

Especificação e Validação do Protocolo de Comunicação DCP

Egídio Neto Alves de Araújo



CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2025

Egídio Neto Alves de Araújo

Especificação e Validação do Protocolo de Comunicação DCP

Monografia apresentada ao curso Engenharia de Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em
Engenharia de Computação.

Orientador: Prof. Dr. Mardson Freitas de Amorim

Junho de 2025

**Catalogação na publicação
Seção de Catalogação e Classificação**

A666e Araujo, Egídio Neto Alves de.
Especificação e validação do protocolo de
comunicação DCP / Egídio Neto Alves de Araujo. - João
Pessoa, 2025.
104 f. : il.

Orientação: Mardson Freitas de Amorim.
TCC (Graduação) - UFPB/CI.

1. Microcontroladores. 2. Protocolo de comunicação.
3. Rede de computadores. 4. Barramento único. 5. Estudo
de caso. I. Amorim, Mardson Freitas de. II. Título.

UFPB/CI

CDU 004.7



CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Engenharia de Computação intitulado ***Especificação e Validação do Protocolo de Comunicação DCP*** de autoria de Egídio Neto Alves de Araújo, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Henrique Thadeu Baltar de Medeiros Cabral Moraes
Universidade Federal da Paraíba

José Antônio Gomes de Lima
Prof. Dr. José Antônio Gomes de Lima

Universidade Federal da Paraíba

Prof. Dr. Mardson Freitas de Amorim
Universidade Federal da Paraíba

Coordenador(a) do Curso de Engenharia de Computação
Josilene Aires Moreira
CI/UFPB

João Pessoa, 23 de maio de 2025

“Being smart doesn’t hurt. And a little luck now and then is nice. But the key is patience and hard work.”

The Elder Scrolls

Aos meus pais, que me deram inúmeros votos de confiança e o aporte necessário para que eu pudesse me dedicar de maneira prioritária às minhas atividades universitárias.

AGRADECIMENTOS

Agradeço à minha família, por me dar imenso apoio e a todos que acreditarem na minha capacidade.

Agradeço à minha tia Egirlândia e minha avó Marlene, que acolheram e cuidaram de mim como minhas segundas mães.

Aos meus amigos de longa data, Vitor Manuel e Vitória Gabrielle, que ouviram minhas intermináveis reclamações e aturaram meus dias de mau humor causado por noites mal dormidas, assim como também me motivaram e me deram forças nos dias difíceis em face de longas rotinas de resoluções de atividades e projetos. Espero algum dia conseguir ser de fato a pessoa que vocês dizem que eu sou.

Também agradeço aos meus amigos que tornaram a caminhada pela graduação uma tarefa tolerável. Em especial, Igor de Oliveira, João Victor Galvão, Isaac Marinho, Danilo Medeiros, Miguel Elias, Ysmahely Filho, Yvson Nunes, Guilherme Nogueira e Mateus Freitas, graças à vocês, guardo boas memórias desses longos anos e posso recordá-los com nostalgia.

Agradeço de coração à todos os motoristas da linha 9901 que não passaram direto por mim ao me verem desistir de correr, já esbaforido, ao perder a esperança de chegar na parada a tempo. Vocês têm minha gratidão eterna.

A todos os professores que, com dedicação e entusiasmo, demonstraram apreço por compartilhar seu conhecimento e incentivaram o crescimento intelectual de seus alunos.

Ao Prof. Ewerton Salvador, por me dar uma valiosíssima oportunidade de aprofundar meu conhecimento e colocar em prática conceitos de engenharia de sistemas e IoT no LASER-UFPB. Essa experiência foi fundamental para a definição da minha área de especialização e para o aprendizado de conceitos que contribuíram diretamente para a elaboração deste trabalho e que, sem dúvida, terão grande relevância em minha trajetória profissional.

Ao meu Prof. orientador, Mardson Amorim, pela paciência com os diversos atrasos e contatempos ao longo do desenvolvimento deste trabalho, pela confiança depositada em mim ao permitir que eu conduzisse este projeto e pelo apoio contínuo durante todo o processo.

E, por fim, agradeço àqueles que, de alguma forma, colocaram em dúvida minha trajetória - seu ceticismo foi, curiosamente, uma das maiores fontes de motivação.

RESUMO

Neste trabalho será feito e apresentado o manual técnico de apresentação e especificação do protocolo de comunicação DCP, idealizado no laboratório LMI-UFPB, que utiliza um fio único como barramento para realizar a comunicação entre múltiplos alvos e controladores. O documento de especificação torna possível a previsibilidade do comportamento de dispositivos que compartilham o uso do protocolo e fornece os parâmetros a serem atendidos para que ocorram comunicações com sucesso. Em conjunto com o dispositivo de validação, é possível projetar sistemas que utilizam o protocolo e ter a certeza de que será possível integrar o projeto em um sistema que utiliza o protocolo sem esperar problemas mais à frente.

O documento de especificação foi baseado em documentos similares de protocolos muito utilizados na indústria, como: I²C, I³C, UNI/O, entre outros. Este documento contém informações dos parâmetros elétricos e funcionais aceitáveis e são utilizados como base para um dispositivo validador - que utiliza um microcontrolador e sensores de instrumentação para se conectar ao barramento ou ao dispositivo de teste - realizar testes para adquirir os parâmetros do DUT e, ao fim, gerar um relatório que contém o resultado dos testes e possíveis soluções para problemas encontrados.

Por fim, é implementado um estudo de caso com diferentes microcontroladores para demonstrar a funcionalidade do validador e a viabilidade dos parâmetros do documento de especificação.

Palavras-chave: microcontroladores, protocolo de comunicação, DCP, rede de computadores, barramento único, multimestre, teste automático, estudo de caso.

ABSTRACT

This paper will present the technical manual for the introduction and specification of the DCP communication protocol, designed at the LMI-UFPB laboratory, which uses a single wire as a bus to communicate between multiple targets and controllers. The specification document makes it possible to predict the behavior of devices that share the use of the protocol and provides the parameters to be met for successful communications to occur. Together with the validation device, it is possible to design systems that use the protocol and be sure that it will be possible to integrate the project into a system that uses the protocol without expecting problems later on.

The specification document was based on similar documents for protocols widely used in the industry, such as: I²C, I³C, UNI/O, among others. This document contains information on acceptable electrical and functional parameters and is used as a basis for a validating device - which uses a microcontroller and instrumentation sensors to connect to the bus or test device - to perform tests to acquire the DUT parameters and, finally, generate a report containing the test results and possible solutions to problems encountered.

Finally, a case study with different microcontrollers is implemented to demonstrate the functionality of the validator and the feasibility of the specification document parameters.

Keywords: microcontrollers, communication protocol, DCP, computer network, single bus, multimaster, automatic testing, case study.

LISTA DE FIGURAS

1	Esquema de codificação dos bits do protocolo DCP.	32
2	ilustração dos parâmetros de tempo AC de uma carga capacitiva.	40
3	Esquemático do <i>hardware</i> do dispositivo de validação.	43
4	Fluxograma do algoritmo para realização de testes implementado.	44
5	Modelo de relatório de validação de dispositivo.	46
6	Arquitetura conceitual do estudo de caso.	48
7	Esquemático do sistema de sensoriamento.	49
8	Esquemático do sistema de interface.	50
9	Esquemático do sistema de agregação.	51
10	Frontend do dispositivo de validação.	54
11	Dispositivo de validação (à direita) ligado à um DUT de 3.3V.	55
12	Implementação do sistema de sensoriamento.	57
13	Diagrama do pacote genérico com os dados dos sensores.	57
14	Transmissão de um pacote genérico com os dados dos sensores.	57
15	Implementação do sistema de interface.	58
16	Implementação do sistema agregador.	59
17	Codificação de um bit 0 e de um bit 1.	60
18	Exemplo de <i>backoff</i> de mensagem.	60
19	Máquina de estados finitos do protocolo DCP.	62

LISTA DE TABELAS

1	características elétricas do protocolo I ² C	24
2	tempos do protocolo I ² C	25
3	características elétricas do protocolo I ³ C	26
4	tempos do protocolo I ³ C	27
5	características DC do protocolo UNI/O	29
6	características AC do protocolo UNI/O	29
7	Tabela de velocidades definidas para o protocolo DCP.	32
8	Enquadramento de dados da camada de enlace	34

LISTA DE ABREVIATURAS

AQI - Air Quality Index

BCD - Binary-Coded Decimal

CI - Centro de Informática

DCP - Device Communication Protocol

DHCP - Dynamic Host Configuration Protocol

DNS - Domain Name System

DUT - Device Under Test

FSM - Finite State Machine

GPIO - General Purpose Input and Output

HTTP - Hypertext Transfer Protocol

I/O - Input/Output

I²C - Inter-Integrated Circuit

I³C - Improved Inter-Integrated Circuit

LED - Light-Emitting Diode

LMI - Laboratório de Medidas e Instrumentação

OLED - Organic Light-Emitting Diode

RX - Receive Across

SCL - Serial Clock

SDA - Serial Data

SPI - Serial Periferal Interface

TX - Transmit Across

UART - Universal Assynchronous Receiver Transmitter

USART - Universal Synchronous Asynchronous Receiver Transmitter

USB - Universal Serial Bus

eCO₂ - Equivalent CO₂

tVOC - Total Volatile Organic Compounds

Sumário

INTRODUÇÃO	19
1 ESCOPO DO TRABALHO	20
1.1 Definição do Problema	20
1.1.1 Objetivo geral	21
1.1.2 Objetivos específicos	21
1.2 Estrutura da monografia	21
2 CONCEITOS GERAIS E REVISÃO DA LITERATURA	23
2.1 Especificações de protocolos de comunicação	23
2.1.1 I ² C	23
2.1.2 I ³ C	25
2.1.3 Onewire	27
2.1.4 UNI/O [®]	28
2.1.5 Contagem Regressiva Binária	30
2.1.6 Hot-Join	30
2.1.7 DCP	31
2.1.8 Documento de especificação	34
2.2 Validação de dispositivo	35
2.2.1 Objetivos da validação	35
2.2.2 Corretude	36
2.2.3 Tipos de validação	36
2.2.4 Estratégias de validação	37
3 METODOLOGIA	38
3.1 Especificação	38
3.2 Validação	41
3.2.1 Metodologia de Validação	41
3.2.2 Dispositivo de Validação	42

3.3	Estudo de Caso	47
3.3.1	Contextualização	47
3.3.2	Implementação	49
4	APRESENTAÇÃO E ANÁLISE DOS RESULTADOS	52
4.1	Especificação	52
4.2	Dispositivo de Validação	53
4.3	Estudo de Caso	56
4.4	Biblioteca de protocolo	61
5	CONCLUSÕES E TRABALHOS FUTUROS	63
REFERÊNCIAS		63
APÊNDICE A - MANUAL TÉCNICO DO PROTOCOLO DCP		66
APÊNDICE B - CÓDIGO-FONTE DO <i>FRONTEND</i> DO DISPOSITIVO DE VALIDAÇÃO		70
APÊNDICE C - CÓDIGO-FONTE DAS ROTINAS DE TESTE DO DISPOSITIVO DE VALIDAÇÃO		84
APÊNDICE D - RESULTADO DE TESTE DE DISPOSITIVO UTILIZANDO O VALIDADOR		91
APÊNDICE E - CÓDIGO-FONTE DO MÓDULO GENÉRICO DO PROTOCOLO DCP		93
APÊNDICE F - CÓDIGO-FONTE DO <i>PORT</i> DO PROTOCOLO DCP PARA O RP2350		101

INTRODUÇÃO

Quando pensamos em um computador tendemos a pensar em uma máquina de uso geral que espera a interação de um ser humano para realizar alguma atividade, porém, computadores são qualquer máquina que busca e executa instruções de uma memória, e transfere o resultado dessas instruções para algum lugar. Em contraste, um sistema embarcado é qualquer dispositivo que inclui um computador programável mas não se destina a ser um computador de uso geral, portanto, o computador pessoal não é um sistema embarcado, entretanto, podemos embarcar computadores em praticamente tudo o que utilizamos no nosso dia-a-dia (Wolf, 2017).

Com o avanço em miniaturização e consumo de energia, computadores deixaram de ser entidades monolíticas que ocupam porção de uma mesa, ou até mesmo toda uma sala, e se tornaram ubíquos no nosso dia-a-dia por meio dos apetrechos que utilizamos diretamente ou indiretamente, como por exemplo: telefones celulares, veículos e eletrodomésticos (Roychoudhury, 2009). E, ao contrário do que se imagina, estes dispositivos são muitas vezes compostos de mais de um computador embarcado, criando um sistema distribuído, no qual cada computador realiza uma etapa da tarefa total e por meio da comunicação, age como um sistema único e coerente (Tanenbaum; Steen, 2006).

A comunicação eficiente entre dispositivos é um pilar fundamental em sistemas distribuídos. Microcontroladores frequentemente necessitam trocar dados e comandos para operar de forma coordenada. Esta troca de informações é realizada através de protocolos de comunicação, conjuntos de regras que definem o formato dos dados, a sequência de eventos e os mecanismos de controle necessários para garantir uma interação confiável (Buchanan, 2000).

A escolha do protocolo de comunicação adequado impacta diretamente no desempenho, custo e complexidade do sistema. Protocolos como UART, SPI e I²C são amplamente utilizados, mas apresentam limitações em cenários específicos, como distância, taxa de transferência ou necessidade de flexibilidade.

O rápido crescimento da indústria de dispositivos embarcados, impulsionado pela demanda por sistemas cada vez mais complexos, exige o desenvolvimento de tecnologias adaptadas aos diversos nichos de mercado. Para garantir que essas tecnologias atendam às necessidades específicas e resolvam os problemas propostos, o processo de criação é estruturado em etapas como proposição, teste, especificação e validação (Roychoudhury, 2009).

A especificação e a validação, em particular, são cruciais para assegurar a conformidade do projeto com os requisitos estabelecidos, sendo especialmente importantes em sistemas embarcados críticos. No entanto, mesmo aplicações não-críticas se beneficiam

de componentes bem definidos e rigorosamente validados (Roychoudhury, 2009).

Diante da problemática apresentada, este trabalho tem como objetivo ratificar a especificação do *Devices Communication Protocol* (DCP) - um protocolo de comunicação desenvolvido por Araújo (2022) que utiliza um único fio para transmissão de dados - através da criação de um manual de referência técnica e de um dispositivo validador. Adicionalmente, é apresentada a implementação de um estudo de caso demonstrando a aplicação do protocolo DCP na comunicação entre dispositivos.

1 ESCOPO DO TRABALHO

A área de microcontroladores é fundamental para o desenvolvimento de dispositivos eletrônicos do dia a dia, como eletrodomésticos, automóveis, *wearables*, e sistemas de controle e automação. Microcontroladores são pequenos computadores em um único chip, contendo processador, memória e periféricos integrados, que são utilizados para controlar processos e realizar tarefas específicas de forma eficiente. Sua versatilidade e baixo custo fazem deles uma solução amplamente adotada para aplicações que exigem controle e automação, desde projetos lúdicos até sistemas industriais complexos.

No contexto de microcontroladores, os protocolos de comunicação desempenham um papel crucial, permitindo a troca de dados entre dispositivos e periféricos. Protocolos como I²C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface) e USART (Universal Synchronous/Asynchronous Receiver/Transmitter) são amplamente utilizados para interconectar sensores, atuadores, memórias, e outros componentes eletrônicos (White, 2011).

Esses protocolos variam em termos de velocidade, complexidade e número de fios necessários para comunicação, oferecendo flexibilidade conforme a demanda da aplicação. Além disso, em sistemas mais complexos, protocolos como CAN (Controller Area Network), USB, e Ethernet são utilizados para a comunicação de longa distância ou em redes maiores, garantindo maior robustez e confiabilidade na transmissão de dados (Catsoulis, 2009).

1.1 Definição do Problema

Com a popularização de dispositivos inteligentes, cada vez mais aparelhos são embarcados com mais de um microcontrolador ou subsistemas digitais, criando assim, sistemas distribuídos. Como todo sistema distribuído, um dos maiores problemas está na comunicação entre microprocessadores, dada a demanda de realizar a comunicação entre os componentes do sistema de maneira coerente, e comunicação externa ao sistema, que deve ser opaca para a arquitetura do sistema, aparentando ser um sistema monolítico.

Entretanto, dada a demanda de compactação de dispositivos, espaço físico em placas de circuito se tornaram um recurso limitado, aumentando a complexidade do posicionamento de chips e roteamento de trilhas.

Em contrapartida, protocolos de comunicação geralmente utilizam múltiplos fios para realizar o envio de dados. O protocolo SPI consegue realizar comunicação *full-duplex*, porém necessita de, no mínimo, 4 fios interligando os dispositivos, e 1 fio extra para cada dispositivo extra adicionado. De maneira similar, o protocolo *UART* utiliza, no mínimo, 2 fios, porém utiliza uma arquitetura ponto-a-ponto, como resultado, uma nova conexão

UART com 2 fios devem ser adicionados para cada dispositivo extra na rede (White, 2011).

O aumento de circuitos digitais nos sistemas embarcados traz a demanda de interligar a *CPU* do circuito a uma maior quantidade de chips, com cada chip demandando mais de um único fio. Além do problema da escassez de espaço físico mencionado, existe o problema da escassez da quantidade de pinos de entrada e saída da *CPU* que devem ser alocados para a utilização de protocolos de comunicação (White, 2011).

Dadas as demanda de utilização de vários pinos para utilização de um ou mais protocolos pela *CPU*, dependendo da quantidade de conexões necessárias, não é possível conectar todos os chips diretamente, criando a necessidade de um microprocessador de intermédio, que se conecta a um, ou vários sistemas e realiza um processamento extra para diminuir a demanda ao processador principal, ou para realizar a tradução de um protocolo para outro.

Esta abordagem, conhecida como hardware adaptador, apesar de efetiva, aumenta o espaço físico utilizado, o preço total dos componentes, e o consumo energético (Roychoudhury, 2009).

1.1.1 Objetivo geral

O seguinte trabalho se propõe em contribuir para o processo de formalização do protocolo *Devices Communication Protocol*, criado por Araújo (2022), por meio da concepção de um documento de especificação e da criação de um dispositivo que realize a validação de um dispositivo com base nos valores determinados na especificação. De maneira que seja fornecido um ponto de partida para desenvolvedores que buscam incluir o protocolo em seus projetos.

1.1.2 Objetivos específicos

1. Criação de documento de especificação na forma de manual de referência técnica do protocolo *DCP*;
2. Produção de plataforma de validação automatizada do protocolo;
3. Implementação de estudo de caso demonstrando a utilização do protocolo e o processo de validação de dispositivo.

1.2 Estrutura da monografia

Descrito nesse capítulo se encontram: a introdução, a definição do problema, objetivos gerais e específicos deste trabalho.

No capítulo 2, temos a definição de conceitos gerais e revisão da literatura correlata, em específico: funções das camadas física e de enlace, implementações de protocolos de comunicação, como: I²C, I³C, OneWire, UNI/O [®], e DCP. Assim como metodologias de validação.

O capítulo 3, apresenta a metodologia proposta para o trabalho, composta pela explicação do procedimento envolvido na criação do manual de referência, atribuições do dispositivo de validação e detalhamento do estudo de caso.

O capítulo 4, apresenta os resultados do trabalho.

E, por fim, o capítulo 5, traz as discussões e conclusões retiradas dos resultados do trabalho.

2 CONCEITOS GERAIS E REVISÃO DA LITERATURA

Este capítulo trará o embasamento teórico para o desenvolvimento do trabalho, demonstrando, em duas seções, conceitos e exemplos que serão utilizados no prosseguimento das atividades. A primeira seção está relacionada à especificação do protocolo, enquanto a segunda seção traz os fundamentos utilizados na etapa de validação.

2.1 Especificações de protocolos de comunicação

Nesta seção temos exemplos de protocolos utilizados na indústria, com foco em suas especificações elétricas e funcionais. Serão apresentados os protocolos: I²C, I³C, UNI/O®, Onewire e DCP.

2.1.1 I²C

O barramento I²C (*Inter-Integrated Circuit*), criado pela *Philips Semiconductors* é um padrão amplamente adotado para comunicação serial bidirecional, sendo utilizado em mais de mil circuitos integrados de diferentes fabricantes e em várias arquiteturas de controle. Como por exemplo, pode-se citar: *System Management Bus* (SMBus), criado pela Intel e Duracell, utilizado para se comunicar com dispositivos de baixa velocidade nas placa-mãe de computadores (SBS, 1998); *Power Management Bus* (PMBus) derivado do SMBus, utilizado para gerenciamento de fontes de alimentação; *Display Data Channel* (DDC) utilizado por *displays* para descrever suas capacidades para adaptadores de vídeo; assim como outras aplicações em arquiteturas de alto nível, como IPMI e ATCA (NXP, 2021).

O I²C possui duas linhas principais: uma linha de dados serial bidirecional, SDA e uma linha de *clock*, SCL, dirigida pelo controlador do barramento e conectadas a uma fonte de tensão positiva por meio de resistores *pull-up* (NXP, 2021). Cada dispositivo no barramento possui um endereço único e a comunicação segue uma relação controlador-alvo, com suporte para múltiplos controladores e detecção de colisão para evitar corrupção de dados em caso de múltiplos controladores transmitindo simultaneamente.

O protocolo permite transferência de dados em diferentes modos, com taxas que vão de até 100 kbit/s no modo padrão, até 3,4 Mbit/s no modo de alta velocidade e 5 Mbit/s no modo ultra-rápido (NXP, 2021). Por padrão, são utilizados 7 bits para o endereçamento dos dispositivos conectados ao barramento, totalizando 128 dispositivos. Entretanto, é possível utilizar endereçamento de 10 bits em qualquer velocidade de barramento, aumentando o número de dispositivos para 1024, porém, o número total de dispositivos também é limitado pela constante de tempo RC do barramento (NXP, 2021).

Em sua forma padrão, o I²C não especifica um valor de tensão para os sinais, sendo estes definidos com base em um valor V_{DD} comum aos dispositivos, entretanto, versões específicas do protocolo, como SMBus e PMBus, por exemplo, fixam as tensões altas e baixas em valores numéricos (SBS, 1998). A tabela 1 mostra os valores elétricos máximos e mínimos do protocolo para cada modo de velocidade, sendo estes os valores de tensão altos e baixos, a tensão de histerese, corrente de vazamento no nível lógico baixo e tempo de descida de sinal.

Tabela 1: características elétricas do protocolo I²C

Characteristics of the SDA and SCL I/O stages

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
V_{il}	Low-level input voltage	-	-0.5	$0.3V_{DD}$	-0.5	$0.3V_{DD}$	-0.5	$0.3V_{DD}$	V
V_{ih}	High-level input voltage	-	$0.7V_{DD}$	-	$0.7V_{DD}$	-	$0.7V_{DD}$	-	V
V_{hys}	Hysteresis of Schmitt Trigger input	-	-	-	$0.05V_{DD}$	-	$0.05V_{DD}$	-	V
I_{ol}	Low-level output current	$V_{ol} = 0.4 \text{ V}$	3	-	3	-	20	-	mA
t_{of}	Output Fall Time	-	-	250	$20 * (V_{DD} / 5.5 \text{ V})$	250	$20 * (V_{DD} / 5.5 \text{ V})$	120	ns

Autoria: NXP Semiconductors

A tabela 2, com estrutura similar à tabela anterior, apresenta os sinais principais do protocolo e parâmetros pertinentes à configuração escolhida, como a frequência do *clock* do barramento, e tempos mínimos e máximos de sinais altos e baixos, condição *START*, tempos de subida e de descida, e tempos máximos de *acknowledgement*, que deve ser enviado pelo destinatário da mensagem para atestar a validade dos dados.

Tabela 2: tempos do protocolo I²C

Characteristics of the SDA and SCL I/O bus lines for Standard, Fast, and Fast-mode Plus I²C-bus devices

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
f _{SCL}	SCL clock frequency	-	0	100	0	400	0	1000	kHz
t _{HOLD;STA}	hold time (repeated) START condition	After this period, the first clock pulse is generated.	4.0	-	0.6	-	0.26	-	μs
t _{LOW}	LOW period of the SCL clock	-	4.7	-	1.3	-	0.5	-	μs
t _{HIGH}	HIGH period of the SCL clock	-	4.0	-	0.6	-	0.26	-	μs
t _{SU;STA}	set-up time for a repeated START condition	-	4.7	-	0.6	-	0.26	-	μs
t _{SU;DAT}	data set-up time	-	250	-	100	-	50	-	ns
t _r	rise time of both SDA and SCL signals	-	-	1000	20	300	-	120	ns
t _f	fall time of both SDA and SCL signals	-	-	300	20 * (V _{DD} / 5.5 V)	300	20 * (V _{DD} / 5.5 V)	120	ns
t _{SU;STO}	set-up time for STOP condition	-	1.4	-	0.6	-	0.26	-	μs
t _{BUF}	bus free time between a STOP and START condition	-	4.7	-	1.3	-	0.5	-	μs
t _{VD;DAT}	data valid time	-	-	3.45	-	0.9	-	0.45	μs
t _{VD;ACK}	data valid acknowledge time	-	-	3.45	-	0.9	-	0.45	μs

Autoria: NXP Semiconductors

2.1.2 I³C

O I³C (*Improved Inter-Integrated Circuit*) é um protocolo de comunicação desenvolvido pela *MIPI Alliance*, fundada pelas empresas: ARM, Nokia, STMicroelectronics e Texas Instruments, porém que atualmente tem diversas empresas parceiras, como por exemplo: Google, Intel, Samsung e Qualcomm (MIPI, 2024). O protocolo foi criado para atender às necessidades de dispositivos embarcados, especialmente em sistemas de sensores em que há requisitos de baixa potência e alta velocidade de comunicação. Ele representa uma evolução do protocolo I²C (*Inter-Integrated Circuit*), mantendo a compatibilidade com dispositivos I²C legados, mas introduzindo melhorias significativas em termos de velocidade, capacidade de gerenciamento de barramento e eficiência energética. Assim como o I²C, o I³C utiliza um barramento de dois fios: SDA e SCL, que opera em modos de push-pull e open-drain, oferecendo flexibilidade em diversas aplicações (MIPI, 2022).

