



Universidade Federal de Uberlândia
Programa de Pós-Graduação em Engenharia Elétrica

Criando gêmeos digitais no Renode: um estudo de caso com microcontrolador STM32L4 e rádio Zigbee AT86RF233

Felipe Narimatsu Presti

Uberlândia
2026



Universidade Federal de Uberlândia
Programa de Pós-Graduação em Engenharia Elétrica

Felipe Narimatsu Presti

**Criando gêmeos digitais no Renode: um estudo de caso
com microcontrolador STM32L4 e rádio Zigbee
AT86RF233**

Dissertação apresentada à Universidade
Federal de Uberlândia como parte dos re-
quisitos para a obtenção do título de Mes-
tre em Engenharia Elétrica.

Orientador: Márcio José da Cunha
Coorientador: Marcelo Barros de Almeida

Uberlândia
2026

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

P936
2026

Presti, Felipe Narimatsu, 2001-
Criando gêmeos digitais no Renode: um estudo de caso com
microcontrolador STM32L4 e rádio Zigbee AT86RF233 [recurso
eletrônico] / Felipe Narimatsu Presti. - 2026.

Orientador: Márcio José da Cunha.

Coorientador: Marcelo Barros de Almeida.

Dissertação (Mestrado) - Universidade Federal de Uberlândia,
Pós-graduação em Engenharia Elétrica.

Modo de acesso: Internet.

DOI <http://doi.org/10.14393/ufu.di.2026.263>

Inclui bibliografia.

Inclui ilustrações.

1. Engenharia elétrica. I. Cunha, Márcio José da, 1978-, (Orient.).
II. Almeida, Marcelo Barros de, 1972-, (Coorient.). III. Universidade
Federal de Uberlândia. Pós-graduação em Engenharia Elétrica. IV.
Título.

CDU: 621.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

Nelson Marcos Ferreira - CRB6/3074

Abstract

The consolidation of cyber-physical systems and *Internet of Things* (IoT) applications has increased the demand for verification and validation methods that reduce cost, risk, and development time in resource-constrained devices. Although *digital twins* are often employed at higher abstraction layers (e.g., cloud integration, visualization, and system-level analytics), faithfully reproducing low-level hardware behavior remains a bottleneck for *firmware* validation on microcontrollers, especially for low-power families that are still underrepresented in simulation platforms.

In this context, this work presents an extension of the Renode simulator, an open-source development framework designed to accelerate embedded system and *IoT* development by enabling emulation of physical hardware, by implementing a new *firmware*-level digital twin for the STM32L4 family. The proposed platform is instantiated and validated on the STM32L496 microcontroller and simulated with the IEEE 802.15.4 AT86RF233 transceiver. The proposed approach follows incremental, test-oriented development, building a complete new platform through `.repl` descriptions, C# peripheral models, and `.resc` automation scripts, covering functional integration across subsystems. In addition, this work introduces a newly developed power consumption estimator that accumulates energy statistics over the simulation *virtual time*, selecting typical current profiles according to frequency, voltage, and *run/sleep* operating modes based on device datasheet metrics.

Experimental results demonstrate the execution, without modifications, of a production firmware binary (`.elf`), whose state machine simultaneously exercises initialization, ADC acquisition with DMA transfers, *timer-based* timing, radio communication, RX/TX windows and acknowledgment handling, and transitions to low-power operation. Instrumentation via a *packet hook* enabled frame interception and UDP forwarding for external decoding of telemetry, increasing protocol observability and supporting the validation of RX/TX flows in the simulated medium. Within the defined scope, the power estimation was applied only to the STM32L496 microcontroller and did not include the radio module consumption, serving as a proof of concept for reproducible energy analyses during simulation.

Keywords: Digital twins; Renode; STM32L496; AT86RF233; Firmware simulation; Energy consumption estimation.

Resumo

A consolidação de sistemas ciberfísicos e aplicações de *Internet of Things* (IoT) tem ampliado a demanda por métodos de verificação e validação que reduzam custo, risco e tempo de desenvolvimento em dispositivos com recursos restritos. Embora *digital twins* sejam frequentemente empregados em camadas de alto nível (por exemplo, integração em nuvem, visualização e análises), a reprodução fiel do comportamento de baixo nível do hardware permanece um gargalo para a validação de *firmware* em microcontroladores, especialmente em famílias de baixo consumo de energia ainda pouco atendidas por plataformas de simulação.

Neste contexto, este trabalho apresenta uma extensão do simulador Renode, um *framework* de desenvolvimento de código aberto projetado para acelerar o desenvolvimento de sistemas embarcados e de *IoT* ao permitir a emulação de hardware físico, por meio da implementação de um novo gêmeo digital em nível de *firmware* para a família STM32L4. A plataforma proposta é instanciada e validada no microcontrolador STM32L496 e simulada com o transceptor IEEE 802.15.4 AT86RF233. A abordagem segue um desenvolvimento incremental orientado a testes, construindo uma nova plataforma completa a partir de descrições `.repl`, modelos de periféricos em C# e scripts de automação `.resc`, contemplando a integração funcional entre subsistemas. Adicionalmente, este trabalho introduz um estimador de consumo energético desenvolvido especificamente para a plataforma, que acumula estatísticas de energia ao longo do tempo virtual de simulação, selecionando perfis típicos de corrente conforme frequência, tensão e modos *run/sleep*, com base em métricas do datasheet do dispositivo.

Os resultados demonstram a execução, sem modificações, do binário (`.elf`) de uma aplicação em produção, cuja máquina de estados exercita simultaneamente inicialização, aquisição via ADC com transferências por DMA, temporização por *timers*, comunicação via rádio, janelas de recepção/espera por confirmação e transições para baixo consumo. A instrumentação por *packet hook* viabilizou a interceptação e o encaminhamento de quadros via UDP para decodificação externa de telemetria, aumentando a observabilidade do protocolo e apoiando a validação dos fluxos de RX/TX no meio simulado. Por escopo, a estimativa de consumo foi aplicada apenas ao microcontrolador STM32L496 e não abrangeu o consumo do módulo de rádio, caracterizando uma prova de conceito para análises reprodutíveis de energia durante a simulação.

Palavras-chave: Gêmeos digitais; Renode; STM32L496; AT86RF233; Simu-

lação de *firmware*; Estimativa de consumo energético.

Prefácio

Esta dissertação de mestrado foi submetida à Universidade Federal de Uberlândia (UFU) como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

A dissertação foi desenvolvida no Programa de Pós-Graduação em Engenharia Elétrica (PPGELT), tendo como orientador o Prof. Dr. **Márcio José da Cunha**. O Prof. Dr. **Marcelo Barros de Almeida** foi coorientador deste trabalho.

