam um efeito memória elevado, conhecido como vício de bateria ou seja, perdem a capacidade de manter sua carga e saúde com um número menor de ciclos. As baterias de LiPo, apresentam boa relação de peso-potência, contudo correm o risco de explodirem ou incendiarem se não utilizado um carregador especial que garante o balanceamento da carga em sua carga e descarga. A química das células de Li-Ion a princípio, atendem os requisitos. Portanto foram analisadas suas características mais relevantes e apresentadas compiladas na tabela 2 abaixo:

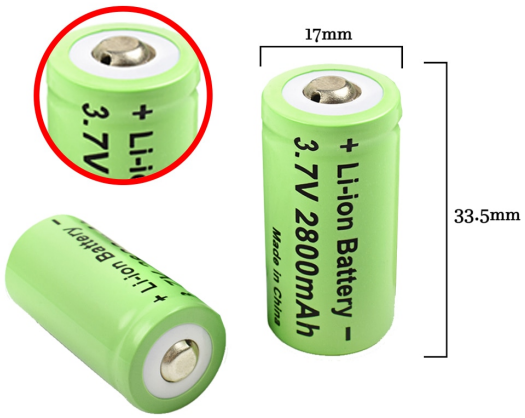
Tabela 2 – Células para baterias.

	CR123A Recarregável	**CR123A Não-Recarregável**	**NCR-18650B**
Fabricante	Bonacell	Panasonic	Panasonic
Tensão	3.7 V	3 V	3.6 V
Capacidade	2800 mAh	1550 mAh	3350 mAh
Ciclo	500	0	500
Peso	17 g	17 g	47.5 g
Diâmetro	17 mm	17 mm	18.5 mm
Altura	33.5 mm	34.2 mm	65.3 mm

Fonte: Autoria Própria (2020)

Com essas características apresentadas, as células de *Lithium-Ion* ficaram com a preferência, pois esse tipo de célula possuem mais de 500 ciclos de carga e recarga, e também tem dimensões que atenderão o dispositivo com tamanho reduzido. Assim, dentro da gama de células disponíveis com essa química foi selecionado a célula CR123A que é recarregável, de dimensão 16340, e que possui capacidade de 2800mA/h, como pode ser visto na figura 6.

Figura 6 – Dimensões da célula, CR123A, número 16340.



Fonte: Aliexpress, 2021

Considerou-se também que o ESP32-TTGO possuí um circuito carregador de bateria que atende os requisitos para recarregar esse tipo de célula de Li-Ion, como visto na secção sobre o ESP32, não havendo a necessidade de inserir um circuito de carga externo.

Outro ponto importante que aumenta a qualidade dispositivo, dá-se também pelo fato de que a própria célula fornece proteções contra diversos casos de uso como visto na figura 7. São elas:

- Proteção contra curto-circuito;
- Proteção contra Sobrecarga;
- Proteção contra sobrecorrente;
- Proteção para descargas elevadas;
- Certificada para União Europeia (CE) e Américas (FCC);

Figura 7 – Proteções da célula, CR123A, número 16340.

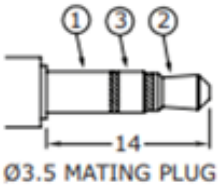
Fonte: Bonacell, 2021

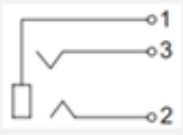
4.1.9 CONECTOR TRS 3,5 MM

Segundo (ANDROID, 2020), a conexão de fone de ouvido de 3.5mm pode ser estérea (canal esquerdo e direito), possuindo 4 conexões, sendo elas canal esquerdo, canal direito, o ground e microfone, respetivamente da ponta até o cabo do conector, seguindo o padrão CTIA. No caso desse trabalho, será utilizado um conector com três saídas, o TRS de 3.5 mm, onde o mesmo não possui um conector para microfone.

A Figura 8 a seguir foi retirada do *datasheet* do conector TRS 3.5 mm fêmea SJ-3523-SMT e alterada para relacionar a conexão elétrica do conector com o padrão estabelecido pela Android. Esse modelo segue as especificações encontradas no módulo GY-PCM5102, e mostra um conector TRS macho relacionando as conexões entre o conector fêmea mostrado na tabela.

Figura 8 – Esquemático do conector SJ-3523-SMT.



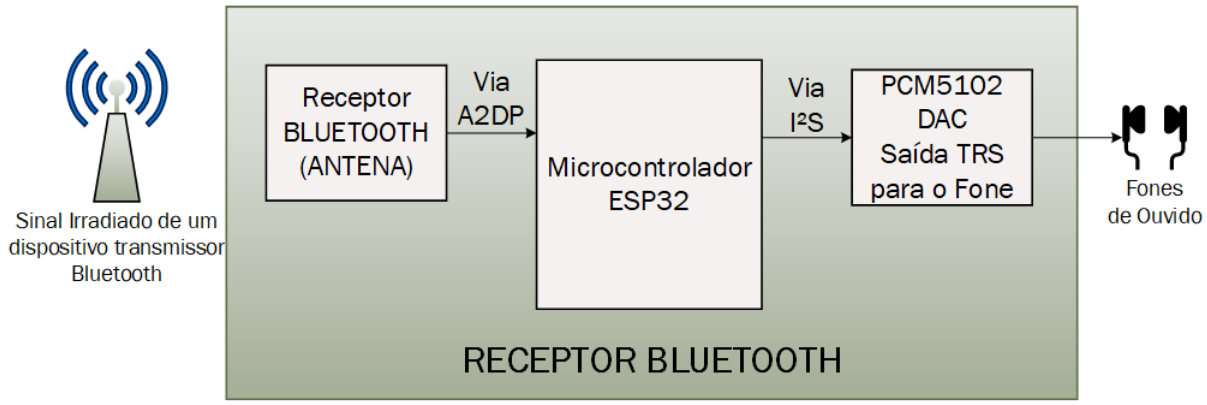
Model No.	SJ-3523-SMT
Schematic	
PIN	
1	Sleeve/Ground
2	Tip/Esquerdo
3	Ring/Direito

Fonte: (CUIDEVICES, 2021)

4.2 TOPOLOGIA DO RECEPTOR BLUETOOTH

Com os dados obtidos após a realização da pesquisa sobre o microcontrolador ESP32, é possível criar um fluxograma que descreve os conceitos do protótipo do RECEPTOR BLUETOOTH que consiga receber áudio em tempo real através do protocolo Bluetooth como é indicado na figura 9.

Figura 9 – Fluxograma do Protótipo para Receptor Bluetooth



Fonte: Autoria Própria, 2020

Partindo do fluxograma acima, foram utilizados duas configurações com ESP32 para o desenvolvimento do protótipo.

A primeira configuração foi feita com a interface de desenvolvimento TTGO T-DISPLAY ESP32 da LILYGO, pois o mesmo seria capaz de atender os requisitos do receptor, possuindo dois núcleos para o processamento do sinal, uma memória *flash* de 4MB, uma antena no próprio PCB, regulador de 5V para 3.3V possibilitando a alimentação via USB e um circuito recarregador com conector para baterias.

A segunda configuração é constituída por um ESP32-WROOM-32D, que compõe apenas o módulo principal do microcontrolador ESP32, sem os subsistemas que o TTGO possui, contudo em sua construção, o modelo escolhido vem de fábrica com uma memória *flash* maior, de 16MB.

Para ambos microcontroladores, existe a necessidade de reproduzir o sinal digital de forma analógica para tocá-los no fone de ouvido. Isto foi realizado buscando atender pelo menos 16 bits de resolução para o sinal de áudio, como mencionado no livro *The Audio Programming Book*, onde não seria possível utilizar o modulo DAC interno do ESP32, por possuir apenas 8 bits de resolução. Sendo assim, o PCM5102 foi escolhido, por possuir entrada I²S, saída analógica para um conector TRS 3.5 mm e a simplicidade de poder utilizar o sinal de sincronia do I²S como *clock* interno, diminuindo um fio na conexão com o CI. Esse conversor digital-analógico foi escolhido como responsável em entregar esse áudio em boa qualidade ao fone de ouvido ou

caixa de som. Para realização dos teste, foi utilizado um módulo DAC chamado GY-PCM5102, no qual seria possível testar as funcionalidades do CI PCM5102 já integrado em uma placa.

4.3 PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO DO SOFTWARE

O ESP32 possui várias maneiras para se implementar o código *firmware* em sua memória, em uma delas é possível utilizar o software ESP-IDF (*Espressif IoT Development Framework*) provido pela Espressif, outro método é pela IDE do Arduino, e o terceiro método seria uma extensão chamada PlataformIO IDE dentro do Visual Studio Code (VSCode) (Editor de texto, propriedade da Microsoft).

O ESP-IDF providência ao usuário um maior controle sobre os registradores presentes no chip em desenvolvimento. Entretanto, ele é somente a ponte entre o código e o chip, implementando o código já criado. Para criar e editar o firmware, é necessário um editor de texto como o Visual Studio Code.

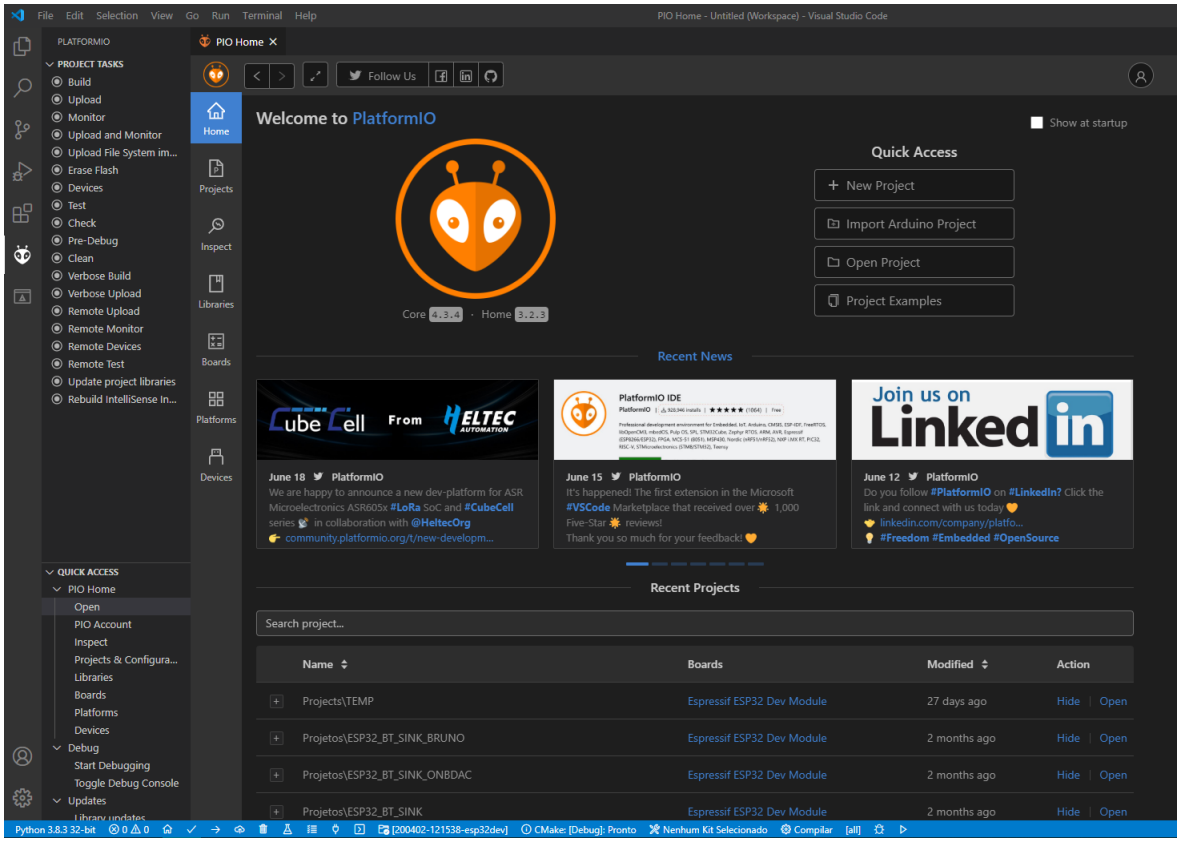
Outra possibilidade, a da IDE do Arduino, como visto na figura 10, com bibliotecas disponibilizadas por usuários para a criação e desenvolvimento do *firmware*. Esse método traz ao usuário o mesmo tipo de linguagem utilizada para programar Arduino, possuindo um editor de texto na mesma IDE, entretanto o mesmo possui um controle menor de seus registradores.

Figura 10 – Arduino IDE

Fonte: Arduíno IDE (2020)

Criado em 2014, o PlataformIO é um extensão do VSCode capaz de combinar vários tipos de *framework*, no mesmo ambiente para facilitar a programação para desenvolvedores (PLATFORMIO, 2020). Chips de empresas como a Texas Instruments, Microchip e Espressif são suportados no PlataformIO. Com uma interface simplificada e possuindo a opção de poder utilizar tanto as bibliotecas do Arduino quanto a do ESP-IDF, o PlatformIO IDE é uma ferramenta que se obtém mais versatilidade para o desenvolvimento um firmware para o ESP32. O software pode ser visto na figura 11

Figura 11 – PlatformIO

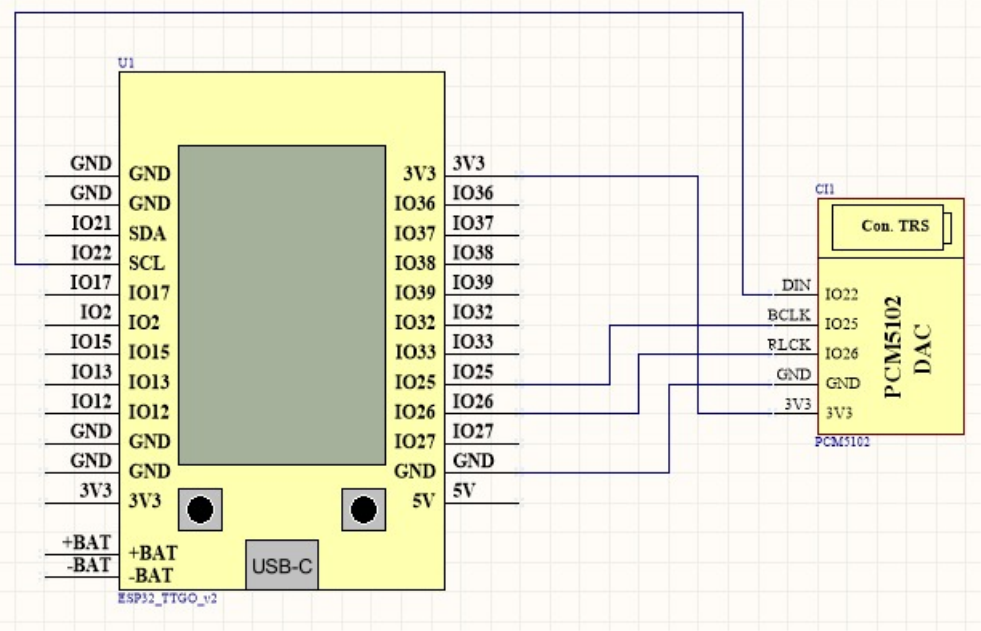


Fonte: PlatformIO Tela de Home (2020)

4.4 DESENVOLVIMENTO DO PROTÓTIPO BLUETOOTH

Para a construção do protótipo, foi inicialmente proposto o circuito contendo as placa de desenvolvimento TTGO T-Display ESP32 em ligação com o módulo DAC (GY-PCM5102) mostrado na figura 12.

Figura 12 – Circuito do Protótipo BLUETOOTH

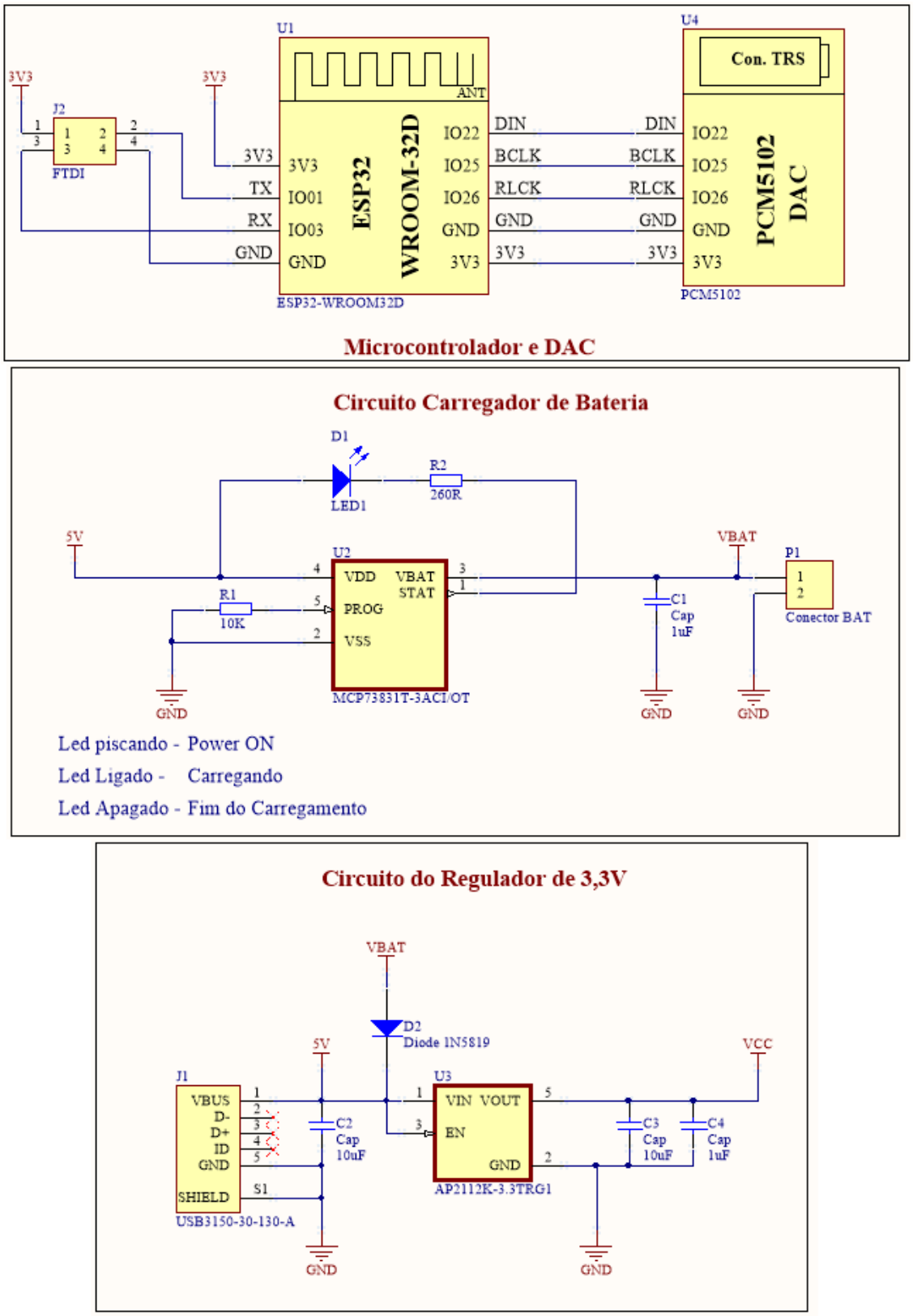


Fonte: Autoria Própria, 2020

Além de utilizar o TTGO, outra opção de desenvolvimento foi a montagem do receptor com o ESP32-WROOM-32D.

A figura 13 a seguir, é a versão da placa do protótipo, onde é apresentado o esquema elétrico com toda a ligação entre os módulos GY-PCM5102, o microcontrolador ESP32-WROOM-32D. Além disso a placa contém o circuito carregador de bateria e o circuito de regulação de tensão para 3,3V.

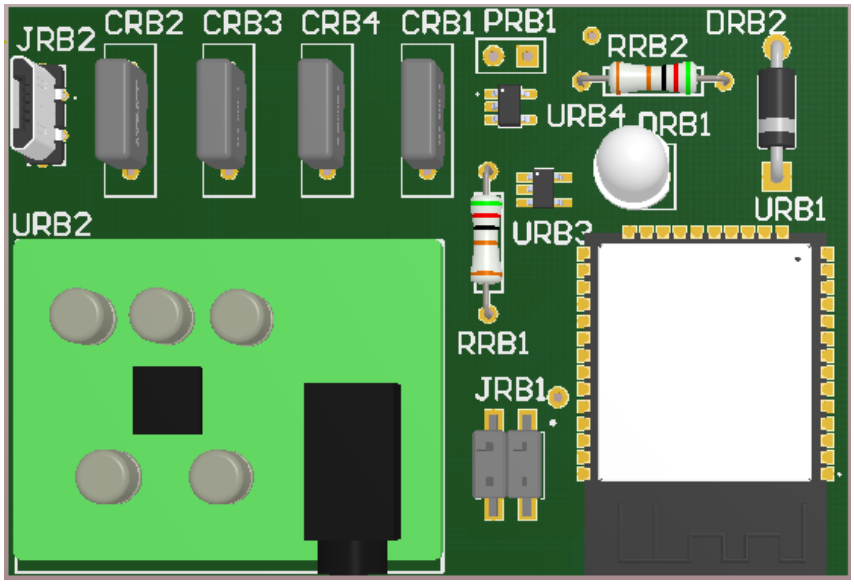
Figura 13 – Esquema Elétrico, Receptor Bluetooth.



Fonte: Autoria Própria, 2021

Com o esquemático definido, têm-se as placas de circuito impresso apresentado na figura 14 onde constitui o Receptor Bluetooth. E devido a pandemia, o protótipo foi montado apenas na matriz de contatos.