Diferentemente do I²C, o documento de especificação do I³C apresenta valores de

tensão bem definidos para tensões mínima, máxima e típica, que são utilizados como o V_{DD} , que é base para os outros valores apresentados. Como tensão V_{DD} típica, são apresentados os valores de 1,2 V, 1,8 V e 3,3 V, que são utilizados para definir as regiões de valores aceitáveis para tensão para o nível alto e tensão de nível baixo no estágio de entrada, que são 30% e 70% do valor de V_{DD} , respectivamente. Além disso, também são especificados: o valor máximo para o nível de tensão baixo na saída, que, similarmente aos outros protocolos, é baseado no *ground* do circuito, o valor mínimo para o nível de tensão alto na saída para o modo *push-pull*; valores de capacitância por pinos de *I/O* e a máxima diferença de capacitância entre os fios do barramento.

A tabela 3 descreve as características elétricas específicas dos estágios de entrada e saída do protocolo, tanto para modos de *push-pull* quanto para *open-drain*. Os parâmetros apresentados incluem: tensões operacionais e limites de tensão de entrada e saída.

Tabela 3: características elétricas do protocolo I³C

I³C I/O Stage Characteristics Common to Push-Pull mode and Open Drain Mode

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
V_{DD}	Operating Voltage	-	1.10	1.20	1.30	V
			1.65	1.80	1.95	V
			2.97	3.30	3.63	V
V_{IL}	Low-Level Input Voltage	-	-0.1 * V_{DD}	-	0.3 * V_{DD}	V
V_{IH}	High-Level Input Voltage	-	0.7 * V_{DD}	-	1.1 * V_{DD}	V
V_{OL}	Output Low Level	For $V_{DD} < 1.4V$: $I_{OL} = 2 \text{ mA}$	-	-	0.18	V
		For $V_{DD} \geq 1.4V$: $I_{OL} = 3 \text{ mA}$	-	-	0.27	V
V_{OH}	Output High Level (push-pull only)	For $V_{DD} < 1.4V$: $I_{OL} = -2 \text{ mA}$	$V_{DD} - 0.18$	-	-	V
		For $V_{DD} \geq 1.4V$: $I_{OL} = -3 \text{ mA}$	$V_{DD} - 0.27$	-	-	V

Autoria: MIPI Alliance

A tabela de tempos 4, de maneira similar ao I²C, traz a região de frequência do *clock*, que se estende de 100 kHz a 12,9 MHz, assim como os tempos mínimos dos sinais básicos, que são 24 ns e 32 ns nos modos de *push-pull* e *open-drain*, respectivamente. Além dos tempos máximos de subida, adiciona os tempos adequados para configurações exclusivas de *push-pull* e tempos máximos para situações específicas encontradas durante o funcionamento do protocolo em determinada configuração.

Tabela 4: tempos do protocolo I³C

I³C Push-Pull Timing Parameters for SDR, ML, HDR-DDR, and HDR-BT Modes

Symbol	Parameter		Min	Typ	Max	Unit
f_{scl}	SCL clock frequency		0.01	12.5	12.9	MHz
t_{LOW}	LOW period of the SCL clock		24	-	-	ns
$t_{\text{DIG_L}}$			32	-	-	ns
$t_{\text{HIGH_MIXED}}$	HIGH period of the SCL clock (mixed bus)		24	-	-	ns
$t_{\text{DIG_H_MIXED}}$			32	-	45	ns
t_{HIGH}	HIGH period of the SCL clock		24	-	-	ns
$t_{\text{DIG_H}}$			32	-	-	ns
t_{sco}	Clock In to Data Out for Target		-	-	12	ns
t_{cr}	Rise time of SCL		-	-	$150e06^* \frac{1}{f_{\text{scl}}}$	ns
t_{cf}	Fall time of SCL		-	-	$150e06^* \frac{1}{f_{\text{scl}}}$	ns
$t_{\text{SU_PP}}$	SDA Signal Data Setup in Push-Pull Mode		3	-	n/a	ns
$t_{\text{HD_PP}}$	Controller	SDA Signal Data Hold in Push-Pull Mode	$t_{\text{cr}} + 3$ and $t_{\text{cf}} + 3$	-	-	-
$t_{\text{HD_PP}}$	Target		0	-	-	-
t_{CASS}	Clock After Repeated START (Sr) Condition		$t_{\text{CASmin}} / 2$	-	-	ns
t_{CBSr}	Clock Before Repeated START (Sr) Condition		$t_{\text{CASmin}} / 2$	-	-	ns
$t_{\text{BT_FREQ}}$	HDR-BT SCL Clock Frequency		0.1	12.5	12.9	MHz

Autoria: MIPI Alliance

2.1.3 Onewire

O protocolo 1-Wire, também conhecido por *Onewire*, foi criado pela Dallas Semiconductor, que posteriormente foi incorporada à Analog Devices, é uma tecnologia de comunicação que utiliza apenas um fio para sinalização e alimentação, oferecendo uma solução simples e econômica para conectar múltiplos dispositivos. A comunicação no barramento é assíncrona e opera em modo *half-duplex*, onde o fluxo de dados ocorre em uma direção por vez. A arquitetura segue uma estrutura estrita de controlador-alvo: o controlador é o único responsável por iniciar a comunicação, enquanto um ou mais dispositivos alvos podem responder, permitindo a conexão simultânea de vários alvos ao barramento (DALLAS, S.I.).

A sinalização no barramento 1-Wire ocorre em intervalos de tempo específicos, denominados *slots*, com duração de 60 µs. Cada bit de dado é transmitido em um único *slot*, e os dispositivos alvos podem ter uma base de tempo que difere da nominal. Isso exige que o controlador tenha um controle de tempo preciso para garantir a comunicação correta, especialmente com alvos que possuam temporizações distintas. Portanto, é fundamental que o controlador respeite os limites de tempo especificados para garantir a sincronização adequada (ATMEL, 2004).

Os níveis lógicos no barramento 1-Wire seguem padrões *CMOS/TTL* convencio-

nais, com máximo de 0,8 V para o nível lógico baixo e um mínimo de 2,2 V para o lógico alto. A linha de dados é conectada à fonte de 5 V do controlador por meio de um resistor *pull-up* fraco, que consiste em um resistor de alto valor, criando um barramento que pode ser controlado utilizando pouca corrente e garantir tempo de bateria maior e menor dissipação de calor. Além disso, os dispositivos conectados ao barramento obrigatoriamente devem ser capazes de utilizar saídas *open-drain* para levar a tensão do barramento ao nível de *ground* (DALLAS, S.I.).

2.1.4 UNI/O °

Similarmente ao Onewire, o protocolo UNI/O, desenvolvido pela Microchip (2009) para comunicação de baixa velocidade em sistemas embarcados, também permite a troca de dados através de um único sinal de I/O, suportando a conexão de múltiplos dispositivos em um barramento compartilhado, porém, sem a capacidade de realizar alimentação desses dispositivos. Esse protocolo segue uma estrutura controlador-alvo. Portanto, em um sistema típico, um único controlador é conectado a um ou vários alvos, como por exemplo, sensores e memórias, sem suporte a mais de um controlador. O protocolo remove a necessidade de um fio exclusivo para *clock* em decorrência da codificação Manchester - código de linha onde cada bit de dado é representado por uma transição entre níveis baixo e alto durante um período igual de tempo -, que possui como maior característica o auto-sincronismo (Microchip, 2009).

O controle do barramento é responsabilidade do controlador, que define o período do *clock*, controla o acesso ao barramento e inicia todas as operações. Portanto, apesar dos dispositivos alvos terem a capacidade de se comunicar com o controlador, o modo ativo é determinado pelo mesmo e o dispositivo alvo deve sincronizar com o controlador, e esperar a arbitragem para poder enviar seus dados. O barramento UNI/O suporta taxas de operação de 10 kbps a 100 kbps e não impõe restrições quanto a faixas de tensão, temperatura ou processos de fabricação (Microchip, 2009).

Todos os dispositivos conectados ao barramento devem utilizar pinos de I/O *tri-state* com *push-pull* e manter o pino em alta impedância quando não estiverem em controle do barramento. Dado o uso de pinos *push-pull*, todos os dispositivos alvo devem ser limitados em corrente para impedir alta fuga de corrente causada por um curto-circuito em uma situação de colisão (Microchip, 2009).

Na tabela 5, o protocolo apresenta seus valores DC de maneira similar aos outros protocolos, porém, apresenta parâmetros diferentes para dispositivos de baixa potência com nível de tensão $V_{CC} < 2.5$ V e dispositivos com $V_{CC} \geq 2.5$ V. Notoriamente, valores de corrente de saída do barramento são diferentes para as 2 categorias.

Tabela 5: características DC do protocolo UNI/O

I/O Structure DC Characteristics

Symbol	Parameter	Conditions	Min	Max	Unit
V_{IH}	High-Level Input Voltage	-	$0.7V_{cc}$	$V_{cc}+1$	V
V_{IL}	Low-Level Input Voltage	$V_{cc} \geq 2.5V$	-0.3	$0.3V_{cc}$	V
		$V_{cc} < 2.5V$	-0.3	$0.2V_{cc}$	V
V_{OL}	Low-Level Output Voltage	$I_{OL} = 300\mu A, V_{cc} \geq 2.5V$	-	0.4	V
		$I_{OL} = 200\mu A, V_{cc} < 2.5V$	-	0.4	V
V_{OH}	High-Level Output Voltage	$I_{OH} = 300\mu A, V_{cc} \geq 2.5V$	$V_{cc}-0.5$	-	V
		$I_{OH} = 200\mu A, V_{cc} < 2.5V$	$V_{cc}-0.5$	-	V
I_o	Slave Output Current Limit	$V_{cc} \geq 2.5V$	-	± 4	mA
		$V_{cc} < 2.5V$	-	± 3	mA
I_{LI}	Input Leakage Current (SCIO)	$V_{IN} = V_{ss} \text{ or } V_{cc}$	-	± 10	μA

Autoria: Microchip

Na tabela 6 de parâmetros AC temos a apresentação de parâmetros comuns, como frequência, período, e tempos de subida e descida, porém, também temos a apresentação de parâmetros de limite de tolerância de deriva de frequência: T_{DRIFT} e T_{DEV} , que especificam a margem de erro associada a cada transmissão de bit e comando respectivamente. Estes parâmetros servem para alertar todos os desenvolvedores sobre a concordância das tolerâncias de comunicação, que possui uma elevada importância dado o fato que o protocolo utiliza apenas um fio para comunicação e a interpretação dos sinais depende diretamente da duração dos pulsos.

Tabela 6: características AC do protocolo UNI/O

AC characteristics

Symbol	Parameter	Min	Max	Unit
F_{BIT}	Serial bit frequency	10	100	kHz
T_E	Bit period	10	100	μs
T_R	Rise time of SCIO	-	100	ns
T_F	Fall time of SCIO	-	100	ns
T_{DRIFT}	Serial bit frequency drift rate tolerance per byte	-	± 0.75	%
T_{DEV}	Serial bit frequency drift limit per command	-	± 5	%
T_{STBY}	Standby pulse time	600	-	μs
T_{SS}	Start header setup time	10	-	μs
T_{HDR}	Start header low pulse time	5	-	μs
T_{SP}	Input filter spike suppression	-	50	ns
T_{HLD}	Hold time	T_E	-	ns

Autoria: Microchip

2.1.5 Contagem Regressiva Binária

A contagem regressiva binária (*Binary Countdown*) surge como um mecanismo eficiente de prevenção de colisão em nível de camada de enlace, utilizado em redes com um grande número de dispositivos nas quais o *overhead* de outros mecanismos de prevenção de colisão se tornam proibitivos. A base da eficiência deste método reside no uso de endereços binários únicos para identificar os dispositivos e na combinação das transmissões através de operações lógicas *OR* ou *AND* no canal de comunicação (Tanenbaum; Wetherall, 2010).

O processo inicia com cada dispositivo transmitindo seu endereço em formato binário, começando pelo bit mais significativo. A transmissão simultânea desses bits é combinada no barramento, resultando em um valor lógico resultante da operação AND entre todos os bits transmitidos. A premissa fundamental para o funcionamento correto do mecanismo é a negligibilidade dos atrasos de transmissão, garantindo que todos os dispositivos percebam os bits transmitidos quase instantaneamente (Tanenbaum; Wetherall, 2010).

Para evitar colisões, uma regra de arbitragem é aplicada: sempre que um dispositivo identifica um bit de ordem superior em seu endereço foi sobreescrito por um bit 0 no canal, ele desiste da transmissão. O processo continua bit a bit até que apenas um dispositivo permaneça transmitindo, sendo esse o detentor do endereço mais baixo. Esse dispositivo que conseguiu a arbitragem do barramento pode então transmitir seus dados e, ao liberar o barramento, os outros dispositivos podem reiniciar o processo de contagem para enviarem seus dados. A eficiência teórica deste método é dada por $\frac{d}{(d+\log_2 N)}$, onde "d" representa o tempo necessário para transmitir uma frame e "N" o número total de dispositivos (Tanenbaum; Wetherall, 2010).

2.1.6 Hot-Join

O mecanismo de *Hot-Join* é uma funcionalidade projetada para permitir a adição dinâmica de dispositivos a um barramento de comunicação enquanto o sistema está em operação, sem a necessidade de reinicialização ou interrupção do funcionamento. Essa capacidade é particularmente útil em cenários onde a adição ou remoção de periféricos precisa ser feita de forma flexível e transparente para o usuário final.

Em sistemas que implementam *Hot-Join*, um novo dispositivo pode se conectar ao barramento, se identificar e negociar sua comunicação com o controlador principal sem interromper as operações já em andamento. Este processo geralmente envolve a detecção da presença do novo dispositivo pelo controlador, seguido por uma fase de identificação e configuração para estabelecer a comunicação adequada (MIPI, 2022).

O *Hot-Join* foi implementado no protocolo I³C como um recurso padrão, permiti-

tindo que dispositivos se conectem e desconectem do barramento sem afetar o funcionamento dos demais componentes. No I³C, o processo de *Hot-Join* envolve a emissão de um sinal específico pelo novo dispositivo para anunciar sua presença e solicitar uma conexão com o controlador. O controlador então gerencia a alocação de endereços e recursos para o novo dispositivo, garantindo uma integração suave e eficiente no sistema. A implementação do *Hot-Join* no I³C contribui para aumentar a flexibilidade e robustez dos sistemas embarcados, permitindo a adaptação dinâmica às necessidades de hardware e software (MIPI, 2022).

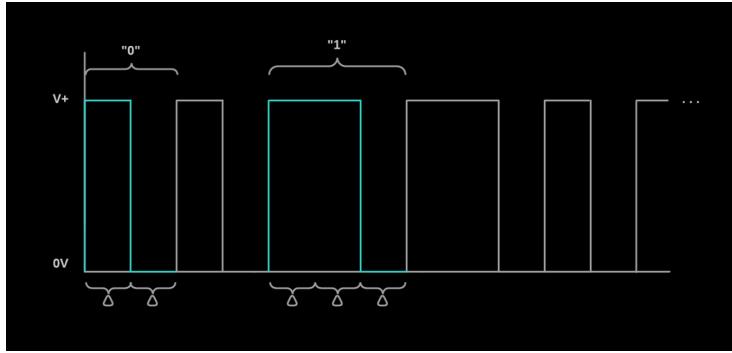
2.1.7 DCP

O protocolo DCP (*Devices Communication Protocol*), desenvolvido no LMI-UFPB, é um protocolo de comunicação bidirecional de um único fio, projetado para suportar uma arquitetura multimestre. Diferente do protocolo UNI/O, o DCP permite que dispositivos se comuniquem sem a necessidade de uma arbitragem ativa por parte de um controlador de barramento. Para isso, o protocolo utiliza a estratégia de Acesso Múltiplo com Detecção de Portadora e Detecção de Colisão (CSMA/CD) combinada com um mecanismo de contagem regressiva binária. Nesse esquema, os dispositivos monitoram o barramento antes de transmitir; caso o barramento esteja ocupado, o dispositivo entra em espera por um período, definido com base em seu endereço. Se ainda houver colisões, o mecanismo de contagem regressiva binária garante que a mensagem de maior prioridade seja transmitida, enquanto o dispositivo cuja mensagem foi sobreescrita detecta a colisão e aguarda antes de retransmitir (Araújo, 2022).

A especificação desenvolvida por Araújo (2022) não define um nível alto de tensão específico, no entanto, para prevenir erros de leitura digital e problemas elétricos, recomenda-se que todos os dispositivos trabalhem em uma faixa de tensão V_{DD} similar, apesar que, com as devidas precauções, é possível conectar dispositivos de tensões diferentes no mesmo barramento. Por outro lado, o nível baixo do barramento corresponde aproximadamente ao nível de referência *ground* do circuito.

A codificação das informações é realizada com base em diferentes durações do nível alto no barramento: o bit 0 é representado por 1 unidade de tempo Δ no estado alto e 1 unidade no estado baixo, enquanto o bit 1 é representado por 2 unidades de tempo Δ no estado alto e 1 unidade no estado baixo. Como ilustrado na figura 1. Caso não haja nenhuma transmissão no barramento, o barramento se mantém no nível alto de tensão por meio da utilização de um resistor *pull-up*.

Figura 1: Esquema de codificação dos bits do protocolo DCP.



Autoria: Araújo (2022)

O valor da unidade de tempo Δ muda dependendo do modo do barramento, e todos os dispositivos associados ao barramentos devem levar essa característica em conta ao realizar uma comunicação no barramento. Realizar uma comunicação em um modo diferente do barramento acarretará em uma comunicação inválida. Os modos do barramento e a duração de uma unidade de tempo Δ estão descritos na tabela 7, com os modos sendo: *Slow*, *Fast I*, *Fast II* e *High Speed*. Os dispositivos conectados devem manter a temporização baseados em seus *clocks* internos, entretanto, os pulsos de sincronismos enviados nas transmissões podem ser utilizados para recalibração e sincronização dos dispositivos para a transmissão que se segue.

Tabela 7: Tabela de velocidades definidas para o protocolo DCP.

Modo	Velocidade	Descrição
Slow	$\Delta = 20\mu s$	Dispositivos operando em 4MHz
Fast I	$\Delta = 4\mu s$	Dispositivos operando em 20MHz
Fast II	$\Delta = 2,5\mu s$	Dispositivos operando em 32MHz
High speed	$\Delta = 1,25\mu s$	Dispositivos operando em 64MHz

Autoria: Araújo (2022)

O DCP apresenta 3 sinais fundamentais: o *Start Sync*, composto por nível baixo para identificar se o dispositivo enviando os dados é um controlador ou alvo e durando 25Δ s para controladores e 50Δ s para alvos; o *Start Bit*, composto por nível alto e nível baixo em sequência, ambos com duração de 7.5Δ s e sinaliza o início da transmissão do pacote; e por fim, o *Pulso de Sincronismo*, que serve para alertar os dispositivos no barramento a sincronizarem, possui duração de 50Δ s (Araújo, 2022).

Cada dispositivo deve possuir um endereço único de 8 bits (*Id*) para acesso ao barramento, este endereço é decidido pelo projetista e define a prioridade de acesso ao barramento pelo dispositivo. Um endereço menor incide em uma maior prioridade devido ao mecanismo de contagem regressiva binária utilizada pelo protocolo (Araújo,

2022). O protocolo de enlace *L3* (Amorim, 2017) tem endereços reservados para dispositivos controladores e alvo, em que um dispositivo controlador possui endereço entre *0xFA* e *0xFF*, portanto, dispositivos controladores utilizam seu endereço *L3* como endereço de acesso ao barramento (*MasterId*) (Araújo, 2022).

Em arquiteturas de meio de comunicação compartilhado existe o problema de colisão de mensagens, que é causado quando 2 ou mais dispositivos enviam mensagens ao mesmo tempo, corrompendo todas as mensagens sendo transmitidas. Para evitar esse problema, o protocolo utiliza uma estratégia baseada em 2 níveis.

Primeiramente é utilizada a estratégia CSMA, a qual dita que, antes de enviar uma mensagem, o dispositivo deve analisar o estado do barramento para detectar alguma transmissão. Caso o barramento esteja livre, o dispositivo deve aguardar $\frac{(Id+6)*\Delta}{4}$ s caso esteja enviando um pacote genérico, ou $\frac{(MasterId-249)*\Delta}{4}$ s caso esteja enviando um pacote *L3*, dado que no segundo caso, o dispositivo possui endereço maior que 250 por especificação. E caso um dispositivo que esteja em espera perceba uma mudança de nível no barramento, ele deve voltar a "escutar" o barramento esperando sua liberação (Araújo, 2022).

Para iniciar uma transmissão, um dispositivo envia os sinais: *Start Sync*, *Start Bit*, endereço (*Id*) e pulso de sincronismo. Em seguida, para realizar a transmissão de dados, o dispositivo deve reenviar os sinais: *Start Sync*, *Start Bit*, endereço, seguido dos dados organizados em um dos esquemas de enquadramento da camada de enlace.

Após o início da transmissão, os dispositivos também devem "escutar" o barramento confirmado se o nível do barramento corresponde ao nível do bit que foi enviado. Caso o dispositivo detecte uma colisão, ele deve guardar sua mensagem para reenvio e ir para o modo de "escuta" aguardando a liberação do barramento (Araújo, 2022).

Explorando a característica do barramento onde o nível baixo é dominante, e alinhado com a estratégia de detecção de colisões implementada após o início da transmissão, adota-se a contagem regressiva binária (Tanenbaum; Wetherall, 2010). Este método garante que, na ocorrência de uma disputa pelo canal, o dispositivo transmitindo o nível lógico baixo não detectará a colisão, enquanto os demais identificarão o conflito e entrerão em estado de espera. Assim, mesmo diante de colisões, assegura-se a possibilidade de envio bem-sucedido da mensagem por um dos dispositivos

Na prática, como os dispositivos enviam seus endereços antes de enviar o pacote de dados no barramento, os dispositivos de endereço menor possuem prioridade na escrita do barramento em caso de colisão, enquanto dispositivos de endereços maiores detectam a colisão ainda no início da transmissão e abrem mão da arbitragem do barramento.

A camada de enlace do protocolo especifica um enquadramento simples, porém flexível, podendo ser estendido para aplicações específicas. O enquadramento é formado

por um campo de cabeçalho de 1 byte seguido do *payload* com os dados da mensagem.

O campo de cabeçalho pode conter 3 tipos de valores:

- Valor 0: que indica que o empacotamento não possui dados e é composto por um sinal de sincronismo;
- Valor 1: que indica que o empacotamento é composto por um pacote L3 de 12 bytes de comprimento;
- Valor $1 < n \leq 255$: que indica o tamanho em bytes do *payload* que se segue.

O *payload* genérico deve obedecer a quantidade de bytes anunciado pelo cabeçalho e contém dados binários sem interpretação específica associada. A tabela 8 demonstra o enquadramento genérico e sua simplicidade, oferecendo à aplicação a capacidade de realizar qualquer interpretação nos dados.

Tabela 8: Enquadramento de dados da camada de enlace

Cabeçalho	<i>Payload</i>
Byte 0	Byte 1 a n

Autoria: Araújo (2022)

2.1.8 Documento de especificação

O desenvolvimento bem-sucedido de qualquer protocolo de comunicação depende fundamentalmente da existência de um manual técnico detalhado e abrangente. Este documento serve como guia essencial para desenvolvedores, fornecendo todas as informações necessárias para realizar uma implementação correta e garantir a interoperabilidade entre diferentes dispositivos que utilizam o protocolo. Em essência, o manual técnico atua como um contrato entre os projetistas do protocolo e aqueles que buscam integrá-lo em suas aplicações, provendo dados cruciais sobre seu funcionamento interno, requisitos elétricos e temporais, e mecanismos de comunicação.

Os protocolos apresentados neste trabalho seguem uma estrutura comum em seus documentos de especificação. A organização geral é composta por seções distintas, cada uma abordando aspectos específicos do protocolo: uma seção introdutória que contextualiza o protocolo, detalhando seus objetivos, aplicações pretendidas e capacidades; uma seção dedicada às características específicas do protocolo, quando aplicável, explorando funcionalidades diferenciadas ou modos de operação particulares; uma seção crucial que

define as especificações elétricas (tanto AC quanto DC) dos barramentos utilizados pelo protocolo e das interfaces dos dispositivos conectados a eles; e, finalmente, uma seção que descreve os sinais básicos do protocolo, incluindo seus tempos mínimos e máximos, conforme exemplificado em Microchip (2009), MIPI (2022) e NXP (2021). Embora alguns documentos adotem uma estrutura mais granular, como demonstrado em SBS (1998) e DALLAS (S.I.), a essência da organização permanece consistente.

As seções de introdução, características específicas e sinais básicos são intrinsecamente dependentes do protocolo em questão, variando significativamente em conteúdo para refletir as particularidades de cada implementação. No entanto, a seção dedicada às informações elétricas tende a apresentar um conjunto comum de dados relevantes em todos os documentos, incluindo: velocidade máxima absoluta e relativa da comunicação, níveis de tensão de entrada aceitáveis, correntes de fuga e características de entrada de carga, tempos de subida e descida dos sinais, capacidade do barramento, tempo de ciclo do protocolo e relação sinal-ruído.