Esta pesquisa foi financiada pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). O trabalho contou ainda com a infraestrutura do Laboratório de Estruturas Mecânicas (LMEST) e com o apoio financeiro da Petrobras.

A partir desta dissertação, foi criado o artigo científico "Creating Digital Twins in Renode: A Case Study with STM32L4 and AT86RF233 Zigbee radio" para a Conferência Internacional sobre Aplicações Industriais Induscon 2025. (Presti et al., 2025).

Agradecimentos

Agradeço profundamente ao Laboratório de Estruturas Mecânicas (LMEST) e a todos os seus membros, pelo ambiente de pesquisa desafiador e acolhedor, que me proporcionou uma grande oportunidade de aprendizado e crescimento acadêmico e profissional. Em especial, agradeço aos professores Marcelo Barros e Márcio Cunha pela orientação não apenas acadêmica, mas também ao longo da minha jornada como Engenheiro de Computação.

Agradeço também aos meus colegas de laboratório, Lucas Araújo, Lucas Lofrano, Luiz Pedro, Stephan, Lukas Gabriel e Eduardo, pelos momentos de pesquisa e descontração. São amizades que quero manter por toda a vida.

Agradeço à minha família por ser meu porto seguro, por sempre me incentivar a estudar e a buscar ser uma pessoa melhor. Sou grato por todos os conselhos e pelo amparo que sempre recebi.

Agradeço à minha namorada Cecília por ser meu apoio emocional durante o mestrado e em todos os outros momentos, sempre que precisei de auxílio, ela esteve presente para escutar, ajudar e acolher.

Espero poder retribuir a todos. Meu sincero e verdadeiro obrigado.

*“Nós mudamos quando crescemos,
somos jovens e não sabemos, que
um dia envelhecemos e distraídos
venceremos.”*

– Selvagens à Procura de Lei

Sumário

Abstract	ii
Resumo	iii
Prefácio	v
Agradecimentos	vi
Alinhamento com a Linha de Pesquisa	xii
Lista de Tabelas	xiii
Lista de Figuras	xv
Lista de Abreviações	xvi
1 Introdução	1
1.1 Proposta e Objetivos	2
1.1.1 Proposta do trabalho	2
1.1.2 Objetivo geral	3
1.1.3 Objetivos específicos	3
1.2 Contribuições	4
1.3 Organização do Trabalho	4
2 Revisão Bibliográfica	6
2.1 Gêmeos digitais (<i>Digital Twins</i>)	6
2.1.1 Evolução do conceito e distinções fundamentais	6
2.1.2 Como gêmeos digitais funcionam: componentes e ciclo de re- troalimentação	7
2.1.3 Para que servem: aplicações e objetivos recorrentes	7
2.1.4 Relação com Sistemas Ciber-Físicos	8
2.1.5 Síntese dos elementos mínimos	8
2.2 Utilização de gêmeos digitais em sistemas embarcados	9
3 Renode	12
3.1 Sobre o Renode	12

3.2	Componentes do Renode	13
3.2.1	Arquivos de descrição de plataforma	14
3.2.2	Arquivos de descrição de periféricos	14
3.2.3	Arquivos de descrição de simulação	15
3.2.4	Arquivos executáveis <code>.ELF</code>	15
3.2.5	Arquivos de descrição de sistema (<i>System View Description</i>)	15
3.2.6	Arquivos Python	16
3.3	Detalhamento dos arquivos <code>.repl</code>	16
3.4	Scripts de Execução (<code>.resc</code>) e Automação da Simulação no Renode	18
3.5	Detalhamento dos arquivos <code>.cs</code>	20
3.5.1	Estrutura interna típica de um modelo (<code>.cs</code>) no Renode	21
3.6	Tempo no Renode	24
4	Metodologia	26
4.1	Cenário de aplicação do <i>firmware</i> de produção	26
4.1.1	Firmware de produção	27
4.2	Visão geral do método	28
4.3	Ambiente e ferramentas	29
4.4	Crerios de validação	30
4.5	Testes de verificação	31
4.5.1	Primeira fase de testes: alternância de GPIO (<i>toggle</i>)	31
4.5.2	Segunda fase de testes: comunicação via UART e SPI	32
4.5.3	Terceira fase de testes: integração com o rádio AT86RF233	34
4.5.4	Quarta fase de testes: execução do <i>firmware</i> de produção	36
5	Desenvolvimento	38
5.1	Criação e compilação de <i>firmwares</i> para serem utilizados no Renode	38
5.2	Criação dos arquivos <code>.repl</code> do STM32L496	39
5.2.1	Base utilizada e customizações a partir do <code>stm32f4.repl</code>	39
5.3	Organização do arquivo <code>stm32l496.repl</code> por grupos funcionais de periféricos	40
5.3.1	Núcleo de Processamento e Controle de Execução	41
5.3.2	Memória e Armazenamento Não Volátil	42
5.3.3	Gerenciamento de <i>clock</i> , energia e <i>reset</i> do sistema	44
5.3.4	Infraestrutura de Interrupções e Eventos Externos	46
5.3.5	Interfaces de Comunicação Serial	48
5.3.6	Grupo de Entrada e Saída Digital (GPIO)	50
5.3.7	Conversão Analógica e Aquisição de Dados	51
5.3.8	Temporizadores e Geração de Eventos Temporais	52
5.3.9	Acesso Direto à Memória (DMA)	53
5.3.10	Periféricos de Suporte, Diagnóstico e Segurança	54
5.3.11	Infraestrutura de Simulação e Integração com Ferramentas	55
5.4	Arquivo <code>stm32l496_kit.repl</code> : composição de placa e anexação do rádio	56