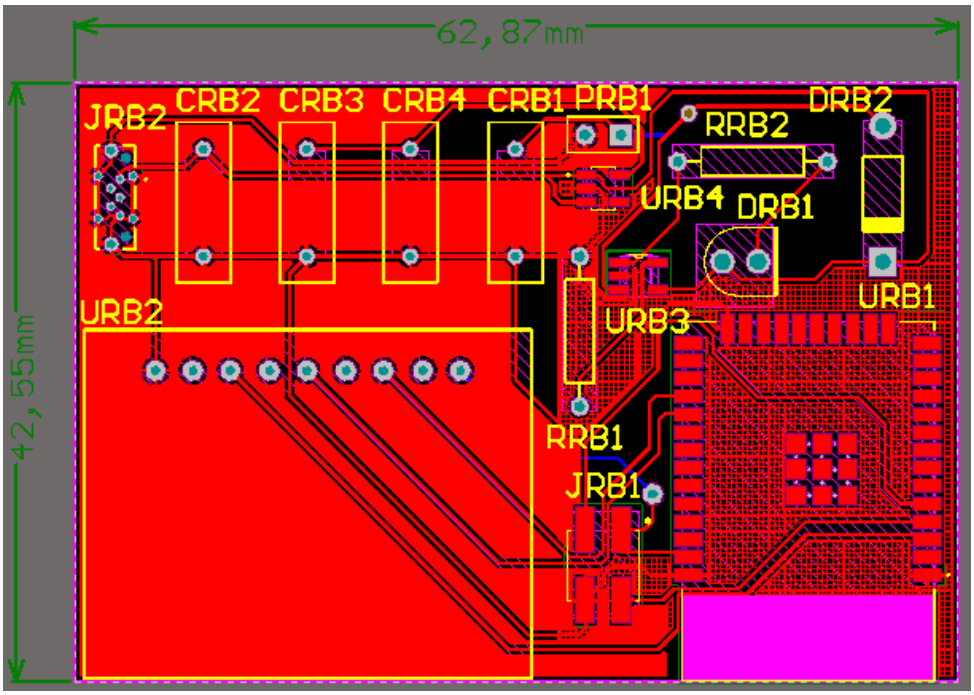
Figura 14 – Placa do Receptor Bluetooth.



Fonte: Autoria Própria, 2021

Além disso, a placa com as trilhas de ligação do circuito podem ser vistas na figura 15 abaixo.

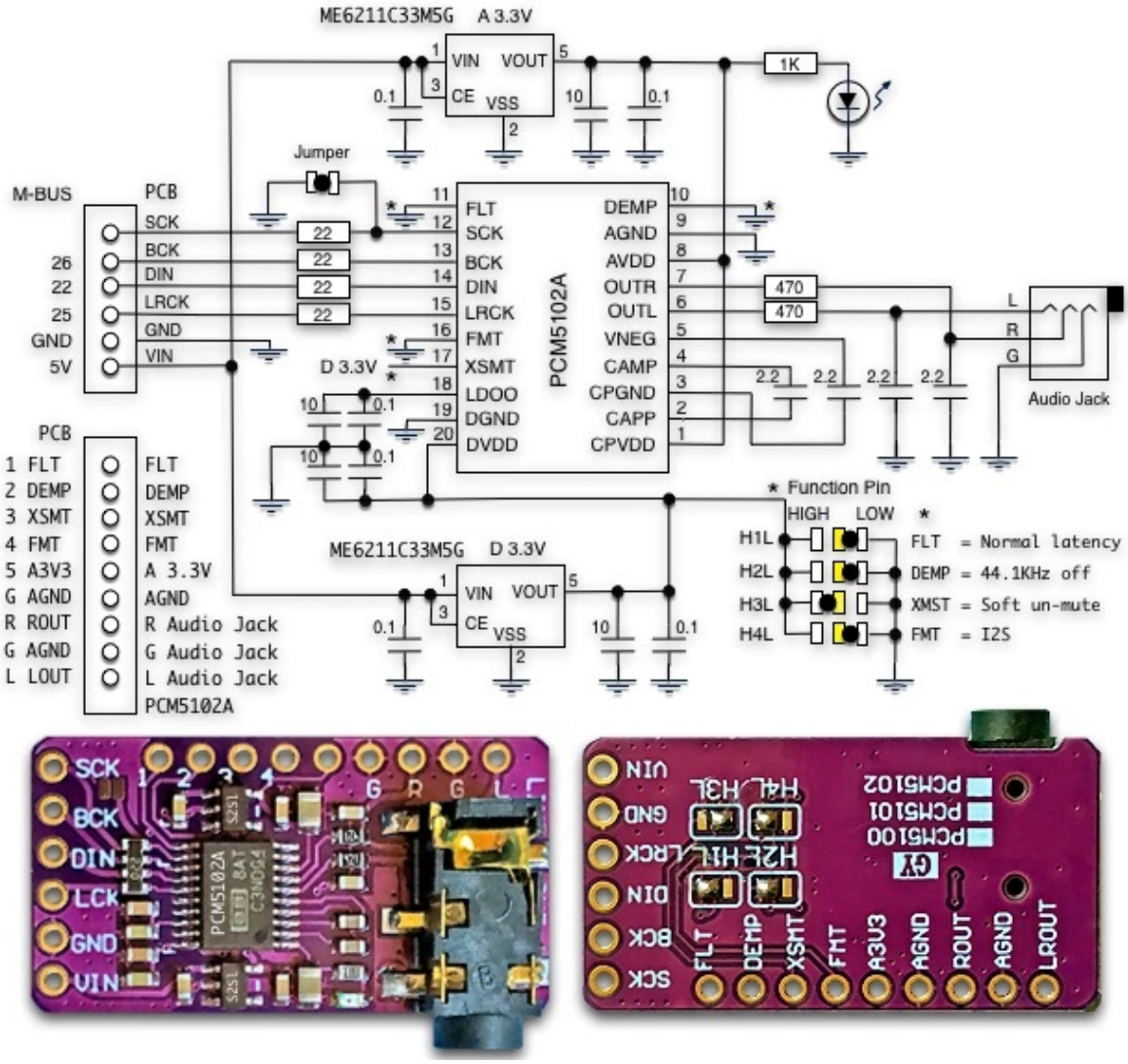
Figura 15 – Circuito do Receptor Bluetooth.



Fonte: Autoria Própria, 2021

Para poder operar corretamente, módulo DAC GY-PCM5102 foi configurado para operar com latência normal, 44.1 kHz, *soft un-mute* e com protocolo I²S para transmissão de dados, proposto por (MACSBUG, 2021). O pino SCK do módulo foi ligado ao *ground* para que o PCM5102 utilize o sinal de *clock* do pino da comunicação I²S como sinal *clock* interno. A conexão com o *ground* pode ser feita pela soldagem de dois *pads* próximos a entrada SCK no módulo GY-PCM5102. A figura 16 mostra o circuito elétrico proposto por MACSBUG.

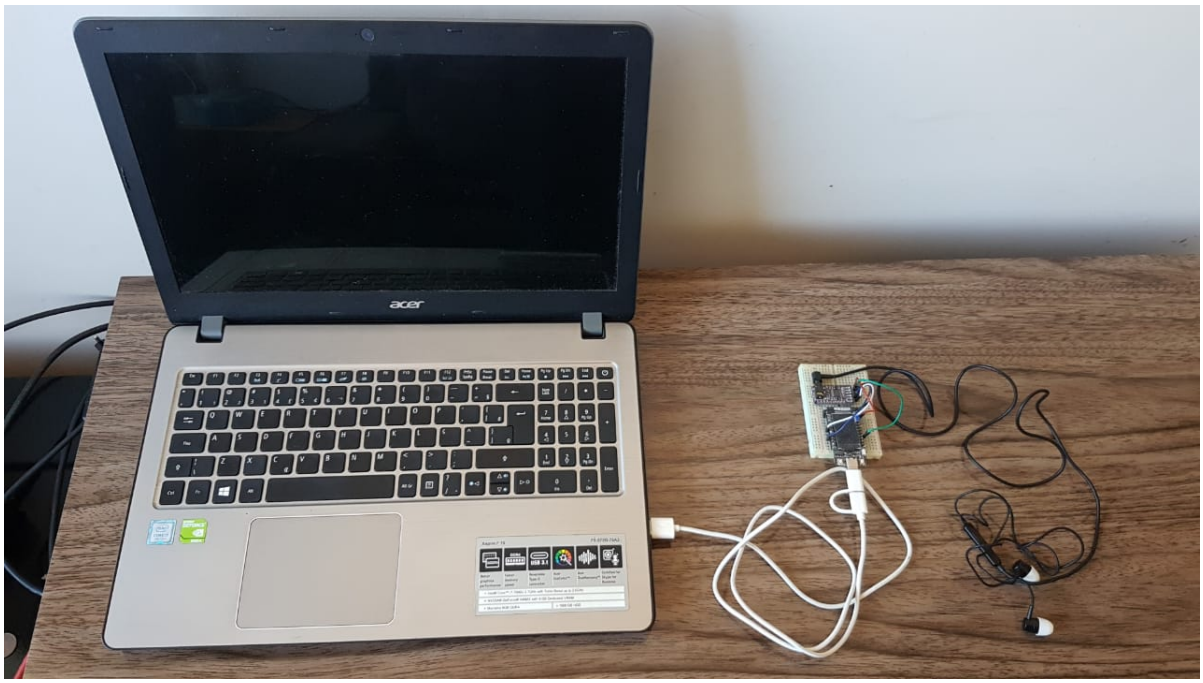
Figura 16 – Configuração módulo GY-PCM 5102



Fonte: (MACSBUG, 2021)

Para a montagem do setup de teste, a placa de desenvolvimento TTGO é alimentado com 5V através da entrada USB-C conectada em um notebook, sendo convertida internamente de 5V para 3.3V para abastecer o ESP32. Além disso a entrada USB-C também é utilizada para inserir o *firmware* no ESP32. O GY-PCM5102 possui um conector TRS 3.5mm no qual um fone de ouvido foi inserido. As figuras 17 e 18 mostram a implementação do ESP32 para o teste do protótipo com TTGO.

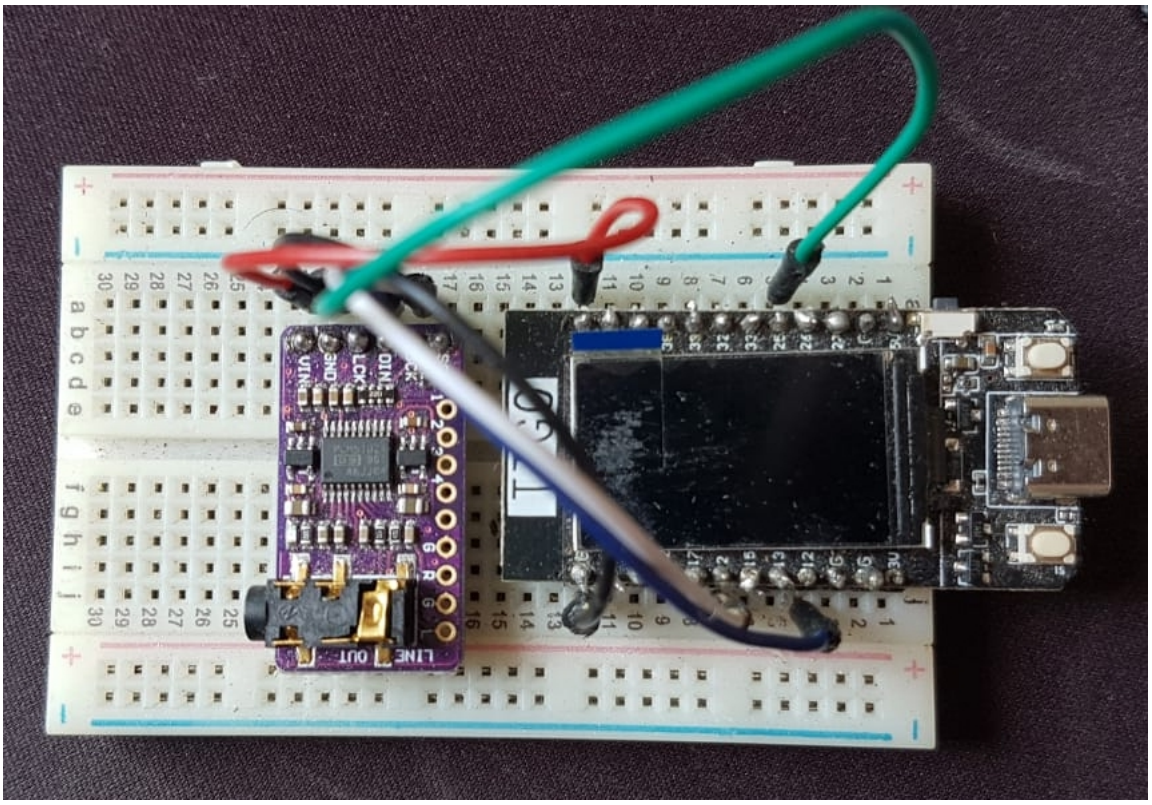
Figura 17 – Implementação para o teste do protótipo BLUETOOTH



Fonte: Autoria Própria, 2020

A realização do primeiro teste com o ESP32 TTGO, foi executado utilizando um celular Samsung Galaxy S7 possuindo sistema operacional Android. Além desse teste também foi executado o mesmo teste com o ESP32-WROOM-32D. Já para o segundo teste, um notebook com sistema operacional Windows 10, foi utilizado para se conectar individualmente tanto com o ESP32 TTGO quanto com o ESP32 WROOM e então transmitir dados de áudio pelo protocolo A2DP para os protótipos.

Figura 18 – Protótipo BLUETOOTH



Fonte: Autoria Própria, 2020

4.5 DESENVOLVIMENTO DO SOFTWARE PARA O PROTÓTIPO BLUETOOTH

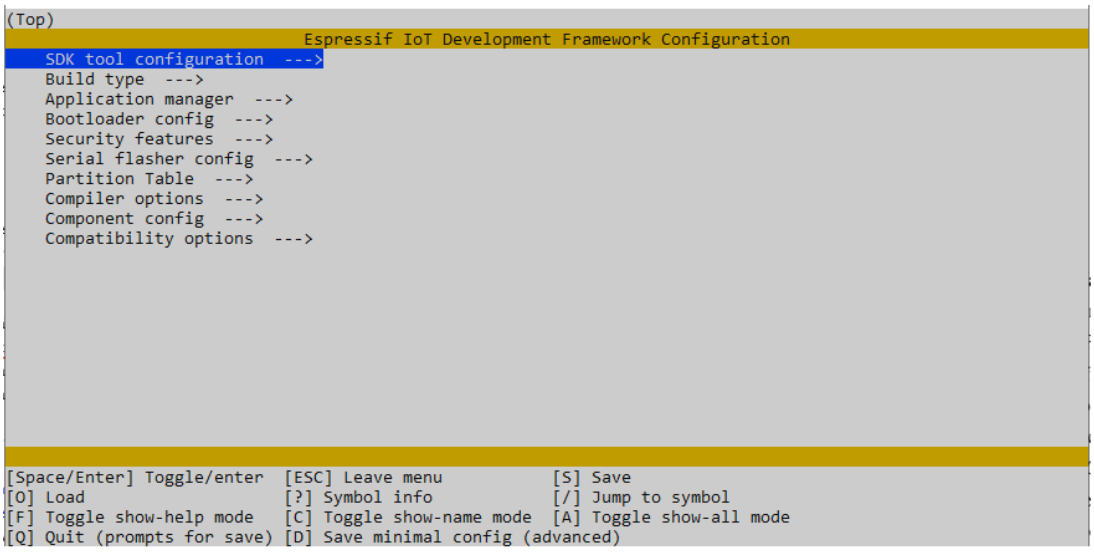
Para o desenvolvimento de um software capaz de realizar a recepção do sinal de áudio pelo protocolo Bluetooth, foi utilizado um código exemplo de recepção Bluetooth A2DP Sink proveniente da Espressif.

Esse código foi modificado para atender as especificações de projeto e interagir com o DAC para conversão digital-analógica. Inicialmente o código foi inserido no Visual Studio Code para edição e compilado pela extensão PlatformIO como visto na capítulo 4.3 de Preparação do Ambiente de Desenvolvimento de Software.

Ao compilar este *firmware*, um aviso de erro de compilação é apresentado. Essa mensagem informa que as configurações de Bluetooth não estão ativadas no ESP32.

Com isso, verificou-se que as configurações de Bluetooth não vem previamente ativadas de fábrica. Para ativá-las, será necessário acessar um Menu de Configuração do ESP32. Isso é possível utilizando o código *"platformio run -t menuconfig"* no terminal de comando, no contexto do endereço da pasta do arquivo a ser compilado. No Menu de Configuração estão localizados as configurações mais avançadas relacionadas, como *Bootloader*, *Serial Flash* e *Component Config*, onde pode ser visto na figura 19.

Figura 19 – Menu de Configuração do ESP32 aberta com o comando menuconfig

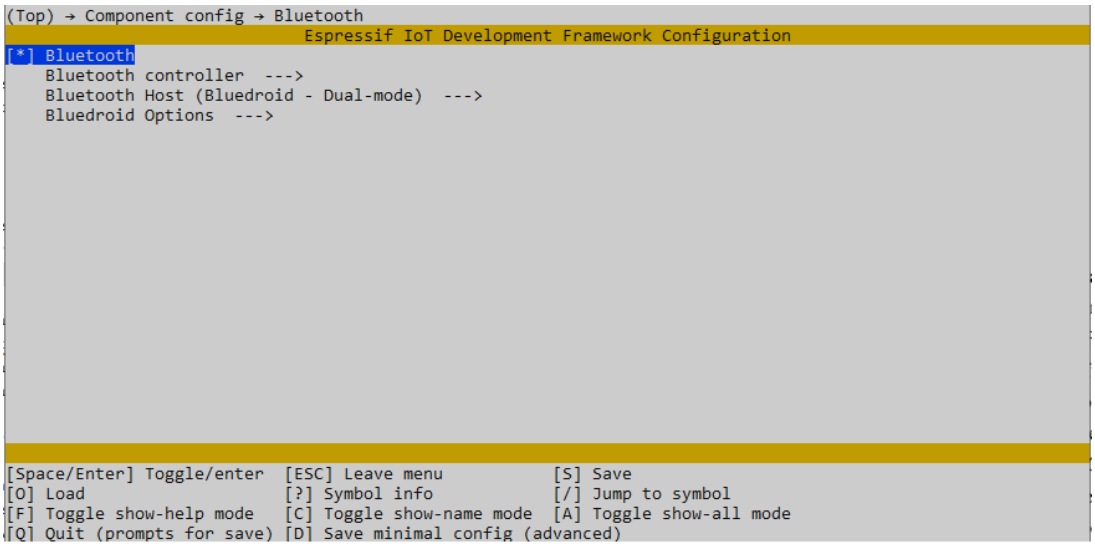


Fonte: Autoria Própria, 2021

Ao abrir o *Component Config*, existirá uma aba que redirecionará para as configurações Bluetooth, onde estará desativada de fábrica. Ao ativa-las, é aberto mais menus para as configurações, como pode ser visto na figura 20, onde nas opções de *Bluetooth Options*, deve ser ativado as funcionalidades do *Classic Bluetooth* e depois do A2DP. Em *Bluetooth Controller*, a configuração de fabrica é o *Bluetooth Controller Mode* é *BLE* e deve ser alterada para *BR/EDR Only*, pois o exemplo *A2DP* é uma aplicação de um exemplo do *Bluetooth Classic* e não da versão *BLE*, responsável por transmissões Bluetooth de baixo consumo de energia. Também

é possivel notar em *Bluetooth Controller* existe uma configuração para atribuir qual core do ESP32 irá ser responsável, no nível hardware, pelo processamento dos dados Bluetooth. Para a realização deste trabalho, o core 0 é configurado para essa função.

Figura 20 – Component Config do Menu de Configuração do ESP32



Fonte: Autoria Própria, 2021

Após ativar as configurações necessárias para o Bluetooth A2DP, será possível compilar o código exemplo da Espressif. Este *firmware* está dividido entre a parte de transmissão de dados Bluetooth e do protocolo A2DP e AVRCP. A parte A2DP do código deverá ser alterada e as outras partes serão mantidas como de fábrica.

Inicialmente, o código de fábrica começa estabelecendo as configurações Bluetooth e atribuindo uma tarefa em FreeRtos para lidar com a transferência de dados do protocolo. Essa tarefa será responsável por se comunicar com o dispositivo, trocando as informações básicas do protocolo Bluetooth. Para criação dessa tarefa em FreeRtos, foi trocado para um comando do ESP32 chamado xTaskCreatePinnedToCore, que possibilita atribuir uma tarefa a um core especifico para processamento da tarefa configurada em software, no caso, utilizamos o core 1 para realizar essa função, pois o core 0 é utilizado nas configurações de hardware do Bluetooth.

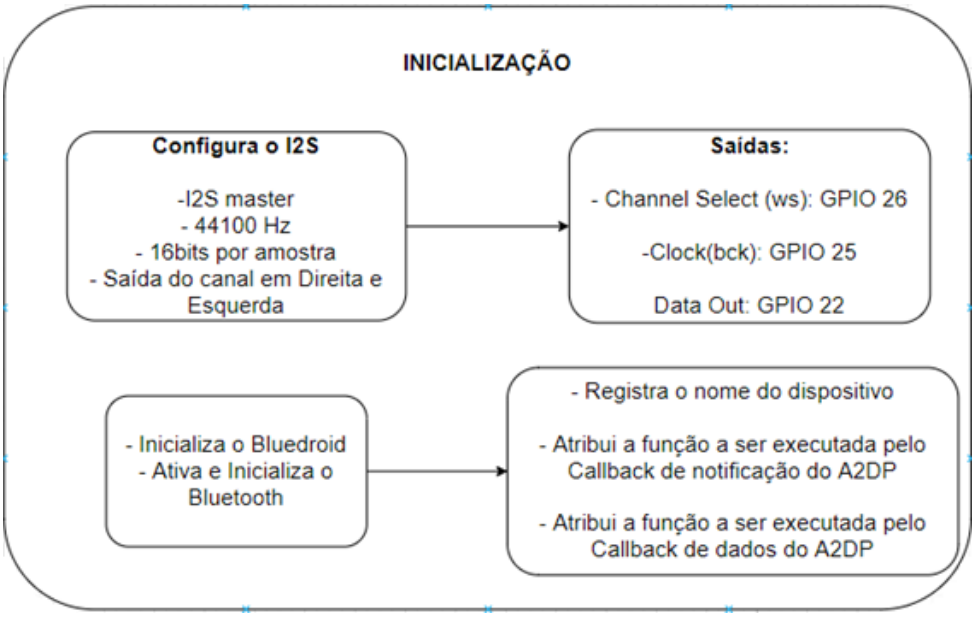
Outra tarefa, também em FreeRtos, foi criada pelo fabricante, para lidar com o recebimento de áudio através do protocolo A2DP, na qual, acrescentou-se o processamento dos dados. Essa tarefa foi criada utilizando o xTaskCreate, que no caso do ESP32, ira utilizar o core que estiver livre para processamento.