Complementando a descrição das especificações elétricas, a seção de sinais básicos apresenta, com auxílio de diagramas e tabelas, os sinais utilizados pelo protocolo. Tabelas com limites de tolerância são fornecidas para garantir que os sinais estejam dentro da faixa aceitável, enquanto diagramas de tempo ilustram exemplos práticos de comunicações básicas, auxiliando o desenvolvedor na compreensão do fluxo de dados e na implementação correta dos protocolos de troca de informações. Essa abordagem combinada de especificações quantitativas e representações visuais facilita a interpretação das informações e minimiza o risco de erros durante a fase de desenvolvimento.

2.2 Validação de dispositivo

Nesta seção será apresentada a importância do processo de validação, junto da teoria e prática do processo.

2.2.1 Objetivos da validação

Para que um protocolo de comunicação seja utilizado, implementações robustas são essenciais. Estas implementações necessitam de especificações inequívocas e livres de erros. Para garantir essas necessidades, especificações formais são utilizadas para evitar problemas de má interpretação que surgem em especificações em linguagem natural, além de criar a capacidade de validar implementações acerca da conformidade com a especificação (Palmer; Sabnani, 1986). A validação busca identificar e corrigir erros antes da implementação final, assegurando a confiabilidade e robustez do sistema em operação. Além disto, procura-se garantir que os sistemas atendam a restrições de tempo e que a interação com o ambiente físico ocorra de maneira eficaz e segura (Roychoudhury, 2009).

2.2.2 Corretude

A garantia da corretude de um sistema é fundamental para assegurar seu comportamento esperado em todas as circunstâncias. Em outras palavras, busca-se demonstrar que um programa ou protocolo executa as ações pretendidas e não apresenta falhas inesperadas. Diversas abordagens podem ser empregadas para validar essa corretude, incluindo testes funcionais, simulações e verificação formal. A escolha da metodologia mais adequada depende criticamente do nível de confiabilidade exigido pelo sistema em questão (Apt; Olderog, 2019).

Dentro das técnicas de verificação formal, destaca-se a Lógica de Hoare, desenvolvida por Robert Floyd na década de 1960. Essa lógica fornece uma base matemática para provar a exatidão de algoritmos, estabelecendo um rigoroso framework para analisar o comportamento do código. O conceito central da Lógica de Hoare é a "trípla de Hoare", representada como $\{P\} C \{Q\}$, onde "C" representa um comando ou bloco de instruções, "P" é uma asserção que descreve o estado do sistema antes da execução de "C" (pré-condição), e "Q" é uma asserção que define o estado após a execução de "C" (pós-condição) (Apt; Olderog, 2019).

A trípla de Hoare expressa a ideia de que, se a pré-condição "P" for verdadeira antes da execução do comando "C", então a pós-condição "Q" será verdadeira após a sua conclusão. Para garantir a corretude total, é necessário que o comando termine sua execução (termino) além do atendimento das condições de pré e pós-condição. Em essência, a Lógica de Hoare permite transformar um programa em uma série de afirmações lógicas que podem ser verificadas formalmente, assegurando a ausência de erros e comportamentos indesejados (Apt; Olderog, 2019).

2.2.3 Tipos de validação

Existem diferentes abordagens para prover especificações formais para protocolos de comunicação. E cada técnica de especificação necessita de técnicas específicas para realizar a validação (Palmer; Sabnani, 1986). Se um protocolo é modelado como uma coleção de máquinas de estado finitos (FSM), então é utilizada a técnica de análise de acessibilidade, ou seja, se o dispositivo consegue acessar todos os estados possíveis do protocolo. Se um protocolo é especificado utilizando uma linguagem de programação, então é utilizada o método de Floyd-Hoare para verificar a exatidão da implementação. E, por fim, para especificações que utilizam uma combinação de FSMs e segmentos de programas são utilizadas técnicas híbridas, que testam cada caso em separado (Palmer; Sabnani, 1986).

Para a área de validação de dispositivos, existem diversas técnicas, dentre algumas delas temos: o teste, que verifica se um sistema se comporta conforme o esperado para

entradas específicas; a simulação, a qual envolve a execução de um modelo completo do sistema para observar seu comportamento em condições controladas; e a verificação formal, que assegura que o sistema funcione adequadamente para todas as entradas possíveis, geralmente através da análise estática de modelos. Por fim, a análise de desempenho busca estimar e garantir as propriedades de tempo de execução do sistema, proporcionando garantias matemáticas sobre seu desempenho (Roychoudhury, 2009).

2.2.4 Estratégias de validação

As estratégias de validação variam conforme a criticidade dos componentes. Para sistemas de alta criticidade, como os que controlam os freios e a direção em automóveis, são necessárias abordagens rigorosas, incluindo modelagem formal e verificação. Para componentes menos críticos, como janelas elétricas, modelagem e testes extensivos são frequentemente suficientes. Já para recursos de entretenimento, métodos de análise de desempenho são aplicados para garantir que as restrições de tempo suave sejam atendidas (Roychoudhury, 2009).

Testes funcionais representam uma etapa crucial na validação de sistemas, focando na verificação do comportamento esperado frente a entradas específicas e cenários predefinidos. Diferentemente de testes estruturais, que avaliam o código internamente, os testes funcionais focam no comportamento externo do sistema, verificando se as funcionalidades descritas na especificação foram corretamente implementadas (Kaner; Falk; Nguyen, 1999). Essa abordagem permite identificar falhas lógicas, inconsistências na interface ou desvios em relação à especificação.

Neste trabalho será feita a abordagem que mistura testes funcionais e lógica de Floyd-Hoare para verificar implementações do protocolo e atestar a acordança das especificações garantindo a funcionalidade em um cenário de pré-integração com outros circuitos. V

3 METODOLOGIA

Nesta seção será descrita a metodologia de formulação de um manual técnico do protocolo DCP, que possui função de especificação, baseado em informações já apresentadas no trabalho de Araújo (2022), além de algumas mudanças que foram vistas como benéficas para a adoção e utilização do protocolo. Também será descrita a criação de um dispositivo para validação do protocolo, que será utilizado para validar um estudo de caso da aplicação do protocolo, com base nas especificações do documento.

3.1 Especificação

O documento de manual técnico do protocolo utilizado para especificação deve atender o formato descrito na revisão da literatura, contendo todas as informações pertinentes à implementação do protocolo. O documento é iniciado com a introdução do protocolo, trazendo informações como a intenção de uso e as principais características do protocolo, voltado para que o leitor saiba se o protocolo atende os requisitos de sua aplicação.

Ainda na introdução são apresentadas as características de funcionamento do protocolo e seus mecanismos como: modos de velocidade, o modelo padrão de comunicação no barramento, mecanismo de *hot-join*, configuração de multicontrolador, e mecanismo de detecção e evitação de colisões (Araújo, 2022).

Em seguida, são apresentadas as características elétricas do protocolo. Esta seção apresenta os valores aceitáveis de tensão, corrente, e capacidade por pino de I/O. Os seguintes dados são apresentados em uma tabela no documento, assim como condições e notas para melhor interpretação:

- V_H (Tensão de Nível Alto):

É a tensão no pino de entrada do receptor quando um sinal lógico alto é aplicado. Este sinal é dependente do v_{DD} escolhido e geralmente é definido como sendo, no mínimo, 70% do valor de V_{DD} .

- V_L (Tensão de Nível Baixo):

Similarmente à V_H , é a tensão no pino de entrada do receptor quando um sinal lógico baixo é aplicado. Como o protocolo não utiliza valores de tensão simétricos, o valor de referência *ground* é utilizado como base, especificando, assim, o valor mínimo de V_i em 0 V, e o valor máximo como sendo 30% do valor de v_{DD} .

- I_{leak} (Corrente de Fuga de Entrada):

É a corrente que entra no pino de entrada de um dispositivo em operação. É dependente do V_{DD} e do resistor *pull-up* utilizado, um resistor de baixo valor, também conhecido como *pull-up* forte, cria uma maior corrente, o que possibilita lidar com barramentos de maior capacidade, porém, à mercê de menor eficiência energética.

Com os parâmetros DC apresentados, o documento segue a demonstração trazendo os parâmetros AC do protocolo, que dizem respeito às frequências e tempos dos sinais de comunicação transmitidos no barramento. Os valores de frequência e de tempo são dependentes da configuração de velocidade utilizada no barramento, com maiores velocidades necessitando tempos e tolerâncias menores. Similarmente aos parâmetros DC, é apresentado uma tabela com os parâmetros pertinentes:

- Velocidade do barramento: Apresenta a região de velocidade aceitável para a transmissão de dados do barramento com valores mínimos e máximos para cada uma das configurações possíveis de velocidade do barramento.
- Rising Time (Tempo de Subida) É o tempo máximo aceitável de subida de um sinal até 90% do valor máximo de tensão do barramento. Não existe um valor mínimo. Entretanto, o tempo de subida é fisicamente limitado pela constante de tempo RC do barramento, e um sinal com tempo de subida muito lento acarreta em problemas de transmissão caso passe do limite indicado.

Dado que o barramento sobe a tensão utilizando o resistor *pull-up* e que a resistência interna do barramento é muito menor que a do resistor *pull-up*, ou seja $R_{bus} \ll R_{PUN}$, o tempo de subida é diretamente relacionado ao valor do resistor e à capacidade do barramento, constituindo a constante de tempo RC de subida.

- Falling Time (Tempo de Descida) É o tempo máximo aceitável de descida de um sinal até 10% do valor máximo de tensão do barramento, e, similarmente ao tempo de subida, o tempo de descida não tem um valor mínimo, porém também é limitado pela constante de tempo RC.

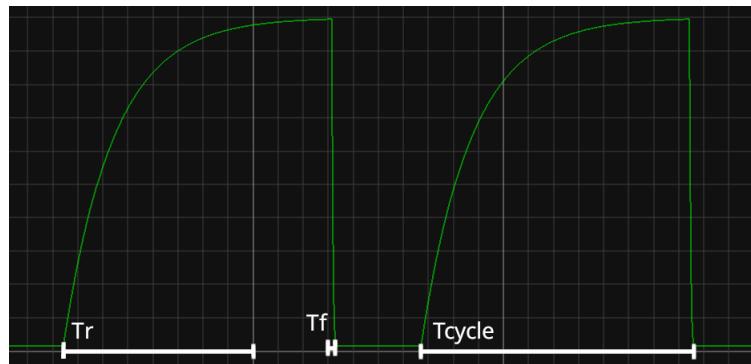
Entretanto, diferentemente do nível alto de tensão, o barramento tem sua tensão abaixada pelo dispositivo que está controlando o barramento. Logo, assumindo que o resistor *pull-up* possua magnitude muito maior que a resistência interna do pino de dispositivo, ou seja $R_{PUN} \gg R_{PDN}$, a constante de tempo é dado pela capacidade do barramento somada à capacidade do pino de I/O do dispositivo e a resistência de ambos até o terminal de tensão de referência *ground* comum.

- Cycle Time (Tempo de Ciclo) O tempo de ciclo é o tempo que leva para a tensão completar um ciclo completo, indo de 10% ao valor máximo e de volta a 10%. Esta é uma métrica de especial importância para o protocolo pois, dada a utilização de

um único fio, o sinal do barramento está sempre demarcando o ciclo de *clock* com os dados modulados no sinal. O tempo de ciclo leva em consideração o tempo de subida, tempo de acomodação e tempo de descida do barramento.

Estes parâmetros são ilustrados na figura 2, que demonstra o ciclo de carga e descarga de um circuito RC.

Figura 2: ilustração dos parâmetros de tempo AC de uma carga capacitiva.



Fonte: Autoria própria.

Com a apresentação dos parâmetros AC e DC, o documento apresenta os sinais básicos do protocolo. Estes sinais foram especificados no trabalho de Araújo (2022) e são variáveis de acordo com a escolha do valor de Δ . Os sinais básicos do protocolo são definidos como:

- bit 0: Pulso de nível alto seguido de um pulso de nível baixo, ambos de duração Δ s.
- bit 1: Pulso de nível alto, de duração 2Δ s, seguido de um pulso de nível baixo de duração Δ s.
- *Start Sync*: Este sinal diferencia um dispositivo controlador de um dispositivo alvo e deve ser enviado no início das transmissões. Para um dispositivo dispositivo controlador o *Start Sync* é definido como um pulso de nível baixo com duração de 25Δ s. E, para um dispositivo alvo, o *Start Sync* é definido como um pulso de nível lógico baixo de duração de 50Δ s.
- *Start Bit*: Este sinal indica o fim do *Start Sync* e início da transmissão de dados. É composto por um nível alto de duração 7.5Δ s, seguido de um nível baixo também de duração 7.5Δ s.

No trabalho de Araújo (2022), também é apresentado um sinal chamado de Pulso de Sincronismo, que tinha como objetivo realizar a sincronia dos *clocks* dos dispositivos. Este sinal foi julgado como redundante neste trabalho e foi removido da especificação do

manual técnico do protocolo. Sua funcionalidade foi integrada ao sinal de *Start Sync*, que terá função dual de identificação de dispositivo e sincronização.

Por fim, o documento apresenta diagramas de tempos mostrando os sinais apresentados anteriormente e demonstrando os tempos que devem ser respeitados, de acordo com os valores mostrados nas tabelas de parâmetros AC e DC. E finaliza com exemplos ilustrativos de circuitos utilizando o barramento para elucidar possíveis dúvidas sobre conexões no design da implementação.

3.2 Validação

3.2.1 Metodologia de Validação

Dada a abordagem escolhida e a arquitetura da aplicação, o dispositivo de validação é construído de maneira que ele se conecta a um barramento de dispositivos, ou diretamente a um único dispositivo, realiza testes elétricos e funcionais de maneira sequencial, e guarda o resultado de cada teste para ser transmitido no fim para a visualização dos resultados. O barramento, os dispositivos, ou o dispositivo único, são o Dispositivo Sob Teste: DUT (Device Under Test), e futuras menções a estes dispositivos utilizarão a terminologia DUT.

O dispositivo validador realiza os seguintes passos:

- Configuração do Ambiente de Teste: O validador recebe uma configuração do teste a ser realizado e reajusta seus parâmetros para realizar os testes.
- Emulação de Controlador ou Alvo: Dada a configuração recebida, o validador modifica seu comportamento para agir como um dispositivo controlador e enviar comandos ao barramento; como dispositivo alvo, enviando informações para o barramento apenas quando requisitado; ou realizar o teste primeiro como um controlador, e depois como um dispositivo alvo.
- Medição dos parâmetros elétricos: O validador realiza testes checando os valores dos níveis de tensão alto e baixo, visando, além do atendimento ao protocolo, enriquecer a coleta de informações sobre o DUT para a criação do relatório final. Assim como a medição da corrente de carga, que é útil para estimativas de consumo energético.
- Medição dos tempos de comunicação: O validador realiza a medição dos tempos dos sinais: tempo de subida, tempo de descida e tempo de ciclo; checando se estão dentro da faixa de sinais aceitáveis especificadas no protocolo.
- Medição dos parâmetros funcionais: O validador realiza testes funcionais do protocolo, como o envio de sinais adequados em momentos adequados, por exemplo, o

envio Start Sync no início de uma transmissão; o atendimento às regras de controle do barramento, que não permite escrever no barramento enquanto outro dispositivo escreve; e realizar o *backoff* do barramento caso seja arbitrado que o dispositivo não pode escrever no barramento em determinado momento.

- Teste de falso positivo: O validador realiza testes com sinais que não estão adequados ao protocolo e analisa o comportamento do DUT perante esses sinais.
- Documentação e Relatório: As informações coletadas pelos testes são disponibilizadas para serem visualizadas na forma de um relatório com os testes realizados e informações sobre os motivos de falhas.

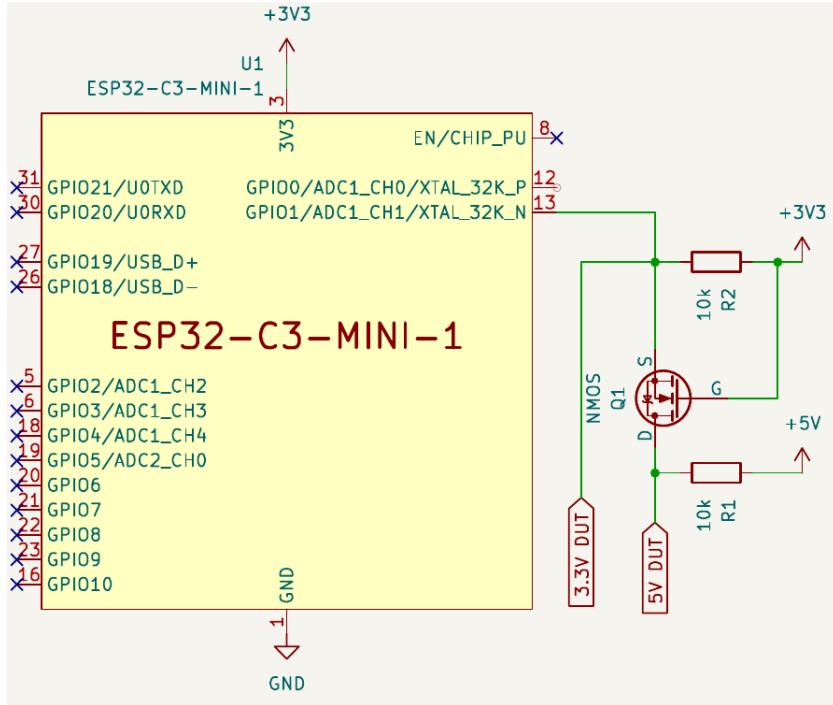
Os testes têm formato de asserção, onde, dado o estado atual do DUT, após um sinal, o dispositivo deve assumir um outro estado sem entrar em uma condição de travamento, atendendo à metodologia de Floyd-Hoare híbrida. Os testes podem ser divididos em 3 etapas com objetivo de: configurar o DUT tal que ele atenda à pré-condição do teste; iniciar um contador de *timeout* que acusa uma condição de travamento; e realizar checagem de atendimento da pós-condição esperada, que, dado o formato predicativo, possui uma resposta de verdadeiro ou falso, que condizem com teste bem sucedido ou mal sucedido, respectivamente.

3.2.2 Dispositivo de Validação

***Hardware* do dispositivo:**

O dispositivo é composto por: um microcontrolador ESP32-C3, escolhido pela sua capacidade de acessar a Internet com seu adaptador Wi-Fi e capacidade de hospedar um servidor web simples sem necessidade de adição de componentes externos. Também temos um circuito conversor de nível composto por um *MOSFET* e resistores, utilizado para atender à necessidade de se conectar a barramentos que utilizam uma tensão nominal maior que 3,3 V. O esquemático do dispositivo demonstrando a utilização dos componentes está ilustrado na figura 3.

Figura 3: Esquemático do *hardware* do dispositivo de validação.



Fonte: Autoria própria.

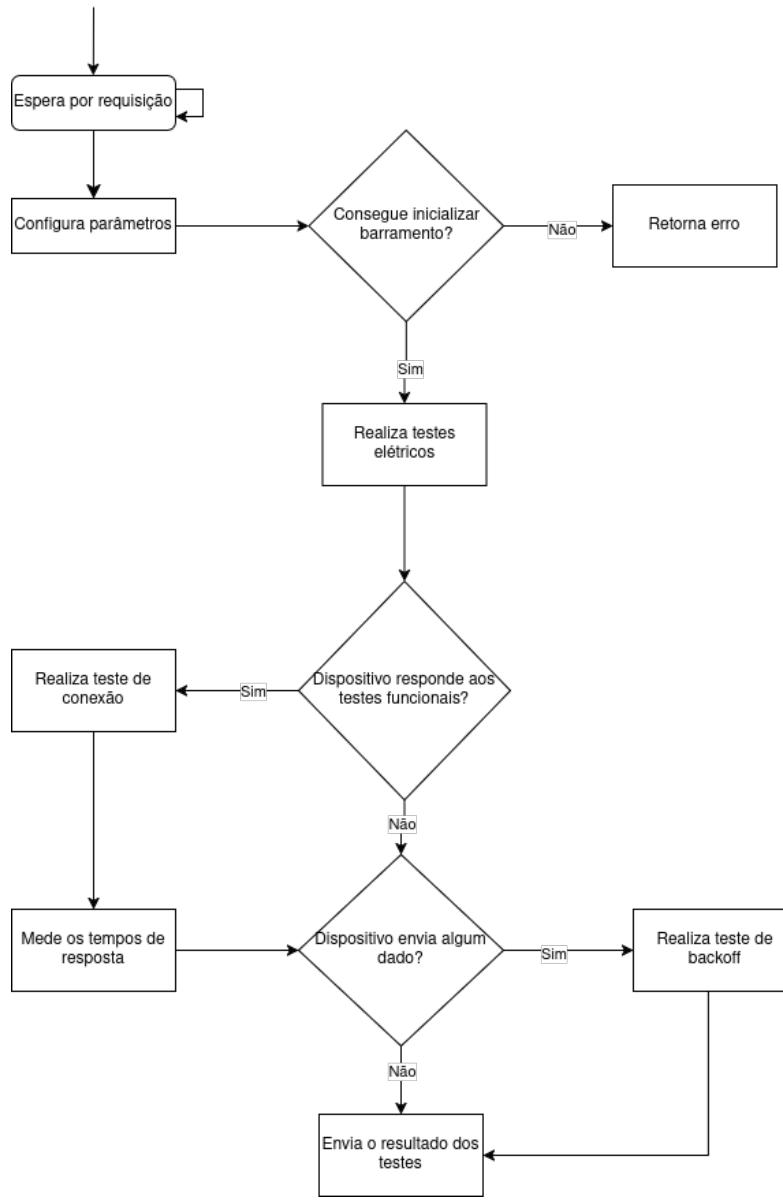
Firmware do dispositivo:

O firmware do dispositivo de validação pode ser separado em 4 componentes: configuração, testes, *backend* e *frontend* do *webserver*. O componente de configuração diz respeito à inicialização e configuração básica do microcontrolador para que seja possível executar os componentes subsequentes de maneira correta e é composto por inicialização da memória *flash*, interface de rede e da funcionalidade *Wi-Fi*. Assim como a subsequente conexão à estação base configurada nos parâmetros de configuração contidos na memória *flash*, configuração de *DNS* e, por fim, inicialização do serviço *HTTP*.

O componente de *backend* é iniciado em conjunto com o serviço *HTTP*. Este componente é responsável por receber e responder à requisições *HTTP* utilizando a rotina adequada. Dada a simplicidade do validador, apenas 2 *endpoint* são expostos: um para enviar os arquivos do *frontend* e um para receber os parâmetros do DUT, realizar a rotina de testes e enviar os resultados.

O fluxograma da figura 4 ilustra o algoritmo utilizado na rotina de teste do DUT. Quando inicializada, a rotina espera pela requisição repassada pelo *backend* com os parâmetros do DUT, ao receber, a rotina configura os parâmetros de acesso ao barramento baseados nos parâmetros do DUT e tenta inicializar suas rotinas de acesso ao barramento, caso não consiga, retorna erro para a rotina de *backend*. Caso consiga acessar o barramento, o validador realiza em sucessão testes elétricos, funcionais e de *backoff*, guardando os resultado de cada teste para, ao final, retornar os resultados para o *backend*.

Figura 4: Fluxograma do algoritmo para realização de testes implementado.



Fonte: Autoria própria.

Os testes elétricos são responsáveis por realizar as medições e avaliações da camada física do DUT. O validador realiza as medições dos parâmetros elétricos DC e AC listados no manual técnico utilizando o conversor A/D sem se preocupar com a forma ou significado do pulso, apenas seus valores quantitativos. Após realizar as medições o dispositivo testa os dados coletados com os limites da especificação do protocolo. Essa etapa traz similaridades e pode ser pensada como uma análise sintática do protocolo.

De maneira semelhante, os testes funcionais avaliam o desempenho do DUT para garantir que ele funcione adequadamente em relação a outros dispositivos. Nesta etapa os valores de tensão não são levados em consideração, entretanto a forma e duração dos pulsos no barramento são analisados, assim como a capacidade do DUT responder nas

janelas de tempo adequadas quando solicitado, e não responder quando não for solicitado, assim como a capacidade do dispositivo de atender às regras de acesso ao barramento. Assim como a etapa anterior, essa etapa tem similaridades e pode ser pensada como a análise semântica do protocolo.

Por fim, o teste de *backoff* necessita que o DUT envie alguma mensagem no barramento. Quando o DUT realiza a transmissão, o validador realiza uma colisão proposital para testar a capacidade do DUT de perceber uma colisão e abortar sua transmissão.

E finalmente, o componente *frontend* é o responsável por implementar a interface de usuário do validador e é composto pelo conjunto padrão de arquivos *frontend*: JavaScript, CSS e HTML. O conjunto de arquivos que compõe o componente são enviados pelo *backend* para o computador cliente do *webserver*, que renderiza a interface constituída pelo modelo interativo de relatório, o qual é automaticamente preenchido de acordo com os parâmentros do DUT fornecidos pelo usuário. Além disso, o *frontend* fornece a capacidade de salvar o relatório para arquivagem ou comparação dentre DUTs.