5.4.1	Instanciação do rádio no barramento SPI	56
5.4.2	Arquivo <code>stm32l496_kit.repl</code> completo	56
5.4.3	Conexões por <i>GPIO</i> (controle e interrupção)	57
5.5	Arquivos de descrição de periférico (<code>.cs</code>)	57
5.5.1	O papel dos <code>.cs</code> na simulação e utilização do STM32L496	58
5.5.2	Base utilizada de modelos STM32 já existentes	59
5.5.3	Relação entre os <code>.cs</code> implementados e o <i>firmware</i> alvo	59
5.5.4	Estrutura de um modelo de periférico em C# no Renode	60
5.5.5	Detalhamento do modelo (<code>.cs</code>)	60
5.5.6	Demais periféricos implementados (apenas visão das funções principais)	64
5.6	Extensões no modelo <code>AT86RF233.cs</code>	65
5.6.1	Implementações no <code>AT86RF233.cs</code>	65
5.7	Estimativa de consumo energético durante a simulação	66
5.8	Scripts de Execução (<code>.resc</code>) criados	67
5.9	Desenvolvimento e geração dos <i>firmwares</i> com STM32CubeIDE	69
5.9.1	Depuração com GDB integrada ao Renode	69
5.10	Fases de teste	70
5.10.1	Primeira fase de testes	70
5.10.2	Segunda fase de testes	70
5.10.3	Terceira fase de testes	71
5.10.4	Quarta fase de testes	72
6	Resultados	75
6.1	Fase 1: Validação inicial de temporização e execução básica (<i>GPIO toggle</i>)	75
6.2	Fase 2: Validação de comunicação serial via UART e SPI	76
6.3	Fase 3: Integração com o rádio AT86RF233 e validação de RX/TX no meio IEEE 802.15.4	78
6.3.1	Firmware A: validação de recepção (<i>RX</i>) com injeção manual de quadros	78
6.3.2	Firmware B: comunicação <i>ping-pong</i> entre dois nós (TX e RX)	78
6.4	Fase 4: Execução do <i>firmware</i> de produção e interceptação/decodificação de dados via UDP	80
6.4.1	Evidência de TX e RX no rádio dentro da máquina de estados	81
6.4.2	Estimativa de consumo energético durante a execução do <i>firmware</i> de produção	82
7	Discussão	84
7.1	Interpretação dos resultados por fases de teste	84
7.1.1	Fase 1: consistência de execução e temporização	84
7.1.2	Fase 2: periféricos de comunicação	85
7.1.3	Fase 3: integração com o rádio e validação de fluxo fim a fim no meio	85

7.1.4	Fase 4: execução do <i>firmware</i> real como critério de fidelidade prática	85
7.2	Observabilidade, depuração e reprodutibilidade do ambiente simulado	86
7.3	Estimativa de consumo energético: utilidade e limitações	86
7.4	Limitações do modelo criado	87
8	Conclusões	88
8.1	Trabalhos futuros	89
	Referências	91
A	Arquivo stm32l496.repl desenvolvido	94
B	Arquivo stm32l4_spi.cs desenvolvido	97

Alinhamento com a Linha de Pesquisa

Linha de Pesquisa: Sistemas Computacionais Industriais

A presente dissertação alinha-se à Linha de Pesquisa *Sistemas Computacionais Industriais* ao investigar métodos e ferramentas para o desenvolvimento, a validação e a depuração de sistemas embarcados utilizados em contextos industriais. O trabalho propõe e avalia a construção de uma plataforma de simulação com foco em fidelidade de baixo nível, capaz de executar *firmware* real sem modificações, integrando a modelagem incremental de periféricos de um microcontrolador e sua comunicação com um transceptor sem fio. Essa abordagem contribui para a engenharia de sistemas computacionais industriais ao reduzir custos e riscos associados a testes em hardware, ampliar a rastreabilidade de falhas, apoiar testes reproduzíveis e acelerar ciclos de desenvolvimento e integração. Dessa forma, os resultados obtidos fortalecem práticas e infraestruturas de verificação, integração e confiabilidade de componentes de hardware e *software* típicos de sistemas industriais.

Lista de Tabelas

5.1	Mapa de registradores do SPI (endereços calculados para SPI1 em 0x40013000).	61
5.2	Periféricos STM32L4 implementados no Renode	64
6.1	Estimativa de consumo na Fase 4 (HCLK=8 MHz, $V_{DD} = 3.0$ V, $T = 25^{\circ}\text{C}$).	83

Lista de Figuras

3.1	Seção do (.repl) nativo STM32F4 do Renode.	17
3.2	Arquivo (.repl) nativo utilizando o STM32F4 em uma <i>board</i> do Renode.	18
3.3	Exemplo de script .resc nativo do Renode.	19
3.4	Cabeçalho e estrutura da classe no arquivo nativo STM32F4_FlashController do Renode.	22
3.5	Enum dos registradores do STM32F4_FlashController do Renode.	23
3.6	Função DefineRegisters do STM32F4_FlashController do Renode.	23
4.1	Diagrama de comunicação do sistema de monitoramento.	27
4.2	Fluxograma dos estágios do <i>firmware</i> do módulo sensor.	27
4.3	Tela inicial do Monitor do Renode.	30
4.4	Trecho do <i>firmware</i> da primeira fase de testes.	32
4.5	Trecho mínimo do <i>firmware</i> da segunda fase de testes: transmissões por SPI e UART com <i>toggle</i> e atraso.	33
5.1	Periférico RCC do STM32F4 nativo do Renode.	40
5.2	Periférico SRAM e FLASH do STM32F4 nativo do Renode.	40
5.3	Descrição CPU e NVIC do STM32L4 no .repl.	41
5.4	Descrição da <i>flash</i> , <i>flash_controller</i> , SRAM e UID do STM32L4 no .repl.	43
5.5	Erro ao iniciar a simulação com uid com exatos 96 bits.	44
5.6	Descrição RCC, PWR e RTC STM32L4 no .repl.	45
5.7	Descrição EXTI do STM32L4 no .repl.	47
5.8	Descrição da UART e SPI do STM32L4 no .repl.	49
5.9	Descrição dos GPIO do STM32L4 no .repl.	50
5.10	Descrição do ADC do STM32L4 no .repl.	51
5.11	Descrição dos <i>timers</i> do STM32L4 no .repl.	52
5.12	Descrição do DMA do STM32L4 no .repl.	53
5.13	Descrição dos periféricos DBGMCU, SYSCFG, CRC e RNG do STM32L4 no .repl.	54
5.14	Utilização do SVD e <i>tags</i> do STM32L4 no .repl.	55
5.15	Trecho do arquivo <i>stm32l496_kit.repl</i> : anexação do rádio AT86RF233 e conexões por <i>GPIO</i>	57
5.16	Offsets de registradores do SPI no arquivo STM32L4_SPI.cs.	61