Em seguida, foram modificados as configurações do protocolo I²S, configurando o ESP32 como *MASTER* para funcionar a uma taxa de amostragem de 44.1KHz de 16 bits, modelo PCM, com amostragem de saída para os canais esquerdo e direito. Por fim, foi configurado também os pinos de saída.

Após configurado as pinagens do I²S, tem-se a inicialização com as configurações das funções de *Callback*, responsáveis por realizar uma tarefa e retornar a sequência do programa.

Para esse trabalho, uma função é atribuída como *Callback* para o tratamento de dados. Além dessa operação, também serão adicionadas funções *Callback*, sendo elas para (i) A2DP para dados e (ii) A2DP de notificação. Ou seja, serão criadas funções que realizarão o tratamento e processamento dos dados de áudio recebido por A2DP e as notificações recebidas pelo protocolo A2DP. A figura 21 mostra um fluxograma descrevendo a inicialização do código.

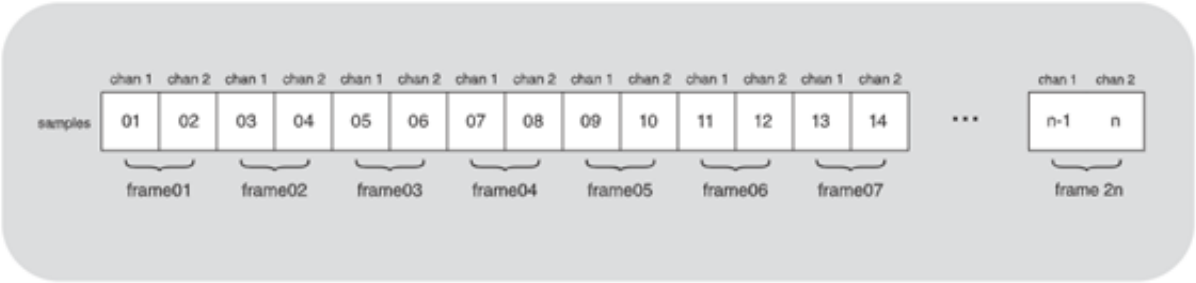
Figura 21 – Inicialização do firmware



Fonte: Autoria própria, 2021

Após a inicialização e ativação do protocolo Bluetooth, se inicia o processamento de dados. De acordo com o livro *The Audio Programming Book*, pág. 191, (BOULANGER; LAZZARINI, 2011), tipicamente, áudios digitais estão formatados em pacotes de 16 bits, sendo que esses dados são alternados entre o canal esquerdo e direito do fone respectivamente, como mostra a figura 22.

Figura 22 – Buffer de áudio

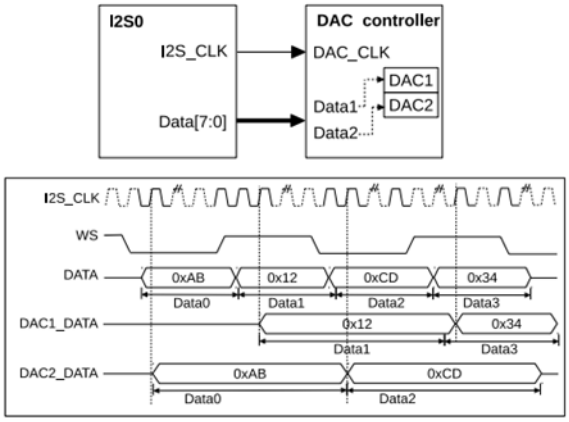


Fonte: (BOULANGER; LAZZARINI, 2011)

Analisando o código da Espressif, nota-se que os valores recebidos do *Callback* de dados proveniente da transmissão via Bluetooth, é caracterizado por um *buffer* contendo 2650 Bytes de tamanho. Portanto, pode-se concluir que os dados de 16 bits de áudio serão recebidos em dois espaços diferentes, ou seja, o primeiro receberá os MSB (bits mais significativos) e o segundo, receberá o restante LSB (bits menos significativos).

O protocolo I²S do ESP32 trabalha com a escrita primeiramente no canal esquerdo e, em seguida, no direito. A figura 23 mostra como os dados são escritos no I²S proveniente do *technical reference manual* do ESP32, onde o Data 0 é escrito no canal Esquerdo e o Data 1 no canal Direito.

Figura 23 – Escrita de dados do I²S do ESP32

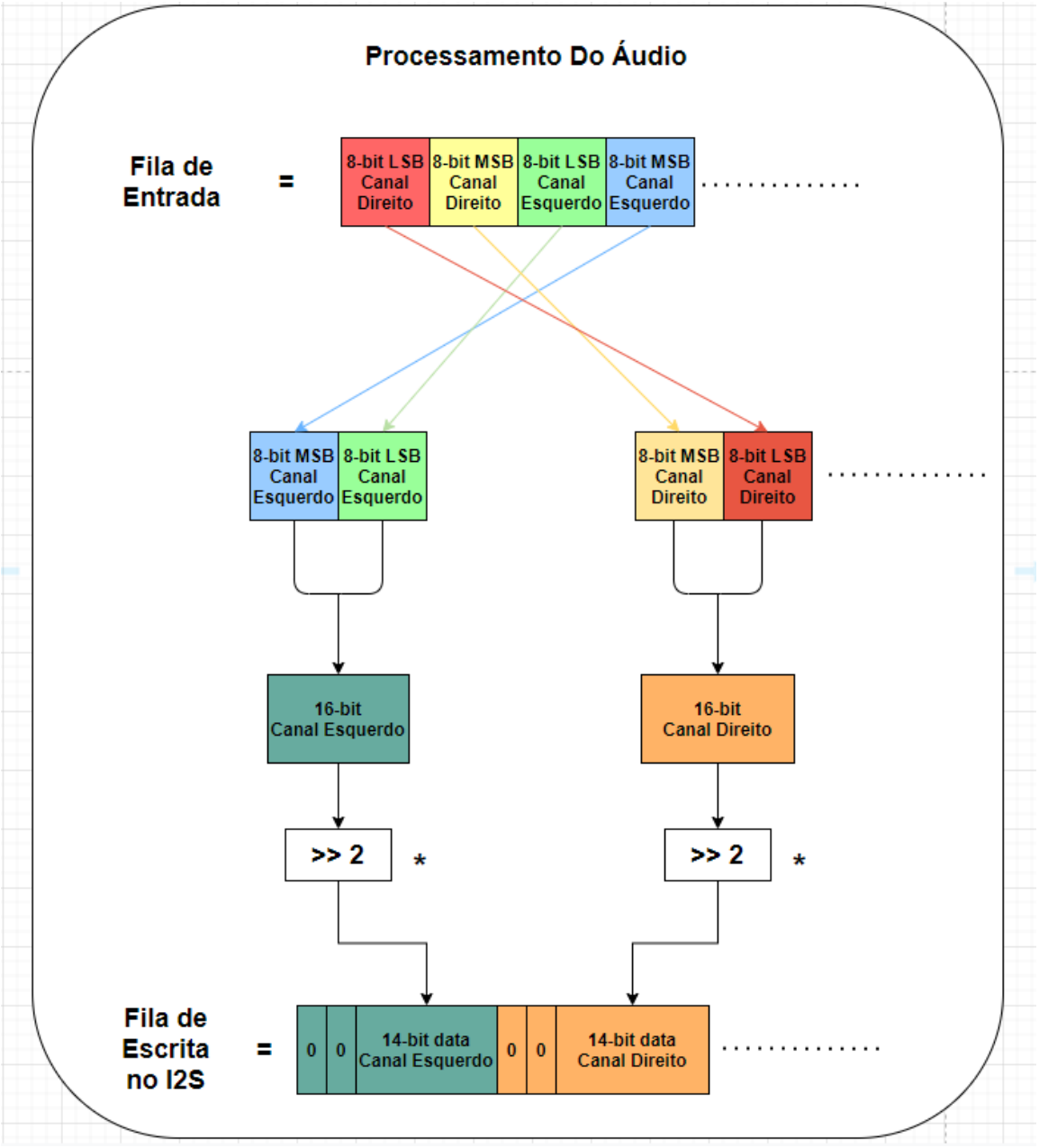


Fonte: (ESPRESSIF, 2021)

Após a realização de testes, nota-se que os dados de áudio recebidos estavam embaralhados e consequentemente, através de mais testes, foi possível descobrir a organização dos dados presentes *buffer* para então, processa-los e escreve-los no canal correto. Entretanto, implementando somente a reorganização de dados é possível notar um ruído gaussiano branco constante na recepção, independente do valor de volume.

Como a documentação disponível não analisa esse caso de ruído, não foi possível identificar exatamente a causa dessa gaussiana. Contudo, uma solução para tirar esse ruído foi deslocar 2 bits para direita do *buffer* de 16 bits que está sendo escrito no I²S. Diminuindo assim, a resolução do áudio, acrescentando zeros a esquerda do dado de áudio, fazendo com que os dados escritos no I²S passem a ter resolução de 14 bits, valor mais que o suficiente para uma transmissão de áudio, sendo somente 2 bits a menos do que é indicado pelo *The Audio Programming Book* (BOULANGER; LAZZARINI, 2011), como a resolução tipica de áudio digital. A figura 24 descreve o processamento do dado efetuado.

Figura 24 – Processamento de dados efetuado



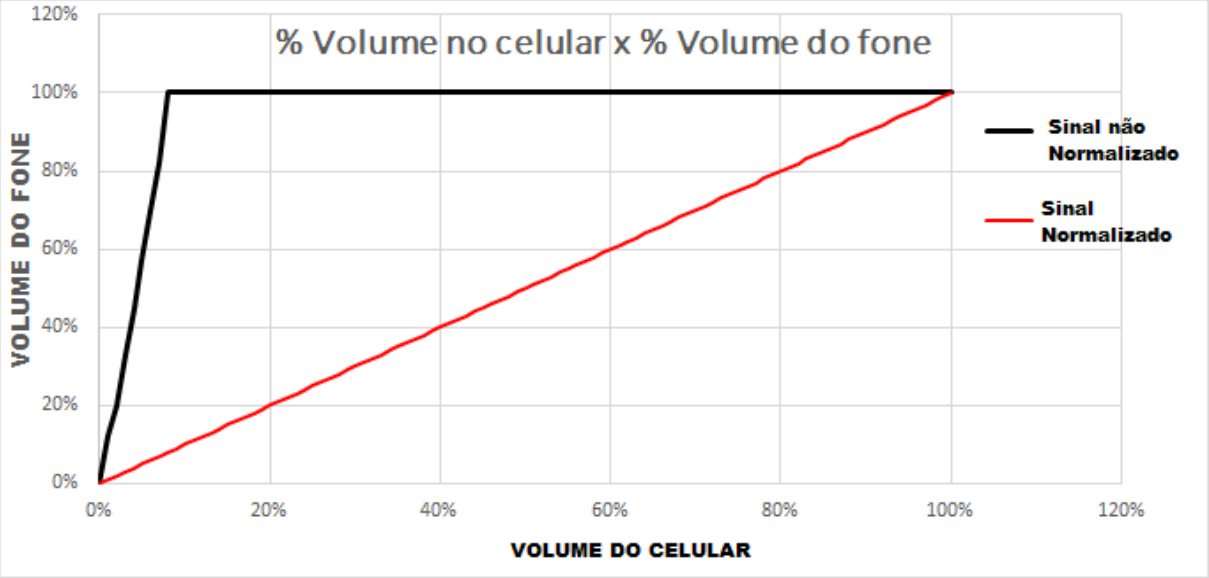
Fonte: Autoria própria, 2021

Após o processamento de dados, outro problema encontrado foi em relação ao volume. Onde, verificou-se que dispositivos com sistema operacional Android ou Windows o sinal de áudio recebido, fica saturado, fazendo com que a progressão linear de volume vá até 8% e comece a saturar. Ou seja, qualquer volume acima de 8% gera um sinal de saída no fone como se fosse 100%. Para corrigir esse problema, foi acrescentado uma função linear para ajustar o volume dos dispositivos Android e Windows utilizando um dado um sinal de notificação Bluetooth de volume pelo protocolo AVRCP.

- gainVol = 0.0110 * SVolume + 0.2
- dadoDir = dadoDir * gainVol
- dadoEsq = dadoEsq * gainVol

Como nesses casos a notificação de volume é recebida pelo protocolo AVRCP, pode-se requisitar a porcentagem de volume (SVolume) do dispositivo e utilizá-la na função abaixo. Portanto, o sinal de áudio de cada lado (dadoDir e dadoEsq) é multiplicado pelo ganho proporcional ao volume (gainVol). Assim, por exemplo, caso o valor máximo volume registrado seja de 0 a 8% originalmente, após o calculo de progressão esse áudio fica com escala de 0% a 100%. Ou seja, o sinal do celular, foi normalizado para o sinal de entrada do fone, ajustando linearmente a curva conforme figura 25. O *firmware* desenvolvido pode ser encontrado no Apêndice A.

Figura 25 – Gráfico de adequação do volume

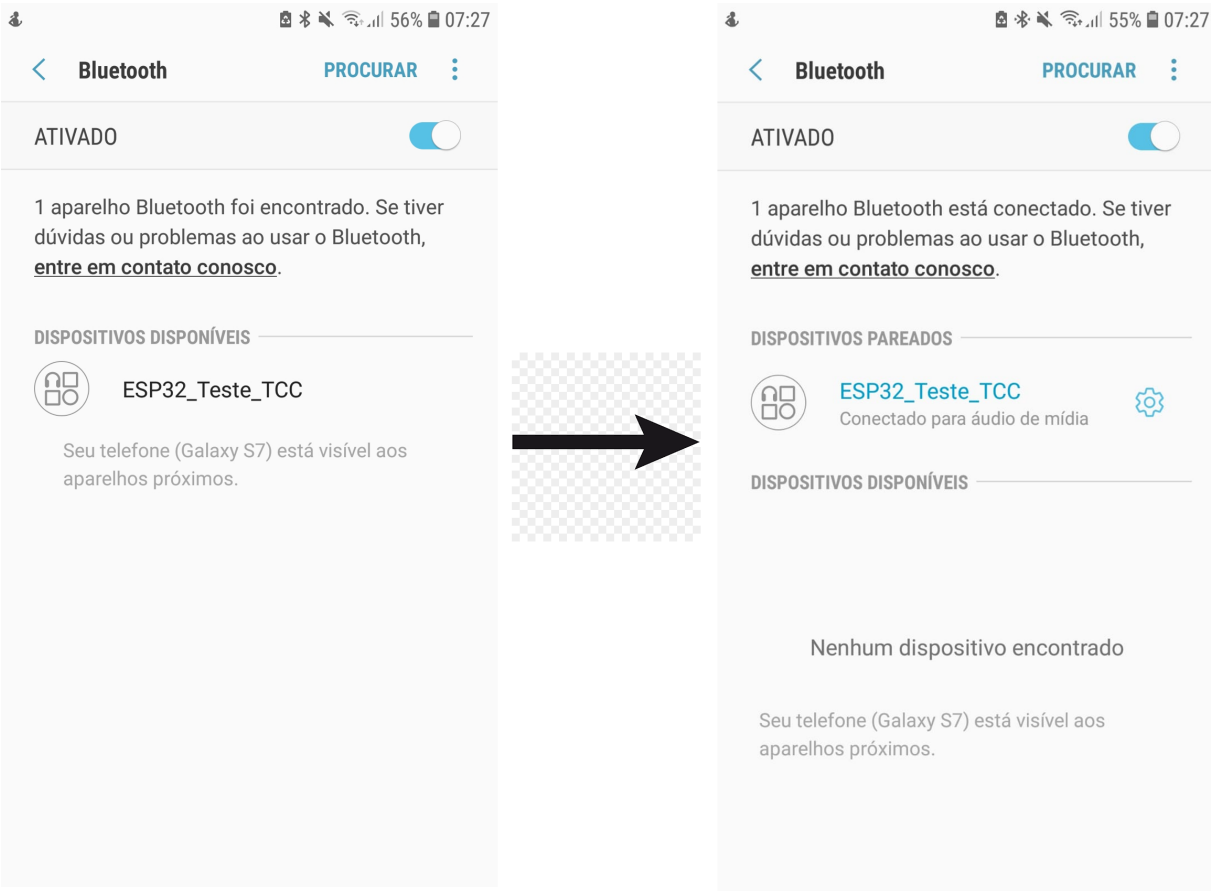


Fonte: Autoria própria, 2021

4.6 RESULTADOS

Ao implementarmos o circuito de teste, conectando o ESP32, configurado como *ESP32-Teste-TCC*, com o celular Samsung Galaxy S7 através do protocolo Bluetooth, foi observado que o protótipo era reconhecido como um dispositivo de áudio de saída. Ao tentar se conectar com o dispositivo criado, foi possível sincronizar o protótipo com o celular e em seguida transmitir os dados de áudio.

Figura 26 – Conexão do Celular com o Protótipo

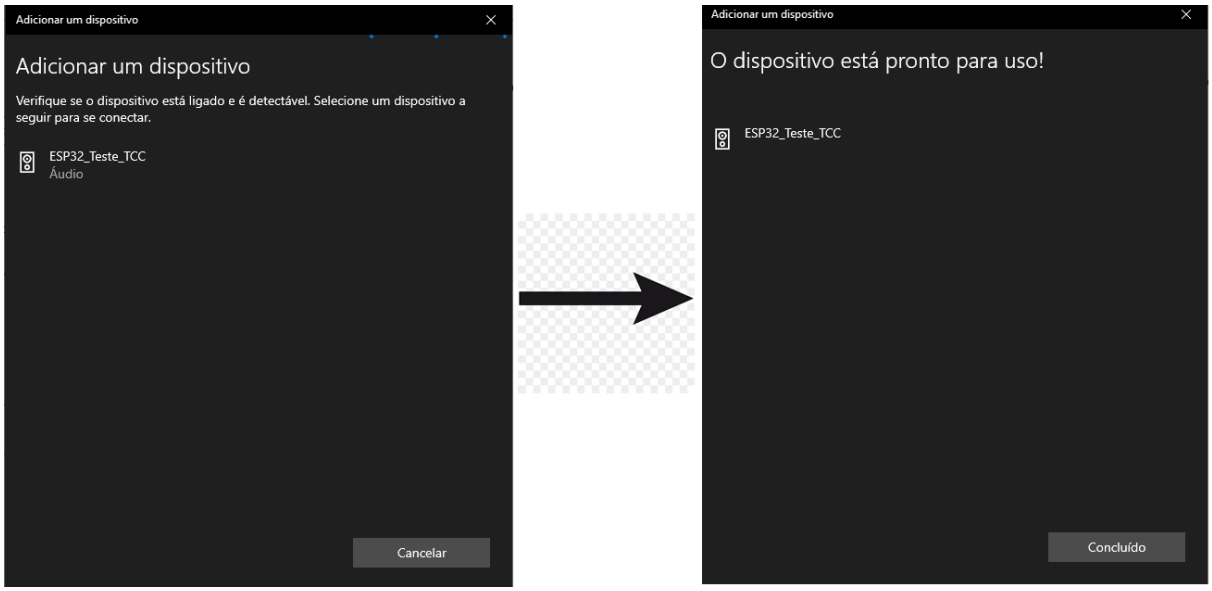


Fonte: Autoria própria, 2021

Após o processamento de dados descritos no capítulo 4.5, a saída de áudio do protótipo ficou com uma qualidade próxima a apresentada no próprio conector TRS do celular, comprovando assim o funcionamento do dispositivo criado.

Em seguida, o protótipo foi conectado com um notebook com Windows 10. As Figuras 26 e 27 a seguir, mostram o processo de conexão obtido entre o ESP32 e o notebook. Novamente, como observado na conexão entre o protótipo e o celular, a qualidade da saída de áudio da conexão TRS do protótipo foi igual a obtida diretamente do TRS do notebook.