Modelo de Relatório:

O modelo de relatório de validação que será gerado pelo dispositivo validador é apresentado na figura 5 e exemplifica as principais etapas da validação DUT. O documento inclui informações iniciais cruciais, como o nome do dispositivo sob teste, a velocidade de comunicação, o tipo de acesso ao barramento, a versão da implementação e a data de emissão. Os resultados dos testes realizados são organizados em duas tabelas, comparando os valores esperados com os obtidos, e fornecendo uma conclusão sobre a conformidade do dispositivo. Em caso de falhas durante a validação, há uma seção dedicada que detalha cada falha, explicando sua causa e sugerindo possíveis melhorias quando cabível.

Figura 5: Modelo de relatório de validação de dispositivo.

DCP-IF	Date of Emission: May 13, 2025	Validation Version: 0.1																																		
DCP Validation Report																																				
Name: Device Name	Device Type: Target																																			
Device Version: 1.0	Device Speed: 4 MHz																																			
Specification Conformity																																				
<table border="1"> <thead> <tr> <th>Parameter</th><th>Expected</th><th>Got</th><th>Result</th></tr> </thead> <tbody> <tr> <td>Speed Test</td><td>4 MHz</td><td>4 MHz</td><td>Pass</td></tr> <tr> <td>Bit High</td><td>40 μs</td><td>40.32 μs</td><td>Pass</td></tr> <tr> <td>Bit Low</td><td>20 μs</td><td>19.73 μs</td><td>Pass</td></tr> <tr> <td>Sync Time</td><td>250 μs</td><td>200 μs</td><td>Fail¹</td></tr> <tr> <td>Bit Sync</td><td>300 μs</td><td>300 μs</td><td>Pass</td></tr> <tr> <td>Bit Sync High</td><td>150 μs</td><td>150 μs</td><td>Pass</td></tr> <tr> <td>Bit Sync Low</td><td>150 μs</td><td>150 μs</td><td>Pass</td></tr> <tr> <td>Bus Yield</td><td>Yes</td><td>No</td><td>Fail²</td></tr> </tbody> </table>	Parameter	Expected	Got	Result	Speed Test	4 MHz	4 MHz	Pass	Bit High	40 μ s	40.32 μ s	Pass	Bit Low	20 μ s	19.73 μ s	Pass	Sync Time	250 μ s	200 μ s	Fail ¹	Bit Sync	300 μ s	300 μ s	Pass	Bit Sync High	150 μ s	150 μ s	Pass	Bit Sync Low	150 μ s	150 μ s	Pass	Bus Yield	Yes	No	Fail ²
Parameter	Expected	Got	Result																																	
Speed Test	4 MHz	4 MHz	Pass																																	
Bit High	40 μ s	40.32 μ s	Pass																																	
Bit Low	20 μ s	19.73 μ s	Pass																																	
Sync Time	250 μ s	200 μ s	Fail ¹																																	
Bit Sync	300 μ s	300 μ s	Pass																																	
Bit Sync High	150 μ s	150 μ s	Pass																																	
Bit Sync Low	150 μ s	150 μ s	Pass																																	
Bus Yield	Yes	No	Fail ²																																	
Transmission information Electrical Information																																				
<table border="1"> <thead> <tr> <th>Parameter</th><th>Result</th></tr> </thead> <tbody> <tr> <td>Transmission Type</td><td>L3</td></tr> <tr> <td>Valid Header</td><td>Pass</td></tr> <tr> <td>Valid Source ID</td><td>Pass</td></tr> <tr> <td>Valid Padding</td><td>Pass</td></tr> <tr> <td>Valid CRC</td><td>Pass</td></tr> </tbody> </table>	Parameter	Result	Transmission Type	L3	Valid Header	Pass	Valid Source ID	Pass	Valid Padding	Pass	Valid CRC	Pass																								
Parameter	Result																																			
Transmission Type	L3																																			
Valid Header	Pass																																			
Valid Source ID	Pass																																			
Valid Padding	Pass																																			
Valid CRC	Pass																																			
<table border="1"> <thead> <tr> <th>Parameter</th><th>Value</th></tr> </thead> <tbody> <tr> <td>Bus Max Speed</td><td>117 MHz</td></tr> <tr> <td>VIH (High-level input voltage)</td><td>3.3 V</td></tr> <tr> <td>VIL (Low-level input voltage)</td><td>0.7 V</td></tr> <tr> <td>Ileak (Leakage current)</td><td>20 μA</td></tr> <tr> <td>Rise Time</td><td>1 μs</td></tr> <tr> <td>Falling Time</td><td>0.2 μs</td></tr> <tr> <td>Cycle Time</td><td>1.5 μs</td></tr> </tbody> </table>	Parameter	Value	Bus Max Speed	117 MHz	VIH (High-level input voltage)	3.3 V	VIL (Low-level input voltage)	0.7 V	Ileak (Leakage current)	20 μ A	Rise Time	1 μ s	Falling Time	0.2 μ s	Cycle Time	1.5 μ s																				
Parameter	Value																																			
Bus Max Speed	117 MHz																																			
VIH (High-level input voltage)	3.3 V																																			
VIL (Low-level input voltage)	0.7 V																																			
Ileak (Leakage current)	20 μ A																																			
Rise Time	1 μ s																																			
Falling Time	0.2 μ s																																			
Cycle Time	1.5 μ s																																			
Failure Details																																				
<ol style="list-style-type: none"> Sync Time did not hold the bus low for the specified time. The device failed to yield the bus in a collision scenario. 																																				

Fonte: Autoria própria.

3.3 Estudo de Caso

3.3.1 Contextualização

Para estudo de caso, é proposto um sistema composto de 3 tipos de dispositivos, os 3 tipos podem ser definidos como: Agregador, Sensor, e interface, interligados por meio de um barramento *DCP*. Estes 3 dispositivos formam uma arquitetura que exemplifica tipos padrões de dispositivos encontrados pela indústria, no qual o sensor realiza amostragens de grandezas físicas e envia para outro local; a interface cuida das entrada e saída do dispositivo, realizando o controle de componentes de comunicação entre máquinas, ou entre humanos; e o agregador demonstra o funcionamento de um dispositivo controlador central, que organiza, regula e sincroniza o funcionamento de periféricos ou dispositivos ligados ao mesmo.

Estes 3 dispositivos representam, entre si, subsistemas que podem ser analisados como sistemas por si mesmos, cada subsistema é composto de um microcontrolador responsável por orquestrar seus periféricos e realizar a comunicação com o barramento DCP. Neste caso, cada subsistema precisa se importar apenas com a comunicação com o agregador, sem se preocupar com os outros sistemas conectados ao barramento e sem se importar com a implementação do próprio agregador. Cada subsistema, necessita atender às especificações do protocolo de comunicação e implementar o mecanismo de *hot-join*. Dessa maneira, mesmo que neste estudo de caso só 3 subsistemas sejam implementados, pela natureza do protocolo e arquitetura do projeto, teoricamente mais subsistemas similares podem se conectar ao barramento, contanto que atendam aos requisitos.

Neste estudo de caso, o subsistema sensor é composto de um microcontrolador, e 1 ou mais periféricos sensores. Este subsistema é responsável por realizar amostras de variáveis e enviar para o agregador, com funcionamento parametrizado de acordo com comandos do agregador. Seu comportamento padrão é enviar dados de maneira autônoma e periódica, e receber poucos dados de configuração.

O subsistema de interface é composto de um microcontrolador, e, no mínimo, um periférico de entrada e um de saída. O subsistema é responsável por receber dados do agregador e utilizar seu periférico de saída de dados para mostrar ou enviar os dados recebidos, e similarmente, os dados recebidos pelo periférico de entrada devem ser repassado ao agregador assim que possível. Seu comportamento padrão é receber dados do agregador, possivelmente provenientes do subsistema sensor, e enviar poucos dados para o agregador.

Por fim, o subsistema agregador é o centralizador nesta aplicação. Este subsistema, como supracitado, é o responsável pelo intermédio da comunicação entre o subsistema sensor e o subsistema de interface, além de ser o componente que realiza a configuração

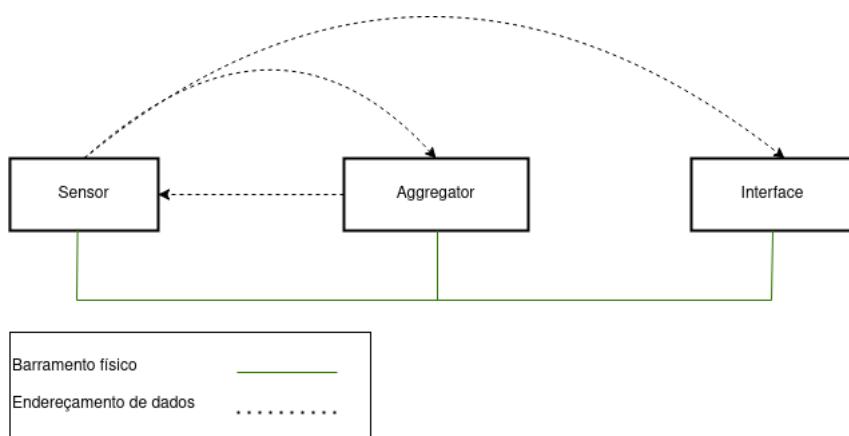
dos outros subsistemas. É importante ressaltar que o intermédio do agregador não é estritamente necessário para o funcionamento de um dispositivo que utiliza o DCP.

Aludindo ao nome do sistema, este subsistema pode ser utilizado para agrregar informações de vários subsistemas sensores, e decidir para quais possíveis subsistemas de interface enviar as informações, como também, pode ser responsável por realizar uma função de *log*, gerar estatísticas sobre o fluxo de dados, ou guardar informações em uma possível memória não-volátil. Além disso, dada a natureza do protocolo e arquitetura do projeto, mesmo que haja um subsistema agregador, dois subsistemas podem se comunicar diretamente utilizando seus respectivos endereços.

Os sistemas se comunicam utilizando um barramento comum. Por conta disto, uma mensagem enviada por um sistema é recebida por todos os outros sistemas. Porém, quando um dispositivo não se importa com a mensagem de outro dispositivo em específico, ele pode ignorá-la. Assim, podemos ter um sistema mandando mensagem para um ou mais sistemas em específico, mesmo que as mensagens não sejam explicitamente endereçadas para eles. Portanto, é possível trazer o conceito de fluxo de mensagens entre sistemas, mesmo que na prática todos os sistemas recebam as mensagens.

A figura 6 demonstra a arquitetura do sistema em questão, onde 3 subsistemas estão conectados por um barramento DCP comum e eles geram mensagens entre si, criando um fluxo de mensagens pelo barramento, representado pela linha contínua. Ainda que fisicamente as mensagens sejam recebidas por todos os dispositivos no barramento, podemos utilizar endereçamento e filtragem de mensagens para criar um fluxo de mensagens entre dispositivos específicos utilizando conceitos da camada de enlace, representado pela linha tracejada. Neste caso, o todos os outros subsistemas esperam e recebem mensagens do subsistema sensor, porém, quando o agregador envia uma mensagem no barramento, o subsistema de interface a ignora, afinal, não é de seu interesse, enquanto o sistema sensor recebe e processa a mensagem.

Figura 6: Arquitetura conceitual do estudo de caso.



Fonte: Autoria própria.

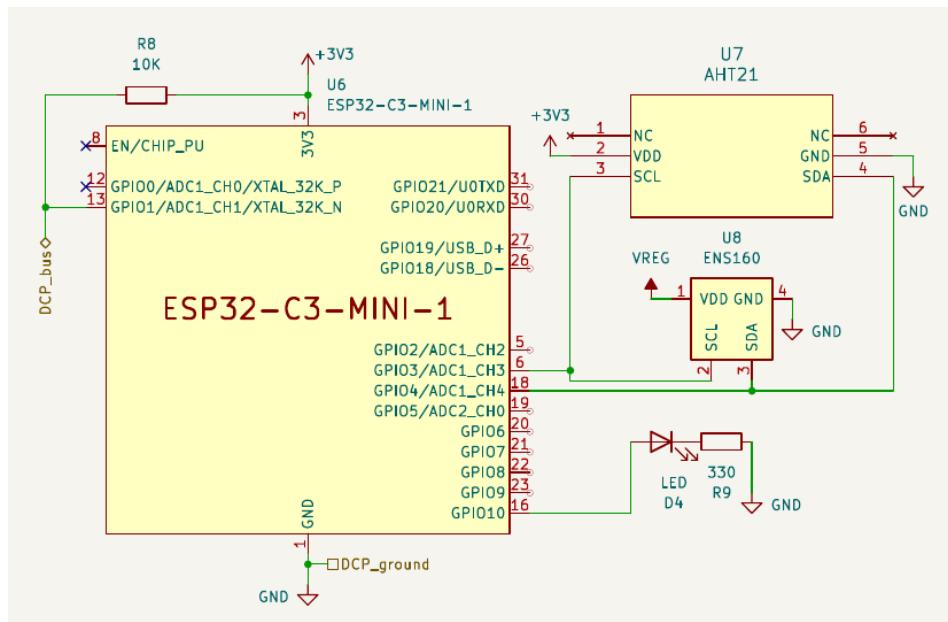
3.3.2 Implementação

Para a implementação do estudo de caso, é proposto um sistema composto por 3 microcontroladores: ESP32-C3, RP2350 e STM32F411, que serão utilizados para implementar os tipos de sistemas discutidos na subseção anterior. Cada microcontrolador é responsável por controlar seu subsistema e que se comunicam utilizando um barramento DCP comum. A escolha de diferentes microcontroladores foi feita para enfatizar a agnoscitidade do protocolo à uma tecnologia específica.

O sistema de sensoriamento é composto pelo ESP32-C3, o sensor de temperatura e umidade relativa AHT21, e o sensor de qualidade do ar ENS160, que fornece o índice de qualidade do ar (AQI), quantidades de partículas voláteis no ar (TVOC), e valor equivalente de partículas de CO₂ no ar (eCO₂). O microcontrolador se comunica com ambos sensores utilizando um barramento I²C, lendo os dados de ambos sensores a cada segundo e enviando os dados pelo barramento DCP.

Além disso, nessa implementação, o sistema de sensoriamento também recebe dados do barramento para demonstrar a capacidade do protocolo de também receber dados pela mesma interface em que se enviam dados. Para a demonstração desta capacidade, é utilizado um LED, que tem seu acionamento ativado pelo sistema agregador. O esquemático do sistema é ilustrado na figura 7.

Figura 7: Esquemático do sistema de sensoriamento.



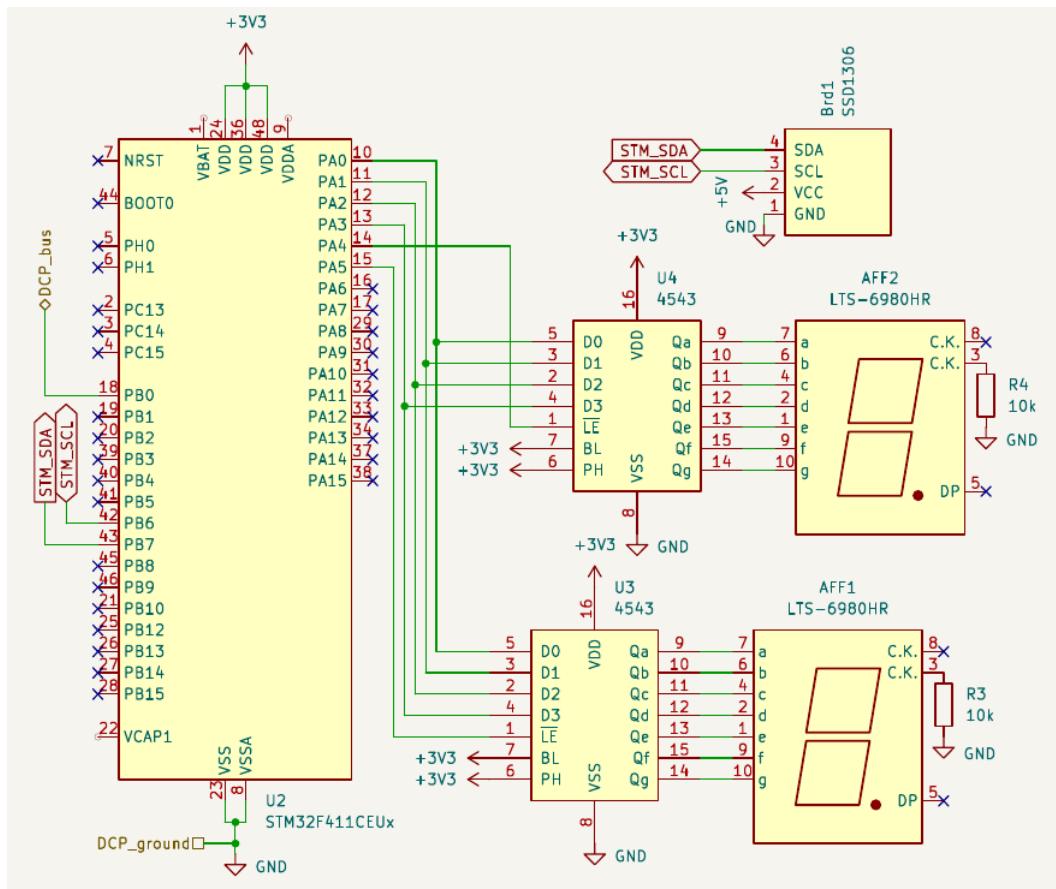
Fonte: Autoria própria.

O sistema de interface é composto de um STM32F411, conectado a um display *OLED* com resolução de 128×64 pixels, com o driver SSD1306 (Systech, 2008), que provê conectividade I²C, além de 2 displays de 7 segmentos, controlados por chips HEF4511B

(Philips, 1995), que decodificam dados *Binary-Coded Decimal* para as combinações de LEDs que representam o algarismo indo-árabe do valor decimal. Os chips de drivers não são estritamente necessários para interface com os displays, porém, nessa aplicação eles são utilizados primariamente para *multiplexar* pinos de dados do microcontrolador, sendo os pinos de dados de ambos *drivers* conectados em paralelo, e o pino *Enable Latch* (\overline{EL}) utilizado para selecionar qual display receberá os dados no barramento. Esses componentes são ilustrados no esquemático do sistema na figura 8.

Decorrente da resolução do display *OLED*, é possível criar uma *string* que consiga demonstrar todas as informações que percorrem o barramento. Portanto, este display é utilizado para demonstrar os dados dos sensores do sistema de sensoriamento, em conjunto de suas respectivas unidades de medida. Enquanto, dado o fato que os displays de 7 segmentos consegue apenas representar 1 número cada, eles serão utilizados para mostrar a parcela inteira do valor de temperatura, também enviado pelo sistema de sensoriamento.

Figura 8: Esquemático do sistema de interface.



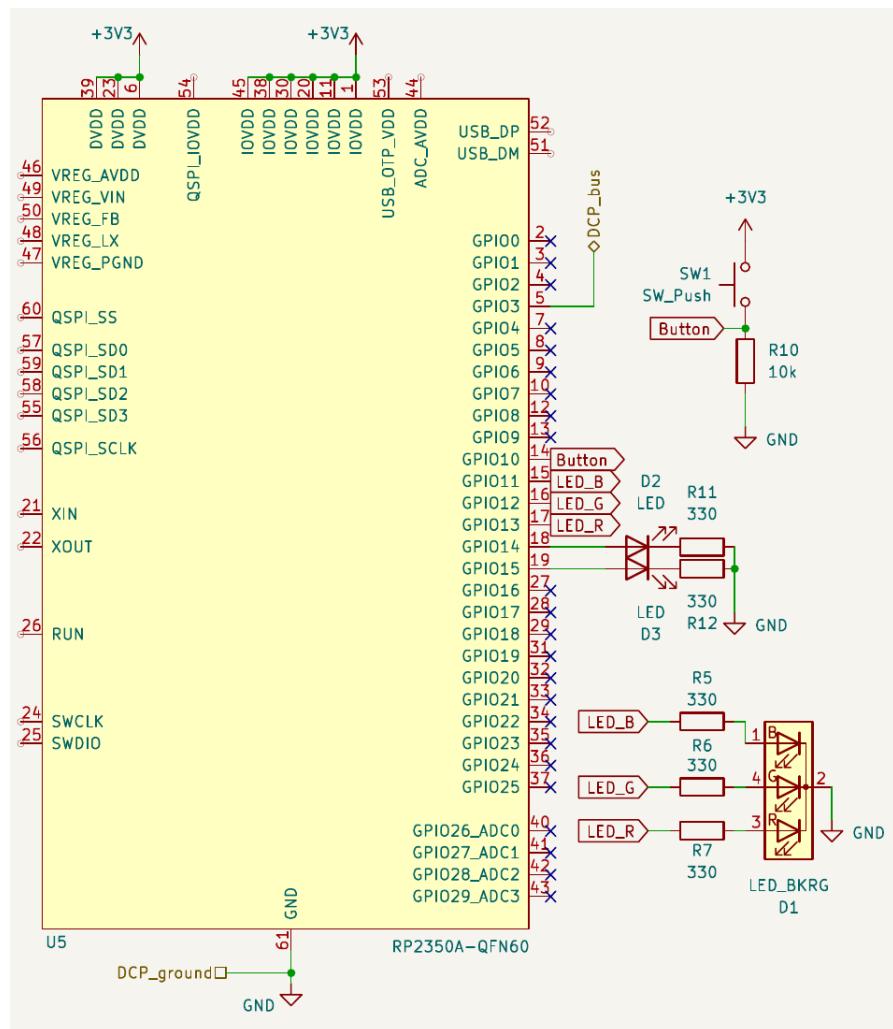
Fonte: Autoria própria.

O sistema de agregação é formado por um RP2350, embutido na placa Raspberry Pico 2, e possui 3 LEDs e 1 botão conectados aos seus pinos. Este sistema lê os dados de temperatura no barramento, e aciona os LEDs caso os valores de temperatura e eCO₂

ultrapassem limites pré-configurados, e também indica continuamente qual é o nível da qualidade do ar utilizando um LED RGB. O esquemático do sistema é ilustrado na figura 9.

O botão é utilizado para controlar o LED no sistema de sensoriamento. Este controle é feito enviando uma mensagem com codificação L3 endereçada ao ESP32 com um comando para ligar o LED quando o RP2350 detecta uma borda de descida em seu pino de entrada, e uma mensagem com um comando para desligar o LED ao detectar uma borda de subida. Esta abordagem tem a vantagem de delegar a leitura do botão e controle do LED para sistemas que estão separados fisicamente e não mantém o barramento ocupado enquanto o botão está ativo, apenas enviando mensagens quando ocorre um evento notável, neste caso, a mudança de estado do botão.

Figura 9: Esquemático do sistema de agregação.



Fonte: Autoria própria.

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

4.1 Especificação

No Apêndice A é apresentado o manual técnico do protocolo, que compila de maneira resumida informações importantes para a implementação e utilização do protocolo, sendo assim, um documento útil para contextualização de desenvolvedores que planejam utilizar o protocolo, ou apenas conhecê-lo. O documento foi feito na língua inglesa pelo fato que atualmente é a língua franca para desenvolvimento de hardware, tornando o documento acessível para uma maior audiência.

O documento apresenta uma breve introdução da ideia do protocolo e contextualização de possíveis usos. Seguido de uma introdução às capacidades do protocolo e suas características. Por fim, traz o contexto da unidade de tempo Δ e sua importância para a utilização do protocolo. Essa última seção foi adicionada decorrente da dependência de todos os outros parâmetros ao Δ , explicitando a importância da priorização de sua escolha como um dos passos iniciais do desenvolvimento de um sistema.

Em seguida se inicia a parte técnica do documento, que entra em maiores detalhes nas características do protocolo, trazendo informações sobre como ocorre a comunicação, o mecanismo de *Hot-Join*, a configuração multicontrolador e a detecção de colisão.

Após isso, são apresentadas as tabelas de especificações elétricas e de temporização. Essas seções trazem valores a serem considerados durante o desenvolvimento de sistemas que utilizam o protocolo. Estas versões das tabelas foram feitas para trazer atenção a valores de parâmetros que, caso estejam fora do esperado, geram erros durante a utilização do protocolo. Os valores são flexíveis para modificação caso o projetista ache melhor, porém, seguir os parâmetros garantem que o dispositivo irá funcionar sem problemas, e, principalmente, garante o funcionamento com outros dispositivos que seguem a especificação. Como deve ser o papel de qualquer especificação.

Em seguida, é apresentado um exemplo mostrando a forma de todos os sinais fundamentais apresentados no documento, sendo o início de uma transmissão na velocidade *SLOW*, porém, mesmo que a velocidade seja modificada, dada a dependência dos parâmetros ao valor de Δ , as formas de todos os sinais se mantêm a mesma, mudando apenas seu tamanho no domínio do tempo.

Ao fim, é demonstrado um exemplo de configuração de dispositivos em um barramento, contendo 3 dispositivos ligados ao barramento, cada um com um resistor de *pull-up* para compensar pela capacitância extra adicionada pela sua presença no barramento. Esta não é a única configuração possível, porém dá uma ideia genérica de como conectar dispositivos ao barramento.

Esse documento não é uma especificação completa e comprehensiva, e muitas informações ainda devem ser adicionadas, porém, como foi dito, já pode fazer o papel de um documento para primeiro contato com o protocolo. Trabalhos futuros podem complementar esse documento para que se torne, de fato, uma especificação comprehensiva. Além disso, mais exemplos e diagramas de temporização mais comprehensivos se fazem necessário.

4.2 Dispositivo de Validação

Para o dispositivo de validação, foi implementada uma arquitetura baseada em sistemas *RESTful*, que é utilizado pela grande maioria dos *websites* dos dias atuais (Richardson; Amundsen; Ruby, 2013). O sistema pode ser separado em 2 partes: um *frontend* executado no navegador do dispositivo cliente que realiza o acesso ao *website* e um *backend* feito em C, executado no microcontrolador e espera as requisições do *frontend*.