5.17	Estrutura típica de classe de periférico no Renode para o SPI (trecho reduzido).	62
5.18	Definição de registradores no <code>STM32L4_SPI.cs</code> (trecho reduzido). . . .	63
5.19	Recorte dos detalhes do registrador no manual de referência.	63
5.20	Exemplo de script <code>.resc</code> para inicialização da simulação e carga do <code>.elf</code>	67
5.21	Arquivo <code>.repl</code> da <i>board</i> da primeira fase de testes.	70
5.22	Adição feita ao <code>.repl</code> da <i>board</i> para a segunda fase de testes.	71
5.23	Arquivo <code>.repl</code> da <i>board</i> da terceira fase de testes.	72
5.24	Rotina adicionada ao modelo do AT86RF233 para injetar manualmente um <i>frame</i> de teste (RX).	73
5.25	Configurações adicionadas no <code>.resc</code> da quarta seção de testes.	73
5.26	Configurações do <i>script</i> python adicionadas no <code>.resc</code> da quarta seção de testes.	74
6.1	<i>Log</i> obtido na fase 1 de testes.	76
6.2	Analisador UART nativo do Renode com o resultado da transmissão. . . .	77
6.3	<i>Log</i> obtido na fase 2 de testes.	77
6.4	<i>Log</i> obtido do início da UART.	78
6.5	<i>Log</i> da saída no analisador UART após inicialização do Firmware A. . . .	79
6.6	<i>Log</i> da saída <i>ping-pong</i> no analisador UART do <i>master</i>	79
6.7	<i>Log</i> da saída <i>ping-pong</i> no analisador UART do <i>slave</i>	80
6.8	<i>Log</i> da saída do <i>hub</i> parcialmente simulado.	81
6.9	<i>Log</i> da transmissão e leitura do <i>frame</i> recebido pelo STM32 via rádio simulado.	81
6.10	Dados de consumo exibidos no Renode Monitor.	82

Lista de Abreviações

Abreviação	Descrição
ACK	Confirmação de recebimento (<i>Acknowledgment</i>)
ADC	Conversor Analógico-Digital (<i>Analog-to-Digital Converter</i>)
AF	Função alternativa (<i>Alternate Function</i>)
APB	Barramento avançado de periféricos (<i>Advanced Peripheral Bus</i>)
ARM	Arquitetura ARM (ex.: <i>Cortex-M</i>)
AT86RF233	Transceptor IEEE 802.15.4 (AT86RF233)
CD	Entrega contínua (<i>Continuous Delivery</i>)
CI	Integração contínua (<i>Continuous Integration</i>)
CMSIS-SVD	Descrição de periféricos e mapa de registradores (<i>System View Description</i>)
CNC	Controle numérico computadorizado (<i>Computerized Numerical Control</i>)
CRC	Verificação por redundância cíclica (<i>Cyclic Redundancy Check</i>)
DBGMCU	Módulo de depuração (<i>Debug MCU</i>)
DMA	Acesso direto à memória (<i>Direct Memory Access</i>)
DT	Gêmeo digital (<i>Digital Twin</i>)
DTP	Protótipo de gêmeo digital (<i>Digital Twin Prototype</i>)
ELF	Formato executável e linkável (<i>Executable and Linkable Format</i>)
EXTI	Interrupção externa (<i>External Interrupt</i>)
GCC	Compilador GNU (<i>GNU Compiler Collection</i>)
GDB	Depurador GNU (<i>GNU Debugger</i>)
GPIO	Entrada/saída de uso geral (<i>General-Purpose Input/Output</i>)
HCLK	Clock do barramento AHB (HCLK)
IDE	Ambiente de desenvolvimento integrado (<i>Integrated Development Environment</i>)
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos (<i>Institute of Electrical and Electronics Engineers</i>)
IoT	Internet das Coisas (<i>Internet of Things</i>)
IRQ	Requisição de interrupção (<i>Interrupt Request</i>)
LL	Camada <i>Low-Layer</i> (STM32)
MAC	Controle de acesso ao meio (<i>Medium Access Control</i>)
MCU	Unidade microcontroladora (<i>Microcontroller Unit</i>)
NVIC	Controlador de interrupções (<i>Nested Vectored Interrupt Controller</i>)

PAN	Rede de área pessoal (<i>Personal Area Network</i>)
PWR	Controle de energia (PWR)
RCC	Controle de reset e clocks (<i>Reset and Clock Control</i>)
RNG	Gerador de números aleatórios (<i>Random Number Generator</i>)
RTC	Relógio de tempo real (<i>Real-Time Clock</i>)
RX	Recepção (<i>Receive</i>)
SPI	Interface serial de periféricos (<i>Serial Peripheral Interface</i>)
SysTick	Temporizador do sistema (<i>System Tick</i>)
TIM6	Temporizador básico 6 (<i>Timer 6</i>)
TIM7	Temporizador básico 7 (<i>Timer 7</i>)
TRGO	Saída de gatilho do temporizador (<i>Trigger Output</i>)
TX	Transmissão (<i>Transmit</i>)
UART	Interface serial assíncrona (<i>Universal Asynchronous Receiver/-Transmitter</i>)
UDP	Protocolo de datagramas do usuário (<i>User Datagram Protocol</i>)
UID	Identificador único do dispositivo (<i>Unique ID</i>)
USART	Interface serial síncrona/assíncrona (<i>Universal Synchronous/Asynchronous Receiver/Transmitter</i>)
VDD	Tensão de alimentação (VDD)
V&V	Verificação e validação (<i>Verification and Validation</i>)

Capítulo 1

Introdução

Digital twins (gêmeos digitais) têm se tornado cada vez mais relevantes no desenvolvimento e na validação de sistemas embarcados e aplicações de *Internet of Things* (IoT), pois permitem replicar sistemas físicos em um ambiente virtual e viabilizar testes mais seguros e escaláveis ainda nas fases iniciais de desenvolvimento (Javaid et al., 2023). Contudo, a maior parte das implementações de gêmeos digitais concentra-se em camadas de abstração de alto nível como integração em nuvem, visualização 3D ou análises em nível de sistema e, frequentemente, negligencia a simulação precisa do comportamento de baixo nível do hardware (He et al., 2020). Essa lacuna é particularmente crítica para desenvolvedores de *firmware* em plataformas baseadas em microcontroladores, que demandam modelos com elevada fidelidade para testar e validar código embarcado de forma confiável. No contexto de controladores de baixo consumo amplamente empregados em IoT, observa-se, por exemplo, a ausência de uma solução acessível de gêmeo digital em nível de *firmware* para a família STM32L4, o que motiva o escopo deste trabalho.

No contexto de sistemas embarcados, um gêmeo digital pode ser entendido como uma representação do hardware real executando em um ambiente virtual, capaz de simular o microcontrolador e seus periféricos em tempo de execução. Diferentemente de modelos de alto nível usados em sistemas em nuvem ou em automação industrial, gêmeos digitais de baixo nível devem reproduzir, com precisão, o mapa de registradores do controlador, a lógica dos periféricos e o comportamento de interrupções, de modo a refletir como o *software* embarcado interage com o hardware real. Entre os principais benefícios, destacam-se a possibilidade de testar *firmware* sem a necessidade de protótipos físicos e a redução do tempo de desenvolvimento em ambientes de IoT com restrições de recursos (Kellermüller et al., 2022). Ainda assim, a efetividade do gêmeo digital depende diretamente do grau de fidelidade com que o comportamento de baixo nível do hardware é representado (do Amaral et al., 2023).