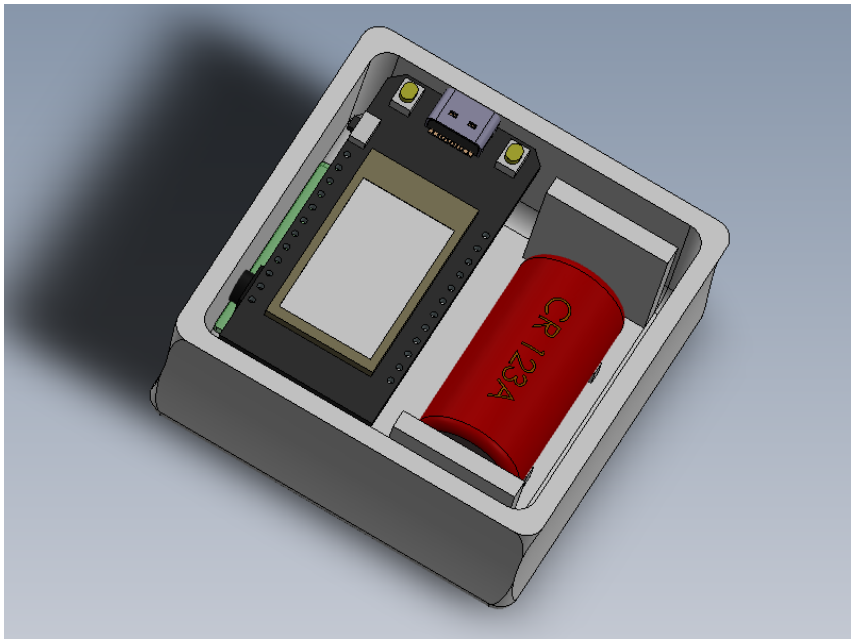
Figura 27 – Conexão do Notebook com o Protótipo



Fonte: Autoria própria, 2021

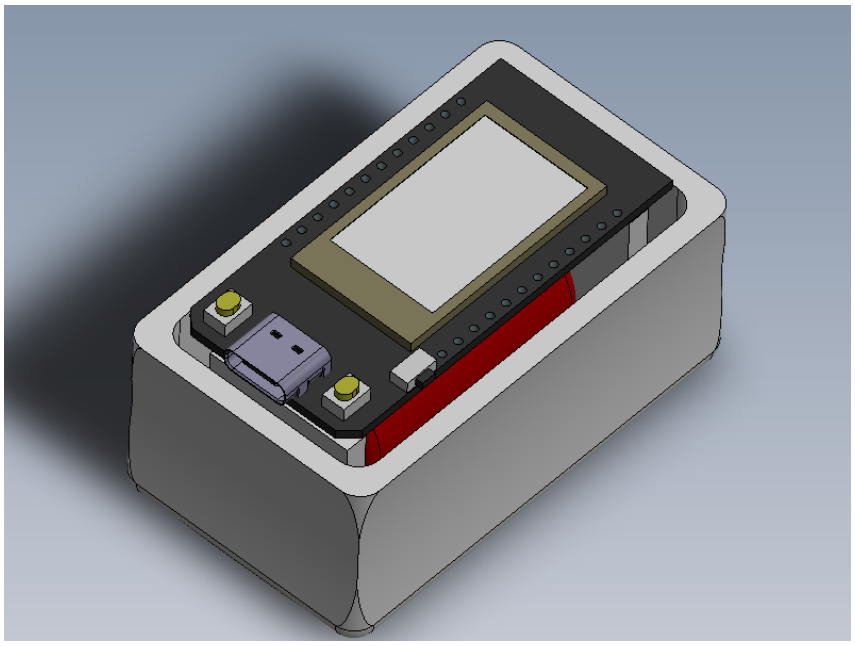
Com o teste do *firmware* realizado e com os conhecimentos adquiridos no estudo de baterias, foi possível criar duas possibilidades de estrutura a serem impressa em 3D, para armazenar o sistema proposto, visando a criação de um dispositivo móvel, como mostra as figuras 28 e 29. E por conta da logística durante a pandemia não foi possível imprimir os modelos.

Figura 28 – Cases Mecânicos para Impressão 3D, Receptor Bluetooth.



Fonte: Autoria Própria, 2021

Figura 29 – Cases Mecânicos para Impressão 3D, Receptor Bluetooth.



Fonte: Autoria Própria, 2021

Ao final da implementação do *firmware*, utilizando as configurações de *hardware* como ESP32 TTGO T-Display 240 MHz, 320KB RAM, 4 MB *flash*, notou-se, através da notificação do console, que o *flash* ocupado era de 95% com máxima capacidade de aproximadamente 1 MB. Acredita-se que o restante do *flash* seja usado para guardas as configurações feitas através do Menu de Configuração do ESP32. Caso esse código fosse implementado utilizando uma aplicação onde fosse necessário mais processamento de dados, um ESP32 com maiores capacidades de memória, como o Módulo ESP32-WROOM-32D, contendo 16 MB de Flash deverá ser utilizado para a função.

Figura 30 – Tamanho ocupado pelo firmware na flash do Protótipo

```
Linking .pio\build\esp32dev\firmware.elf
Retrieving maximum program size .pio\build\esp32dev\firmware.elf
Checking size .pio\build\esp32dev\firmware.elf
Building .pio\build\esp32dev\firmware.bin
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [==        ]  23.6% (used 77340 bytes from 327680 bytes)
Flash: [==========]  95.0% (used 996632 bytes from 1048576 bytes)
esptool.py v3.0
========================================================================================== [SUCCESS] Took 211.54 seconds ====

Terminal will be reused by tasks, press any key to close it.
```

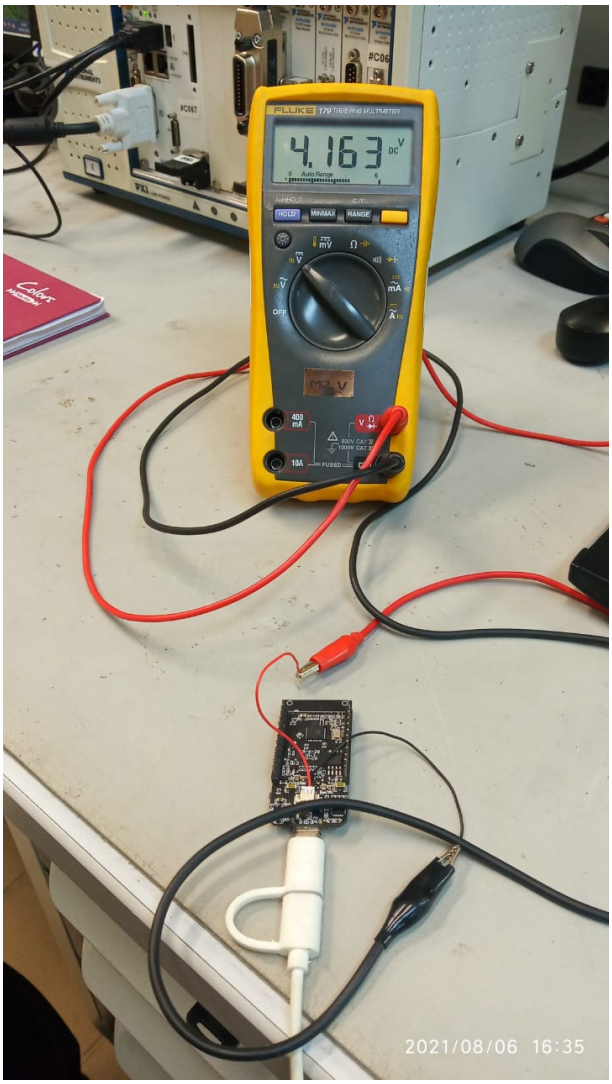
Fonte: Autoria própria, 2021

Com esse teste então pode-se definir o ESP32-WROOM-32D como um hardware melhor preparado para as aplicações desse trabalho, mesmo sendo necessário implementa-lo com alguns poucos circuitos adicionais, como por exemplo o regulador de 3.3V, o conector FTDI para embarcar o firmware e o circuito carregador de bateria. O principal benefício que ele possui em relação ao TTGO, fica sendo sua capacidade de memória quatro vezes maior, garantindo um

bom espaço para implementações adicionais. Por ser um circuito menor, pode-se criar melhores composições dele com o PCM5102, além da possibilidade de utilizar versões mais recentes do Bluetooth, coisa que com o TTGO fica-se limitado as escolhas da fabricante LILYGO.

Para garantir o funcionamento correto do carregador de bateria, o circuito apresentado na figura 13 foi testado, sua tensão nominal com o voltímetro em paralelo, necessita estar entre 4.1V e 4.2V. O valor medido estava regulado como apresentado na figura 31 abaixo em 4.15V.

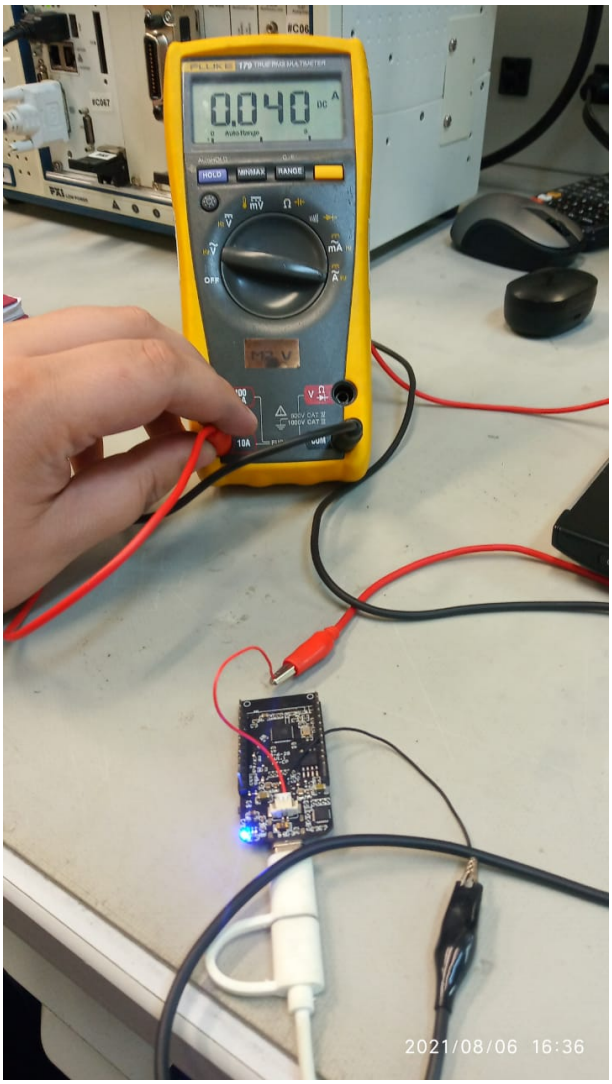
Figura 31 – Tensão do circuito carregador de bateria



Fonte: Autoria própria, 2021

A corrente de carga ficou limitado entre 40 e 50 mA, pois ao se testar o circuito carregador com o amperímetro em série, e utilizando sua resistência interna como carga. Foi simulado a condição de bateria totalmente descarregada, onde a resistência série da bateria é muito pequena. Com isso, o circuito carregador disponibiliza 10% de sua corrente nominal, processo denominado carga lenta, afim de evitar danos as baterias. Essa corrente pode ser configurada até 500mA dependendo da bateria a ser carregada e pode ser analisada na figura 32.

Figura 32 – Corrente do circuito carregador de bateria



Fonte: Autoria própria, 2021

Sendo assim, atinge-se todos os objetivos propostos para a conclusão do desenvolvimento do Receptor Bluetooth, onde, englobando suas particularidades e problemas encontrados anteriormente, houve uma importante contribuíram para a conclusão desse protótipo.

5 DISPOSITIVO RF

Neste capítulo, será apresentado um segundo dispositivo, focado em radio-frequência, visando a possibilidade de enviar os sinais de áudio em interface de equipamentos que não possuem Bluetooth integrado. Porém durante o desenvolvimento, apenas foi possível enviar um bit único entre o par de módulos. Esses conceitos de transceptores e o protótipo funcional serão demonstrados mais a frente.

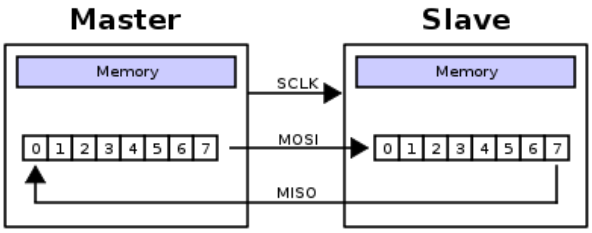
5.1 MATERIAIS UTILIZADOS

- Interface serial de periféricos SPI
- Transceptor de RFM69HW
- Conversor Analógico Digital

5.1.1 SPI - SERIAL PERIPHERAL INTERFACE

A interface SPI - *Serial Peripheral Interface* (Interface Serial para periféricos) é um protocolo que permite a comunicação de um microcontrolador com outras placas, sensores e dispositivos periféricos. Esse protocolo opera fazendo a interligação dos dispositivos com quatro fios. MISO (*Master In Slave Out*) e MOSI (*Master Out Slave In*) focados na transmissão os dados, SS(*Chip select*) para seleção de qual periféricos será acessado e SCLK(*source clock*) para sincronização. A comunicação de dados pode ser representado na figura 33 onde o SS está omitido.

Figura 33 – Fluxo da comunicação SPI



Fonte: (LOWPOWERLABS, 2018)

5.1.2 RFM69HW - TRANSCEPTOR RF

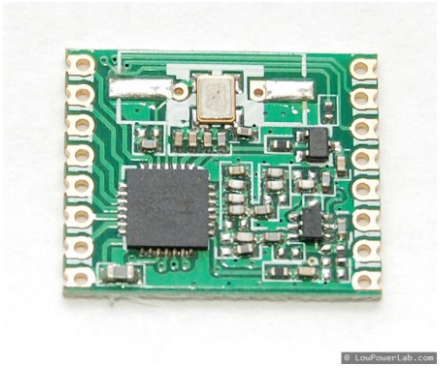
Este componente, tratado como periférico, será responsável pela função de transceiver (Transmissor/Receptor). Sua comunicação e controle, ocorre pela interface SPI, de onde receberá do microcontrolador, um pacote de dados e um comando identificando qual será a função executada (transmissão ou recepção).

Este transceptor possui inúmeras características que serão levadas em conta durante a concepção do projeto. Uma delas seria a possibilidade de executar a modulação do sinal transmitido em alguns métodos, como FSK, GFSK, MSK, GMSK e OOK.

Cada um dos métodos supracitados tem suas vantagens e desvantagens, mas o principal ponto levado em consideração é a taxa de transmissão. Sendo FSK método que trabalha na maior faixa para ser escolhida, entre 10kbps a 300kbps. Logo o método utilizado será o FSK.

Para este processo de transmissão com FSK, o RFM receberá dados do microcontrolador formatados como PCM, para cada pacote de dados, e assim sucessivamente para todos os pacotes. A placa de RFM69HW pode ser visto na figura 34. Este transceiver trabalhará a 915MHz e com 300kbps como taxa máxima de transmissão de dados (HOPERF, 2020).

Figura 34 – RFM69HW



Fonte: (LOWPOWERLABS, 2018)

5.1.3 CONVERSÃO ANALÓGICA DIGITAL (ADC)

Para realizarmos a amostragem do sinal analógico proveniente de um conector TRS de 3.5mm, é necessário utilizar um Conversor Analógico Digital (ADC), para que os dados de áudio possam ser interpretados pelo microcontrolador e assim processados e trabalhados.

Conversores ADC especializados na área de áudio não são novidade, podem ser encontrados em vários modelos, e empresas como a Texas Instruments, umas das maiores desenvolvedora de pastilhas de silício fabricam CIs como o PCM1802, PCM4220 e PCM1808 visando aplicações nessa área. A tabela 3 a seguir mostra uma comparação de alguns desses ADC que podem atender a necessidade de adquirir o áudio analógico.

5.2 TOPOLOGIA DO TRANSMISSOR RF

Após o desenvolvimento do trabalho com os elementos referentes ao dispositivo Bluetooth, inicia-se as discussões visando propor uma solução para atender aparelhos que não possuem as facilidade do Bluetooth implementadas. Primeiramente nesse capítulo, foi definido o RFM69HW e o ESP32 que irão trabalhar como receptor e transmissor de dados digitais. Além

Tabela 3 – Conversores Analógico Digital (ADC)

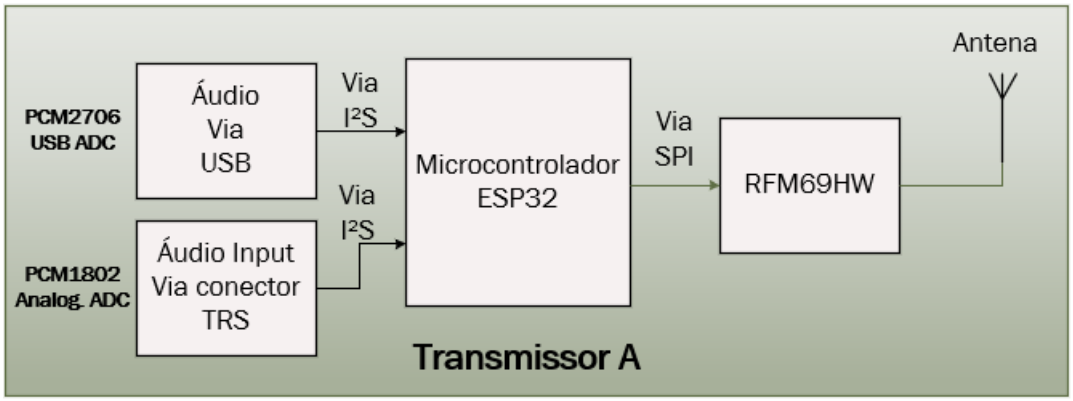
PCM1802	**PCM4220**	**PCM1808**
• 16kHZ a 96kHZ de Sample rate; • Necessita de um Clock na entrada SCK; • Resolução de 24bits; • 5V Analógico e 3.3V Digital; • Configurado pelos pinos no hardware;	• 8kHz até 216kHz de Sample rate; • Necessita de um Clock na entrada SCK; • Resolução de 24bits; • 4V Analógico e 3.3V Digital; • Configurado pelos pinos no hardware;	• 8kHz até 96kHz de Sample rate; • Necessita de um Clock na entrada SCK; • Resolução de 24bits; • 5V Analógico e 3.3V Digital; • Configurado pelos pinos no hardware;

Fonte: Autoria Própria (2020)

disso, como é necessário a ligação do microcontrolador com um Conversor Analógico-Digital, têm-se os esquemas elétricos de ligação e a placa de circuito que representa esse esquemático. Levando em consideração que este desenvolvimento contempla o Transmissor RF que será detalhado e discutido mais a frente, apresenta-se também, suas aplicações e problemas encontrados. Para que a transmissão posso ocorrer sem danificar a saída de radio-frequência faz-se necessário o dimensionamento de uma antena. No caso, um fio qualquer enrolado funcionará provisoriamente como um dispositivo passivo, unipolar e omnidirecional.

Tendo os componentes principais selecionados, a transmissão RF que será executado com o RFM69HW, e para isso foi montado um fluxograma com o intuito de representar o desenvolvimento do Transmissor A, como mostra a figura 35.

Figura 35 – Fluxograma da transmissão RF

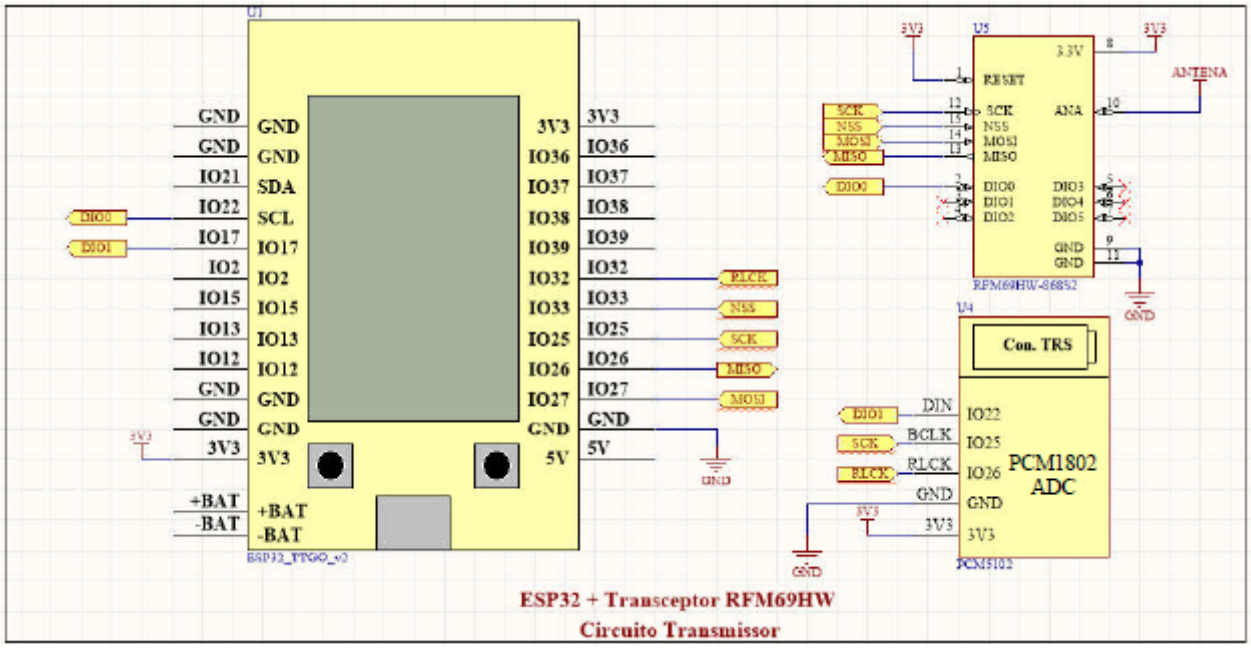


Fonte: Autoria Própria, 2020

O sinal elétrico que está em um conector TRS do equipamento que está emitindo o som de interesse necessita ser amostrado. Frequentemente esse sinal pode ser encontrado em conectores denominados *audio output* (Saída de áudio analógico). Utilizando um cabo comumente conhecido como P2-P2, pode-se transportar esse sinal de saída para a entrada do conversor analógico-digital. Após o processo de aquisição e conversão do sinal analógico, o PCM1802, envia via I²S, o pacote de dado digital contendo as informações do áudio para o microcontrolador, que processará o sinal e enviará através do protocolo SPI para o RFM69HW. Esse processo é executado se repetindo ao longo do tempo.

Essa configuração do ESP32 TTGO com o RFM é responsável por modular e transmitir o sinal, apresentado anteriormente. Sua configuração compõe o Transmissor A e o esquema elétrico com a ligação entre os componentes está abaixo na figura 36.

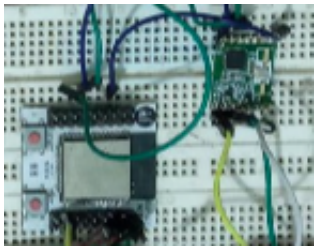
Figura 36 – Esquema elétrico da transmissão RF



Fonte: Autoria Própria, 2020

A figura 37 abaixo mostra o protótipo do RFM69HW com o ESP32 instalado.

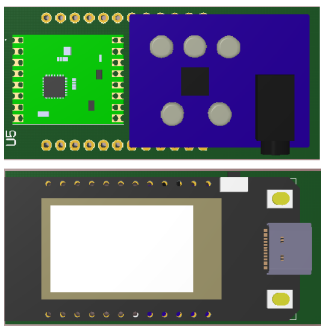
Figura 37 – Protótipo do RFM



Fonte: Autoria Própria, 2020

Com isso, pode-se executar a concepção da placa de circuito impresso do transmissor, sendo sua principal particularidade, a opção de inserir o RFM no lado inferior, e utilizar o circuito do regulador que o ESP32 TTGO possui, além de trazer a possibilidade de usar a alimentação de qualquer porta USB, de qualquer dispositivo, para fazer a alimentação do dispositivo transmissor. Na figura 38 abaixo, encontra-se o circuito impresso mencionado. E devido a pandemia, o protótipo foi montado apenas na matriz de contatos.