Dada a simplicidade da página, o *frontend* do *webserver* foi implementado utilizando HTML, CSS, e JavaScript, sem a necessidade de utilizar um *framework* de programação. Os arquivos foram salvos em uma partição da memória flash do ESP32-C3 e servidos utilizando as funções de servidor HTTP do *framework* ESP-IDF (Espressif, 2016). Na imagem 10 temos a apresentação do *frontend*, cujo código-fonte pode ser encontrado no Apêndice B.

Ao ser ligado, o dispositivo de validação se conecta à rede *Wi-Fi* e abre um servidor local, que pode ser acessado por qualquer dispositivo que esteja na mesma rede. Acessando o endereço do dispositivo, recebido por *DHCP*, o dispositivo apresenta o formulário de validação, com campos do texto que podem ser preenchidos livremente pelo usuário, e campos com seleções com os parâmetros necessários para realizar os testes do dispositivo.

A figura 11 apresenta o dispositivo validador conectado à outro dispositivo, um ESP32-C3, que implementa o protocolo. E, dado que o ESP32-C3 utiliza lógica em 3,3V, o conversor de nível não se faz necessário, porém ele também é apresentado na imagem. Pela imagem pode-se ver que o validador se conecta com o DUT em 3 pontos, sendo eles: 5V para alimentação do DUT; GND, para tensão de referência; e o pino de interface com o barramento DCP, neste caso, já com um resistor *pull-up* de 10 kΩ.

Figura 10: Frontend do dispositivo de validação.

DCP Validation Report

DCP-IF Date of Emission: April 22, 2025
Validation Version: 0.1

Name:	<input type="text" value="Enter device name"/>
Device Version:	<input type="text" value="Enter version"/>
Device Type: *	<input type="text" value="Selecione"/>
Device Speed: *	<input type="text" value="Selecione"/>

* Required fields for validation

Specification Conformity

Parameter	Expected	Got	Result
Speed Class	-	-	-
Bit High Time	-	-	-
Bit Low Time	-	-	-
Sync Time	-	-	-
Bit Sync Time	-	-	-
Bit Sync High	-	-	-
Bit Sync Low	-	-	-
Bus Yield	-	-	-

Transmission Information

Parameter	Result
Transmission Type	-
Sync	-
BitSync	-
Size	-

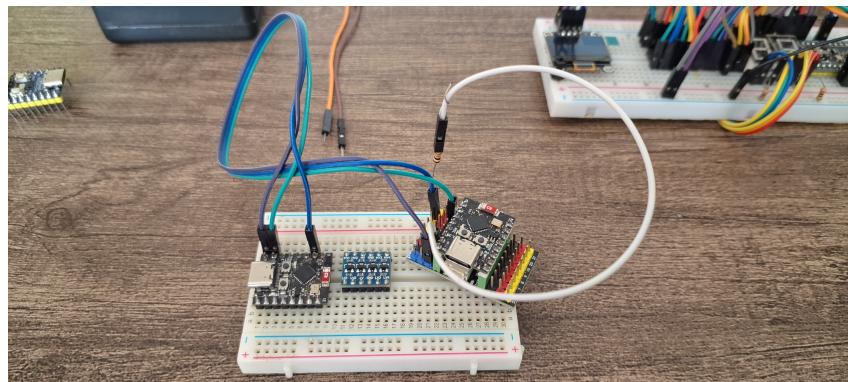
Electrical Information

Parameter	Value
VIH	-
VIL	-
Rise Time	-
Falling Time	-
Cycle Time	-
Bus Max Speed	-

[Fetch Data](#) [Download as PDF](#)

Fonte: Autoria própria.

Figura 11: Dispositivo de validação (à direita) ligado à um DUT de 3.3V.



Fonte: Autoria própria.

Ao selecionar os parâmetros do dispositivo, a página preenche os parâmetros padrões que são esperados do *DUT*, e libera a funcionalidade do botão de teste, rotulado "*Fetch Data*". Clicar nesse botão ativa uma requisição para o validador com os parâmetros selecionados e inicia a rotina de testes. Dentre estes testes estão testes elétricos e funcionais cujo código-fonte é apresentado no Apêndice C.

Caso o dispositivo esteja conectado ao *DUT*, conforme a figura 11, o validador realiza os testes no dispositivo com um *timeout* de 10 segundos. Caso não ocorra nenhuma transmissão nesse intervalo de tempo ou caso o validador esteja desconectado do barramento, o servidor responderá *Internal Server Error* para a requisição. Caso contrário, responderá com os resultados dos testes em formato JSON. Um exemplo, com valores fantasia pode ser encontrado a seguir:

```
{
  electricalInfo: {
    "VIH (High-level input voltage)": 5.1,
    "VIL (Low-level input voltage)": 0.2,
    "Rise Time": 12.033,
    "Falling Time": 0.474,
    "Cycle Time": 12.542,
    "Bus Max Speed": 105.2892,
  },
  transmissionInfo: {
    Type: 0,
    Sync: {status: true},
    BitSync: {status: true},
    Size: {status: true},
    L3: {status: true}
  },
  specConformity: {
    "Speed": 32,
    "Bit High Time": 5,
    "Bit Low Time": 2.5,
    "Sync Time": 62,
    "Bit Sync Time": 37.5,
    "Bit Sync High": 19,
    "Bit Sync Low": 19,
  }
}
```

```

    "Bus Yield": true
}
}

```

Ao receber a resposta do servidor, o *frontend* mostra uma mensagem de erro, ou, no caso de êxito, popula os valores nas tabelas da página com os valores recebidos, demonstrando também se o parâmetro está conforme o esperado seguindo o modelo na figura 5.

Por fim, ao clicar no botão "Download as PDF", é possível baixar a página como um arquivo em formato PDF ou enviar para impressão para fins de arquivamento. Ademais, dada a natureza do sistema, o endpoint "api/v1/validation", que inicia os testes, está disponível para requisições com método *POST*, tornando possível que outro *frontend* ou aplicação seja desenvolvida para consumir os dados e dê outro destino aos resultados, como por exemplo: guardar os dados em um banco de dados ou realizar automação do processo de validação de vários dispositivos.

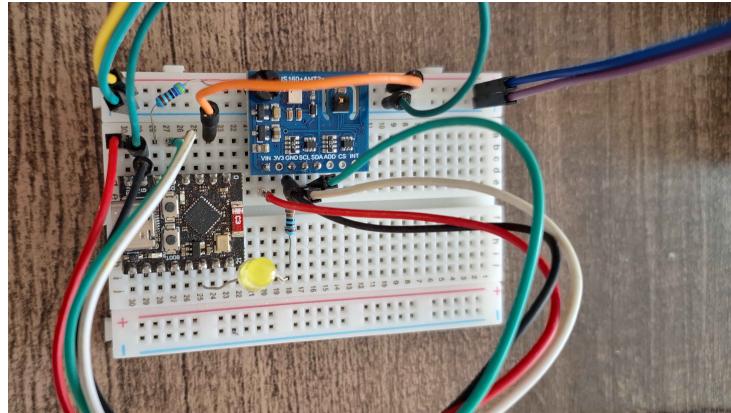
O Apêndice D apresenta um caso de teste de validação, no qual o dispositivo não atinge a conformidade com a especificação, ilustrando as funcionalidades do *frontend*. O apêndice detalha a interface do formulário de validação, exibindo os parâmetros esperados e os valores medidos durante o teste, assim como os resultados obtidos. Em testes onde os parâmetros do DUT estão fora da margem de erro da especificação, é indicado um resultado de falha acompanhado de uma referência ao índice correspondente em uma lista detalhada que explica a causa da não conformidade.

4.3 Estudo de Caso

Como apresentado na seção de metodologia, os sistemas foram implementados por: ESP32-C3, STM32F411, e RP2350, e, como a comunicação ocorre entre os microcontroladores, seus respectivos sistemas serão referidos pelo microcontrolador que está em sua implementação.

Para coleta de dados, a ESP32-C3 se comunica com o conjunto de sensores por meio de um barramento I²C, utilizando bibliotecas disponibilizadas pela *Adafruit Industries* (Industries, 2025) e Sciosense (Sciosense, 2025). O controlador requisita a medição das variáveis ambientais; enquanto os sensores realizam as medições, o controlador realiza a checagem se alguma mensagem foi recebida pelo barramento; e, após 1 segundo, tempo necessário para medição de compostos voláteis, os dados dos sensores são enviados pelo barramento. A implementação física do sistema está exemplificada na figura 12, que contém o ESP32-C3, um módulo combinado dos sensores conectados por I²C, resistor *pull-up* do barramento, e um LED amarelo.

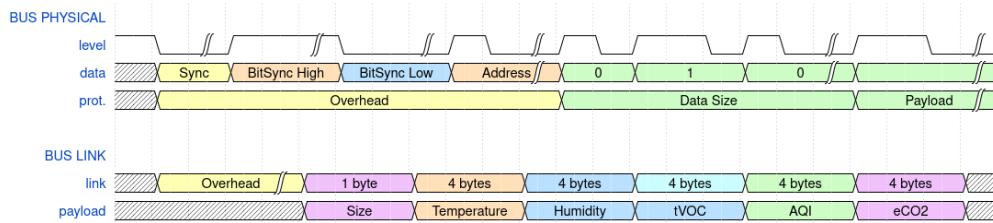
Figura 12: Implementação do sistema de sensoriamento.



Fonte: Autoria própria.

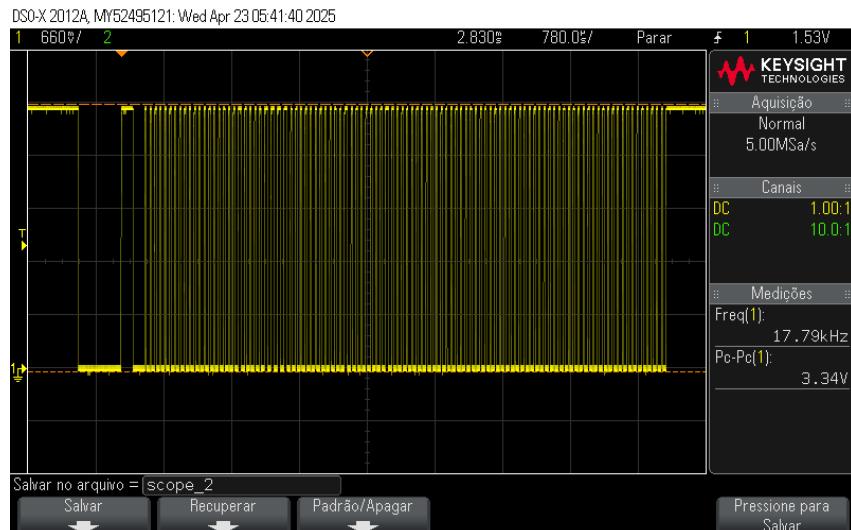
Os dados dos sensores são codificados como um payload genérico de 20 bytes, composto por 5 valores: 2 em formato de *float* de 32 bits, e 3 em formato *unsigned int*, também de 32 bits. Estes são: a temperatura, a umidade relativa, tVOC, índice AQI, e eCO₂, respectivamente, ilustrado na figura 13. A figura 14 demonstra uma medição por osciloscópio do envio da mensagem pelo sistema físico utilizando um Δ de 20 μs .

Figura 13: Diagrama do pacote genérico com os dados dos sensores.



Fonte: Autoria própria.

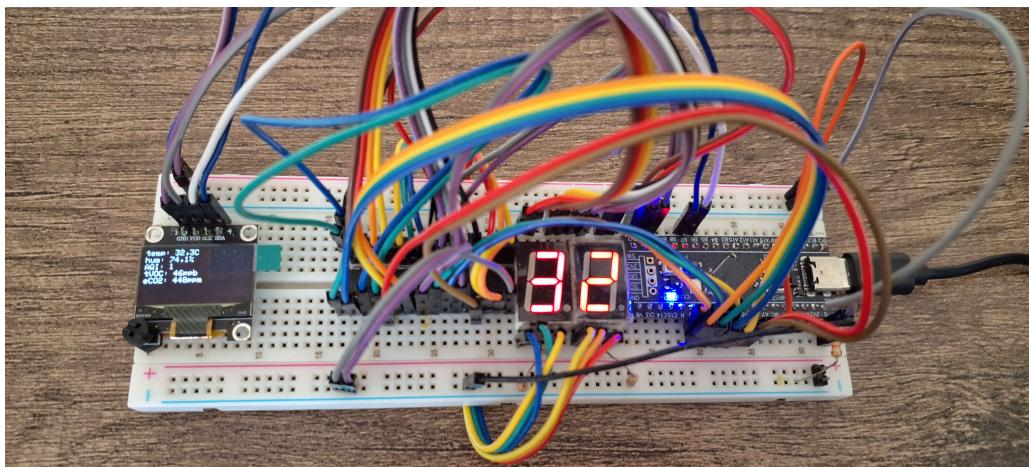
Figura 14: Transmissão de um pacote genérico com os dados dos sensores.



Fonte: Autoria própria.

Depois de publicados no barramento, os dados são recebidos pelos STM32F411 e RP2350. O STM32F411 realiza a leitura dos dados e o devido processamento para exibi-los no display OLED, com legenda e unidades de grandeza. Assim como também envia os dígitos inteiros da temperatura em *BCD* para os chips dos displays de 7 segmentos, como pode ser visto na imagem 15.

Figura 15: Implementação do sistema de interface.



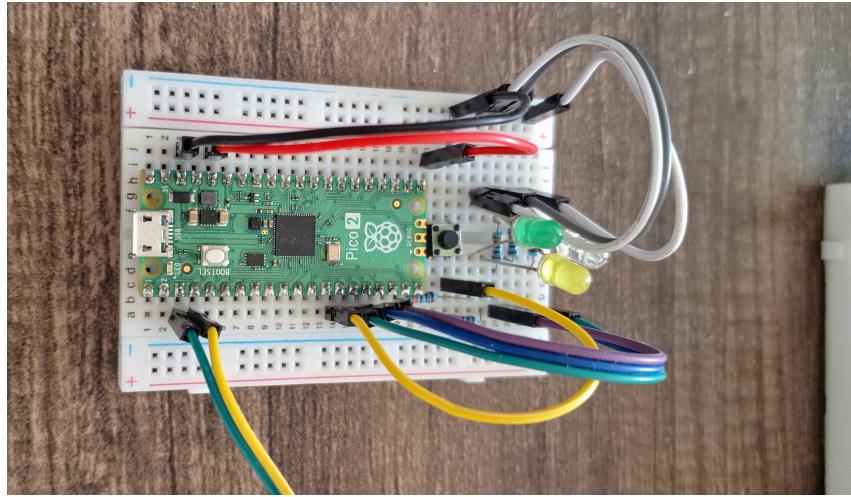
Fonte: Autoria própria.

Já o RP2350 recebe os dados dos sensores e checa se a temperatura está acima de 35°C , acionando um dos LEDs caso positivo. Se o nível de eCO₂ for maior que 600 ppm, outro LED também é acionado; e o nível de qualidade do ar é sempre indicado em um LED RGB, que muda sua tonalidade para tons avermelhados com o aumento do AQI, que indica uma piora na qualidade do ar. O sistema com o RP2350, LEDs e botão pode ser visto na imagem 16.

Além disso, caso haja alguma interação com o botão, o RP2350 envia um pacote L3 endereçado ao ESP32-C3, que possui endereço 0xF0 no barramento, com um comando de mudança do estado do LED, e para qual estado deve ir, representado no campo de dados. Este evento é acionado por meio de uma interrupção no *GPIO* atrelado ao botão, que ao detectar uma borda de descida, após *debounce*, envia uma mensagem com o comando para ligar o LED, e, reciprocamente, envia um comando para desligar o LED quando detecta uma borda de subida.

Os dispositivos do estudo de caso foram configurados para funcionar na classe de velocidade *SLOW*, o que consiste em $\Delta = 20 \mu\text{s}$. Esta classe de velocidade foi escolhida por motivos de estabilidade. Transmissões na classe *FAST II* ($\Delta = 2,5 \mu\text{s}$) chegaram a ocorrer, porém com o aumento da velocidade, as margens de erro se tornam menores, o que causa um aumento nos erros de transmissão, e requer ajustes mais finos nas rotinas de temporização de cada sistema. O sistema foi concebido pensando em atingir uma classe de velocidade do nível *FAST* ($\{\Delta \mid \Delta \in (4, 2.5)\mu\text{s}\}$), porém, não foi possível realizar os

Figura 16: Implementação do sistema agregador.



Fonte: Autoria própria.

ajustes necessários para manter transmissões na classe *FAST* totalmente livres de erros a tempo do término deste trabalho.

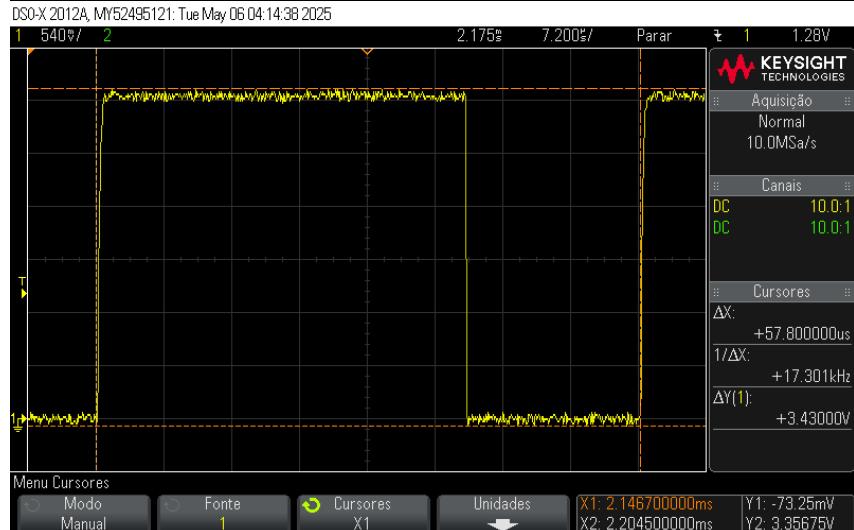
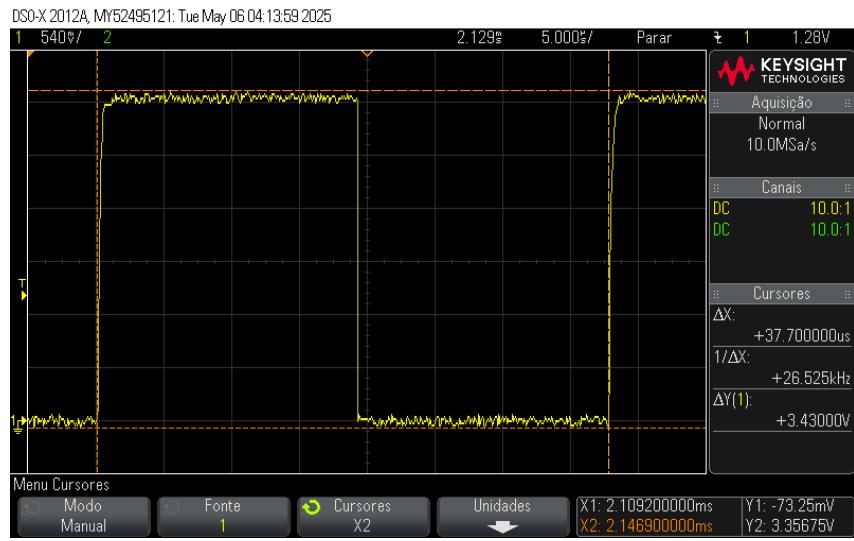
Ademais, dadas as especificações dos sensores utilizados, transmissões de dados na classe *SLOW* não interferem no funcionamento do dispositivo. Bits codificados na classe de velocidade *SLOW* são demonstrados na figura 17, junto des medidas de tempo.

Em condições normais, uma condição de colisão é muito improvável de acontecer, visto que foram escolhidos endereços bem espaçados, o que acarreta em diferentes tempos de *CSMA*, e, caso uma transmissão se inicie no mesmo tempo, o *backoff* acontece de maneira transparente, com o dispositivo que enviou um nível não-dominante (bit 1) no barramento detectando a colisão ao ler um nível dominante (bit 0 ou *SYNC*) que foi enviado por outro dispositivo.

Na figura 18 temos um exemplo de uma colisão causada de maneira artificial, forçando um *backoff* decorrente de um dispositivo que não respeitou a regra do *CSMA* de checar se há transmissões no barramento. Em primeiro momento há uma transmissão no barramento, e em seguida, essa transmissão é interrompida por outra transmissão, que ao enviar o sinal *SYNC* alerta o primeiro dispositivo de colisão no barramento, que por sua vez, realiza o *backoff*.

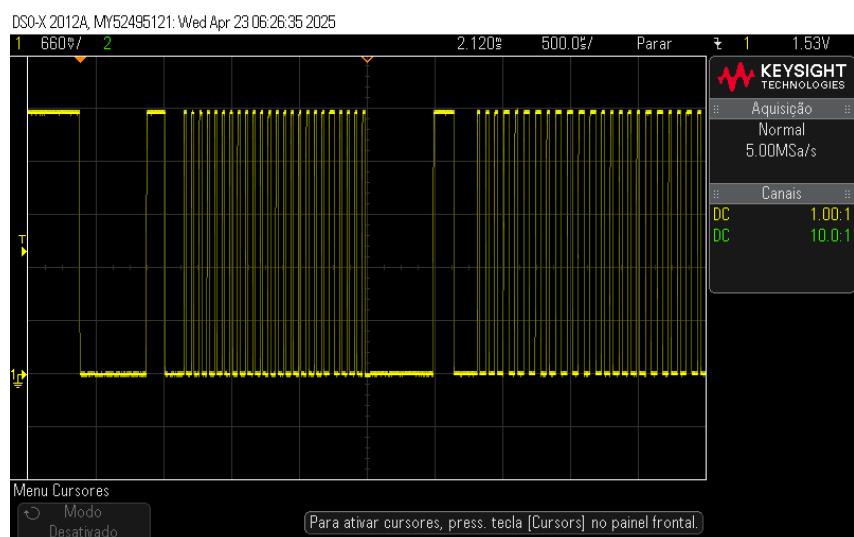
Demonstrando, pelo fato de que a mensagem que não respeitou o *CSMA* não foi corrompida, é possível concluir que caso ocorresse uma colisão própria, na qual os dispositivos começassem a transmissão no exato mesmo instante, quando um dispositivo detectasse o nível dominante no barramento quando ele mandou um nível não-dominante, seria realizado o *backoff* e a mensagem com prioridade não seria corrompida.

Figura 17: Codificação de um bit 0 e de um bit 1.



Fonte: Autoria própria.

Figura 18: Exemplo de *backoff* de mensagem.



Fonte: Autoria própria.

4.4 Biblioteca de protocolo

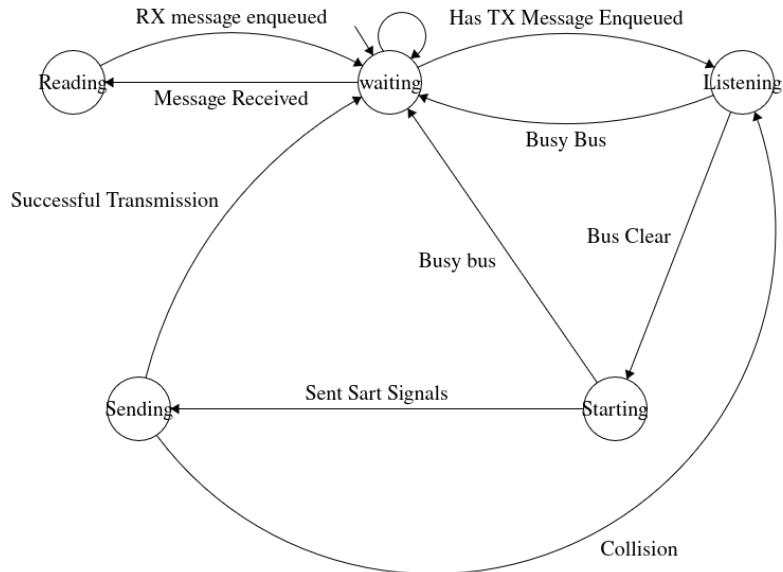
Dado o fato de que o estudo de caso utiliza 3 chips diferentes, visando evitar reimplementar a mesma funcionalidade três vezes, e, buscando melhorar a qualidade deste trabalho, também foi desenvolvida uma biblioteca em C, que implementa boa parte da funcionalidade do protocolo de maneira genérica. E deixa necessário realizar apenas a implementação de algumas seções específicas que geralmente utilizam bibliotecas de baixo nível para acessar o hardware do chip. Estas seções são denominadas *port* e são implementadas para cada chip alvo em um arquivo *port.c*.

Visando facilitar a integração com várias plataformas e mantendo em mente a disponibilidade crescente de chips com *Simultaneous Multithreading (SMT)*, foram utilizadas algumas abstrações provenientes do sistema operacional FreeRTOS (Amazon, 2025), que oferece suporte para diversas arquiteturas. Entre essas abstrações estão: Tasks, que são rotinas que executam concorrentemente realizando *time share* pelo escalonador do sistema operacional, ou executam paralelamente quando o sistema possui mais de um núcleo; Filas, para o envio de dados entre diferentes *tasks* e rotinas de interrupção; e rotinas de sincronização para zonas críticas de código.