Para atender a esse objetivo, este projeto utiliza o Renode como base para implementar um gêmeo digital em nível de *firmware* (Renode Project, 2026f). O Renode é uma estrutura de simulação de código aberto capaz de executar binários embarcados sem modificações e de simular periféricos de microcontroladores em

tempo de execução, oferecendo recursos de depuração, *scripting* e automação de testes repetíveis. Apesar dessas vantagens, não havia suporte nativo no Renode para a família STM32L4, o que exigiu a implementação manual de periféricos essenciais. Neste estudo de caso, o microcontrolador STM32L496 é virtualmente recriado e integrado a um transceptor IEEE 802.15.4 AT86RF233, possibilitando a execução de um *firmware* sem alterações, a simulação do comportamento de hardware com precisão em nível de registradores e a comunicação completa entre o controlador e o módulo de rádio.

Além da ausência de suporte nativo para a família STM32L4, observou-se uma barreira prática para a implementação de novos microcontroladores no Renode: há relativa escassez de exemplos completos e de implementações de referência produzidas por outros desenvolvedores e pesquisadores. Como consequência, a construção do modelo do STM32L496 demandou a consulta direta ao código-fonte do próprio Renode, tanto para identificar padrões de modelagem adotados em plataformas existentes quanto para compreender detalhes de infraestrutura e integração entre periféricos, em conjunto com a documentação oficial do simulador.

Assim, este trabalho apresenta o projeto e a implementação de um gêmeo digital para um sistema sem fio baseado no STM32L496, combinando a simulação do microcontrolador com a simulação do rádio. Os periféricos necessários ao funcionamento do STM32L4 são implementados no Renode, e o modelo do AT86RF233 é estendido para suportar interação em tempo real com o *firmware*, incluindo operações via SPI e sinalização de interrupções. A plataforma resultante permite executar código embarcado integralmente em simulação, viabilizando testes de comunicação sem fio, depuração e *logging* sem dependência de hardware físico. Como resultado, o gêmeo digital construído estabelece uma base reutilizável e extensível para validação, em nível de *firmware*, de sistemas IoT baseados na família STM32L4.

1.1 Proposta e Objetivos

1.1.1 Proposta do trabalho

Este trabalho se insere no contexto de digital twins aplicados a sistemas embarcados, nos quais uma representação virtual do hardware permite executar, depurar e validar firmwares com maior reprodutibilidade, rastreabilidade e capacidade de automação de testes. Em projetos de firmware, a dependência de placas físicas, a instrumentação limitada e a variabilidade do ambiente (revisões de hardware, disponibilidade de bancada e periféricos) frequentemente elevam custos e atrasam ciclos de desenvolvimento, especialmente nas etapas de integração e depuração.

A proposta desta dissertação é implementar um sistema de simulação de firmwares no Renode, com foco em viabilizar a execução e a depuração de aplicações embarcadas por meio de um digital twin funcional. A abordagem baseia-se na modelagem de periféricos em nível de registradores, eventos e interrupções, permitindo que o firmware interaja com o ambiente simulado de forma compatível com o comportamento esperado no hardware real. Além de oferecer um ambiente controlado

para depuração, a proposta busca apoiar testes repetíveis e a evolução incremental do gêmeo digital conforme os requisitos do firmware.

Adicionalmente, o ambiente utilizado para estimar o consumo energético do controlador em diferentes cenários de operação, fornecendo subsídios para estudos de viabilidade de alimentação por fontes renováveis e maior autonomia energética.

1.1.2 Objetivo geral

Implementar um sistema de simulação de firmwares, no Renode, para controladores da família STM32L4, de modo a permitir a execução, a depuração e a validação funcional de aplicações embarcadas em um ambiente reprodutível baseado em *digital twin*.

1.1.3 Objetivos específicos

1. Definir os requisitos funcionais do *digital twin*, identificando as interfaces e periféricos necessários para suportar a execução dos firmwares-alvo (inicialização, clock, interrupções, comunicação e temporização).
2. Modelar e integrar, no Renode, os periféricos essenciais ao boot e ao funcionamento do firmware, com ênfase em comportamento observável por meio de registradores, flags, interrupções e eventos.
3. Implementar periféricos customizados adicionais no Renode (em C#), ampliando a cobertura necessária para simular corretamente as rotinas do firmware STM32, incluindo RCC, PWR, DBGMCU, SYSCFG, EXTI, DMA, RTC, UART, SPI, ADC, TIMERS, RNG e controlador de Flash. e demais periféricos requeridos pelo firmware.
4. Construir e padronizar a plataforma de simulação (arquivos de descrição e scripts de execução), garantindo inicialização consistente e eficiente, carregamento de binários e configuração de mecanismos de observabilidade, como logs e rastreamento de transações em barramentos.
5. Simular no *digital twin* criado o *firmware* de produção utilizado no projeto real do laboratório LMEST.
6. Avaliar limitações e o nível de fidelidade alcançado, identificando divergências relevantes entre simulação e hardware e delimitando o escopo do digital twin.
7. Estimar o consumo energético do STM32L496 no ambiente simulado, considerando diferentes modos de operação e combinações de periféricos habilitados, de modo a apoiar análises de viabilidade de soluções com maior autonomia energética e potencial uso de fontes renováveis.

1.2 Contribuições

As principais contribuições deste trabalho são:

- Implementação de uma plataforma de simulação no Renode para a família STM32L4 (com instanciação e validação em um alvo STM32L496), incluindo descrição de memória (SRAM/Flash), mapeamento de barramento e integração com NVIC e linhas de interrupção.
- Implementação, em C#, de modelos customizados de periféricos STM32L4 no Renode, abrangendo: RCC, PWR, DBGMCU, SYSCFG, EXTI, DMA, RTC, UART, SPI, ADC, temporizadores, RNG e controlador de Flash.
- Integração funcional entre periféricos simulados, incluindo roteamento de interrupções (por exemplo, agregação de linhas EXTI e encaminhamento para o NVIC), sinalização de alarmes do RTC, e interações de aquisição de dados envolvendo ADC e requisições de DMA.
- Construção do artefato de plataforma no Renode (arquivo `.repl`), consolidando a composição do sistema, endereçamento dos periféricos, conexões de IRQ/DMA e parâmetros de frequência/temporização necessários para executar firmwares-alvo.

Os conteúdos produzidos neste trabalho estão disponíveis em repositório (Presti, 2026).