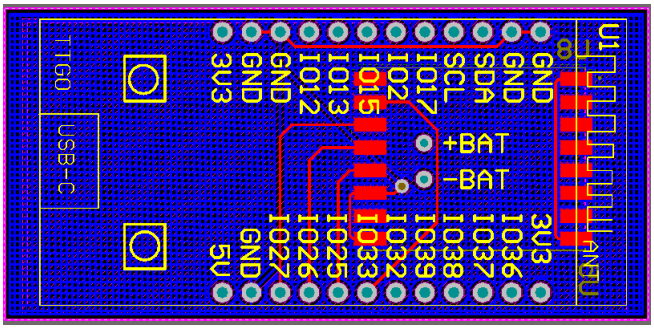
Figura 38 – Placa de Circuito da transmissão RF



Fonte: Autoria Própria, 2020

Além disso, o principal para a confecção da placa de circuito é o esquema de ligação que pode ser visto na figura 39, abaixo:

Figura 39 – Circuito da transmissão RF

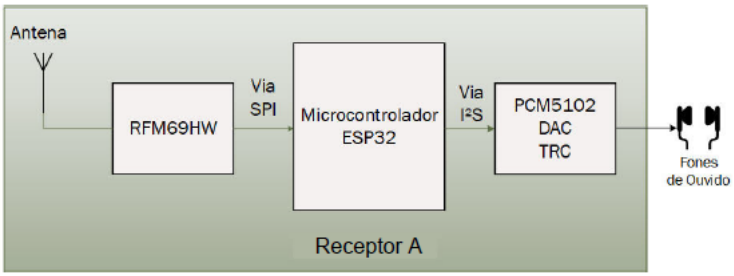


Fonte: Autoria Própria, 2020

5.2.1 TOPOLOGIA DO RECEPTOR RF

O sinal que o transmissor emitirá, tem como proposito encaminhar o dado do som sem utilizar fios, e com o intuito de captar esse sinal e reproduzi-lo a uma determinada distância do emissor, faz-se necessário um dispositivo de recepção. O fluxograma com esse processo é visto na figura 40.

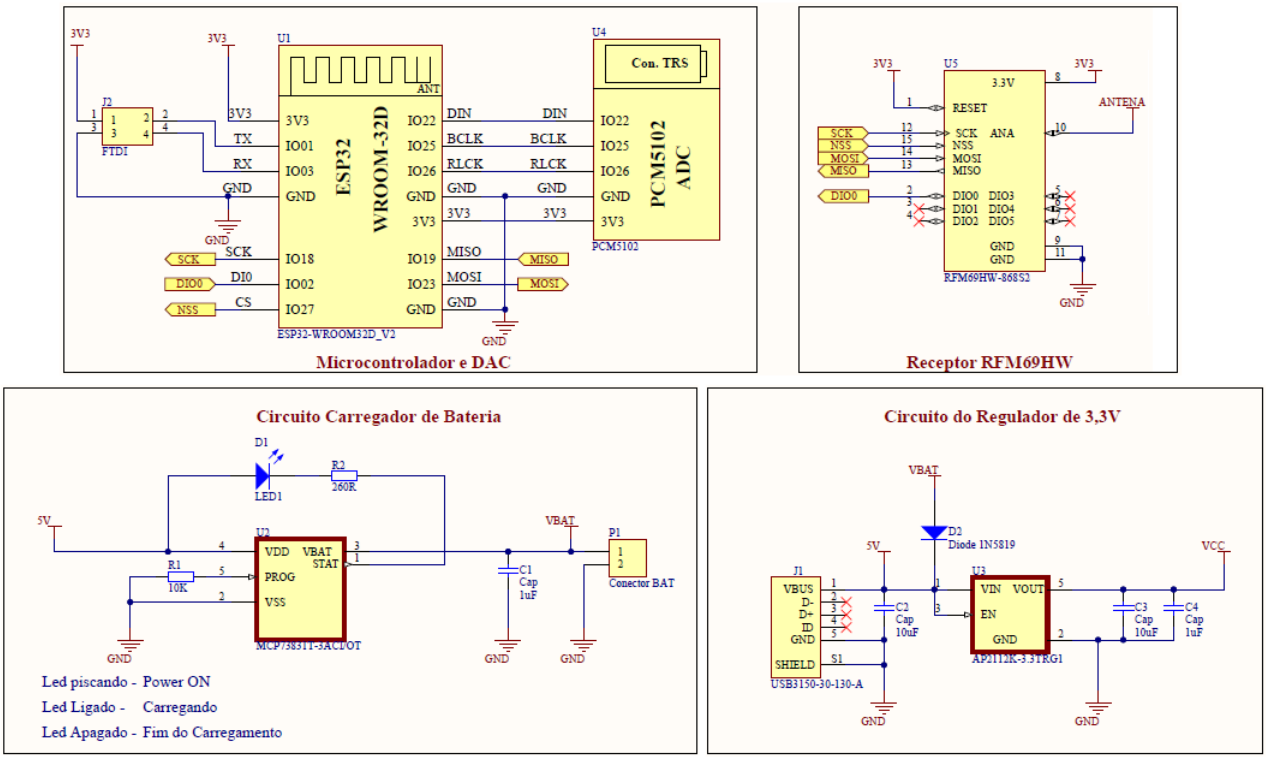
Figura 40 – Fluxograma do Receptor A



Fonte: Autoria Própria, 2020

O esquema elétrico da ligação do RFM com o ESP32, o circuito regulador e carregador de bateria e o conversor digital analógico (PCM5102) pode ser visto na figura 41 abaixo.

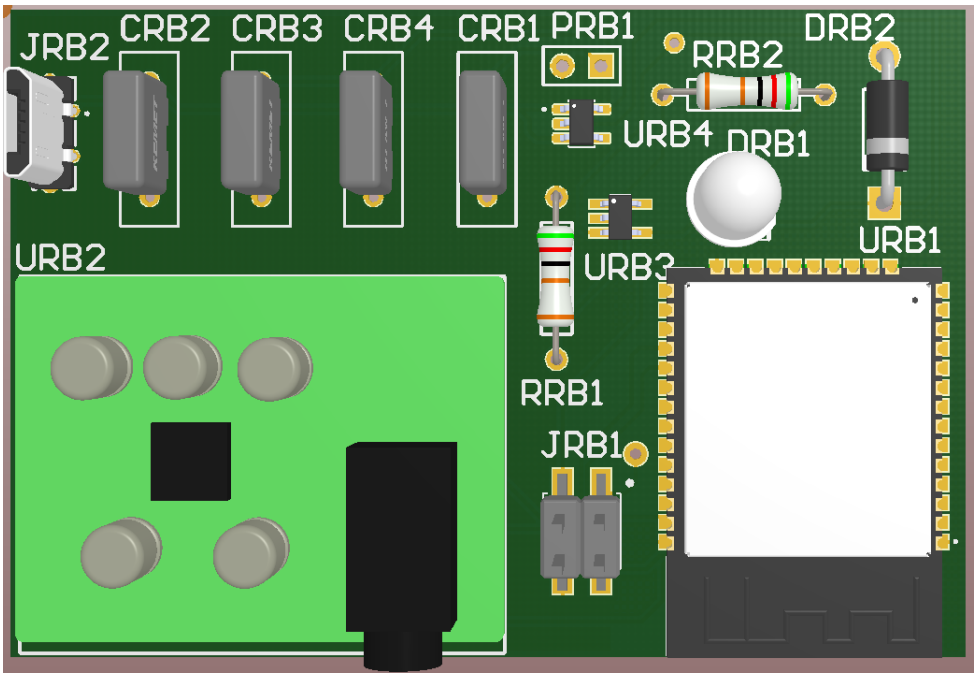
Figura 41 – Esquema Elétrico do RECEPTOR A



Fonte: Autoria Própria, 2020

O Circuito acima, foi utilizado para o desenvolvimento do protótipo. Esse receptor tem sua placa de circuito apresentado na figura 42.

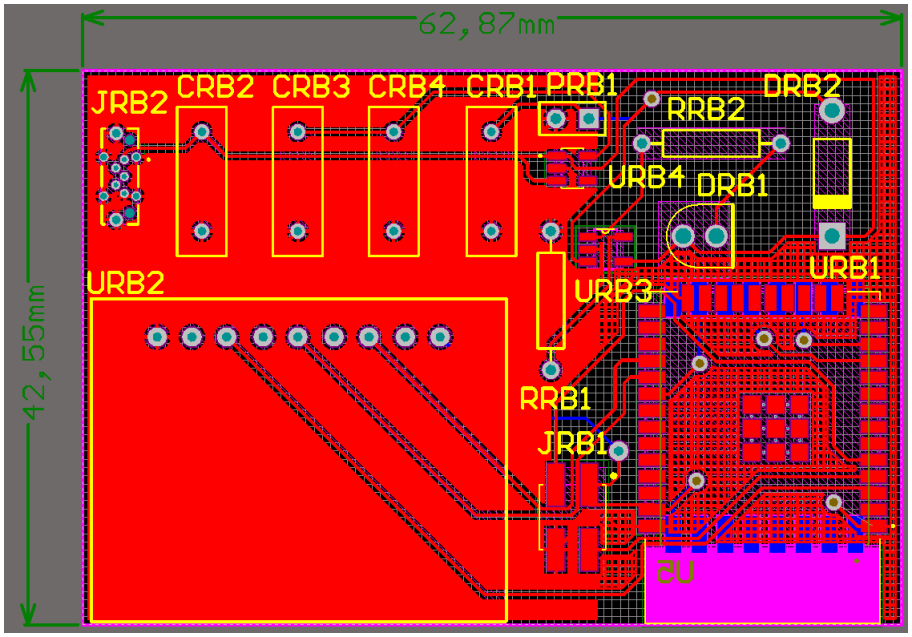
Figura 42 – Placa de circuito do RECEPTOR A



Fonte: Autoria Própria, 2020

Além disso, as trilhas do circuito pode ser visto na placa da figura 43abaixo.

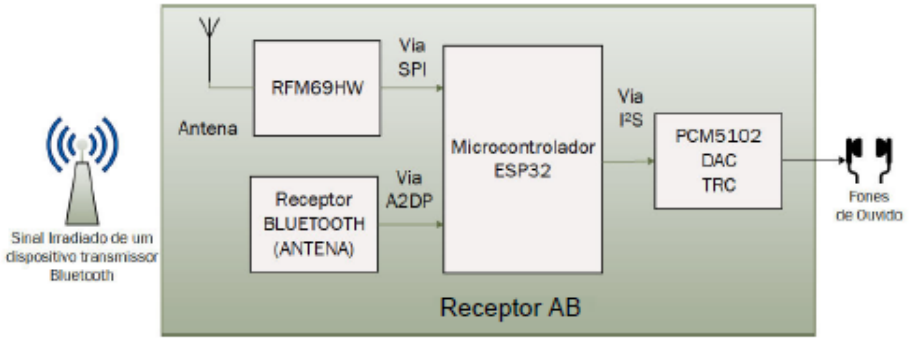
Figura 43 – Circuito do RECEPTOR A



Fonte: Autoria Própria, 2020

Esse dispositivo denominado RECEPTOR A pode conter além da aplicação RF, a operação do Bluetooth também, e quando isso ocorrer, o protótipo anterior já adicionando o RFM em seu circuito, será chamado RECEPTOR AB. Tendo seu fluxo operacional apresentado abaixo na figura 44.

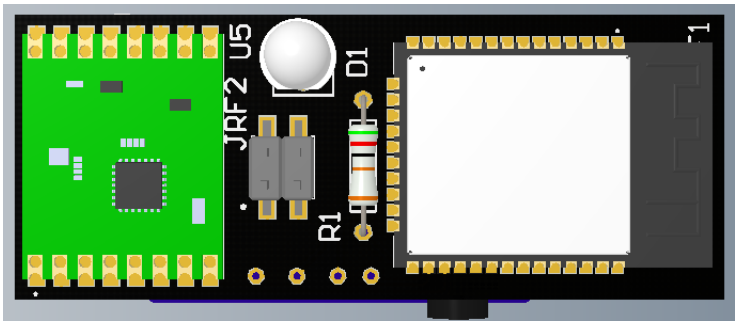
Figura 44 – Fluxograma do Receptor AB



Fonte: Autoria Própria, 2020

Além dos circuitos acima mencionado, existe mais um circuito de controle focado na configuração e no desenvolvimento do programa de comunicação do dispositivo. Esse circuito é constituído de um ESP32, o RFM69HW e por fim, um LED, utilizado para executar a recepção de um simples bit, o que garante o correto funcionamento da transmissão. Esse receptor pode ser visto na figura 45.

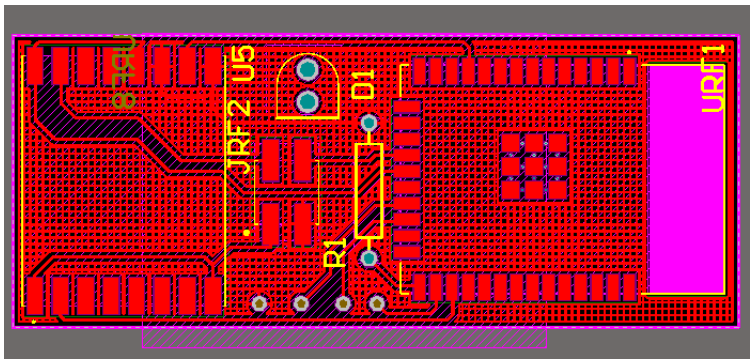
Figura 45 – Placa de circuito do RECEPTOR com LED



Fonte: Autoria Própria, 2020

Além disso, pode-se encontrar o circuito de ligação entre os componentes na figura 46 abaixo.

Figura 46 – Placa de circuito do RECEPTOR com LED



Fonte: Autoria Própria, 2020

5.3 PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE PARA RFM

Para a compilação do *firmware* que controla e opera o RFM, foi utilizado os mesmos ambientes do desenvolvimento do dispositivo Bluetooth. A primeira versão foi executada dentro do Platformio, utilizando as ferramentas da Espressif. Já a segunda versão do *firmware* foi trabalhada utilizando a IDE do Arduíno.

5.4 DESENVOLVIMENTO DO PROTÓTIPO DO TRANSMISSÃO E RECEPÇÃO RF

O esquema de ligação é o mesmo tanto para o receptor quanto para o transmissor, diferenciando-se apenas pelo *firmware* instalado nos respectivos ESP32 e o DAC que o receptor possui para tocar o som no fone.

Para generalizar e exemplificar o funcionamento de todo o processo de transmissão e recepção, tem-se um Conversor Analógico-Digital captando o sinal do som, convertendo e enviando para o ESP32 via I²S, que por sua vez envia para RFM69HW através do protocolo SPI. O transmissor RFM69HW, tem a função de modular um pacote de dados, e transmiti-lo por radiofrequência onde será capitado pelo RFM69HW do módulo receptor. Após a transferência, o ESP32 do módulo recepctor é notificado pelo RFM69HW relatando que a recepção ocorreu bem e, em seguida, o microcontrolador acessa os registradores do transceptor contendo os dados recebidos. Após os recebimento e acesso aos dados nos registradores internos, o ESP32 envia os dados para o conversor digital que tocará os dados no fone, além de encaminhar uma mensagem de volta para o transmissor informando o fim do processamento.

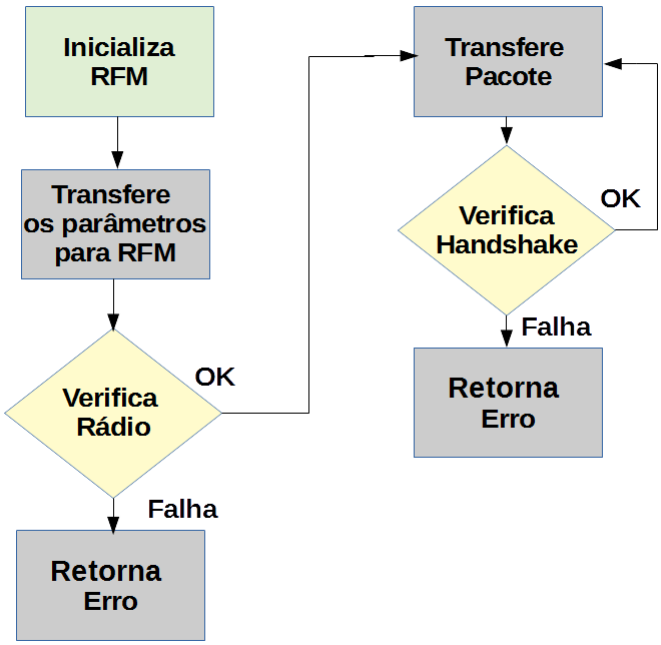
Com esse processo sendo reproduzido continuamente durante todo o funcionamento dos transceptores, têm-se todo o pacote de dados (Áudio Sonoro) transmitido, ficando disponivel para ser processado.

Até a realização desse documento, os CIs necessários para transmissão estão ao alcance e podem ser encontrados tanto no Aliexpress quanto na Mouser.

5.5 DESENVOLVIMENTO DO SOFTWARE PARA O PROTÓTIPO RF

Para o desenvolvimento do software que controla o RFM faz-se necessário seguir os seguinte fluxograma apresentado a seguir. Primeiramente será apresentado, na figura 47, o fluxo de trabalho do transmissor:

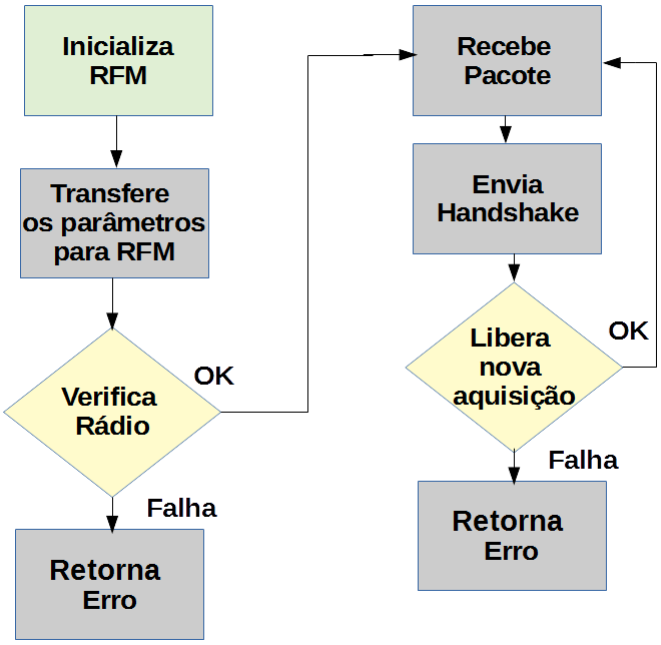
Figura 47 – Fluxograma do programa de transmissão do RFM69HW



Fonte: Autoria Própria, 2020

Posteriormente, para o receptor, o software deve funcionar com o seguinte fluxo de operações apresentado na figura 48.

Figura 48 – Fluxograma do programa de recepção do RFM69HW



Fonte: Autoria Própria, 2020

Baseando nos objetivos proposto, foi iniciado o desenvolvimento de ambos os softwares utilizando a *toolkit IDF* disponibilizada pela Espressif. Essa solução fez-se necessária pois o dispositivo proposto visava atender, em conjunto, a opção RF e a opção Bluetooth que também foi desenvolvida com a linguagem de Espressif.

Como apresentado na topologia do receptor, o software responsável por receber o pacote de áudio deve funcionar captando os dados proveniente da amostragem digital de um dispositivo ADC, no caso do protótipo proposto o PCM1802. Esse módulo envia o sinal amostrado por I²S, então durante o desenvolvimento do software, uma seção contém a configuração do I²S, de suas pinagens de entrada e saída de sinal, e se o áudio é do canal da direita e da esquerda.

Com o pacote de dados já capturados, o ESP32 necessita transmitir o dado para o RFM69HW, através de um barramento de comunicação de dados, denominado SPI. Esse barramento, como visto anteriormente, trabalha com o preenchimento e deslocamento de registradores nos endereços de memória do ESP32 para o RFM69HW. Com esses valores e endereços específicos torna-se possível realizar as configurações do RFM69HW e também transmitir o pacote de dados.

Os principais registradores utilizados para executar a configuração do RFM69HW foram os seguintes:

- radio.Modulation = FSK; //Seleciona a modulação FSK.
- radio.COB = RFM69; //Define o modelo do transceptor.

- radio.Frequency = 915000; //Seleciona a frequência de operação do radio.
- radio.OutputPower = 10+18; //Define a potência de saída.
- radio.PreambleLength = 16; //Comprimento do cabeçalho.
- radio.FixedPktLength = true; //Define se o pacote tem tamanho fixo ou variável.
- radio.PayloadLength = 21; //Define o tamanha dos dados em bytes.
- radio.CrcDisable = true; //Desabilita a verificação cruzada de integridade de dados.
- radio.AesOn = false; //Desabilita a criptografia AES.
- radio.SymbolTime = 192.5; //taxa de transmissão de dados em Kbps.
- radio.Deviation = 35; //Limita o desvio de frequência.
- radio.BandWidth = 100; //Banda disponível.
- radio.SyncLength = 3; //Define o tamanho da palavra de sincronismos, o valor padrão é 0xAA2DD4.

  radio.SyncWord[0] = 0xAA;

  radio.SyncWord[1] = 0x2D;

  radio.SyncWord[2] = 0xD4;
- radio.vInitialize(); //Inicializa o rádio
- radio.vGoStandby(); // Entra com o rádio em modo de espera.

Os comandos acima são apenas alguns dos disponíveis na documentação técnica do RFM69, existem ainda configurações específicas que o *datasheet* e o *application note* (AN2001-RF69) do RFM detalham de forma mais exclusiva.