A biblioteca é composta, principalmente, por um par de arquivos: *DCP.h*, que provê o cabeçalho das funções implementadas, e o *generic/DCP.c*, que possui as implementações das funções. O arquivo fonte implementa funções de interface: *SendMessage* e *ReadMessage*, que interagem com as filas de entrada e saída de dados da *task* do protocolo; a *task* do protocolo, *BusHandler*, que executa indefinidamente implementando a máquina de estados finita apresentada na figura 19 e realiza a entrada e saída de dados através das filas *RX* e *TX*; a rotina de interrupção para a leitura de dados recebidos no barramento, que é executada em um evento de interrupção por borda de descida no pino do barramento e que envia dados para a *task* do protocolo por meio da fila *ISR_queue*; e, por fim, rotinas de envio de dados binários no barramento, que convertem dados binários na codificação de dados do protocolo e rotina de *delay* com precisão na ordem de nanosegundos. O arquivo de cabeçalho e o arquivo de código-fonte podem ser encontrados no Apêndice E.

À cargo dos arquivos *port*, ficam a implementação de uma rotina de inicialização, que inicializa os periféricos necessários, realiza a criação das filas do *FreeRTOS*, registra a rotina de interrupção, e configura os pinos *GPIO*; rotinas de mudança e leitura de estado de *GPIO*; rotina de inicialização de algum periférico medidor de tempo para temporização dos sinais, que pode ser um *timer* veloz o suficiente, ou contador autoincrementável; e, por fim, opcionalmente uma maneira de realizar *log* dos dados, que pode ser útil para *debug*. Um exemplo de arquivo de port implementado para o *RP2350* pode ser encontrado no Apêndice F.

Figura 19: Máquina de estados finitos do protocolo DCP.



Fonte: Autoria própria.

Para realizar a compilação dos arquivos selecionando o *port* correto para o chip alvo, é utilizada a ferramenta *CMake*, que já é comumente utilizada em projetos que utilizam as linguagens C, C++ e ASM. Utilizando um arquivo *CMakeLists.txt* na pasta da biblioteca é possível realizar a definição e exportação dos arquivos fonte e de cabeçalho como uma biblioteca *CMake*, tornando possível que a biblioteca seja adicionada em outros projetos com trabalho mínimo e abrindo a possibilidade de delegar a seleção do arquivo de *port* adequado para o projeto que utiliza a biblioteca. Portanto, para um projeto que vai ser compilado para o chip STM32F411, é possível adicionar o arquivo *STM32F4/port.c* como uma dependência da biblioteca.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como objetivo criar um documento de um manual técnico para o protocolo DCP, desenvolver um dispositivo validador capaz de testar a conformidade de dispositivos externos com as configurações estabelecidas - além de identificar erros e sugerir possíveis correções - e demonstrar o funcionamento do protocolo por meio de um estudo de caso.

Para abordar a problemática, foi elaborado um manual de referência técnica que fornece informações básicas para projetistas iniciantes no protocolo DCP. Embora seja possível implementar dispositivos com base apenas neste documento, não foi possível completar uma especificação abrangente em tempo hábil para a apresentação deste trabalho. Portanto, projetistas que utilizarem este documento podem encontrar problemas específicos não abordados no manual.

O estudo de caso demonstrou o funcionamento do validador, confirmando a operação correta do barramento e do *DUT*. Também foi apresentado um exemplo de dispositivo que não atendia aos parâmetros especificados para demonstração do validador, demonstrando a capacidade de identificar potenciais problemas de comunicação em fase de implementação, reduzindo riscos de retrabalho e defeitos na produção.

Adicionalmente, o estudo de caso validou a funcionalidade do protocolo através da implementação bem-sucedida de três sistemas que utilizam o barramento como único meio de comunicação. Esses sistemas demonstraram comunicação bidirecional multimestre, além da implementação dos mecanismos *Hot-Join* e prevenção de colisão. O resultado culminou na criação de uma biblioteca que pode ser reutilizada em outros projetos, facilitando a adaptação do protocolo a sistemas sem suporte nativo.

Trabalhos futuros podem utilizar o protocolo e o validador para criar sistemas completos com aplicações práticas, tendo em vista as capacidades do protocolo de suportar acesso multicontrolador e do mecanismo de *Hot-Join*. Além de adicionar ao documento de especificação informações sobre problemas encontrados, entrar em maiores detalhes sobre cada característica do protocolo e adicionar mais diagramas mostrando formato e limites dos sinais fundamentais. Existe também, a possibilidade de trabalhos em aumentar o rol de chips suportados na biblioteca do protocolo.

REFERÊNCIAS

AMAZON. *Real-time operating system for microcontrollers and small microprocessors.* FreeRTOS. 2025. Disponível em: <https://www.freertos.org/>. Acesso em: 22 abr. 2025.

AMORIM, Mardson Freitas de. L3: Protocolo de Comunicação Para Telemetria. João Pessoa, 2017.

APT, Krzysztof R.; OLDEROG, Ernst-Rüdiger. L3: Protocolo de Comunicação Para Telemetria. Varsóvia, 2019. DOI: arxiv.org/abs/1904.03917.

ARAÚJO, Thiago Alves de. *Desenvolvimento de um protocolo de comunicação serial com um único fio.* Trabalho de Conclusão de Curso – Universidade Federal da Paraíba, João Pessoa, 2022.

ATMEL. *AVR318: Dallas 1-Wire master.* [S. l.], 2004.

BUCHANAN, William. *Computer busses design and application.* Boca Raton: CRC, 2000. DOI: [10.1201/9781420041682](https://doi.org/10.1201/9781420041682).

CATSOULIS, John. *Designing embedded hardware.* 2. ed. Sebastopol, CA: O'Reilly Media, Inc, 2009.

DALLAS. *Quick Guide to 1-Wire net. Using PCs and Microcontrollers.* [S. l.], S.I.

ESPRESSIF. *HTTP Server.* Espressif. 2016. Disponível em: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/api-reference/protocols/esp_http_server.html. Acesso em: 20 abr. 2025.

INDUSTRIES, Adafruit. *Adafruit AHTX0: Arduino library for AHT10 and AHT20 sensors!* [S. l.: s. n.], 2025. Disponível em: https://github.com/adafruit/Adafruit_AHTX0.git.

KANER, Cem; FALK, Jack L.; NGUYEN, Hung Quoc. *Testing Computer Software, Second Edition.* 2nd. USA: John Wiley & Sons, Inc., 1999. ISBN 0471358460.

MICROCHIP. *UNI/O Bus Specification.* [S. l.], 2009.

MIPI. *MIPI Alliance Overview: About Us.* MIPI Alliance. 2024. Disponível em: <https://www.mipi.org/about-us>. Acesso em: 9 nov. 2024.

MIPI. *MIPI I3C Basic Specification.* 1.1.1. ed. [S. l.], mar. 2022.

NXP. *I²C-bus specification and user manual.* 7.0. ed. [S. l.], 2021.

PALMER, J. W.; SABNANI, Krishan. A Survey of Protocol Verification Techniques. In: MILCOM 1986 - IEEE Military Communications Conference: Communications-Computers: Teamed for the 90's. [S. l.: s. n.], 1986. v. 1, p. 1.5.1–1.5.5. DOI: 10.1109/MILCOM.1986.4805652.

PHILIPS. *BCD to 7-segment latch/decoder/driver*. [S. l.], 1995.

RICHARDSON, Leonard; AMUNDSEN, Mike; RUBY, Sam. *RESTful Web APIs*. [S. l.]: O'Reilly Media, Inc., 2013. ISBN 1449358063.

ROYCHOUDHURY, Abhik. *Embedded Systems and Software Verification*. 1. ed. Amsterdã: Elsevier Science, 2009.

SBS. *System Management Bus Specification*. SBS Implementers Forum. 1998. Disponível em: <https://smbus.org/specs/smbus110.pdf>. Acesso em: 9 nov. 2024.

SCIOSENSE. *ScioSense ENS160*: Driver for the ScioSense ENS160 digital gas sensor. [S. l.: s. n.], 2025. Disponível em: https://github.com/sciosense/ENS160_driver.git.

SYSTECH, Solomon. *SSD1306 Advance Information*: 128 x 64 Dot Matrix OLED/PLED Segment/Common Driver with Controller. [S. l.], 2008.

TANENBAUM, Andrew S.; STEEN, Maarten van. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

TANENBAUM, Andrew S.; WETHERALL, David J. *Computer Networks*. 5th. USA: Prentice Hall Press, 2010. ISBN 0132126958.

WHITE, Elecia. *Making embedded systems*. 1. ed. Sebastopol, CA: O'Reilly Media, Inc, 2011.

WOLF, Marilyn. *Computers as components: principles of embedded computing system design*. 3. ed. Amsterdã: Elsevier Science, 2017.

APÊNDICE A - MANUAL TÉCNICO DO PROTOCOLO DCP

Introduction

The Devices Communication Protocol (DCP) is a bidirectional, single-wire communication protocol developed at LMI-UFPB. It's designed to support a multi-master architecture for device communication. The primary goal of DCP is to enable reliable data exchange between devices without requiring complex arbitration mechanisms typically found in other protocols. It is engineered to reduce wiring complexity while ensuring robust, low-power, and scalable communication. DCP is particularly suited for applications such as automotive telematics, industrial sensor networks, and modular control systems, where both reliability and minimal physical wiring are crucial.

DCP Purpose

DCP provides an efficient and cost-effective solution for inter-device communication by combining clock and data in a single time-modulated signal. It incorporates robust collision detection and avoidance methods based on the CSMA/CD paradigm with binary countdown arbitration, which effectively manages simultaneous transmissions. Additionally, DCP enables hot-join capabilities so that devices can dynamically enter or exit the bus without interrupting ongoing communications. The protocol also supports multiple speed modes (Slow, Fast I, Fast II, and High Speed) to accommodate diverse system requirements and electrical conditions.

Protocol Overview and the Role of Δ

At the heart of DCP is the concept of a base time unit, Δ , which serves as the fundamental timing reference for defining the duration of all pulses and intervals on the bus. The value of Δ is determined by system constraints—such as bus capacitance, pull-up resistor values, and the desired bit rate—and is used to scale the pulse durations for both data and control signals. By adjusting Δ , the protocol can be tuned for either slower transmissions (using a larger Δ) to tolerate noise and capacitance or for high-speed communications (using a smaller Δ) that require tighter timing control.

Every time-related parameter in DCP—including bit pulse lengths, synchronization intervals, rise and fall times, and overall cycle duration—is expressed as a multiple of Δ , providing a scalable and adaptable communication framework.

Protocol Features

Bus Speeds

DCP defines four operational classes based on Δ , each suited for different application scenarios. The table below summarizes each mode, its recommended Δ time unit, and typical device clock frequencies:

Mode	Δ (Time Unit)	Description
Slow	$\Delta = 20 \mu s$	Devices operating at 4 MHz
Fast I	$\Delta = 4 \mu s$	Devices operating at 20 MHz
Fast II	$\Delta = 2.5 \mu s$	Devices operating at 32 MHz
High Speed	$\Delta = 1.25 \mu s$	Devices operating at 64 MHz

Table 1: DCP Speed Classes and Corresponding Δ Values

Bus Communication

DCP transmits data using single-wire communication that embeds clock and data signals via time modulation. At the physical layer, the communication occurs on a single conductor that is pulled high by a resistor connected to the supply voltage (V_{DD}). Devices actively drive the bus low to assert a logical "0," while a passive pull-up network maintains a logical "1." The voltage levels are defined such that the high-level voltage, V_h , is at least 70% of V_{DD} , and the low-level voltage, V_l , remains below 30% of V_{DD} .

Signal encoding in DCP relies entirely on the timing reference Δ . A logical "0" is transmitted as a high-level pulse lasting Δ followed by a low-level pulse also lasting Δ , whereas a logical "1" uses a high-level pulse of duration 2Δ followed by a low-level pulse of duration Δ .

In addition, synchronization signals are defined to mark the start of a transmission: the Start Sync signal, which lasts 25Δ for transmitting controllers and 50Δ for target devices, initiates the transmission and synchronizes the receiver's clock; and the Start Bit, composed of a high-level pulse of 7.5Δ immediately followed by a low-level pulse of 7.5Δ , signals the transition to the data payload.

- **Start Sync:** Initiates a transmission and synchronizes the receiver's clock. For the transmitting controller, it is a low-level pulse lasting 25Δ , while target devices use 50Δ to ensure proper clock recovery.
- **Start Bit:** Marks the end of synchronization and the beginning of data payload transmission. It consists of a high-level pulse of 7.5Δ immediately followed by a low-level pulse of 7.5Δ .

The bit stream is framed within a clearly defined cycle time that includes a rise time, a brief settling period, and a fall time, thereby eliminating the need for a separate clock line.

Hot-Join Mechanism

The hot-join mechanism in DCP allows devices to connect to an operational bus without disrupting active communications. When a new device is powered up, it initially enters a passive listening state, monitoring the bus for the Start Sync signal to acquire the timing information it needs. Upon detecting the Start Sync, the device synchronizes its internal clock and configures its transmission parameters according to the appropriate Δ value. Only after confirming an idle state on the bus does the device transition to active communication mode. This careful integration ensures that hot-joining does not disturb ongoing data exchanges.

Multi-Controller Setup

DCP supports multiple controllers on the same bus without causing interference. Each controller first performs carrier sensing by listening to the bus, which is maintained high by the pull-up resistor when idle. When multiple controllers attempt to transmit simultaneously, DCP employs a binary countdown arbitration process. During this process, each controller sends its unique binary address along with the data. While transmitting, a controller monitors the bus; if it sends a logical "1" but detects a logical "0," it infers that a higher-priority controller is active and withdraws from the arbitration process. Controllers that lose arbitration then wait for a random backoff period before retrying. This method ensures orderly access and minimizes the probability of repeated collisions.

Collision Detection and Avoidance Strategies (CSMA/CD)

The collision detection and avoidance strategy in DCP is a multi-layered approach based on CSMA/CD and a priority-based binary backoff mechanism.

Before transmitting any data, each device must monitor the bus to determine if it's currently occupied. If the bus is busy (logical Low), the device must defer transmission and wait for the bus to become idle. When a device detects an idle bus, it doesn't immediately transmit. Instead, it waits for a specific period before attempting transmission. This waiting time depends on the device's address:

- Generic Packets: Devices intending to send a generic packet wait $\frac{(d+6)\Delta}{4}$ seconds.
- L3 Packets (Addresses 250-255): Devices sending L3 packets wait $\frac{(MasterId-249)\Delta}{4}$ seconds, where 'MasterId' is the device's address. This shorter waiting time gives priority to devices in this address range.

After the deferral period, the device begins transmitting its message bit-by-bit. Crucially, during transmission, each device continuously monitors the bus. It compares the level it is writing to the bus with the actual level present on the bus.

If a device detects a discrepancy between the level it's sending and the level on the bus, it indicates a collision has occurred. Specifically, if a device writes a '0' (Low) to the bus but reads a '1' (High), it means another device is simultaneously transmitting a '1'. This signals a collision. The device emitting the '0' does not detect the collision because the dominant low level overwrites the high level. The device detecting the discrepancy immediately stops transmission and enters an error state, signaling loss of arbitration.

After detecting a collision, the colliding devices enter a binary backoff procedure. This involves waiting for a random period determined by their address before attempting to retransmit. The lower the address, the higher the priority in regaining access to the bus. This mechanism helps resolve contention and prevents continuous collisions.

Devices must be able to detect communication failures during any phase of transmission – Start Sync, Start Bit, or Payload. Signals outside a defined tolerance margin (2% of error) are considered invalid, and the communication is terminated to prevent corrupted messages from being received.

Electrical specifications and timing for I/O stages and bus lines

Electrical Specification

Table 2: Electrical characteristics of the DATA line

Symbol	Parameter	Conditions	Min	Max
V_{IL}	LOW-level input voltage		$V_{GND} V$	$V_{GND} + 0.3 * V_{DD} V$
V_{IH}	HIGH-level input voltage		$0.7 * V_{DD} V$	-
t_{fall}	Output fall time	V_{IHmin} to V_{ILmax}	-	400ns ⁽¹⁾
t_{rise}	Output rise time	V_{ILmax} to V_{IHmin}	-	400ns ⁽¹⁾
R_p	Pullup Resistor	-	$\frac{V_{DD} - V_{OL}}{I_{OL}} \Omega$	$\frac{4\mu}{0.8473 \cdot C_{bus}} \Omega$ ⁽²⁾

⁽¹⁾ Calculated by 2% (moe) of $20\mu s$, which is Δ_{SLOW} .

⁽²⁾ 4μ is (moe) Δ_{SLOW} , which is our max rise time, for other speeds the function is given as $\frac{(moe_{speed})}{0.8473 \cdot C_{bus}} \Omega$.

Timing Specification

Table 3: Characteristics of the DATA line for different speed configurations

Symbol	SLOW	FAST I	FAST II	ULTRA
t_Δ	$20\mu s$	$4\mu s$	$2.5\mu s$	$1.25\mu s$
t_0	$20\mu s$	$4\mu s$	$2.5\mu s$	$1.25\mu s$
t_1	$40\mu s$	$8\mu s$	$5\mu s$	$2.5\mu s$
$tMinbit0$	$32\mu s$	$6.4\mu s$	$4\mu s$	$2\mu s$
$tMaxbit0$	$48\mu s$	$9.6\mu s$	$6\mu s$	$3\mu s$
$Tr_{MAX}^{(1)}$	49.952 kbps	249.952 kbps	399.952 kbps	799.952 kbps
$Tr_{MIN}^{(2)}$	24.976 kbps	124.976 kbps	199.976 kbps	399.976 kbps

⁽¹⁾ Max theoretical throughput for a 1 second transmission, consisting only of bit 0. ⁽²⁾ Min theoretical throughput for a 1 second transmission, consisting only of bit 1.

Timing Diagram

The timing diagram depicted in Figure 1 illustrates the fundamental signal structure of a typical DCP transmission sequence. The communication begins with the *Start Sync* pulse, a prolonged low-level signal whose duration is determined by the transmitter's role on the bus. This signal allows all listening devices to synchronize their internal clocks and determine the class of the transmitter—whether it is a controller or a peripheral node. Once synchronization is achieved, a distinct high-low sequence of equal duration follows, known as the *Start Bit*. This bit serves as a delimiter, clearly indicating the conclusion of the synchronization phase and the imminent start of data transmission. Subsequent to this preamble, the payload is transmitted as a time-modulated sequence of logical "0"s and "1"s, each defined by specific high and low pulse durations that are multiples of the base unit Δ .

An oscilloscope capture is shown in Figure 2 to illustrate real conditions. This image provides a physical representation of the DCP waveform during transmission, confirming the accuracy of the protocol's time-domain characteristics. Both the synchronization and data segments adhere to the specified

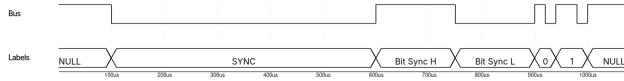


Figure 1: DCP protocol fundamental signals diagram

pulse widths, demonstrating stable transitions and clear signal definition. These characteristics are essential for ensuring reliable communication, particularly in systems with variable electrical loading or bus capacitance.

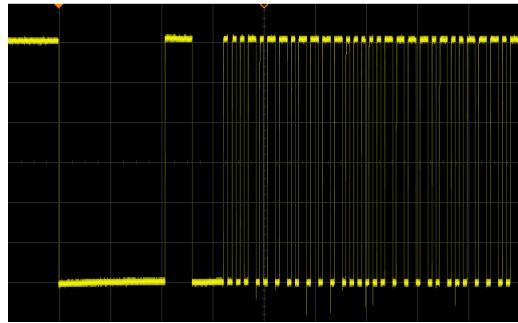


Figure 2: Oscilloscope capture of a DCP transmission, showing Start Sync, Start Bit, and payload.

Wiring Example

The wiring configuration for a typical DCP setup is shown in Figure 3. In this example, three devices are interconnected via a shared single-wire bus. A pull-up resistor is placed between the signal line and the supply voltage VDD for each device. These resistors ensure that the bus remains at a high logic level accounting for the extra capacitance added on each node.

This configuration allows any device to monitor the line state and participate in communication when appropriate. The use of multiple pull-up resistors in parallel does not disrupt the protocol as long as the total effective resistance complies with the electrical constraints defined earlier. It is important to size the pull-up resistors with the expected bus capacitance in mind, maintaining rise times within the allowed specifications for the selected speed class. This simple yet robust topology supports the protocol's hot-join and multi-master features by facilitating seamless electrical integration and reliable signal propagation.

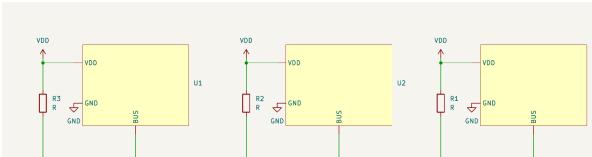


Figure 3: Example wiring configuration with three DCP-compatible devices sharing a single communication bus.

APÊNDICE B - CÓDIGO-FONTE DO *FRONTEND* DO DISPOSITIVO DE VALIDAÇÃO

Código HTML do frontend

```
1      <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>DCP Validation Report</title>
7     <style>
8         body {
9             font-family: Arial, sans-serif;
10            line-height: 1.6;
11            color: #333;
12            max-width: 800px;
13            margin: 0 auto;
14            padding: 20px;
15        }
16
17        .header {
18            text-align: center;
19            margin-bottom: 20px;
20        }
21
22        .header h1 {
23            margin-bottom: 5px;
24        }
25
26        .header p {
27            color: #666;
28            font-size: 14px;
29            margin: 5px 0;
30        }
31
32        .device-info {
33            margin-bottom: 20px;
34        }
35
36        .device-info div {
37            margin-bottom: 10px;
38        }
39
40        .device-info label {
41            display: inline-block;
42            width: 150px;
43            font-weight: bold;
44        }
45
46        .device-info input, .device-info select {
47            width: 250px;
48            padding: 5px;
49            border: 1px solid #ddd;
50            border-radius: 4px;
51        }
52
```

```

53     .required::after {
54         content: "*";
55         color: red;
56         margin-left: 5px;
57     }
58
59     table {
60         width: 100%;
61         border-collapse: collapse;
62         margin-bottom: 20px;
63     }
64
65     table, th, td {
66         border: 1px solid #ddd;
67     }
68
69     th, td {
70         padding: 10px;
71         text-align: left;
72     }
73
74     th {
75         background-color: #f2f2f2;
76     }
77
78     .failure-details {
79         margin-bottom: 20px;
80         display: none;
81     }
82
83     .failure-details h2 {
84         font-size: 18px;
85         color: #721c24;
86     }
87
88     .failure-details ol {
89         margin-left: 20px;
90     }
91
92     .button-container {
93         display: flex;
94         justify-content: center;
95         gap: 15px;
96         margin: 30px 0;
97     }
98
99     .button {
100         padding: 10px 20px;
101         background-color: #007bff;
102         color: white;
103         border: none;
104         border-radius: 4px;
105         cursor: pointer;
106         font-size: 16px;
107     }
108
109     .button:hover {
110         background-color: #0056b3;
111     }

```

```

112     .pass {
113         color: green;
114     }
115
116     .fail {
117         color: red;
118         font-weight: bold;
119     }
120 }
121
122 #loading-indicator {
123     display: none;
124     text-align: center;
125     margin: 20px 0;
126 }
127
128 .spinner {
129     border: 4px solid rgba(0, 0, 0, 0.1);
130     border-radius: 50%;
131     border-top: 4px solid #007bff;
132     width: 20px;
133     height: 20px;
134     animation: spin 1s linear infinite;
135     display: inline-block;
136     margin-right: 10px;
137     vertical-align: middle;
138 }
139
140 @keyframes spin {
141     0% { transform: rotate(0deg); }
142     100% { transform: rotate(360deg); }
143 }
144
145 .notification {
146     display: none;
147     padding: 10px;
148     margin: 10px 0;
149     border-radius: 4px;
150     text-align: center;
151 }
152
153 .notification.success {
154     background-color: #d4edda;
155     color: #155724;
156     border: 1px solid #c3e6cb;
157 }
158
159 .notification.error {
160     background-color: #f8d7da;
161     color: #721c24;
162     border: 1px solid #f5c6cb;
163 }
164
165 .note {
166     font-size: 12px;
167     color: #666;
168     margin-top: 5px;
169 }
170

```

```

171     .side-by-side-container {
172         display: flex;
173         justify-content: space-between;
174         margin-bottom: 20px;
175         width: 100%;
176         clear: both;
177     }
178
179     .table-container {
180         width: 48%;
181     }
182
183     /* Hide buttons when printing */
184     @media print {
185         .button-container, #loading-indicator, .notification, .note {
186             display: none !important;
187         }
188         body {
189             padding: 0;
190         }
191     }
192 </style>
193 </head>
194 <body>
195     <div class="header">
196         <h1>DCP Validation Report</h1>
197         <p id="date-of-emission">DCP-IF Date of Emission: February 5, 2025</p>
198         <p>Validation Version: 0.1</p>
199     </div>
200
201     <div id="notification" class="notification"></div>
202
203     <div class="device-info">
204         <div>
205             <label for="device-name">Name:</label>
206             <input type="text" id="device-name" placeholder="Enter device name">
207         </div>
208         <div>
209             <label for="device-version">Device Version:</label>
210             <input type="text" id="device-version" placeholder="Enter version">
211         </div>
212         <div>
213             <label for="device-type" class="required">Device Type:</label>
214             <select id="device-type">
215                 <option value="Seleccione">Seleccione</option>
216                 <option value="controlador">controlador</option>
217                 <option value="alvo">alvo</option>
218             </select>
219         </div>
220         <div>
221             <label for="device-speed" class="required">Device Speed:</label>
222             <select id="device-speed">
223                 <option value="Seleccione">Seleccione</option>
224                 <option value="4 MHz">4 MHz</option>
225                 <option value="20 MHz">20 MHz</option>
226                 <option value="32 MHz">32 MHz</option>
227                 <option value="64 MHz">64 MHz</option>
228             </select>
229         </div>