1.3 Organização do Trabalho

O Capítulo 1 contextualiza o problema e a motivação da pesquisa, apresenta a proposta e os objetivos e delimita o escopo do gêmeo digital em nível de *firmware* para a família STM32L4. Em seguida, o Capítulo 2 reúne a revisão bibliográfica, discutindo conceitos e classificações de gêmeos digitais aplicados a sistemas embarcados, além de situar o Renode como infraestrutura de simulação e *rehosting* para execução de binários embarcados. O Capítulo 3 faz uma introdução sobre a ferramenta Renode, suas características e os tipos de arquivos que compõem os *digital twins* implementados. O Capítulo 4 descreve o método de desenvolvimento incremental orientado a testes, detalhando a construção da plataforma no Renode (arquivos `.repl` e `.resc`), a implementação dos modelos de periféricos em C# (arquivos `.cs`), a integração com o rádio AT86RF233 e os critérios de validação e reprodutibilidade adotados, incluindo a instrumentação para estimativa de consumo energético. Na sequência, o Capítulo 5 apresenta o desenvolvimento da plataforma de simulação e dos modelos de periféricos, descrevendo as decisões de modelagem, a integração entre subsistemas (interrupções, temporização e DMA) e a configuração do ambiente para suportar a execução do *firmware* alvo sem modificações. O Capítulo 6 apresenta os resultados experimentais obtidos nos testes unitários e de integração, bem como na

execução do *firmware* de produção, organizando as principais evidências observáveis (logs, interrupções, comunicação por barramentos e *frames*) e os indicadores utilizados para verificação do comportamento esperado. Por sua vez, o Capítulo 7 discute de forma crítica os resultados, relacionando as evidências coletadas na simulação com as decisões técnicas adotadas em cada etapa do desenvolvimento incremental, e analisando as limitações e o nível de fidelidade alcançado. O Capítulo 8 resume as principais conclusões do trabalho, retoma a motivação e verifica o atendimento dos objetivos propostos, com ênfase nas contribuições técnicas e na validação por meio da execução do *firmware* de produção. Por fim, são apresentadas perspectivas de trabalhos futuros, incluindo o aprimoramento da fidelidade de periféricos e modos de baixo consumo, a ampliação do conjunto de testes automatizados e a extensão da abordagem para novos cenários e microcontroladores.

Capítulo 2

Revisão Bibliográfica

2.1 Gêmeos digitais (*Digital Twins*)

A popularização do conceito de *digital twin* (gêmeo digital) está diretamente ligada ao avanço de sistemas conectados e instrumentados: sensores embarcados em ativos físicos, redes de comunicação, computação em nuvem e técnicas de inteligência artificial ampliaram a capacidade de capturar dados do mundo real e usá-los para construir “réplicas” digitais úteis para engenharia e operação (Barricelli et al., 2019). Apesar do amplo uso do termo na indústria e na literatura, ainda existe heterogeneidade conceitual, com diferentes trabalhos atribuindo significados e níveis de abrangência distintos ao que seria, de fato, um gêmeo digital (Kritzinger et al., 2018). De forma sintética e operacional, um gêmeo digital pode ser entendido como uma representação virtual de um sistema físico que é continuamente atualizada (total ou parcialmente) por dados do mundo real, de modo a apoiar análise e tomada de decisão ao longo do ciclo de vida do ativo. Nessa perspectiva, o gêmeo digital existe simultaneamente ao lado do sistema físico, mantendo um acoplamento informacional que permite, por exemplo, monitorar, prever comportamentos e avaliar alternativas de intervenção (Rasheed et al., 2020). A literatura também registra termos correlatos (como, *computational megamodel*, *digital avatar* e *digital clone*), o que reforça a necessidade de delimitar claramente o conceito adotado nesta pesquisa (Rasheed et al., 2020).

2.1.1 Evolução do conceito e distinções fundamentais

Historicamente, ideias precursoras associam-se a práticas de espelhamento de sistemas para suporte operacional (por exemplo, o uso de réplicas para acompanhar condições de missões de foguetes), mas a formulação moderna do *digital twin* é frequentemente vinculada ao modelo de três elementos proposto por Grieves: o sistema físico, o sistema virtual e o elo de conexão que integra ambos, permitindo fluxos de informação entre os domínios físico e digital (Barricelli et al., 2019). Uma contribuição importante para reduzir ambiguidades é a distinção entre (a) *digital model*, (b) *digital shadow* e (c) *digital twin*, baseada no grau e na direção do acoplamento de da-

dos entre o físico e o virtual. No *digital model*, não existe troca automática de dados; no *digital shadow*, há atualização automática do modelo virtual a partir do físico, porém sem retroalimentação automatizada no sentido inverso; já no *digital twin* há fluxo automatizado bidirecional, permitindo que mudanças no sistema virtual também se reflitam no físico (Kritzinger et al., 2018). Essa gradação é especialmente útil para evitar confundir soluções que são apenas modelos digitais desconectados (ou parcialmente conectados) como se fossem gêmeos digitais completos (Kritzinger et al., 2018).

2.1.2 Como gêmeos digitais funcionam: componentes e ciclo de retroalimentação

A característica central que diferencia um gêmeo digital de uma simulação tradicional é a conexão contínua (ou suficientemente frequente) com o ativo físico. Em aplicações típicas de gêmeos digitais, o sistema físico é instrumentado por sensores e recursos de comunicação que viabilizam uma ligação direta (*seamless*) entre os domínios físico e digital, sustentando a troca contínua de dados (Barricelli et al., 2019). Com isso, o gêmeo digital recebe não apenas dados dinâmicos do ativo físico, mas também dados de contexto (por exemplo, do ambiente), e pode devolver ao sistema físico informações derivadas (predições e prescrições), bem como disponibilizar resultados para especialistas e para outros gêmeos digitais (Barricelli et al., 2019). Do ponto de vista de arquitetura, uma decomposição do sistema pode se dar por, *Physical Space* (PS), *Virtual Space* (VS), canais de conexão entre PS-VS e um repositório de dados (*data repository*) que armazena dados correntes e históricos e serviços que operam sobre esses dados e modelos (por exemplo, diagnóstico, predição, otimização e suporte à decisão) (Barricelli et al., 2019). Além do canal PS-VS, são enfatizados também processos de comunicação: entre gêmeos digitais (DT-DT) e entre gêmeos digitais e usuários especialistas (*domain experts*), o que amplia a utilidade do conceito em ecossistemas complexos (Barricelli et al., 2019). Funcionalmente, um gêmeo digital pode ser entendido como um arranjo em *loop* entre o ativo físico e sua parte virtual: dados coletados por sensores alimentam continuamente a representação digital, que combina modelos e *scripts* para estimar estados, identificar anomalias, antecipar falhas e avaliar cenários. A partir dessas análises, o gêmeo digital produz informações para suporte à decisão e, quando aplicável, pode orientar intervenções no sistema físico por mecanismos de *feedback* (por exemplo, monitoramento e controle, ajustes operacionais e planejamento de manutenção). Assim, a efetividade do gêmeo digital depende da adequação e confiabilidade dos modelos, bem como da qualidade e disponibilidade dos dados e da capacidade de convertê-los em decisões ao longo da operação (Rasheed et al., 2020).