Para que o dispositivo funcione, todos esses comando devem ser escritos em linguagem C++ e compilados com os comandos da biblioteca que a EspressIf disponibiliza. Nesse ponto encontrou-se um forte obstáculo para o atendimento, em sua totalidade, dos objetivos propostos.

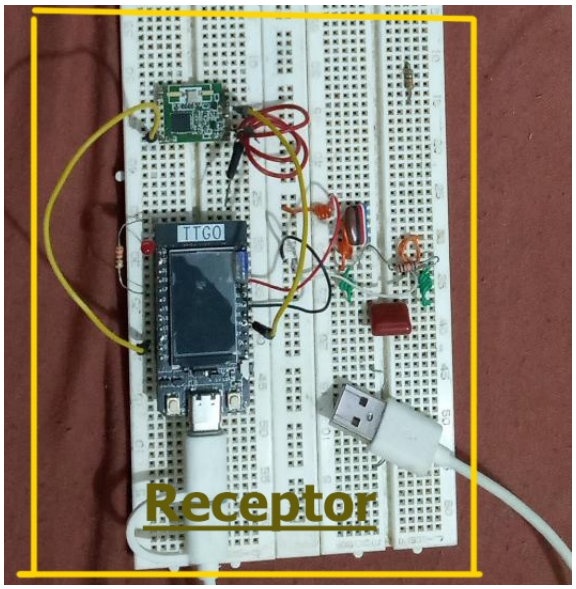
A biblioteca de comandos disponibilizada pela fabricante do ESP32, não apresenta de forma aberta, o fluxo de dados e o controle dos registradores do microcontrolador, com isso, ao encaminhar os comandos de configuração para o RFM69, nem todos os valores são inseridos de forma correta nos registradores do transceptor, fazendo com que o mesmo não inicialize de forma correta.

Com isso, outra abordagem foi trabalha, à possibilidade de executar o controle do microcontrolador e do transceptor com as bibliotecas em Arduíno disponibilizadas pelo fabricante do radio, HopeRF. Utilizando essa abordagem, perde-se a integrabilidade com o dispositivo receptor Bluetooth, contudo habilita a possibilidade de testar se o RFM69 está funcionando e também verificar o rendimento do sensor, para alguns casos específicos.

5.6 RESULTADOS

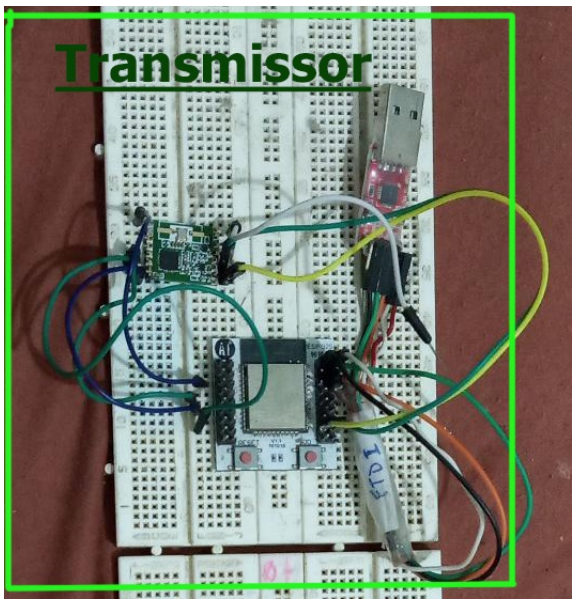
Durante o desenvolvimento do ESP32 com o RFM69HW foi necessário desenvolver dois firmware, um responsável pelo controle do receptor e o outro responsável pelo transmissor. Ambos os dispositivos foram montados e instrumentados na matriz de contatos e podem ser visto nas figuras 49 e 50.

Figura 49 – Setup de teste e desenvolvimento do Receptor RFM69HW



Fonte: Autoria Própria, 2020

Figura 50 – Setup de teste e desenvolvimento do Transmissor RFM69HW



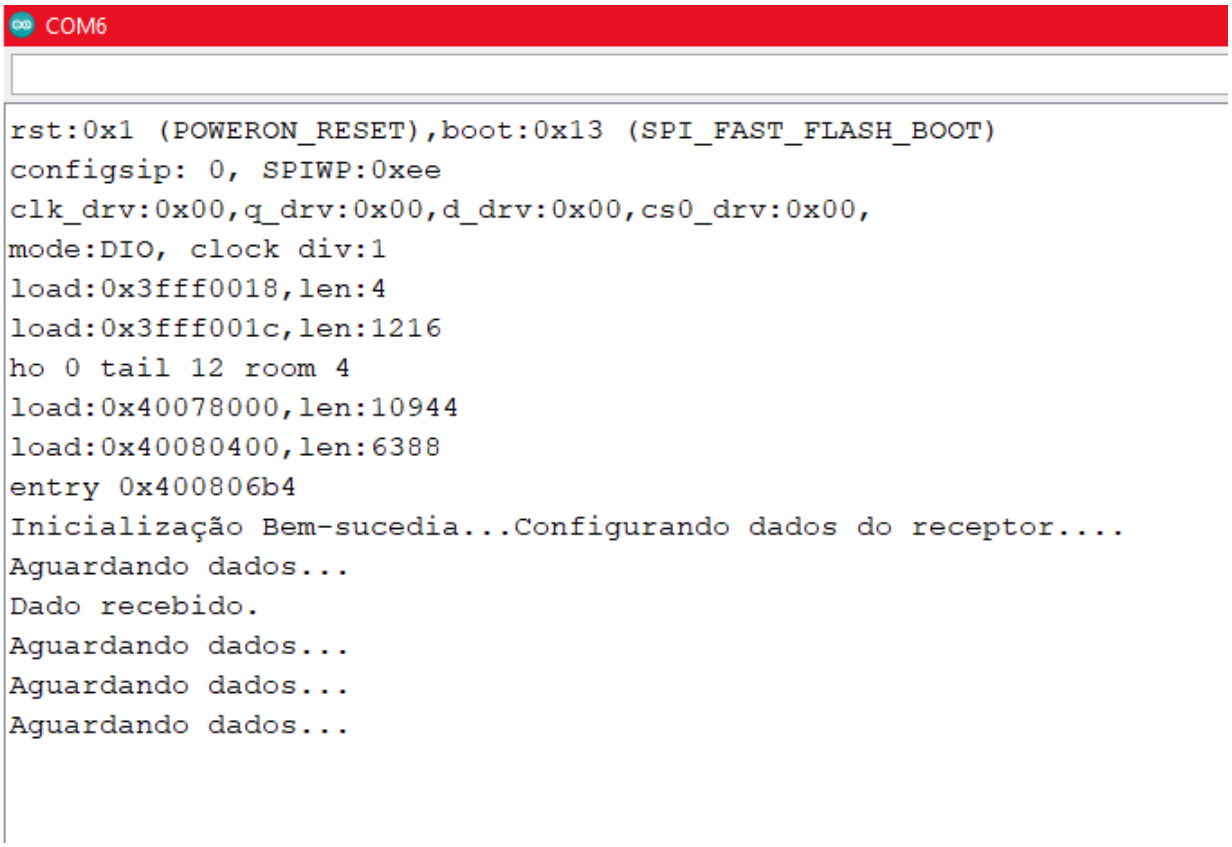
Fonte: Autoria Própria, 2020

Para atender esse objetivo de verificar o funcionamento do RFM69HW, foi proposto a transmissão de um bit único, que foi enviado a uma taxa de 192,5kbps, e pode acionar um LED.

O par de *firmwares* desenvolvidos para a versão Arduíno, tem o mesmo comportamento proposto para a versão em IDF, e foram desenvolvidos para interfacear o ESP32 com o RFM69HW. Os códigos podem ser encontrado no Apêndice B.

Ao embarcarmos o software no ESP32 com a função do receptor, recebemos a mensagem de "Inicialização Bem-sucedia. Configurando dados do receptor....Aguardando dados". Essa mensagem é um forte indicador de que o RFM agora está funcionando em sua plenitude. As mensagens podem ser encontradas no figura 51 abaixo.

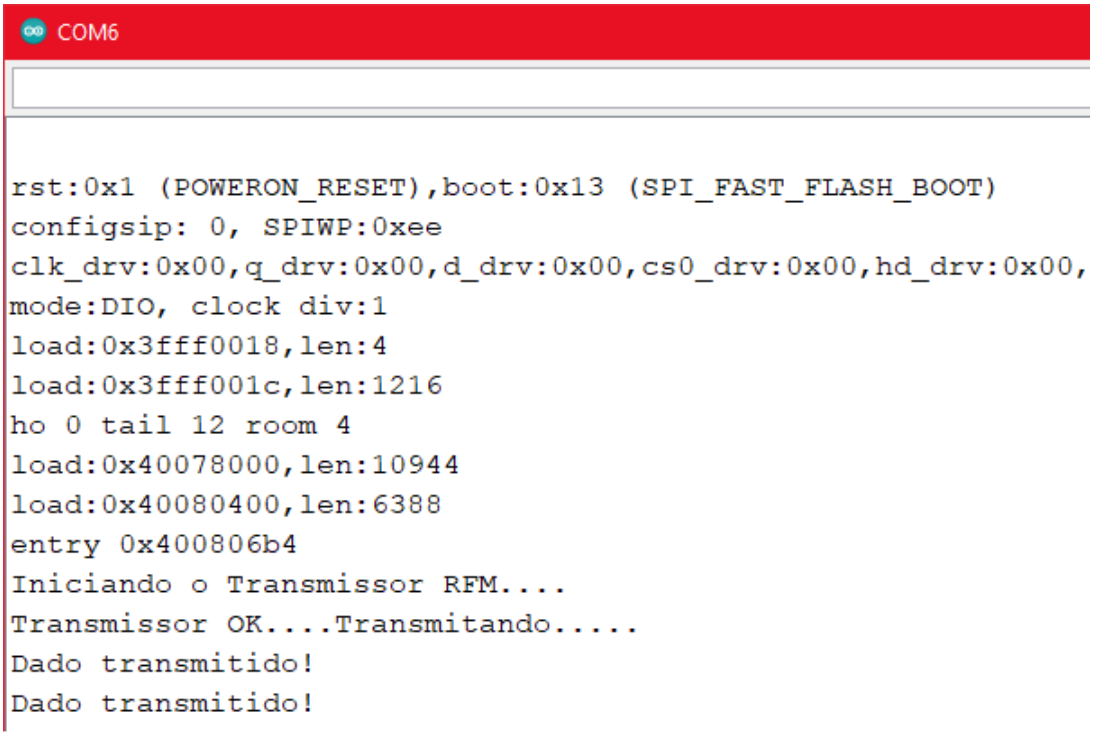
Figura 51 – Resposta da Serial para a recepção com o RFM69HW



Fonte: Autoria Própria, 2020

Após o receptor estar configurado, iniciou-se a compilação do firmware do transmissor. E após finalizado o carregamento, recebeu-se na serial a seguinte mensagem: "Iniciando Transmissor RFM69HW...", "Transmissor OK", "Dado Transmitido...". Essa mensagem também indica que o transmissor agora está funcionando conforme o esperado, e seu resultado pode ser visto na figura 52.

Figura 52 – Resposta da Serial para a transmissão com o RFM69HW



Fonte: Autoria Própria, 2020

Após esse teste de transmissão verificou que o sinal fica acionando e apagando o LED até por volta de cem metros. Esse valor da distância varia e pode ser ajustado alterando algumas configurações do radio RFM. A taxa de transmissão, a potência do transmissor e o tipo de antena utilizado afetam proporcionalmente à máxima distância alcançável do receptor com o sinal ainda integro e sem perdas. Como no caso do teste a antena utilizada foi uma antena passiva, unipolar e Omnidirecional (em todas as direções), o alcance e o rendimento do transmissor fica comprometido, podendo ter um alcance muito maior com uma antena especifica para a frequência de 915MHz, valor de operação do sinal de rádio.

6 CONCLUSÃO

Com a realização das pesquisas apresentadas nesse documento, foi possível fazer um estudo dos principais componentes para criação de um dispositivo eletrônico capaz de realizar a recepção de dados de áudio através do protocolo Bluetooth.

Além disso, houve a criação de um protótipo, no qual foi possível reproduzir áudio recebido pelo protocolo Bluetooth, com uma qualidade igual a do conector TRS dos próprios dispositivos testados. Esse protótipo tinha como principal componente o ESP32 TTGO T-Display, trabalhando como microcontrolador, capaz de executar a recepção e processamento dos dados, além de gerenciar o carregamento da bateria. Contudo no decorrer dos ensaios foi analisado a necessidade de utilizar uma interface ESP32 que possuísse uma memória *flash* maior que 4MB, pois somente o código do Bluetooth já consumia mais que 95% desse valor. Logo o novo microcontrolador escolhido foi o ESP32-WROOM-32D, componente que proporciona 16 MB de *flash*, além de um número maior de pinos de entrada e saída disponível. Esse microcontrolador apesar de possuir uma memória maior, não vem com um circuito regulador de tensão e nem um circuito carregador de bateria, contudo esse problema pôde ser facilmente contornado com um circuito externo como proposto no protótipo final.

Outro ponto importante, levantado durante o trabalho foi a proposição de um protótipo capaz trabalhar juntamente com a opção da operação via Bluetooth. Para isso foi previsto utilizar o RFM69HW como transceptor de radiofrequência com o intuito de integrar ambas as operações no mesmo dispositivo, o que se mostrou inatingível até o fim da elaboração desse documento. Contudo, trabalhos para caraterizar e validar o dispositivo transceptor de radiofrequência foram executados e apresentaram rendimentos satisfatórios de forma individual, sem considerar a junção com a plataforma com o protocolo Bluetooth. Esses rendimentos significam que o radio RFM é sim capaz de transmitir dados, vide o caso do acendimento do LED, e pode ser adaptado para transmitir um volume maior de dados.

Considerou-se importante também para o desenvolvimento do projeto, a escolha de uma bateria capaz de atender os requisitos de mobilidade do dispositivo, e ser segura para poder ser operado. Além de considerar o circuito carregador e o regulador de tensão.

Durante todo o desenvolvimento do trabalho, foi possível implantar diversos conceitos multidisciplinares abordados no decorrer do curso de engenharia eletrônica, sendo as áreas de maior abrangência nesse trabalho, as disciplinas de Microcontroladores, Sistemas Embarcados, Processamento digitais de Sinais, entre outras.

REFERÊNCIAS

ANDROID, S. **Fone de ouvido de 3,5 mm: Especificação de acessório**. 2020. Disponível em: <https://source.android.com/devices/accessories/headset/plug-headset-spec>. Citado na página 26.

BLUETOOTH. **A2DP**. 2019. Disponível em: <https://www.bluetooth.com/specifications/specs/>. Citado na página 21.

BLUETOOTH. **Our History**. 2020. Disponível em: <https://www.bluetooth.com/about-us/our-history/>. Citado na página 20.

BLUETOOTH. **Profiles make Bluetooth technology interoperable**. 2020. Disponível em: <https://www.bluetooth.com/specifications/profiles-overview/>. Citado na página 20.

BLUETOOTH. **Understanding Bluetooth Range**. 2020. Disponível em: <https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/range/>. Citado na página 20.

BOULANGER, R.; LAZZARINI, V. **The Audio Programming Book**. [S.l.: s.n.], 2011. ISSN ISBN 9780262014465. Citado 3 vezes nas páginas 23, 39 e 40.

BYCAST. **Aprenda o que é streaming e como funciona para áudio**. 2018. Disponível em: <https://blog.bycast.com.br/streaming/aprenda-o-que-e-streaming-e-como-funciona-para-audio/>. Citado na página 14.

CUIDEVICES. **Interconnect TRS3,5mm**. 2021. Disponível em: <https://www.cuidevices.com/catalog/interconnect>. Citado na página 26.

ESPRESSIF. **Código Bluetooth Clássico**. 2020. Disponível em: <https://github.com/espressif/esp-idf/tree/178b122/examples/bluetooth/bluedroid/classic_bt>. Citado na página 21.

ESPRESSIF. **ESP32 Modules and Boards**. 2020. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/modules-and-boards.htm>. Citado na página 18.

ESPRESSIF. **Product Ordering Information**. 2020. Disponível em: <https://www.espressif.com/sites/default/files/documentation/espressif_products_ordering_information_en.pdf>. Citado 3 vezes nas páginas 14, 18 e 19.

ESPRESSIF. **Technical reference manualdo ESP32**. 2021. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf>. Citado na página 40.

HOPERF. **RFM69HW**. 2020. Disponível em: <https://www.hoperf.com/modules/rf_transceiver/RFM69HW.html>. Citado na página 49.

LILYGO. **LILYGO® TTGO T-Camera ESP32 WROVER & PSRAM Camera Module ESP32-WROVER-B OV2640 Camera Module 0.96 OLED**. 2020. Disponível em: <http://www.lilygo.cn/prod_view.aspx?TypeId=50030&Id=1273&FId=t3:50030:3>. Citado na página 20.

LOWPOWERLABS. **RFM69 and SPIFlash libraries released via Arduino LibraryManager**. 2018. Disponível em: <https://lowpowerlab.com/2018/05/24/rfm69-and-spiflash-libraries-released-via-arduino-librarymanager/>. Citado 2 vezes nas páginas 48 e 49.

MACSBUG. **Web Radio of M5Stack PCM5102A I2S DAC**. 2021. Disponível em: <https://macsbug.wordpress.com/2021/02/19/web-radio-of-m5stack-pcm5102a-i2s-dac/>. Citado na página 33.

MELLO, D. **Home office foi adotado por 46pandemia**. 2020. Disponível em: <https://agenciabrasil.ebc.com.br/economia/noticia/2020-07/home-office-foi-adotado-por-46-das-empresas-durante-pandemia>. Citado na página 17.

NXP. **I2S bus specification**. 2020. Disponível em: <https://web.archive.org/web/20070102004400/http://www.nxp.com/acrobat_download/various/I2SBUS.pdf>. Citado na página 22.

PLATFORMIO. **PLATFORMIO IDE**. 2020. Disponível em: <https://platformio.org/>. Citado na página 29.

ROTTA MáRIO DANTAS, A. C. G. **Protocolos de comunicação para ambientes de Internet das coisas**. 2020. Disponível em: <https://infranewstelecom.com.br/protocolos-de-comunicacao-para-ambientes-de-internet-das-coisas/>. Citado na página 14.

ZIGBEE. **What is Zigbee?** 2020. Disponível em: <https://zigbeealliance.org/solution/zigbee/>. Citado na página 20.