```

```

230      <p class="note">* Required fields for validation</p>
231  </div>
232
233  <div id="loading-indicator">
234      <div class="spinner"></div> Loading data ...
235  </div>
236
237  <h2>Specification Conformity</h2>
238  <table id="spec-conformity-table">
239      <thead>
240          <tr>
241              <th>Parameter</th>
242              <th>Expected</th>
243              <th>Got</th>
244              <th>Result</th>
245          </tr>
246      </thead>
247      <tbody>
248          </tbody>
249  </table>
250
251  <div class="side-by-side-container">
252      <div class="table-container">
253          <h2>Transmission Information</h2>
254          <table id="transmission-info-table">
255              <thead>
256                  <tr>
257                      <th>Parameter</th>
258                      <th>Result</th>
259                  </tr>
260              </thead>
261              <tbody>
262                  </tbody>
263          </table>
264      </div>
265
266      <div class="table-container">
267          <h2>Electrical Information</h2>
268          <table id="electrical-info-table">
269              <thead>
270                  <tr>
271                      <th>Parameter</th>
272                      <th>Value</th>
273                  </tr>
274              </thead>
275              <tbody>
276                  </tbody>
277          </table>
278      </div>
279  </div>
280  <div class="failure-details">
281      <h2>Failure Details</h2>
282      <ol id="failure-list">
283          </ol>
284  </div>
285
286  <div class="button-container">
287      <button class="button" id="fetch-data-button">Fetch Data</button>
288      <button class="button" id="print-button">Download as PDF</button>

```

```
289     </div>
290
291     <script>
292         // Set current date
293         document.addEventListener('DOMContentLoaded', function() {
294             const today = new Date();
295             const dateStr = today.toLocaleDateString('en-US', {
296                 year: 'numeric',
297                 month: 'long',
298                 day: 'numeric'
299             });
300             document.getElementById('date-of-emission').textContent = 'DCP-IF Date of
Emission: ${dateStr}';
301         });
302     </script>
303     <script type="module" src="../../src/result.js"></script>
304     <script type="module" src="../../src/buttons.js"></script>
305 </body>
306 </html>
```

Código JavaScript do frontend

Script dos botões da tela

```
1 import {fetchReportData, populateDefaultValues} from "../src/result.js";
2
3 // Event listener for download PDF button
4 document.getElementById('print-button').addEventListener('click', function() {
5     window.print();
6 });
7
8 // Event listener for fetch data button
9 document.getElementById('fetch-data-button').addEventListener('click', function() {
10    fetchReportData();
11 });
12
13 document.getElementById('device-type').addEventListener('input', function() {
14
15    const deviceType = document.getElementById('device-type').value;
16    const deviceSpeed = document.getElementById('device-speed').value;
17
18    if (deviceType !== "Selecione" && deviceSpeed !== "Selecione") {
19        populateDefaultValues();
20    }
21 });
22
23 // Event listener for select buttons
24 document.getElementById('device-speed').addEventListener('input', function() {
25
26    const deviceType = document.getElementById('device-type').value;
27    const deviceSpeed = document.getElementById('device-speed').value;
28
29    // If the values are not de default, populate the tables
30    if (deviceType !== "Selecione" && deviceSpeed !== "Selecione") {
31        populateDefaultValues();
32    }
33 });
```

Script da camada de serviço para comunicação com o backend

```
1 const specDelta = [20, 4, 2.5, 1.25];
2 const specParameterNames = [
3     "Speed Class", "Bit High Time", "Bit Low Time", "Sync Time", "Bit Sync Time", "Bit
Sync High", "Bit Sync Low", "Bus Yield"
4 ];
5
6 const specDefaultValues = [
7     // Speed (MHz), Bit High (us), Bit Low (us), Sync Time (us), Bit Sync Time (us), Bit
Sync High (us), Bit Sync Low (us), Bus Yield
8     [4, 2*specDelta[0], specDelta[0], 25*specDelta[0], 15*specDelta[0], 7.5*specDelta
[0], 7.5*specDelta[0], "Yes"],
9     [20, 2*specDelta[1], specDelta[1], 25*specDelta[1], 15*specDelta[1], 7.5*specDelta
[1], 7.5*specDelta[1], "Yes"],
10    [32, 2*specDelta[2], specDelta[2], 25*specDelta[2], 15*specDelta[2], 7.5*specDelta
[2], 7.5*specDelta[2], "Yes"],
11    [64, 2*specDelta[3], specDelta[3], 25*specDelta[3], 15*specDelta[3], 7.5*specDelta
[3], 7.5*specDelta[3], "Yes"]
12 ];
13
14 const mock = {
15     electricalInfo: {
16         "VIH (High-level input voltage)": 5.1,
17         "VIL (Low-level input voltage)": 5.1,
18         "Rise Time": 5.1,
19         "Falling Time": 5.1,
20         "Cycle Time": 5.1,
21         "Bus Max Speed": 5.1,
22     },
23     transmissionInfo: {
24         Type: 0,
25         Sync: {status: true},
26         BitSync: {status: true},
27         Size: {status: true},
28         L3: {status: true}
29     },
30     specConformity: {
31
32         "Speed": 32,
33         "Bit High Time": 5,
34         "Bit Low Time": 2.5,
35         "Sync Time": 62,
36         "Bit Sync Time": 37.5,
37         "Bit Sync High": 19,
38         "Bit Sync Low": 19,
39         "Bus Yield": true
40     }
41 }
42
43 async function fetchReportData() {
44     const deviceType = document.getElementById('device-type').value;
45     const deviceSpeed = document.getElementById('device-speed').value;
46
47     //only fetch if the user selected the params
48     if (deviceType === "Selecione" || deviceSpeed === "Selecione") {
49         showNotification('Please select both Device Type and Device Speed', 'error');
50         return;
51     }
52 }
```

```

52
53     try {
54         document.getElementById('loading-indicator').style.display = 'block';
55
56         const requestBody = {
57             isController: deviceType === 'controlador',
58             deviceSpeed: parseInt(deviceSpeed.split(' ')[0])
59         };
60
61         const response = await fetch('/api/v1/validation', {
62             method: 'POST',
63             headers: {
64                 'Content-Type': 'application/json'
65             },
66             body: JSON.stringify(requestBody)
67         });
68
69         if (!response.ok) {
70             throw new Error(`HTTP error! Status: ${response.status}`);
71         }
72
73         //get validation result in the response
74         const data = await response.json();
75
76         populateSpecConformity(data.specConformity);
77         populateTransmissionInfo(data.transmissionInfo);
78         populateElectricalInfo(data.electricalInfo);
79
80         document.getElementById('loading-indicator').style.display = 'none';
81
82         showNotification('Data loaded successfully', 'success');
83     } catch (error) {
84         console.error('Error fetching report data:', error);
85         document.getElementById('loading-indicator').style.display = 'none';
86         showNotification('Failed to load data from server', 'error');
87     }
88 }
89
90 function checkForFailures(data) {
91     // Check spec conformity for failures
92     if (data.specConformity && data.specConformity.length > 0) {
93         for (const item of data.specConformity) {
94             if (item.result === "Fail") {
95                 return true;
96             }
97         }
98     }
99
100    // Check transmission info for failures
101    if (data.transmissionInfo && data.transmissionInfo.length > 0) {
102        for (const item of data.transmissionInfo) {
103            if (item.result === "Fail") {
104                return true;
105            }
106        }
107    }
108
109    return false;
110 }
```

```

111
112 let nErrors = 1;
113
114 function populateElectricalInfo(electricalInfo) {
115     if (!electricalInfo) return;
116
117     const electricalTable = document.getElementById('electrical-info-table') .
118         getElementsByTagName('tbody')[0];
119     const rows = electricalTable.rows;
120
121     const units = ["V", "V", "us", "us", "us", "KHz"]
122
123     for (let i = 0; i < rows.length; i++){
124         let cells = rows[i].cells;
125
126         const value = electricalInfo[cells[0].textContent];
127         if (value !== undefined){
128             cells[1].textContent = "" . concat(value.toFixed(3), " ", units[i]);
129         }
130     }
131
132 function populateTransmissionInfo(transmissionInfo) {
133     if (!transmissionInfo) return;
134
135     const transmissionTable = document.getElementById('transmission-info-table') .
136         getElementsByTagName('tbody')[0];
137     const rows = transmissionTable.rows;
138
139     for (let i = 0; i < rows.length; i++){
140         let cells = rows[i].cells;
141
142         const value = transmissionInfo[cells[0].textContent];
143         if (value === undefined) continue;
144
145         if (value.status){
146             cells[1].textContent = "Pass";
147             cells[1].className = "pass";
148         } else {
149             cells[1].textContent = "Fail " . concat(nErrors);
150             cells[1].className = "fail";
151
152             AppendFailureDetails(value.reason);
153             nErrors++;
154         }
155
156     const typeCells = rows[0].cells;
157     const transmissionType = transmissionInfo.Type === 0? "L3": "Generic";
158     typeCells[1].textContent = transmissionType;
159     typeCells[1].className = transmissionInfo[transmissionType]? "pass": "fail";
160
161     if (transmissionInfo.type === 0){
162
163         const L3 = ["Header", "Source ID", "Padding", "CRC"]
164
165         if(transmissionInfo["L3"]){
166             L3.forEach(h => {
167                 const row = transmissionTable.insertRow();

```

```

168         row.insertCell(0).textContent = h;
169         const cell = row.insertCell(1);
170
171         cell.textContent = "Pass";
172         cell.className = "pass";
173     });
174 } else{
175     L3.forEach(h => {
176
177         const value = transmissionInfo[h];
178         if (value === undefined) return;
179
180         const row = transmissionTable.insertRow();
181         row.insertCell(0).textContent = h;
182         const cell = row.insertCell(1);
183
184         cell.textContent = value.status ? "Pass": "Fail";
185         cell.className = value.status ? "pass": "fail";
186
187         if (!value){
188             populateFailureDetails([value.reason]);
189         }
190     });
191 }
192 }
193 }
194
195 function populateSpecConformity(specConformity) {
196     if (!specConformity) return;
197
198     const specTable = document.getElementById('spec-conformity-table').
199     getElementsByTagName('tbody')[0];
200     const rows = specTable.rows;
201
202     const reason = " signal out of range";
203
204     for (let i = 0; i < rows.length; i++){
205         let cells = rows[i].cells;
206
207         const value = specConformity[cells[0].textContent];
208         if (value === undefined) continue;
209
210         const splits = cells[1].textContent.split(' ');
211
212         if (cells[0].textContent === "Bus Yield"){
213             cells[2].textContent = value? "Yes": "No";
214
215             if (cells[2].textContent === cells[1].textContent){
216                 cells[3].textContent = "Pass";
217                 cells[3].className = "pass";
218             } else{
219                 cells[3].textContent = "Fail";
220                 cells[3].className = "fail";
221             }
222
223             return;
224         } else {
225             cells[2].textContent = "" + value.toFixed(3) + " " + splits[1];
226         }
227     }
228 }

```

```

226     const v = parseFloat(splits[0]);
227
228     if (value >= v-v*.02 && value <= v+v*.02){
229         cells[3].textContent = "Pass";
230         cells[3].className = "pass";
231     }else{
232         cells[3].textContent = "Fail ".concat(nErrors);
233         cells[3].className = "fail";
234
235         AppendFailureDetails(cells[0].textContent.concat(reason));
236         nErrors++;
237     }
238 }
239 }
240 }
241
242 function AppendFailureDetails(detail) {
243     document.querySelector('.failure-details').style.display = 'block';
244
245     const failureList = document.getElementById('failure-list');
246
247     const li = document.createElement('li');
248     li.textContent = detail;
249     failureList.appendChild(li);
250 }
251
252 function populateDefaultValues() {
253     const deviceSpeed = document.getElementById('device-speed').value;
254     const speedValue = parseInt(deviceSpeed.split(' ')[0]); //get only the number
255
256     let defaultValues;
257
258     const specTable = document.getElementById('spec-conformity-table').
259     getElementsByTagName('tbody')[0];
260     specTable.innerHTML = '';
261
262     //a switch case performs better than a for loop
263     switch(speedValue) {
264         case 4:
265             defaultValues = specDefaultValues[0];
266             break;
267         case 20:
268             defaultValues = specDefaultValues[1];
269             break;
270         case 32:
271             defaultValues = specDefaultValues[2];
272             break;
273         case 64:
274             defaultValues = specDefaultValues[3];
275             break;
276         default:
277             return;
278     }
279
280     const specConformityData = [];
281
282     for (let i = 0; i < specParameterNames.length; i++) {
283         const parameter = specParameterNames[i];
284         let expected = defaultValues[i];

```

```

284
285     if (i === 0) {
286         expected = `${expected} MHz`;
287     } else if (i < 7) {
288         expected = `${expected} us`;
289     }
290
291     const row = specTable.insertRow();
292     const cell1 = row.insertCell(0);
293     const cell2 = row.insertCell(1);
294     const cell3 = row.insertCell(2);
295     const cell4 = row.insertCell(3);
296
297     cell1.textContent = parameter;
298     cell2.textContent = expected;
299     cell3.textContent = "-";
300     cell4.textContent = "-";
301 }
302
303     document.querySelector('.failure-details').style.display = 'none';
304 }
305
306 function initializeTablesWithPlaceholders() {
307     const electricalParameters = [
308         "VIH", "VIL",
309         "Rise Time", "Falling Time", "Cycle Time", "Bus Max Speed"
310     ];
311
312     const electricalTable = document.getElementById('electrical-info-table').
313         getElementsByTagName('tbody')[0];
314     electricalTable.innerHTML = '';
315
316     electricalParameters.forEach(param => {
317         const row = electricalTable.insertRow();
318         const cell1 = row.insertCell(0);
319         const cell2 = row.insertCell(1);
320
321         cell1.textContent = param;
322         cell2.textContent = '-';
323     });
324
325     const transmissionParameters = [
326         "Transmission Type", "Sync", "BitSync", "Size"
327     ];
328
329     const transmissionTable = document.getElementById('transmission-info-table').
330         getElementsByTagName('tbody')[0];
331     transmissionTable.innerHTML = '';
332
333     transmissionParameters.forEach(param => {
334         const row = transmissionTable.insertRow();
335         const cell1 = row.insertCell(0);
336         const cell2 = row.insertCell(1);
337
338         cell1.textContent = param;
339         cell2.textContent = '-';
340     });
341
342     const specTable = document.getElementById('spec-conformity-table').

```

```

getElementsByTagName( 'tbody' )[0];
341 specTable.innerHTML = '';
342
343 specParameterNames.forEach(param => {
344     const row = specTable.insertRow();
345     const cell1 = row.insertCell(0);
346     const cell2 = row.insertCell(1);
347     const cell3 = row.insertCell(2);
348     const cell4 = row.insertCell(3);
349
350     cell1.textContent = param;
351     cell2.textContent = '-';
352     cell3.textContent = '-';
353     cell4.textContent = '-';
354 });
355
356 document.querySelector('.failure-details').style.display = 'none';
357 }
358
359 function showNotification(message, type) {
360     const notification = document.getElementById('notification');
361     notification.textContent = message;
362     notification.className = 'notification ${type}';
363     notification.style.display = 'block';
364
365     setTimeout(() => {
366         notification.style.display = 'none';
367     }, 3000);
368 }
369
370 document.addEventListener('DOMContentLoaded', function() {
371     initializeTablesWithPlaceholders();
372 });
373
374 export { fetchReportData, populateDefaultValues };

```

APÊNDICE C - CÓDIGO-FONTE DAS ROTINAS DE TESTE DO DISPOSITIVO DE VALIDAÇÃO

```
1
2 #include "DCP.h"
3 #include "validator.h"
4
5 #include <freertos/FreeRTOS.h>
6 #include <freertos/task.h>
7 #include <freertos/portmacro.h>
8
9 #include <driver/gpio.h>
10 #include <esp_private/esp_clk.h>
11 #include <esp_log.h>
12 #include <rom/ets_sys.h>
13 #include "esp_cpu.h"
14
15 #include "esp_adcadc_oneshot.h"
16 #include "esp_adcadc_cali.h"
17 #include "esp_adcadc_cali_scheme.h"
18
19 ///////////////////////////////////////////////////////////////////
20
21 extern portMUX_TYPE criticalMutex;
22
23 struct DCP_electrical_t MeasureElectrical(const gpio_num_t pin){
24     struct DCP_electrical_t ret = {0};
25
26     ret.VIH = 0;
27     ret.VIL = 0;
28
29     for(esp_cpu_set_cycle_count(0); esp_cpu_get_cycle_count() < 10UL*esp_clk_cpu_freq();)
30     {
31         if(gpio_get_level(pin) == 1){
32             break;
33         }
34     }
35     if(gpio_get_level(pin) == 1){
36
37         static int adc_raw[2][10];
38         static int voltage[2][10];
39
40         adc_oneshot_unit_handle_t adc1_handle;
41         ESP_ERROR_CHECK(adc_oneshot_new_unit(&(adc_oneshot_unit_init_cfg_t){.unit_id =
42             ADC_UNIT_1}, &adc1_handle));
43
44         ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL_2, &(adc_
45             _oneshot_chan_cfg_t){.atten = ADC_ATTEN_DB_12, .bitwidth = ADC_BITWIDTH_DEFAULT})) ;
46
47         adc_cali_handle_t adc1_cali_chan0_handle = NULL;
48
49         adc_cali_curve_fitting_config_t cali_config = {
50             .unit_id = ADC_UNIT_1,
51             .chan = ADC_CHANNEL_2,
52             .atten = ADC_ATTEN_DB_12,
```

```

52         .bitwidth = ADC_BITWIDTH_DEFAULT,
53     };
54
55     if (adc_cali_create_scheme_curve_fitting(&cali_config, &handle) == ESP_OK) {
56
57         ESP_ERROR_CHECK(adc_oneshot_read(adcl_handle, ADC_CHANNEL_2, &adc_raw[0][0]));
58
59         ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adcl_cali_chan0_handle, adc_raw[0][0], &voltage[0][0]));
60
61         ret.VIH = voltage[0][0];
62
63         gpio_set_level(pin, 0);
64         gpio_set_direction(pin, GPIO_MODE_OUTPUT);
65
66         ESP_ERROR_CHECK(adc_oneshot_read(adcl_handle, ADC_CHANNEL_2, &adc_raw[0][0]));
67
68         ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adcl_cali_chan0_handle, adc_raw[0][0], &voltage[0][0]));
69
70         gpio_set_direction(pin, GPIO_MODE_INPUT);
71
72         ret.VIL = voltage[0][0];
73
74         ESP_ERROR_CHECK(adc_oneshot_del_unit(adcl_handle));
75         example_adc_calibration_deinit(adcl_cali_chan0_handle);
76     } else {
77         ESP_ERROR_CHECK(adc_oneshot_del_unit(adcl_handle));
78     }
79
80     gpio_set_direction(pin, GPIO_MODE_INPUT);
81     gpio_set_level(pin, 0);
82
83     taskENTER_CRITICAL(&criticalMutex);
84
85     esp_cpu_set_cycle_count(0);
86     gpio_set_direction(pin, GPIO_MODE_OUTPUT);
87     while(gpio_get_level(pin) == 1);
88     const esp_cpu_cycle_count_t th = esp_cpu_get_cycle_count();
89
90     esp_cpu_set_cycle_count(0);
91     gpio_set_direction(pin, GPIO_MODE_INPUT);
92     while(gpio_get_level(pin) == 0);
93     const esp_cpu_cycle_count_t tl = esp_cpu_get_cycle_count();
94
95     taskEXIT_CRITICAL(&criticalMutex);
96
97     ESP_LOGV("Electrical", "th: %lu\ttl: %lu", th, tl);
98
99     ret.rise = (double)th/esp_clk_cpu_freq();
100    ret.falling = (double)tl/esp_clk_cpu_freq();
101    ESP_LOGV("Electrical", "rise: %f\tfalling: %f", ret.rise, ret.falling);
102
103    ret.cycle = ret.rise + ret.falling;
104
105    if (ret.cycle != 0){
106        ret.speed = 1/ret.cycle;

```

```

107     }
108
109     return ret;
110 }
111
112 ///////////////////////////////////////////////////////////////////
113
114 extern volatile struct {
115     float delta;      //transmission time unit
116     float moe;        //transmission margin of error
117     esp_cpu_cycle_count_t limits[2];
118 } configParam;
119
120 static DCP_MODE targetParams;
121
122 extern bool s_SendBytes(gpio_num_t const pin, uint8_t const size, uint8_t const data[size
    ], unsigned const delays[restrict 3]);
123 extern uint8_t s_ReadByte(const gpio_num_t pin);
124
125 ///////////////////////////////////////////////////////////////////
126
127 volatile static struct rawCycles_t{
128     esp_cpu_cycle_count_t sync;
129     esp_cpu_cycle_count_t bitSync_low;
130     esp_cpu_cycle_count_t bitSync_high;
131     esp_cpu_cycle_count_t bit0;
132     esp_cpu_cycle_count_t bit1;
133 } rawCycles;
134
135 bool ReadBit(const gpio_num_t pin){
136
137     while (gpio_get_level(pin) == 0)
138         continue;
139
140     //reading high time
141     esp_cpu_set_cycle_count(0);
142     for (esp_cpu_cycle_count_t lim = configParam.limits[1] << 1; gpio_get_level(pin) == 1
143         && esp_cpu_get_cycle_count() < lim;)
144         continue;
145
146     const esp_cpu_cycle_count_t t = esp_cpu_get_cycle_count();
147
148     if (t <= configParam.limits[1]){
149         rawCycles.bit0 = t;
150         return 0;
151     } else {
152         rawCycles.bit1 = t;
153     }
154 }
155
156 uint8_t ReadByte(const gpio_num_t pin){
157
158     uint8_t byte = 0;
159
160     for (int i = 7; i >= 0; —i){
161         byte |= ReadBit(pin) << i;
162     }
163 }
```

```

164     return byte;
165 }
166
167 uint32_t ValidL3(uint8_t* data){return 0;}
168 uint32_t ValidGeneric(uint8_t* data){return 0;}
169
170 struct DCP_Transmission_t TestConnection(const gpio_num_t pin){
171
172     assert(configParam.limits[0] != 0 && configParam.limits[1] != 0);
173
174     gpio_set_direction(pin, GPIO_MODE_INPUT);
175
176     struct DCP_Transmission_t ret = {};
177     static uint8_t data[0xFF];
178
179     rawCycles.sync = 0;
180     rawCycles.bitSync_low = 0;
181     rawCycles.bitSync_high = 0;
182     rawCycles.bit0 = 0;
183     rawCycles.bit1 = 0;
184
185     //only leave trap if no data is on bus
186     //wait for at least 15delta of idle
187     esp_cpu_set_cycle_count(0);
188     while (esp_cpu_get_cycle_count() < 15*configParam.limits[1]){
189         if (gpio_get_level(pin) == 0) esp_cpu_set_cycle_count(0);
190     }
191
192     for(esp_cpu_set_cycle_count(0); esp_cpu_get_cycle_count() < 10UL*esp_clk_cpu_freq();)
193     {
194         if(gpio_get_level(pin) == 0){
195
196             for (esp_cpu_set_cycle_count(0); gpio_get_level(pin) == 0; rawCycles.sync =
197                 esp_cpu_get_cycle_count()){
198                 if (rawCycles.sync > 100*configParam.limits[1]){
199                     ret.errors |= ERROR_sync_inf;
200                 }
201             }
202
203             //check sync limit
204             if(rawCycles.sync > 25*configParam.limits[1]){
205                 ret.errors |= ERROR_sync_tooLong;
206             }else if(rawCycles.sync < 25*configParam.limits[0]){
207                 ret.errors |= ERROR_sync_tooShort;
208             }
209
210             //wait bitsync
211             for (;gpio_get_level(pin) == 1; rawCycles.bitSync_high =
212                 esp_cpu_get_cycle_count()){
213                 if(rawCycles.bitSync_high > 10*configParam.limits[1]){
214                     ret.errors |= ERROR_bitSync_inf;
215                 }
216
217             esp_cpu_set_cycle_count(0);
218             //check bitsync limit
219             if(rawCycles.bitSync_high > 7.5*configParam.limits[1]){

```

```

220         ret.errors |= ERROR_bitSync_tooLong;
221     }else if(rawCycles.bitSync_high < 7.5*configParam.limits[0]){
222         ret.errors |= ERROR_bitSync_tooShort;
223     }
224
225     for(;; gpio_get_level(pin) == 0; rawCycles.bitSync_low =
226 esp_cpu_get_cycle_count()){
227         if(rawCycles.bitSync_low > 10*configParam.limits[1]){
228             ret.errors |= ERROR_bitSync_invalidLow;
229         }
230     }
231
232     data[0] = ReadByte(pin);
233     const uint8_t flag = data[0]? data[0]: sizeof(struct DCP_Message_L3_t)+1;
234
235     for(int i = 1; i < flag; ++i){
236         data[i] = ReadByte(pin);
237     }
238
239     //is there any data being transmitted?
240     uint8_t end = ReadByte(pin);
241     //empty bus is read as FF, anything else is extra data
242     if(end != 0xFF){
243         ret.errors |= ERROR_invalidSize;
244     }
245
246     ret.type = data[0];
247
248     const DCP_Data_t message = {.data = data};
249     ret.errors |= message.message->type? ValidGeneric(message.data): ValidL3(
250     message.data);
251
252     return ret;
253 }
254
255 return (struct DCP_Transmission_t){.errors = ERROR_noTransmission};
256
257 /////////////////////////////////
258
259 struct DCP_timings_t GetTimes(const gpio_num_t pin){
260
261     struct DCP_timings_t ret;
262     const uint32_t freqMHz = esp_clk_cpu_freq()/1e6; //should be 180
263
264     ESP_LOGV("times", "sync: %lu\t BSH: %lu\t BSL: %lu\tB0: %lu\tB1: %lu",
265             rawCycles.sync,
266             rawCycles.bitSync_high,
267             rawCycles.bitSync_low,
268             rawCycles.bit0,
269             rawCycles.bit1
270 );
271
272     ret.speed = 0xFF;
273     ret.sync = (double)rawCycles.sync/freqMHz;
274     ret.bitSync_low = (double)rawCycles.bitSync_low/freqMHz;
275     ret.bitSync_high = (double)rawCycles.bitSync_high/freqMHz;
276     ret.bit0 = (double)rawCycles.bit0/freqMHz;