2.1.3 Para que servem: aplicações e objetivos recorrentes

Do ponto de vista de suas aplicações e benefícios ao longo do ciclo de vida do sistema, gêmeos digitais são empregados para diferentes finalidades, incluindo:

monitoramento e diagnóstico; apoio ao controle; análise e otimização de desempenho; planejamento e suporte à operação e manutenção; além de *what-if analysis* e avaliação de cenários para lidar com riscos e incertezas. Nesse contexto, a literatura também menciona a noção de *digital siblings*, isto é, variações do gêmeo digital utilizadas para explorar alternativas e cenários de decisão (Rasheed et al., 2020). Adicionalmente, gêmeos digitais podem apoiar atividades de engenharia e desenvolvimento, ao viabilizar validação e experimentação em ambiente virtual e a integração de estratégias de controle e software embarcado aos modelos do sistema, reduzindo iterações dependentes de protótipos físicos (Rasheed et al., 2020). Na prática, essas aplicações se conectam a objetivos clássicos como predição e suporte à decisão, por exemplo ao antecipar falhas, estimar comportamentos futuros, avaliar impactos de mudanças e orientar escolhas de intervenção (Rasheed et al., 2020). No escopo desta dissertação, essa perspectiva é explorada especificamente para acelerar o desenvolvimento de *firmwares* de microcontroladores, por meio de um gêmeo digital em nível de execução que permita testar e depurar o software embarcado de forma mais rápida, reproduzível e antecipada em relação à disponibilidade do hardware.

2.1.4 Relação com Sistemas Ciber-Físicos

A noção de gêmeos digitais é frequentemente discutida no contexto de Sistemas Ciber-Físicos (CPS), entendidos como sistemas que integram processos físicos e computacionais, com capacidade de monitorar e controlar o processo físico por meio de *feedback loops*. Nesse enquadramento, gêmeos digitais podem ser interpretados como modelos virtuais de alta fidelidade que representam CPS e seus correspondentes físicos, contribuindo para atividades como projeto, monitoramento, manutenção e otimização em ambientes industriais (Parnianifard et al., 2022).

2.1.5 Síntese dos elementos mínimos

Em convergência, os trabalhos revisados indicam que uma solução tende a se aproximar de um *digital twin* quando reúne, ao menos (Barricelli et al., 2019):

- um ativo/processo físico observável (PS) instrumentado para coleta de dados;
- um artefato digital (VS) capaz de representar estados e comportamentos relevantes;
- um mecanismo de sincronização baseado em dados (idealmente contínuo e automatizado);
- serviços analíticos sobre dados/modelos (diagnóstico, predição, otimização, suporte à decisão);
- alguma forma de retroalimentação do digital ao físico (ação/recomendação/-controle), caracterizando bidirecionalidade quando automatizada.

2.2 Utilização de gêmeos digitais em sistemas embarcados

A adoção de gêmeos digitais em sistemas embarcados tem crescido à medida que aplicações ciberfísicas e de Internet das Coisas (IoT) ampliam a presença de software em dispositivos com recursos restritos, conectividade intermitente e forte acoplamento entre *firmware* e hardware. Nesse tipo de utilização, o “objeto gêmeo” deixa de ser apenas um ativo industrial ou um processo em alto nível e passa a incluir, de forma central, o comportamento do equipamento real: seu *firmware*, seus periféricos, suas interrupções, seus sensores/atuadores e os efeitos do ambiente físico no tempo de execução. Assim, a utilidade do gêmeo digital em embarcados depende diretamente do grau de fidelidade com que o modelo representa as interfaces e as dinâmicas do dispositivo, bem como da capacidade de sincronizar dados e comandos com o sistema real ao longo do ciclo de vida.

No contexto IoT, a literatura enfatiza que gêmeos digitais devem ser compreendidos como um arranjo técnico que integra modelos (do dispositivo, do ambiente e/ou do processo), dados coletados do mundo físico e serviços que exploram essa integração para monitoramento, diagnóstico, simulação de cenários e controle. Revisões sobre gêmeos digitais no contexto IoT ressaltam que a implementação costuma adotar uma organização em camadas, em que a *device layer* concentra os dispositivos e a coleta de dados, enquanto camadas superiores realizam a integração, análise e a oferta de serviços associados ao gêmeo (Minerva et al., 2020). Nessa perspectiva, a conectividade entre o dispositivo e esses recursos (borda/nuvem) sustenta o fluxo de dados necessário para manter o gêmeo coerente ao longo do ciclo de vida, viabilizando monitoramento e suporte a decisões (Minerva et al., 2020).

Ao transportar esses princípios para sistemas embarcados, torna-se relevante explicitar o nível de abstração adotado. Em um extremo, podem existir gêmeos digitais de “alto nível”, focados em métricas e estados derivados (por exemplo, estimativas de energia, estados de um processo e visualizações), em que o *firmware* é apenas fonte de dados. No outro, há gêmeos digitais de “baixo nível”, que visam reproduzir a interação do *firmware* com o hardware, de modo que o próprio binário (ou uma versão equivalente) possa executar no gêmeo, acionando periféricos, temporizadores e rotinas de interrupção. Essa distinção é crítica para embarcados, pois muitos defeitos e regressões surgem justamente no acoplamento *firmware-hardware* (temporalidade, condições de corrida, bordas de protocolo, inicialização de periféricos), que tende a escapar de modelos excessivamente abstratos.

Nesse cenário, duas estratégias técnicas aparecem com frequência. A primeira é a emulação/*rehosting* de *firmware*, na qual o gêmeo digital executa a imagem de *firmware* (ou parte dela) em um ambiente emulado, reproduzindo CPU e subsistemas relevantes e permitindo instrumentação profunda para observar e controlar a execução. Essa abordagem tem sido explorada também sob a ótica de segurança, pois possibilita analisar comportamento e telemetria do *endpoint* sem depender do dispositivo físico, além de facilitar a automação de verificações e respostas a incidentes no

contexto IoT. (Cheng et al., 2024) A segunda estratégia é a modelagem/simulação orientada a componentes, em que sensores, atuadores e interfaces (barramentos, registradores e protocolos) são representados por modelos executáveis que reagem aos estímulos do *firmware*. Em muitos casos práticos, as duas estratégias se combinam: a execução do *firmware* exige algum nível de emulação do processador, enquanto o comportamento do “mundo” (sensores/atuadores e rede) é fornecido por modelos que podem variar em fidelidade e custo computacional.