Apêndices

APÊNDICE A – FIRMWARE BLUETOOTH INSERIDO NO ESP32

A.1 MAIN.C

```
1  // Copyright 2020-2021 Bruno T. Adriano and Gabriel C. Diorio
   //
3  // Licensed under the Apache License, Version 2.0 (the "License");
   // you may not use this file except in compliance with the License.
5  // You may obtain a copy of the License at

7  //      http://www.apache.org/licenses/LICENSE-2.0
   //
9  // Unless required by applicable law or agreed to in writing, software
   // distributed under the License is distributed on an "AS IS" BASIS,
11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
   // See the License for the specific language governing permissions and
13 // limitations under the License.

15 //Original code avaliable at: https://github.com/espressif/esp-idf/tree
       /178b122c145c19e94ac896197a3a4a9d379cd618/examples/bluetooth/bluedroid
       /classic_bt/a2dp_sink

17 //Includes
   #include <stdint.h>
19 #include <stdio.h>
   #include <stdlib.h>
21 #include <unistd.h>
   #include <string.h>
23 #include <stdbool.h>
   #include <math.h>
25 #include <sys/unistd.h>
   #include <sys/stat.h>
27
   #include "freertos/FreeRTOS.h"
29 #include "freertos/task.h"
   #include "freertos/xtensa_api.h"
31 #include "freertos/FreeRTOSConfig.h"
   #include "freertos/queue.h"
33 #include "freertos/ringbuf.h"

35 #include "nvs.h"
   #include "nvs_flash.h"
37 #include "esp_system.h"
   #include "esp_log.h"
```

```
39
   //Includes somente com a ativa  o do BT no menuconfig
41 #include "esp_bt.h"
   #include "esp_bt_main.h"
43 #include "esp_bt_device.h"
   #include "esp_gap_bt_api.h"
45 #include "esp_a2dp_api.h"
   #include "esp_avrc_api.h"
47 #include "driver/i2s.h"

49
   #include "esp_err.h"
51 #include "esp_vfs_fat.h"
   #include "driver/sdmmc_host.h"
53 #include "driver/sdspi_host.h"
   #include "sdmmc_cmd.h"
55

57 //Biblioteca feita pra cuidar da ativa  o e transmiss o de dados BT/
   #include "Lib_bt_sink.h"
59
   //Defines do SD
61 #define PIN_NUM_MISO 33
   #define PIN_NUM_MOSI 26
63 #define PIN_NUM_CLK  21
   #define PIN_NUM_CS   13
65

67 #define BT_APP_EVT_STACK_UP 0

69 /* Ativa  o dos perfil usados */
   static void tarefa_de_inicializacao(uint16_t event, void *p_param);
71
   void SD_INI(void);
73
   void app_main(){
75
     //---------------------------- Inicializa os dados de calibra  o
        -----------------------------//
77   esp_err_t check = nvs_flash_init();
     if (check == ESP_ERR_NVS_NO_FREE_PAGES || check ==
        ESP_ERR_NVS_NEW_VERSION_FOUND) {
79     ESP_ERROR_CHECK(nvs_flash_erase());
       check = nvs_flash_init();
81   }
     ESP_ERROR_CHECK(check);//Checa o erro nos dados de calibra  o.
83
```

```
85      //------------------------------------ Configura  es do i2s
            ------------------------------------//
        i2s_config_t i2s_config = {
87
          .mode = I2S_MODE_MASTER | I2S_MODE_TX,                    //Somente
              transmiss o (TX)
89        .sample_rate = 44100,                                     //Frequencia
              de transmiss o
          .bits_per_sample = 16,                                    //Bits de
              resolu   o
91        .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,             //Usando os 2
              channels(L/R)
          .communication_format = I2S_COMM_FORMAT_PCM,              //Utiliza o
              PCM //Altered by Bruno T. and Gabriel C.
93        .dma_buf_count = 6,
          .dma_buf_len = 60,
95        .intr_alloc_flags = 0,                                    //
              Interrup   o padr o
          .tx_desc_auto_clear = true                                //Auto clear
              do tx ativado
97      };
        //------------------------------------ Instala o Driver do i2s
            ------------------------------------//
99      i2s_driver_install(0, &i2s_config, 0, NULL);
        printf("\n  \n Configurado I2S1\n");
101     //------------------------------------ Configura os pinos do i2s
            ------------------------------------//
        i2s_pin_config_t pin_config = {
103       .bck_io_num = 25, //Also tested with "2"//Altered by Bruno T. and
              Gabriel C.
          .ws_io_num = 26, //Also tested with "25"//Altered by Bruno T. and
              Gabriel C.
105       .data_out_num = 22, //Also tested with "12"//Altered by Bruno T. and
              Gabriel C.
          .data_in_num = -1
107     };
        //----------------------- Ativa o modulo "0" do i2s e seus pinos de
            saida ----------------------//
109     i2s_set_pin(0, &pin_config);

111     //Ativa o PDM do i2s
        //i2s_set_pdm_rx_down_sample(I2S_NUM_0,I2S_PDM_DSR_8S);
113


115
```

```
    //---------------------- Checa os erros da libera  o da memoria do
        bluetooth BLE ----------------//
117 //Chamar esp_bt_controller_mem_release bloqueia o modo bluetooth BLE.
    ESP_ERROR_CHECK(esp_bt_controller_mem_release(ESP_BT_MODE_BLE));
119


121
    //------------- Atribui as configura  es padr es do bletooth para a
        variavel bt_cfg ------------//
123 esp_bt_controller_config_t config_bt_ini =
        BT_CONTROLLER_INIT_CONFIG_DEFAULT();

125 //---------------------- Inicia as consfigura  es padr es e checa por
        erros ----------------------//
    if ((check = esp_bt_controller_init(&config_bt_ini)) != ESP_OK) {
127
      ESP_LOGE(AVRC_BT, "%s Falha na inicializa  o dos controles: %s\n",
          __func__, esp_err_to_name(check));
129   return;

131 }

133 //------------------------ Ativa o modo BT Classico e checa por erros
        -------------------------//
    if ((check = esp_bt_controller_enable(ESP_BT_MODE_CLASSIC_BT)) !=
        ESP_OK) {
135
      ESP_LOGE(AVRC_BT, "%s Falha na ativa  o do modulo: %s\n", __func__,
          esp_err_to_name(check));
137   return;

139 }

141 //------------------------------------ Inicia o Bluetooth
        -------------------------------------//
    if ((check = esp_bluedroid_init()) != ESP_OK) {
143
      ESP_LOGE(AVRC_BT, "%s Falha na iniciliza  o do bluedroid: %s\n",
          __func__, esp_err_to_name(check));
145   return;

147 }

149 //------------------------------------ Ativa o Bluetooth
        -------------------------------------//

151 if ((check = esp_bluedroid_enable()) != ESP_OK) {
```

```
153         ESP_LOGE(AVRC_BT, "%s Falha na ativa   o bluedroid: %s\n", __func__,
                esp_err_to_name(check));
            return;
155
        }
157


159
        //------------------------ Cria uma xtask do freertos para o bluetooth
            -------------------------//
161     tarefa_bt_inicio();

163     //-------------------------- Manda a tarefa de inicializa   o para ser
            executada ----------------//
        tarefa_envio_de_trf_bt(tarefa_de_inicializacao, BT_APP_EVT_STACK_UP,
            NULL, 0, NULL);
165


167


169 }//Final do void main

171


173


175 //----------------------------------------- Fun   es
        ------------------------------------------//

177 void bt_app_gap_cb(esp_bt_gap_cb_event_t event, esp_bt_gap_cb_param_t *
        param){
      switch (event) {
179     case ESP_BT_GAP_AUTH_CMPL_EVT: {
          if (param->auth_cmpl.stat == ESP_BT_STATUS_SUCCESS) {
181         ESP_LOGI(AVRC_BT, "Autenticado: %s", param->auth_cmpl.device_name
                );
            esp_log_buffer_hex(AVRC_BT, param->auth_cmpl.bda, ESP_BD_ADDR_LEN
                );// MOSTRA OS NUMEROS DO ID DO DISPOSITIVO CONECTADO
183       } else {
            ESP_LOGE(AVRC_BT, "Falha na autentica   o, status:%d", param->
                auth_cmpl.stat);
185       }
          break;
187     }

189     #if (CONFIG_BT_SSP_ENABLED == true)
```

```
        case ESP_BT_GAP_CFM_REQ_EVT:
191     ESP_LOGI(AVRC_BT, "ESP_BT_GAP_CFM_REQ_EVT Compare os Valores:: %d",
            param->cfm_req.num_val);
        esp_bt_gap_ssp_confirm_reply(param->cfm_req.bda, true);
193     break;
        case ESP_BT_GAP_KEY_NOTIF_EVT:
195     ESP_LOGI(AVRC_BT, "ESP_BT_GAP_KEY_NOTIF_EVT Codigo::%d", param->
            key_notif.passkey);
        break;
197     case ESP_BT_GAP_KEY_REQ_EVT:
        ESP_LOGI(AVRC_BT, "ESP_BT_GAP_KEY_REQ_EVT Entre com o Codigo:!");
199     break;
        #endif
201
        default: {
203       ESP_LOGI(AVRC_BT, "Evento: %d", event);
          break;
205     }
      }
207   return;
    }
209

211 // Tarefa de inicializa   o
    static void tarefa_de_inicializacao(uint16_t event, void *p_param){
213   ESP_LOGD(AVRC_BT, "%s Evetento %d", __func__, event);
      switch (event) {
215     case BT_APP_EVT_STACK_UP: {//Caso o evento seja definido como
            BT_APP_EVT_STACK_UP
        //------------------------------ Escolhe o nome do dispositivo
            ------------------------------//
217     char *nome_do_esp = "ESP32_Teste_TCC";//Altered by Bruno T. and
            Gabriel C.
        esp_bt_dev_set_device_name(nome_do_esp);
219     esp_bt_gap_register_callback(bt_app_gap_cb);

221     //--------------------------------- Inicializa o perfil A2DP
            ---------------------------------//
        esp_a2d_register_callback(&a2d_notification_callback);
223     esp_a2d_sink_register_data_callback(a2d_data_callback);
        esp_a2d_sink_init();
225
        //--------------------------------- Inicializa o modo AVRCP
            ---------------------------------//
227     esp_avrc_ct_init();
        esp_avrc_ct_register_callback(bt_app_rc_ct_cb);//CT = Esp32,
            control
```

```
229
        //------------------------------ Inicia o alvo do perfil AVRCP
            ------------------------------//
231     assert (esp_avrc_tg_init() == ESP_OK);// Tg = Celular, Target
        esp_avrc_tg_register_callback(bt_app_rc_tg_cb);
233
        //--------------- Define o controle de volume como opera  o do
            perfil AVRCP ---------------//
235     esp_avrc_rn_evt_cap_mask_t evt_set = {0};
        esp_avrc_rn_evt_bit_mask_operation(ESP_AVRC_BIT_MASK_OP_SET, &
            evt_set, ESP_AVRC_RN_VOLUME_CHANGE);
237     assert(esp_avrc_tg_set_rn_evt_cap(&evt_set) == ESP_OK);

239     //-------- Define como um dispositivo conectavel e visivel para
            qualquer aparelho ---------//
        esp_bt_gap_set_scan_mode(ESP_BT_CONNECTABLE,
            ESP_BT_GENERAL_DISCOVERABLE);
241
        break;
243   }
      default://Caso o evento n o esteja definido como BT_APP_EVT_STACK_UP
245   ESP_LOGE(AVRC_BT, "%s Evento n o registrado %d", __func__, event);
      break;
247  }
 }
```