```

```

277     ret.bit1 = (double)rawCycles.bit1/freqMHz;
278
279     if( ret.bit0 < 2){
280         ret.speed = 64;
281     }else if( ret.bit0 < 3){
282         ret.speed = 32;
283     }else if( ret.bit0 < 6){
284         ret.speed = 20;
285     }else if( ret.bit0 < 23){
286         ret.speed = 4;
287     }
288
289     return ret;
290 }
291
292 ///////////////////////////////////////////////////////////////////
293
294 static const struct DCP_Message_t yieldMessage = (struct DCP_Message_t){
295     .type = 5,
296     .generic = {
297         .addr = 0x0, //highest priority
298         .payload = "test"
299     }
300 };
301
302 enum Collision_e DoesYield(const gpio_num_t pin){
303     enum Collision_e collisionFlag = COL_null;
304     const uint32_t freqMHz = esp_clk_cpu_freq()/1e6;
305
306     gpio_set_direction(pin, GPIO_MODE_INPUT);
307
308     for(esp_cpu_set_cycle_count(0); esp_cpu_get_cycle_count() < 10UL*esp_clk_cpu_freq();)
309     {
310         if(gpio_get_level(pin) == 0){
311             //wait for SYNC to end
312             while(gpio_get_level(pin) == 0) continue;
313
314             for (esp_cpu_set_cycle_count(0); gpio_get_level(pin) == 1; )
315                 if(esp_cpu_get_cycle_count() > 10*configParam.limits[1]){
316                     return collisionFlag;
317                 }
318
319             if(esp_cpu_get_cycle_count() <= 6*configParam.limits[0]){
320                 return collisionFlag;
321             }
322
323             (void)s_ReadByte(pin);
324
325             //let's read one byte and interrupt the transmission
326             (void)s_ReadByte(pin);
327
328             DCP_Data_t message = {.message = &yieldMessage};
329             bool collision = s_SendBytes(pin, message.message->type, message.data,
330                                         (unsigned[3]) {(configParam.delta-3)*freqMHz, (2*configParam.delta-3)*
331                                         freqMHz, 150});
332
333             gpio_set_direction(pin, GPIO_MODE_INPUT);
334             //assert(collisionFlag == COL_null);

```

```
334         collisionFlag = collision? COL_true: COL_false;
335     }
336 }
338
339 return collisionFlag;
340 }
```

APÊNDICE D - RESULTADO DE TESTE DE DISPOSITIVO UTILIZANDO O VALIDADOR

DCP Validation Report	http://192.168.0.2/																																				
<h3>DCP Validation Report</h3> <p>DCP-IF Date of Emission: April 22, 2025 Validation Version: 0.1</p>																																					
Name:	Teste																																				
Device Version:	0.1																																				
Device Type: *	controlador																																				
Device Speed: *	4 MHz																																				
<h4>Specification Conformity</h4> <table border="1"><thead><tr><th>Parameter</th><th>Expected</th><th>Got</th><th>Result</th></tr></thead><tbody><tr><td>Speed Class</td><td>4 MHz</td><td>4.000 MHz</td><td>Pass</td></tr><tr><td>Bit High Time</td><td>40 µs</td><td>40.888 µs</td><td>Fail 1</td></tr><tr><td>Bit Low Time</td><td>20 µs</td><td>18.950 µs</td><td>Fail 2</td></tr><tr><td>Sync Time</td><td>500 µs</td><td>491.500 µs</td><td>Pass</td></tr><tr><td>Bit Sync Time</td><td>300 µs</td><td>300.694 µs</td><td>Pass</td></tr><tr><td>Bit Sync High</td><td>150 µs</td><td>149.837 µs</td><td>Pass</td></tr><tr><td>Bit Sync Low</td><td>150 µs</td><td>150.856 µs</td><td>Pass</td></tr><tr><td>Bus Yield</td><td>Yes</td><td>No</td><td>Fail</td></tr></tbody></table>		Parameter	Expected	Got	Result	Speed Class	4 MHz	4.000 MHz	Pass	Bit High Time	40 µs	40.888 µs	Fail 1	Bit Low Time	20 µs	18.950 µs	Fail 2	Sync Time	500 µs	491.500 µs	Pass	Bit Sync Time	300 µs	300.694 µs	Pass	Bit Sync High	150 µs	149.837 µs	Pass	Bit Sync Low	150 µs	150.856 µs	Pass	Bus Yield	Yes	No	Fail
Parameter	Expected	Got	Result																																		
Speed Class	4 MHz	4.000 MHz	Pass																																		
Bit High Time	40 µs	40.888 µs	Fail 1																																		
Bit Low Time	20 µs	18.950 µs	Fail 2																																		
Sync Time	500 µs	491.500 µs	Pass																																		
Bit Sync Time	300 µs	300.694 µs	Pass																																		
Bit Sync High	150 µs	149.837 µs	Pass																																		
Bit Sync Low	150 µs	150.856 µs	Pass																																		
Bus Yield	Yes	No	Fail																																		
<h4>Transmission Information</h4> <table border="1"><thead><tr><th>Parameter</th><th>Result</th></tr></thead><tbody></tbody></table>		Parameter	Result																																		
Parameter	Result																																				
<h4>Electrical Information</h4> <table border="1"><thead><tr><th>Parameter</th><th>Value</th></tr></thead><tbody></tbody></table>		Parameter	Value																																		
Parameter	Value																																				
1 of 2	4/22/25, 3:40 AM																																				

Parameter	Result
Transmission Type	L3
Sync	Pass
BitSync	Pass
Size	Pass

Parameter	Value
VIH	3.300 V
VIL	0.000 V
Rise Time	16.000 µs
Falling Time	1.969 µs
Cycle Time	1.969 µs
Bus Max Speed	55.652 KHz

Failure Details

1. Bit High Time signal out of range
2. Bit Low Time signal out of range

APÊNDICE E - CÓDIGO-FONTE DO MÓDULO GENÉRICO DO PROTOCOLO DCP

Cabeçalho da biblioteca

```
1 #ifndef __DCP__
2 #define __DCP__
3
4 #ifdef __cplusplus
5 extern "C" {
6 #endif
7
8 #include <stdbool.h>
9 #include <stdint.h>
10 #include <stdlib.h>
11
12 enum e_Flags {
13     FLAG_Instant      = 0b0,
14     FLAG_Assynchronous = 0b1
15 };
16
17 typedef struct DCP_MODE{
18     uint8_t addr;
19     union{
20         enum e_Flags e_flags;
21         unsigned int flags;
22     } flags;
23     bool isController;
24     enum {SLOW = 0, FAST1, FAST2, ULTRA} speed;
25 }DCP_MODE;
26
27 bool DCPIInit(const unsigned int busPin, const DCP_MODE mode);
28
29 struct DCP_Message_L3_t{
30     uint8_t SOH;           //header
31     uint8_t IDS;          //source ID
32     uint8_t IDD;          //destinaiton ID
33     uint8_t COD;          //instruction code
34     uint8_t data[6];       //data
35     uint8_t PAD;          //padding
36     uint8_t CRC_;         //CRC
37 };
38
39 struct DCP_Message_Generic_t{
40     uint8_t addr;
41     uint8_t payload[];
42 };
43
44 struct DCP_Message_t {
45     uint8_t type;
46     union {
47         struct DCP_Message_L3_t L3;
48         struct DCP_Message_Generic_t generic;
49     };
50 };
51
52 //this is done so we can access the message as a continuous byte array
```

```

53 //in this way, it is possible to send everything in one go
54 typedef union {
55     struct DCP_Message_t * const message;
56     uint8_t * data;
57 } DCP_Data_t;
58
59 bool SendMessage(const DCP_Data_t message);
60 struct DCP_Message_t* ReadMessage();
61
62 #ifdef __cplusplus
63 }
64 #endif
65
66 #endif

```

Código fonte da biblioteca

```

1 #include "DCP.h"
2
3 #include <freertos/FreeRTOS.h>
4 #include <freertos/task.h>
5 #include <freertos/queue.h>
6
7 #include <portmacro.h>
8
9 #include <assert.h>
10 #include <stdbool.h>
11 #include <stdint.h>
12 #include <string.h>
13 #include <limits.h>
14
15 extern void Log(char const * const tag, char const * const msg, ...);
16
17 extern void gpio_set_direction(unsigned int pin, unsigned int dir);
18 extern void gpio_set_level(unsigned int pin, unsigned int level);
19 extern int gpio_get_level(unsigned int pin);
20
21 extern uint32_t get_clock_speed();
22 extern void reset_clock_tick();
23 extern uint32_t get_clock_tick();
24 extern void toggle_debug_pin();
25
26 static char* TAG = "DCP Driver";
27
28 volatile DCP_MODE busMode;
29
30 QueueHandle_t RXmessageQueue = NULL;
31 QueueHandle_t TXmessageQueue = NULL;
32 QueueHandle_t isrq = NULL;
33
34 TaskHandle_t busTask = NULL;
35
36 volatile struct {
37     float delta;      //transmission time unit
38     float moe;        //transmission margin of error
39     uint32_t limits[2];
40 } configParam;

```

```

41
42 /*!
43 * @brief generic definition of function that delays for microseconds
44 * @param ticks = delay in us * frequency in MHz
45 */
46 static __attribute__((always_inline)) inline void Delay(const uint32_t ticks){
47     reset_clock_tick();
48
49     taskENTER_CRITICAL();
50
51     while (get_clock_tick() < ticks)
52         asm volatile ("nop");
53
54     taskEXIT_CRITICAL();
55 }
56
57 static inline bool s_ReadBit(const unsigned int pin){
58
59     while (gpio_get_level(pin) == 0)
60         continue;
61
62     //reading high time
63     reset_clock_tick();
64     for (uint32_t lim = configParam.limits[1] << 1;
65          gpio_get_level(pin) == 1 && get_clock_tick() < lim;)
66         continue;
67
68     return get_clock_tick() <= configParam.limits[1]? 0: 1;
69 }
70
71 static uint8_t s_ReadByte(const unsigned int pin){
72
73     uint8_t byte = 0;
74
75     for (int i = 7; i >= 0; —i){
76         byte |= s_ReadBit(pin) << i;
77     }
78
79     return byte;
80 }
81
82
83 void BusISR(void* arg){
84
85     const uint16_t pin = (uint16_t)arg;
86     static uint8_t data[0xFF];
87
88     //reading incoming data
89     gpio_set_direction(pin, 1);
90     toggle_debug_pin();
91
92     //wait for SYNC to end
93     while(gpio_get_level(pin) == 0) continue;
94
95     toggle_debug_pin();
96     for (reset_clock_tick(); gpio_get_level(pin) == 1; ){
97         if(get_clock_tick() > 10*configParam.limits[1]){
98             return;
99         }

```

```

100     }
101
102     if(get_clock_tick() <= 6*configParam.limits[0]){
103         return;
104     }
105
106     toggle_debug_pin();
107     data[0] = s_ReadByte(pin);
108     const uint8_t flag = data[0]? data[0]: sizeof(struct DCP_Message_L3_t)+1;
109
110    for (int i = 1; i < flag; ++i){
111        toggle_debug_pin();
112        data[i] = s_ReadByte(pin);
113        toggle_debug_pin();
114    }
115
116    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
117    xQueueSendFromISR(isrq, data, &xHigherPriorityTaskWoken);
118    toggle_debug_pin();
119
120    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
121 }
122
123 static inline bool s_SendBytes(unsigned int const pin, uint8_t const size,
124                               uint8_t const data[size],
125                               unsigned const delays[restrict 3]){
126
127    for (int i = 0; i < size; ++i){
128        for (int j = 7; j >= 0; --j){
129            //bus modulation
130            // if bit == 0: 1 delta high, 1 delta low
131            // else: 2 delta high, 1 delta low
132
133            gpio_set_direction(pin, 1);
134
135            if (((data[i] >> j) & 0x1) == 0){
136                Delay(delays[0]);
137            }else {
138                Delay(delays[1]);
139            }
140
141            //collision
142            //if any transmission pulled the pin low after
143            //the delay started, they should still be
144            //keeping the delay low.
145            if (gpio_get_level(pin) == 0){
146                return true;
147            }
148
149            //low side of the bit
150            gpio_set_direction(pin, 0);
151            gpio_set_level(pin, 0);
152
153            Delay(delays[0] + delays[2]);
154            gpio_set_direction(pin, 1);
155
156            toggle_debug_pin();
157            //collision
158            if(gpio_get_level(pin) == 0){

```

```

159             return true;
160         }
161     }
162 }
163
164 return false;
165 }
166
167 /*!
168  * @brief task that controls the state machine of the control of the bus
169  */
170 _Noreturn void busHandler(void* arg){
171
172     const unsigned int pin = (unsigned int)arg;
173     enum {STARTING, LISTENING, SENDING, WAITING, READING, END_} state = WAITING;
174
175     // precalculations
176     const uint32_t freqMHz = get_clock_speed() /1e6;
177
178 #if defined(CONFIG_IDF_TARGET_ESP32C3)
179
180     //negative skews in us to be added to the timings
181     const unsigned int skews[4][4] = {
182         //listening , sync , 0 , 1
183         {0, 20, 3, 3},
184         {0, 25, 2, 1},
185         {0, 20, 2, 2},
186         {0, 20, 1, 0}
187     };
188
189 #elif defined(rp2350)
190
191     const unsigned int skews[4][4] = {
192         {0, 10, 3, 4},
193         {0, 0, 0, 0},
194         {0, 0, 0, 0},
195         {0, 0, 0, 0}
196     };
197
198 #else
199
200     const unsigned int skews[4][4] = {
201         {0, 0, 0, 0},
202         {0, 0, 0, 0},
203         {0, 0, 0, 0},
204         {0, 0, 0, 0}
205     };
206
207 #endif
208
209     const uint32_t delays[] = {
210         (busMode.addr + 6) * configParam.delta /4.0 * freqMHz,
211         ((busMode.isController?25:50) *
212             configParam.delta-skews[busMode.speed][1])*freqMHz,
213         (configParam.delta-skews[busMode.speed][2])*freqMHz,
214         (2*configParam.delta-skews[busMode.speed][3])*freqMHz
215     };
216
217     // variables

```

```

218     uint8_t qItem[0xFF];
219     DCP_Data_t message = {0};
220     bool collision = false;
221
222     assert(RXmessageQueue != NULL);
223     assert(TXmessageQueue != NULL);
224
225     gpio_set_direction(pin, 1);
226
227     while(1){
228
229         //toggle_debug_pin();
230
231         switch(state){
232             case LISTENING:
233                 //listening bus for CSMA
234                 if(gpio_get_level(pin) == 0){
235                     state = WAITING;
236                     continue;
237                 }
238                 taskENTER_CRITICAL();
239
240                 toggle_debug_pin();
241                 ulTaskNotifyValueClear(busTask, UINT_MAX);
242                 //protocol priority delay
243                 //devices with smaller addresses will have the priority
244                 Delay(delays[0]);
245
246                 //while in the delay, did someone take the bus?
247                 if(ulTaskNotifyTake(pdTRUE, 0)){
248                     Log(TAG, "someone took the bus");
249                     state = WAITING;
250                     continue;
251                 }
252
253                 toggle_debug_pin();
254                 __attribute__((fallthrough));
255             case STARTING:
256                 //starting communication
257                 if(gpio_get_level(pin) == 0){
258                     taskEXIT_CRITICAL();
259                     state = WAITING;
260                     continue;
261                 }
262
263                 //sync signal
264                 gpio_set_direction(pin, 0);
265
266                 Delay(delays[1]);
267
268                 //bit sync signal
269                 //high part
270                 toggle_debug_pin();
271                 gpio_set_direction(pin, 1);
272                 Delay((uint32_t)(8 * delays[2]));
273
274                 //bit sync signal
275                 //low part
276                 gpio_set_direction(pin, 0);

```

```

277     gpio_set_level(pin, 0);
278     Delay((uint32_t)(8 * delays[2]));
279
280     toggle_debug_pin();
281
282     //leaving the bus still low not to interfere in the first bit
283     __attribute__((fallthrough));
284 case SENDING:
285     //sending data
286     assert(message.data != NULL);
287
288     toggle_debug_pin();
289
290     collision = s_SendBytes(pin,
291                             message.message->type?
292                             message.message->type:
293                             sizeof(struct DCP_Message_t),
294                             message.data,
295                             (unsigned[3]) {delays[2], delays[3], 150});
296
297     taskEXIT_CRITICAL();
298
299     toggle_debug_pin();
300     if (collision){
301         Log(TAG, "Collision detected");
302         state = LISTENING;
303         continue;
304     }
305
306     free(message.data);
307
308     Log(TAG, "successfully sent message, going to wait mode");
309
310     ulTaskNotifyValueClear(busTask, UINT_MAX);
311     gpio_set_direction(pin, 1);
312
313     __attribute__((fallthrough));
314 case WAITING:
315     //waiting for messages to send/receive
316     state = WAITING;
317
318     //message to read
319     if(uxQueueMessagesWaiting(isrq)){
320         state = READING;
321         break;
322     }
323
324     //message to send
325     if (xQueueReceive(TXmessageQueue, &(message.data), 1) == pdPASS){
326         state = LISTENING;
327         break;
328     }
329
330     break;
331 case READING:
332
333     xQueueReceive(isrq, &qItem, pdMS_TO_TICKS(5));
334
335     message.data = malloc(qItem[0]?

```

```

336                     qItem [ 0 ]:
337                     sizeof( struct DCP_Message_t ) * sizeof( uint8_t ) );
338             memmove( message . data , qItem , qItem [ 0 ]?
339                         qItem [ 0 ]:
340                         sizeof( struct DCP_Message_t ) );
341
342             xQueueSend ( RXmessageQueue , &( message . data ) , pdMS_TO_TICKS( 15 ) );
343
344             state = WAITING;
345             break;
346         default :
347             Log(TAG, "this code should not be executed , possible corruption");
348             break;
349     }
350 }
351 }
352
353 bool SendMessage( const DCP_Data_t message){
354
355 #ifdef ESP_LOGD
356     if ( message . message->type ){
357         Log(TAG, "sending message: %s" , message . message->generic . payload );
358     } else {
359         Log(TAG, "sending L3 message" );
360     }
361 #endif
362
363     if ( xQueueSend ( TXmessageQueue , ( void * ) &( message . data ) , portMAX_DELAY ) != pdTRUE ){
364         Log(TAG, "could not send message to queue" );
365         return false ;
366     }
367
368     return true ;
369 }
370
371 struct DCP_Message_t* ReadMessage () {
372
373     DCP_Data_t message = { 0 };
374
375     if ( ( busMode . flags . flags & 0x1 ) == FLAG_Instant ){
376         if ( xQueueReceive ( RXmessageQueue , &( message . data ) , 0 ) == pdTRUE ){
377
378             return message . message ;
379         }
380
381         return NULL;
382     }
383
384     if ( xQueueReceive ( RXmessageQueue , &( message . data ) , portMAX_DELAY ) == pdTRUE ){
385
386         return message . message ;
387     }
388
389     return NULL;
390 }

```

APÊNDICE F - CÓDIGO-FONTE DO *PORT* DO PROTOCOLO DCP PARA O RP2350

```
1
2 #include "DCP.h"
3
4 #include "pico/stl.h"
5 #include "hardware/gpio.h"
6 #include "hardware/timer.h"
7 #include "hardware/clocks.h"
8
9 #include <freertos/FreeRTOS.h>
10 #include <freertos/task.h>
11 #include <freertos/queue.h>
12
13 #include <stdarg.h>
14 #include <stdio.h>
15 #include <assert.h>
16
17 static char* TAG = "DCP port";
18
19 ///////////////////////////////////////////////////
20
21 static const float deltaLUT[] = {20, 4, 2.5, 1.25};
22 extern struct {
23     float delta;      //transmission time unit
24     float moe;        //transmission margin of error
25     uint32_t limits[2];
26 } configParam;
27
28 __Noreturn extern void busHandler(void* arg);
29
30 extern void BusISR(void* arg);
31 ///////////////////////////////////////////////////
32
33 static volatile timer_hw_t* tmr = NULL;
34
35 void Log(char const * const tag, char const * const msg, ...){
36     va_list args;
37
38     printf("%s", tag);
39
40     va_start(args, msg);
41     printf(msg, args);
42     va_end(args);
43 }
44
45 void gpio_set_direction(unsigned int pin, unsigned int dir){
46     gpio_set_dir(pin, dir == 0); //dir == true: out; dir == false: in
47 }
48
49 void gpio_set_level(unsigned int pin, unsigned int level){
50     gpio_put(pin, level);
51 }
52
53 int gpio_get_level(unsigned int pin){
54     //we have to burn some cycles
```

```

56     for (int i = 0; i<20; ++i) asm volatile ("nop");
57
58     return gpio_get(pin);
59 }
60
61 void toggle_debug_pin(){
62     gpio_xor_mask(1u << 25 | 1u << 28);
63 }
64
65 uint32_t get_clock_speed(){
66     return clock_get_hz(clk_sys);
67 }
68
69 void reset_clock_tick(){
70     tmr->timelw = 0x0;
71     tmr->timehw = 0x0;
72 }
73
74 uint32_t get_clock_tick(){
75     return tmr->timelr;
76 }
77
78 static void GPIO_callback(unsigned gpio, uint32_t event){
79     if (gpio == 2){
80         assert(event == 0x4); // event 0x4 == edge fall
81         BusISR((void*)2);
82     }
83 }
84
85 /////////////////////////////////
86 extern DCP_MODE busMode;
87 extern QueueHandle_t RXmessageQueue;
88 extern QueueHandle_t TXmessageQueue;
89 extern QueueHandle_t isrq;
90
91 extern TaskHandle_t busTask;
92
93 /////////////////////////////////
94 bool DCPIInit(const unsigned int busPin, const DCP_MODE mode){
95
96     if (mode.addr == 0) return false;
97
98     if (busMode.addr != 0){
99         busMode = mode;
100        return true;
101    }
102
103    // Initialize GPIO
104    gpio_init(busPin);
105    gpio_pull_up(busPin);
106    gpio_set_dir(busPin, GPIO_IN);
107    gpio_put(busPin, 0);
108
109    gpio_init(25);
110    gpio_set_dir(25, GPIO_OUT);
111
112    tmr = timer_get_instance(1);
113    assert(tmr != NULL);
114    tmr->source = 0x1; //clk_sys

```

```

115     tmr->pause = 0x0; //unpause
116
117     RXmessageQueue = xQueueCreate(8, sizeof(uint8_t *));
118     if (!RXmessageQueue){
119         Log(TAG, "could not create RX message queue");
120
121         return false;
122     }
123     Log(TAG, "RX queue created");
124
125     TXmessageQueue = xQueueCreate(8, sizeof(uint8_t *));
126     if (!TXmessageQueue){
127         Log(TAG, "could not create TX message queue");
128
129         vQueueDelete(RXmessageQueue);
130
131         return false;
132     }
133     Log(TAG, "TX queue created");
134
135     busMode = mode;
136     configParam.delta = deltaLUT[busMode.speed];
137     configParam.moe = .02*configParam.delta;
138
139     configParam.limits[0] = ((configParam.delta - configParam.moe)*1e-6)*
140                             get_clock_speed();
141     configParam.limits[1] = ((configParam.delta + configParam.moe)*1e-6)*
142                             get_clock_speed();
143
144     Log(TAG, "transmission limits: [%lu ~ %lu] ticks",
145         configParam.limits[0], configParam.limits[1]);
146     Log(TAG, "transmission limits: [%2.2f ~ %2.2f] us",
147         configParam.delta - configParam.moe,
148         configParam.delta + configParam.moe);
149
150     isrq = xQueueCreate(3, 0xFF);
151     if (isrq == NULL){
152         Log(TAG, "could not create ISR queue");
153
154         vQueueDelete(RXmessageQueue);
155         vQueueDelete(TXmessageQueue);
156
157         return false;
158     }
159     Log(TAG, "ISR queue created");
160
161     gpio_set_irq_enabled_with_callback(busPin, GPIO_IRQ_EDGE_FALL, true, &GPIO_callback);
162     xTaskCreate(busHandler, "DCP bus handler", 2*1024,
163                 (void*)busPin, configMAX_PRIORITIES-2, &busTask);
164
165     if (!busTask){
166         Log(TAG, "could not create bus arbitrator task");
167
168         vQueueDelete(RXmessageQueue);
169         vQueueDelete(TXmessageQueue);
170
171         // Disable IRQ if we fail
172         gpio_set_irq_enabled(busPin, GPIO_IRQ_EDGE_FALL, false);
173

```

```
174     vQueueDelete( isrq );
175
176     return false;
177 }
178 Log(TAG, "bus arbitrator task created");
179
180 return true;
181 }
```