A.2 LIB_BT_SINK.H

```
1
 /*
3  This example code is in the Public Domain (or CC0 licensed, at your
       option.)

5  Unless required by applicable law or agreed to in writing, this
   software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
7  CONDITIONS OF ANY KIND, either express or implied.
*/
9 #include <stdio.h>
  #include <stdint.h>
11 #include <stdbool.h>
  #include <stdlib.h>
13 #include <string.h>
  #include <sys/unistd.h>
15 #include <sys/stat.h>

17 #include "esp_a2dp_api.h"
```

```
#include "esp_avrc_api.h"
#include "esp_bt_main.h"
#include "esp_bt_device.h"
#include "esp_gap_bt_api.h"
#include "esp_log.h"

#include "freertos/FreeRTOS.h"
#include "freertos/FreeRTOSConfig.h"
#include "freertos/xtensa_api.h"
#include "freertos/task.h"
#include "freertos/ringbuf.h"
#include "freertos/queue.h"
#include "driver/i2s.h"

#include <sys/unistd.h>
#include <sys/stat.h>
#include "esp_err.h"
#include "esp_vfs_fat.h"
#include "driver/sdmmc_host.h"
#include "driver/sdspi_host.h"
#include "sdmmc_cmd.h"


#include "sys/lock.h"


#define div 4 //Numero no qual e dividido a quantidade de dados(bytes) trabalhados no troca de variavel do vetor bt para o vetor teste


#define TAREFA_BT                       "TAREFA_BT"
#define ENVIO_DE_TAREFA_BT              (0x01) //Antigo BT_APP_SIG_WORK_DISPATCH (0x01)
#define AVRC_BT                         "AVRC" //Antigo BT_AV_TAG "BT_AV"
#define TG_RC                           "TGRC" //Antigo BT_RC_TG_TAG "RCTG"
#define CT_RC                           "CTRC" //Antigo BT_RC_CT_TAG "RCCT"


// AVRCP EVENTOS
#define APP_RC_CT_TL_GET_CAPS           (0)
#define APP_RC_CT_TL_GET_META_DATA      (1)
#define APP_RC_CT_TL_RN_TRACK_CHANGE    (2)
#define APP_RC_CT_TL_RN_PLAYBACK_CHANGE (3)
#define APP_RC_CT_TL_RN_PLAY_POS_CHANGE (4)
```

```
61
   //------------------------ Declaracao das variaveis e parametros
      ------------------------------//
63 static uint32_t s_pacotes_enviados = 0;
   static esp_a2d_audio_state_t s_audio_state = ESP_A2D_AUDIO_STATE_STOPPED;
65 static const char *s_a2d_conn_state_str[] = {"Disconnected", "Connecting",
       "Connected", "Disconnecting"};
   static const char *s_a2d_audio_state_str[] = {"Suspended", "Stopped", "
      Started"};
67 static esp_avrc_rn_evt_cap_mask_t s_avrc_peer_rn_cap;
   static _lock_t s_volume_lock;
69 static uint8_t s_volume = 5;
   static bool s_volume_notify;
71
   //Paremetro para a tarefa bt enviada
73 typedef void (* bt_app_cb_t) (uint16_t event, void *param);

75 /* MSG a ser enviada */
   typedef struct {
77  uint16_t              sig;      // Sianl para as tarefas
    uint16_t              event;    // Message event id
79  bt_app_cb_t           cb;       // Contexto para o Callback
    void                  *param;   // Parametros
81 } bt_app_msg_t;

83 //Paremetro para a copia do Callback.
   typedef void (* bt_app_copy_cb_t) (bt_app_msg_t *msg, void *p_dest, void *
      p_src);
85
   static xQueueHandle s_tarefa_bt_queue = NULL; //ANTIGO s_bt_app_task_queue
87 static xTaskHandle s_envio_tarefa_bt = NULL;
   static xTaskHandle s_bt_i2s_task_handle = NULL; //antigo
      s_bt_i2s_task_handle
89 static RingbufHandle_t s_ringbuf_i2s = NULL;;


91
   //---------------------------------- Declaracao de funcoes
      ------------------------------------//
93
   //Envio de tarefas(configuracoes). Fica em um while(1) esperando dados
      tarefas para serem executadas.
95 bool tarefa_envio_de_trf_bt(bt_app_cb_t p_cback, uint16_t event, void *
      p_params, int param_len, bt_app_copy_cb_t p_copy_cback); // antigo
      bt_app_work_dispatch
   void tarefa_bt_inicio(void);
97 static bool envio_msg_bt(bt_app_msg_t *msg);
```

```
    static void tarefa_enviada(bt_app_msg_t *msg);
99  static void tarefa_rec_trf_bt(void *arg);
    void tarefa_rec_trf_bt_shut_down(void);
101
    void filtro(uint8_t *data,uint8_t * teste1,size_t tamanho);
103 static void bt_escreve_no_i2s(void *arg);
    void bt_i2s_start(void);
105 void bt_i2s_end(void);
    size_t envia_ringbuf(const uint8_t *data, size_t size);
107
    void a2d_notification_callback(esp_a2d_cb_event_t event,
       esp_a2d_cb_param_t *param);
109 void a2d_data_callback(const uint8_t *data, uint32_t len);
    void bt_app_alloc_meta_buffer(esp_avrc_ct_cb_param_t *param);
111 void bt_app_rc_ct_cb(esp_avrc_ct_cb_event_t event, esp_avrc_ct_cb_param_t
       *param);
    void bt_app_rc_tg_cb(esp_avrc_tg_cb_event_t event, esp_avrc_tg_cb_param_t
       *param);
113 static void tarefa_a2d_evento(uint16_t event, void *p_param);

115 static void bt_av_new_track(void);
    static void bt_av_playback_changed(void);
117 static void bt_av_play_pos_changed(void);
    void bt_av_notify_evt_handler(uint8_t event_id, esp_avrc_rn_param_t *
       event_parameter);
119 static void bt_av_hdl_avrc_ct_evt(uint16_t event, void *p_param);
    static void volume_set_by_controller(uint8_t volume);
121 static void volume_set_by_local_host(uint8_t volume);
    static void bt_av_hdl_avrc_tg_evt(uint16_t event, void *p_param);
123
    //------------------------------------ Envio de Tarefas BT
       ---------------------------------//
125
    //Cria uma tarefa que ira cuidar da troca de dados bt
127 void tarefa_bt_inicio(void){
     s_tarefa_bt_queue = xQueueCreate(10, sizeof(bt_app_msg_t));
129  xTaskCreatePinnedToCore(tarefa_rec_trf_bt, "Tarefa_BT", 3072, NULL,
        configMAX_PRIORITIES - 3, &s_envio_tarefa_bt,1);

131  return;
    }
133
    //Funcao que trabalha com a Tarefa que se deseja realizar seja executada
135 bool tarefa_envio_de_trf_bt(bt_app_cb_t p_cback, uint16_t event, void *
       p_params, int param_len, bt_app_copy_cb_t p_copy_cback){
     ESP_LOGD(TAREFA_BT , "%s Evento: 0x%x, Param_len: %d", __func__, event,
        param_len);
```

```
137
    bt_app_msg_t msg;
139 memset(&msg, 0, sizeof(bt_app_msg_t));

141 //Tarefa a ser enviada
    msg.sig = ENVIO_DE_TAREFA_BT;
143 msg.event = event;// Dado que da o estado da conecao, usado para navegar
        entre os menus
    msg.cb = p_cback;// dado
145

147 //Checa se contem uma tarefa e se tiver, manda ela para o tarefa_rec
    if (param_len == 0) {
149   return envio_msg_bt(&msg);
    } else if (p_params && param_len > 0) {
151   if ((msg.param = malloc(param_len)) != NULL) {
        memcpy(msg.param, p_params, param_len); // Checa se o dispositivo
            conectado requisitou uma copia do callback.
153     if (p_copy_cback) {
          p_copy_cback(&msg, msg.param, p_params);
155     }
        return envio_msg_bt(&msg);
157   }
    }
159
    return false;
161 }

163 //Funcao que envia a Tarefa que se deseja realizar para o while(1)
    static bool envio_msg_bt(bt_app_msg_t *msg){
165  if (msg == NULL) {
       return false;
167  }

169  if (xQueueSend(s_tarefa_bt_queue, msg, 10 / portTICK_RATE_MS) != pdTRUE)
         {
       ESP_LOGE(TAREFA_BT , "%s Falha no envio do xQueue", __func__);
171    return false;
     }
173  return true;
    }
175
    //Funcao pra mandar a tarefa(msg)_bt para a tarefa ser realizada
177 static void tarefa_enviada(bt_app_msg_t *msg){
     if (msg->cb) {
179    msg->cb(msg->event, msg->param);
     }
```

```
181 }

183 //While(1)
static void tarefa_rec_trf_bt(void *arg){// Tarefa que espera receber uma
    tarefa
185  bt_app_msg_t msg;
 while(1) {
187    if (pdTRUE == xQueueReceive(s_tarefa_bt_queue, &msg, (portTickType)
        portMAX_DELAY)) {
      ESP_LOGD(TAREFA_BT, "Tarefa: :%s, sig 0x%x, 0x%x", __func__, msg.sig
          , msg.event);
189       switch (msg.sig) {
          case ENVIO_DE_TAREFA_BT:
191          tarefa_enviada(&msg);
          break;
193          default:
          ESP_LOGW(TAREFA_BT, "%s, Evento nao registrado sig: %d", __func__,
              msg.sig);
195          break;
       }
197       if (msg.param) {
          free(msg.param);
199       }
    }//Final do while(1)
201  }
}
203
//Funcao que Deleta o tarefa_rec_trf_bt
205 void tarefa_rec_trf_bt_shut_down(void){
 if (s_envio_tarefa_bt) {
207    vTaskDelete(s_envio_tarefa_bt);
    s_envio_tarefa_bt = NULL;
209  }
 if (s_tarefa_bt_queue) {
211    vQueueDelete(s_tarefa_bt_queue);
    s_tarefa_bt_queue = NULL;
213  }
}
215


217
//------------------------------------ I2S
    ---------------------------------//
219


221 //Filtro
```

```
void filtro(uint8_t *data,uint8_t * teste1,size_t tamanho){//Funcao criada
    por Bruno T. e Gabriel D.
223
 //Declara as variaveis para serem trabalhadas
225 short dado_mandar_dir = 0;
 short dado_mandar_esq = 0;
227
 // Trocar variavel malloc Filtro
229 for(int i=0;i<tamanho/div;i++){//Troca os dois primeiros Bytes entre o
    vetor de bt e o de teste e corrige o problema do erro de lado de audio
    .
   //pds dir
231   dado_mandar_dir += data[i*4 + 2];
   dado_mandar_dir = dado_mandar_dir <<8;
233   dado_mandar_dir += data[i*4 + 3];
   dado_mandar_dir = dado_mandar_dir >> 1;
235   dado_mandar_dir = (uint8_t) dado_mandar_dir;

237   //pds esq
   dado_mandar_esq += data[i*4];
239   dado_mandar_esq = dado_mandar_esq <<8;
   dado_mandar_esq += data[i*4 + 1];
241   dado_mandar_esq = dado_mandar_esq >> 1;
   dado_mandar_esq = (uint8_t) dado_mandar_esq;
243


245   teste1[i*div] = dado_mandar_dir << 8;
   //teste1[i*div] = 0;
247   teste1[i*div + 1] = dado_mandar_dir; //Data[3]; Segundo byte de dado a
      ser mandado pro lado direito direito


249   teste1[i*div + 2] = dado_mandar_esq << 8;
   //teste1[i*div + 2] = 0;
251   teste1[i*div + 3] = dado_mandar_esq; //Data[1]; Segundo byte de dado a
      ser mandado pro lado esquerdo


253
 }
255

}
257
void filtro2(uint8_t *data,short *teste1,size_t tamanho){//Funcao criada
    por Bruno T. e Gabriel D.
259
 //Declara as variaveis para serem trabalhadas
261 short dado_mandar_dir = 0;
 short dado_mandar_esq = 0;
```

```
263   float gain_vol = 0;

265   // Trocar variavel malloc Filtro
      for(int i=0;i<tamanho/4;i++){//Troca os dois primeiros Bytes entre o
          vetor de bt e o de teste e corrige o problema do erro de lado de audio
          .
267     //Sinal do lado direito
        dado_mandar_dir = data[i*4 + 3];
269     dado_mandar_dir = dado_mandar_dir << 8;
        dado_mandar_dir += data[i*4 + 2];
271
        dado_mandar_dir = dado_mandar_dir >> 2;// por teste se move o dado de
            16bits recebidos 2 bits pro lado, fazendo assim a precisao cair pra
            14bits
273

275     //teste1[i*2] = (short) dado_mandar_dir * 0.8; //sinal com ganho fixo

277     //Sinal do lado esquerdo
        dado_mandar_esq = data[i*4 + 1];
279     dado_mandar_esq = dado_mandar_esq << 8;
        dado_mandar_esq += data[i*4];
281
        dado_mandar_esq = dado_mandar_esq >> 2;// por teste se move o dado de
            16bits recebidos 2 bits pro lado, fazendo assim a precisao cair pra
            14bits
283
        //teste1[i*2+1] = (short) dado_mandar_esq * 0.8;//sinal com ganho fixo
285

287     gain_vol = 0.0110 * s_volume + 0.2;// Valor do multiplicador maximo =
            1.3, minimo 0.2
        dado_mandar_dir = dado_mandar_dir * gain_vol;
289     dado_mandar_esq = dado_mandar_esq * gain_vol;

291     teste1[i*2] = (short) dado_mandar_dir;
        teste1[i*2+1] = (short) dado_mandar_esq;
293


295


297
    }
299 }

301
```

```
303 static void bt_escreve_no_i2s(void *arg){
    uint8_t *data = NULL; //ponteiro que recebe o endereco em uint8_t do dado
        inicial
305 size_t item_size = 0;
    size_t bytes_written = 0;
307

309 for (;;) {

311   //Recebe o dado PCM do Bluetooth
      data = (uint8_t *)xRingbufferReceive(s_ringbuf_i2s, &item_size, (
          portTickType)portMAX_DELAY);
313   uint8_t *teste = malloc(item_size);//Cria um ponteiro de tamanho
          itemsize para trabalahr com
      short *teste2 = malloc(item_size*sizeof(short));// cria um malloc do
          tipo short  de tamanho itemsaize
315


317


319
      if (item_size != 0){
321

323     filtro2(data,teste2,item_size);


325
        //Escreve no i2s
327     i2s_write(I2S_NUM_0, teste2, item_size, &bytes_written, portMAX_DELAY
            );// Escreve o endereco a ser transformado em i2s


329
        vRingbufferReturnItem(s_ringbuf_i2s,(void *)data);
331
        //Free malock teste
333     free(teste);
        free(teste2);
335
      }
337


339 }
}
341
void bt_i2s_start(void){
343 s_ringbuf_i2s = xRingbufferCreate(8 * 1024, RINGBUF_TYPE_BYTEBUF);
    if(s_ringbuf_i2s == NULL){
```

```
345     return;
    }
347
    xTaskCreate(bt_escreve_no_i2s, "BtI2ST", 1024, NULL, configMAX_PRIORITIES
        - 3, &s_bt_i2s_task_handle);
349   return;
  }
351
  void bt_i2s_end(void){
353   if (s_bt_i2s_task_handle) {
      vTaskDelete(s_bt_i2s_task_handle);
355     s_bt_i2s_task_handle = NULL;
    }
357
    if (s_ringbuf_i2s) {
359     vRingbufferDelete(s_ringbuf_i2s);
      s_ringbuf_i2s = NULL;
361   }
  }
363
  size_t envia_ringbuf(const uint8_t *data, size_t size){//Envia o ponteiro
      do som para ser interpretado pelo i2s
365   BaseType_t done = xRingbufferSend(s_ringbuf_i2s, (void *)data, size, (
        portTickType)portMAX_DELAY);
    if(done){
367     return size;
    } else {
369     return 0;
    }
371 }

373 //------------------------------------ A2DP ---
      ----------------------------------//
  void a2d_notification_callback(esp_a2d_cb_event_t event,
      esp_a2d_cb_param_t *param){
375   switch (event) {
      case ESP_A2D_CONNECTION_STATE_EVT:
377     case ESP_A2D_AUDIO_STATE_EVT:
      case ESP_A2D_AUDIO_CFG_EVT: {
379       tarefa_envio_de_trf_bt(tarefa_a2d_evento, event, param, sizeof(
            esp_a2d_cb_param_t), NULL);
        break;
381     }
      default:
383     ESP_LOGE(AVRC_BT, "Invalid A2DP event: %d", event);
      break;
385   }
```

```
 }

 void a2d_data_callback(const uint8_t *data, uint32_t len){
  envia_ringbuf(data, len);
  if (++s_pacotes_enviados % 100 == 0) {
  ESP_LOGI(AVRC_BT, "Audio packet count %u", s_pacotes_enviados);//imprime
      a cada 100 pacotes de audio mandados


 }
}

static void tarefa_a2d_evento(uint16_t event, void *p_param){
ESP_LOGD(AVRC_BT, "%s evt %d", __func__, event);
esp_a2d_cb_param_t *a2d = NULL;
switch (event) {
  case ESP_A2D_CONNECTION_STATE_EVT: {
    a2d = (esp_a2d_cb_param_t *)(p_param);

    uint8_t *bda = a2d->conn_stat.remote_bda;
    ESP_LOGI(AVRC_BT, "A2DP connection state: %s, [%02x:%02x:%02x:%02x:%02x
        :%02x]",
    s_a2d_conn_state_str[a2d->conn_stat.state], bda[0], bda[1], bda[2], bda
        [3], bda[4], bda[5]);
    if (a2d->conn_stat.state == ESP_A2D_CONNECTION_STATE_DISCONNECTED) {
      esp_bt_gap_set_scan_mode(ESP_BT_CONNECTABLE,
          ESP_BT_GENERAL_DISCOVERABLE);
      bt_i2s_end();
    } else if (a2d->conn_stat.state == ESP_A2D_CONNECTION_STATE_CONNECTED){
      esp_bt_gap_set_scan_mode(ESP_BT_NON_CONNECTABLE,
          ESP_BT_NON_DISCOVERABLE);
      bt_i2s_start();
    }
    break;
  }
  case ESP_A2D_AUDIO_STATE_EVT: {
    a2d = (esp_a2d_cb_param_t *)(p_param);
    ESP_LOGI(AVRC_BT, "A2DP Audio: %s", s_a2d_audio_state_str[a2d->
        audio_stat.state]);
    s_audio_state = a2d->audio_stat.state;
    if (ESP_A2D_AUDIO_STATE_STARTED == a2d->audio_stat.state) {
      s_pacotes_enviados = 0;

    }
    break;
  }
  case ESP_A2D_AUDIO_CFG_EVT: {
    a2d = (esp_a2d_cb_param_t *)(p_param);
```

```
427     ESP_LOGI(AVRC_BT, "A2DP Configuracao da stream de audio, codec type %d"
            , a2d->audio_cfg.mcc.type);
        // for now only SBC stream is supported
429     if (a2d->audio_cfg.mcc.type == ESP_A2D_MCT_SBC) {
          int sample_rate = 16000;
431       char oct0 = a2d->audio_cfg.mcc.cie.sbc[0];
          if (oct0 & (0x01 << 6)) {
433         sample_rate = 32000;
          } else if (oct0 & (0x01 << 5)) {
435         sample_rate = 44100;
          } else if (oct0 & (0x01 << 4)) {
437         sample_rate = 48000;
          }
439
          i2s_set_clk(I2S_NUM_0, sample_rate, I2S_BITS_PER_SAMPLE_16BIT,
              I2S_CHANNEL_STEREO );
441       printf("\n  \n Configurado I2S\n");

443       ESP_LOGI(AVRC_BT, "Configure audio player %x-%x-%x-%x",
          a2d->audio_cfg.mcc.cie.sbc[0],
445       a2d->audio_cfg.mcc.cie.sbc[1],
          a2d->audio_cfg.mcc.cie.sbc[2],
447       a2d->audio_cfg.mcc.cie.sbc[3]);
          ESP_LOGI(AVRC_BT, "Audio player configured, sample rate=%d",
              sample_rate);
449     }
        break;
451   }
      default:
453   ESP_LOGE(AVRC_BT, "%s unhandled evt %d", __func__, event);
      break;
455 }
    }
457
    //---------------------------------- AVRCP
        -------------------------------------//
459 void bt_app_alloc_meta_buffer(esp_avrc_ct_cb_param_t *param){
    esp_avrc_ct_cb_param_t *rc = (esp_avrc_ct_cb_param_t *)(param);
461 uint8_t *attr_text = (uint8_t *) malloc (rc->meta_rsp.attr_length + 1);
    memcpy(attr_text, rc->meta_rsp.attr_text, rc->meta_rsp.attr_length);
463 attr_text[rc->meta_rsp.attr_length] = 0;
    rc->meta_rsp.attr_text = attr_text;
465 }

467 void bt_app_rc_ct_cb(esp_avrc_ct_cb_event_t event, esp_avrc_ct_cb_param_t *
        param){
    switch (event) {
```

```
469     case ESP_AVRC_CT_METADATA_RSP_EVT:
        bt_app_alloc_meta_buffer(param);
471     case ESP_AVRC_CT_CONNECTION_STATE_EVT:
        case ESP_AVRC_CT_PASSTHROUGH_RSP_EVT:
473     case ESP_AVRC_CT_CHANGE_NOTIFY_EVT:
        case ESP_AVRC_CT_REMOTE_FEATURES_EVT:
475     case ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT: {
          tarefa_envio_de_trf_bt(bt_av_hdl_avrc_ct_evt, event, param, sizeof(
              esp_avrc_ct_cb_param_t), NULL);
477       break;
        }
479     default:
        ESP_LOGE(CT_RC, "Invalid AVRC event: %d", event);
481     break;
    }
483 }

485 void bt_app_rc_tg_cb(esp_avrc_tg_cb_event_t event, esp_avrc_tg_cb_param_t *
        param){
    switch (event) {
487     case ESP_AVRC_TG_CONNECTION_STATE_EVT:
        case ESP_AVRC_TG_REMOTE_FEATURES_EVT:
489     case ESP_AVRC_TG_PASSTHROUGH_CMD_EVT:
        case ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT:
491     case ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT:
        tarefa_envio_de_trf_bt(bt_av_hdl_avrc_tg_evt, event, param, sizeof(
            esp_avrc_tg_cb_param_t), NULL);
493     break;
        default:
495     ESP_LOGE(TG_RC, "Invalid AVRC event: %d", event);
        break;
497 }
    }
499

    static void bt_av_new_track(void){
501 // request metadata
    uint8_t attr_mask = ESP_AVRC_MD_ATTR_TITLE | ESP_AVRC_MD_ATTR_ARTIST |
        ESP_AVRC_MD_ATTR_ALBUM | ESP_AVRC_MD_ATTR_GENRE;
503 esp_avrc_ct_send_metadata_cmd(APP_RC_CT_TL_GET_META_DATA, attr_mask);

505 // register notification if peer support the event_id
    if (esp_avrc_rn_evt_bit_mask_operation(ESP_AVRC_BIT_MASK_OP_TEST, &
        s_avrc_peer_rn_cap,
507 ESP_AVRC_RN_TRACK_CHANGE)) {
      esp_avrc_ct_send_register_notification_cmd(APP_RC_CT_TL_RN_TRACK_CHANGE,
          ESP_AVRC_RN_TRACK_CHANGE, 0);
509 }
```

```
}

static void bt_av_playback_changed(void){
if (esp_avrc_rn_evt_bit_mask_operation(ESP_AVRC_BIT_MASK_OP_TEST, &
    s_avrc_peer_rn_cap,
ESP_AVRC_RN_PLAY_STATUS_CHANGE)) {
  esp_avrc_ct_send_register_notification_cmd(
      APP_RC_CT_TL_RN_PLAYBACK_CHANGE, ESP_AVRC_RN_PLAY_STATUS_CHANGE, 0);
}
}

static void bt_av_play_pos_changed(void){
if (esp_avrc_rn_evt_bit_mask_operation(ESP_AVRC_BIT_MASK_OP_TEST, &
    s_avrc_peer_rn_cap,
ESP_AVRC_RN_PLAY_POS_CHANGED)) {
  esp_avrc_ct_send_register_notification_cmd(
      APP_RC_CT_TL_RN_PLAY_POS_CHANGE, ESP_AVRC_RN_PLAY_POS_CHANGED, 10);
}
}

void bt_av_notify_evt_handler(uint8_t event_id, esp_avrc_rn_param_t *
    event_parameter){
switch (event_id) {
  case ESP_AVRC_RN_TRACK_CHANGE:
  bt_av_new_track();
  break;
  case ESP_AVRC_RN_PLAY_STATUS_CHANGE:
  ESP_LOGI(AVRC_BT, "Playback status changed: 0x%x", event_parameter->
      playback);
  bt_av_playback_changed();
  break;
  case ESP_AVRC_RN_PLAY_POS_CHANGED:
  ESP_LOGI(AVRC_BT, "Play position changed: %d-ms", event_parameter->
      play_pos);
  bt_av_play_pos_changed();
  break;
}
}

static void bt_av_hdl_avrc_ct_evt(uint16_t event, void *p_param){
ESP_LOGD(CT_RC, "%s evt %d", __func__, event);
esp_avrc_ct_cb_param_t *rc = (esp_avrc_ct_cb_param_t *)(p_param);
switch (event) {
  case ESP_AVRC_CT_CONNECTION_STATE_EVT: {
    uint8_t *bda = rc->conn_stat.remote_bda;
    ESP_LOGI(CT_RC, "AVRC conn_state evt: state %d, [%02x:%02x:%02x:%02x
        :%02x:%02x]",
```

```
549     rc->conn_stat.connected, bda[0], bda[1], bda[2], bda[3], bda[4], bda
            [5]); // BDA FORMA O ID DO DISPOSITIVO CONECTADO

551     printf("\n ->>>Comando ESP_AVRC_CT_CONNECTION_STATE_EVT chego \n ");

553
        if (rc->conn_stat.connected) {
555       // get remote supported event_ids of peer AVRCP Target
          esp_avrc_ct_send_get_rn_capabilities_cmd(APP_RC_CT_TL_GET_CAPS);
557     } else {
          // clear peer notification capability record
559       s_avrc_peer_rn_cap.bits = 0;
        }
561     break;
      }
563   case ESP_AVRC_CT_PASSTHROUGH_RSP_EVT: {
        ESP_LOGI(CT_RC , "AVRC passthrough rsp: key_code 0x%x, key_state %d",
            rc->psth_rsp.key_code, rc->psth_rsp.key_state);
565     break;
      }
567   case ESP_AVRC_CT_METADATA_RSP_EVT: {

569     ESP_LOGI(CT_RC, "AVRC metadata rsp: attribute id 0x%x, %s", rc->
            meta_rsp.attr_id , rc->meta_rsp.attr_text); //SE rc->meta_rsp.attr_id
             = 0x1->NOME DA MUSICA; 0x2 ARTISTA; 0x4 ALBUM
        free(rc->meta_rsp.attr_text);
571
        break;
573   }
      case ESP_AVRC_CT_CHANGE_NOTIFY_EVT: {
575     ESP_LOGI(CT_RC, "AVRC event notification: %d", rc->change_ntf.event_id)
            ;
        bt_av_notify_evt_handler(rc->change_ntf.event_id , &rc->change_ntf.
            event_parameter);
577     break;
      }
579   case ESP_AVRC_CT_REMOTE_FEATURES_EVT: {
        ESP_LOGI(CT_RC, "AVRC remote features %x, TG features %x", rc->
            rmt_feats.feat_mask, rc->rmt_feats.tg_feat_flag);
581     break;
      }
583   case ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT: {
        ESP_LOGI(CT_RC, "remote rn_cap: count %d, bitmask 0x%x", rc->
            get_rn_caps_rsp.cap_count,
585     rc->get_rn_caps_rsp.evt_set.bits);
        s_avrc_peer_rn_cap.bits = rc->get_rn_caps_rsp.evt_set.bits;
587     bt_av_new_track();
```

```
        bt_av_playback_changed();
589     bt_av_play_pos_changed();
        break;
591   }
      case ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT:{
593
        ESP_LOGI(CT_RC, "AVRC set Abs volume do CT: %d%%", (uint8_t) rc->
            set_volume_rsp.volume * 100/ 0x7f);
595
        break;
597   }

599   default:
      ESP_LOGE(CT_RC, "%s unhandled evt %d", __func__, event);
601   break;
  }
603 }

605 static void volume_set_by_controller(uint8_t volume){

607


609 ESP_LOGI(TG_RC, "Volume is set by remote controller %d%%\n", (uint32_t)
      volume* 100/ 0x7f);
  _lock_acquire(&s_volume_lock);
611 s_volume = volume * 100/ 0x7f;
  _lock_release(&s_volume_lock);
613
  //Sem essa parte no original
615 /*if (s_volume_notify) {// ativado quando s_volume_nitify for true
    esp_avrc_rn_param_t rn_param;
617   rn_param.volume = s_volume;
    printf("\n MANDOU\n ");
619   esp_avrc_tg_send_rn_rsp(ESP_AVRC_RN_VOLUME_CHANGE,
        ESP_AVRC_RN_RSP_CHANGED, &rn_param);
    s_volume_notify = false;
621 }*/


623


625

  }
627
  static void volume_set_by_local_host(uint8_t volume){//Funcao que muda o
      volume no esp. Da pra alterar e mudar pra botao.(Variavel s_volume da o
      volume atual)
629
```

```
ESP_LOGI(TG_RC, "Volume is set locally to: %d%%", (uint32_t)volume * 100 /
    0x7f);
_lock_acquire(&s_volume_lock);
s_volume = volume;
_lock_release(&s_volume_lock);

if (s_volume_notify) {// ativado quando s_volume_nitify for true
  esp_avrc_rn_param_t rn_param;
  rn_param.volume = s_volume;
  printf("\n MANDOU\n ");
  esp_avrc_tg_send_rn_rsp(ESP_AVRC_RN_VOLUME_CHANGE,
      ESP_AVRC_RN_RSP_CHANGED, &rn_param);
  s_volume_notify = false;
}
}

static void bt_av_hdl_avrc_tg_evt(uint16_t event, void *p_param){
ESP_LOGD(TG_RC, "%s evt %d", __func__, event);
esp_avrc_tg_cb_param_t *rc = (esp_avrc_tg_cb_param_t *)(p_param);
switch (event) {
  case ESP_AVRC_TG_CONNECTION_STATE_EVT: {
    uint8_t *bda = rc->conn_stat.remote_bda;
    ESP_LOGI(TG_RC, "AVRC conn_state evt: state %d, [%02x:%02x:%02x:%02x
        :%02x:%02x]",
    rc->conn_stat.connected, bda[0], bda[1], bda[2], bda[3], bda[4], bda
        [5]);
    break;
  }
  case ESP_AVRC_TG_PASSTHROUGH_CMD_EVT: {
    ESP_LOGI(TG_RC, "AVRC passthrough cmd: key_code 0x%x, key_state %d", rc
        ->psth_cmd.key_code, rc->psth_cmd.key_state);
    break;
  }
  case ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT: {
    ESP_LOGI(TG_RC, "AVRC set absolute volume: %d%%", (uint8_t)rc->
        set_abs_vol.volume * 100/ 0x7f);
    volume_set_by_controller(rc->set_abs_vol.volume);

    break;
  }
  case ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT: {
    ESP_LOGI(TG_RC, "AVRC register event notification: %d, param: 0x%x", rc
        ->reg_ntf.event_id, rc->reg_ntf.event_parameter);
    if (rc->reg_ntf.event_id == ESP_AVRC_RN_VOLUME_CHANGE) {
      s_volume_notify = true;// ativacao do software do
          volume_set_by_local_host para ele colcoar o volume indicado no
          rn_param.volume
```

```
        esp_avrc_rn_param_t rn_param;
669     rn_param.volume = s_volume;
        esp_avrc_tg_send_rn_rsp(ESP_AVRC_RN_VOLUME_CHANGE,
            ESP_AVRC_RN_RSP_INTERIM, &rn_param);
671   }
      break;
673   }
      case ESP_AVRC_TG_REMOTE_FEATURES_EVT: {
675     ESP_LOGI(TG_RC, "AVRC remote features %x, CT features %x", rc->
          rmt_feats.feat_mask, rc->rmt_feats.ct_feat_flag);
      break;
677   }
      default:
679   ESP_LOGE(TG_RC, "%s unhandled evt %d", __func__, event);
      break;
681 }
    }
```

APÊNDICE B – FIRMWARE RF INSERIDO NO ESP32

B.1 CÓDIGO DO RECEPTOR COM RFM69HW

/* RadioLib RF69 Receive Example https://jgromes.github.io/RadioLib/ */

```
// include the library
#include <RadioLib.h>
#define PA4 4
#define PA8 2
#define PA9 3
#define PA12 12
// NSS pin:   10 PA4      de acordo com biblioteca RFM69
// DIO0 pin:  2  PA8
// DIO1 pin:  3  PA9
RF69 rf = new Module(PA4, PA8, PA9); //stm32


void setup() {
  Serial.begin(9600);
  pinMode(PC12, OUTPUT);
  delay(100);
  digitalWrite(PC12, LOW);

  // initialize RF69 with default settings
  Serial.print(F("Inicializa   o Bem-sucedia..."));
  // carrier frequency:                   434.0 MHz
  // bit rate:                            48.0 kbps
  // Rx bandwidth:                        125.0 kHz
  // frequency deviation:                 50.0 kHz
  // output power:                        13 dBm
  // sync word:                           0x2D01
  int state = rf.begin(915.0);
  if (state == ERR_NONE) {
    Serial.println(F("Configurando dados do receptor...."));
  } else {
    Serial.print(F("failed, code "));
    Serial.println(state);
    while (true);
  }
}

void loop() {
  Serial.println(F("Aguardando dados..."));

  // you can receive data as an Arduino String
```

```
41    bit LED;
      int state = rf.receive(LED);
43
       if (state == ERR_NONE)
45       {
          Serial.println(LED);
47        Serial.println(F("Dado recebido."));
          digitalWrite(PC12, !digitalRead(PC12));
49       }

51 }
```

B.2 CÓDIGO TRANSMISSOR RFM69HW

/* RadioLib RF69 Transmit Example https://jgromes.github.io/RadioLib/ */

```
   // include the library
2  #include <RadioLib.h>
   #define PA4 4
4  #define PA8 2
   #define PA9 3
6  #define PA12 12

8  // NSS pin:   10 PA4      de acordo com biblioteca RFM69
   // DIO0 pin:  2  PA8
10 // DIO1 pin:  3  PA9
   RF69 rf = new Module(PA4, PA8, PA9);
12
   void setup() {
14    Serial.begin(9600);
      pinMode(PC13, OUTPUT);
16    delay(100);
      digitalWrite(PC12, HIGH);
18    delay(1000);
      digitalWrite(PC12, LOW);
20
     // initialize RF69 with default settings
22   //Serial.print(F("[RF69] Initializing      "));
     // carrier frequency:                   434.0 MHz
24   // bit rate:                            48.0 kbps
     // Rx bandwidth:                        125.0 kHz
26   // frequency deviation:                 50.0 kHz
     // output power:                        13 dBm
28   // sync word:                           0x2D01
     int state = rf.begin(915.0);
30   if (state == ERR_NONE) {
        Serial.println(F("Iniciando o Transmissor RFM...."));
```

```
32    } else {
        Serial.print(F("failed, code "));
34      Serial.println(state);
      while (true);
36    }
}
38
void loop() {
40    Serial.println(F("Transmissor OK.... Transmitando ....."));
    bit LED=1;
42
    int state = rf.transmit(LED);
44
      if (state == ERR_NONE) {
46    // the packet was successfully transmitted
        Serial.println(F("Dado transmitido!"));
48      digitalWrite(PC12, HIGH);

50    } else if (state == ERR_PACKET_TOO_LONG) {

52        Serial.println(F("Timeout!"));

54    } else {
      // some other error occurred
56      Serial.print(F("failed, code "));
        Serial.println(state);
58
    }
60
    // wait for a second before transmitting again
62  delay(500);
    digitalWrite(PC12, LOW);
64  delay(500);
}
```