Um eixo particularmente relevante para embarcados é a utilização de gêmeos digitais como infraestrutura de verificação e validação (V&V) e como suporte a práticas contemporâneas de engenharia de software, como integração contínua (CI) e entrega contínua (CD). A dificuldade de escalar testes dependentes de hardware físico (*hardware-in-the-loop*) motiva soluções que permitam repetir cenários de integração de forma determinística, reproduzível e automatizável. Nessa direção, a noção de *Digital Twin Prototype* (DTP) tem sido proposta para capturar artefatos de gêmeo que são adequados a estágios de desenvolvimento e teste, quando o sistema físico é indisponível, caro ou instável. DTPs visam viabilizar testes de integração e regressão em *pipelines*, mantendo um compromisso pragmático entre fidelidade e viabilidade de execução em escala. (Barbie et al., 2021; Barbie e Hasselbring, 2024)

A formalização do conceito de DTP também contribui para organizar a literatura: em vez de exigir, desde o início, um gêmeo completo e continuamente sincronizado com o mundo físico. Trabalhos recentes distinguem protótipos que suportam atividades de engenharia (testes automatizados, análise de impacto e validação de requisitos) e podem evoluir para gêmeos mais abrangentes conforme amadurece o ciclo de vida do sistema. Essa perspectiva é especialmente útil em sistemas embarcados, onde a diversidade de plataformas e a necessidade de modelar periféricos e tempos de execução tornam custoso a construção de um gêmeo completo já nas fases iniciais. (Barbie e Hasselbring, 2024)

Aplicações em domínios específicos ilustram esse uso pragmático. Em sistemas de agricultura inteligente, por exemplo, DTPs podem representar componentes e integrações típicas (dispositivos de campo, gateways, serviços e fluxos de dados) para executar testes de integração automatizados em cenários representativos, reduzindo o custo de montar bancadas físicas e permitindo variações controladas do ambiente. A literatura aponta que esse tipo de protótipo permite testar interações ponta a ponta e avaliar respostas do sistema sob condições diversas, reforçando a ideia de gêmeos digitais como instrumentos de engenharia e não apenas de monitoramento operacional. (Barbie et al., 2024)

Além de V&V funcional, gêmeos digitais em embarcados têm sido empregados para verificação contínua de requisitos de segurança (no sentido de *safety* e também de *security*). No setor automotivo, o conceito de *cyber digital twin* aparece como base para verificar requisitos de segurança de forma contínua, apoiando a identificação de desvios e a avaliação sistemática de conformidade do sistema frente a ameaças e critérios definidos. (Franco da Silva et al., 2021) Em IoT, a integração entre emulação de *firmware* e rotinas de detecção e resposta também reforça o papel do gêmeo como ambiente seguro para inspeção e experimentação, com impacto direto

na capacidade de triagem e automação de respostas sem expor o dispositivo real a riscos ou indisponibilidade. (Cheng et al., 2024)

Em termos de requisitos técnicos, a literatura sobre DT aplicado ao desenvolvimento de software embarcado destaca necessidades específicas: integração com ferramentas e processos já adotados (como em *pipelines* de *build* e teste), mecanismos de observabilidade que não dependam do hardware físico, reprodutibilidade de cenários e capacidade de evoluir modelos com baixo custo de manutenção. Esse conjunto de requisitos é particularmente sensível em pequenas e médias empresas, onde restrições de tempo e orçamento tornam decisiva a viabilidade de adoção, e onde abordagens graduais (como DTPs) podem ser mais realistas do que a tentativa de construir um gêmeo completo desde o início. (Barbie e Hasselbring, 2023)

A questão da fidelidade do gêmeo em embarcados também se evidencia em trabalhos que relacionam desempenho do gêmeo à representação do *firmware* e de sua execução. Em contextos industriais como controle numérico computadorizado (CNC), discutir a simulação do *firmware* como componente do gêmeo aponta para um entendimento mais “computacional” do DT, no qual o nível de detalhe do software embarcado influencia diretamente a capacidade do gêmeo de produzir resultados úteis e confiáveis. (Kutscher et al., 2020) Ainda que os domínios variem, a implicação é semelhante em IoT e sistemas microcontrolados: quanto maior o acoplamento entre *firmware* e dinâmica física, maior a necessidade de representar interfaces de baixo nível e efeitos temporais para que o gêmeo suporte análise, diagnóstico e teste com confiança.

Por fim, a literatura converge em desafios recorrentes, o custo de modelar e manter periféricos, protocolos e ambientes físicos, o compromisso entre fidelidade e custo computacional, a necessidade de padronização conceitual e de artefatos reutilizáveis e a validação do gêmeo frente ao sistema real, sobretudo em aspectos temporais e de desempenho. Uma síntese produtiva para a revisão em embarcados é reconhecer que gêmeos digitais podem assumir múltiplas “formas” ao longo do ciclo de vida: de protótipos voltados a CI e testes automatizados até gêmeos mais completos para e verificação contínua. Nesse sentido, DTPs e abordagens de emulação/modelagem em camadas oferecem um caminho pragmático para inserir gêmeos digitais no desenvolvimento e na manutenção de sistemas embarcados, preservando o valor central do DT: apoiar decisões e atividades de engenharia por meio da integração consistente entre modelo e dados do sistema físico. (Barbie et al., 2021; Barbie e Hasselbring, 2024; Minerva et al., 2020)

Capítulo 3

Renode

3.1 Sobre o Renode

Renode é um *framework open-source* de simulação que permite desenvolver, depurar e testar sistemas embarcados e IoT em ambientes virtuais, inclusive cenários com múltiplos nós, com foco em repetibilidade e escalabilidade (Renode Project, 2026f,g,h). Em vez de depender exclusivamente de protótipos físicos, o Renode executa binários equivalentes aos carregados no hardware real e fornece um ambiente controlável para inspeção de estado, integração com depuradores e automação de testes (Renode Project, 2026h,b).

Do ponto de vista de funcionamento, o Renode organiza a simulação em máquinas virtuais compostas por CPU, memória, barramentos e periféricos, descritos por arquivos de plataforma (.repl) e scripts de inicialização/execução (.resc), controlados via interface de linha de comando (Monitor) (Renode Project, 2026b). A modelagem de periféricos pode ser implementada em diferentes níveis, incluindo modelos completos em C# com definição de registradores, *bitfields* e efeitos colaterais. (Renode Project, 2026c).

O Renode disponibiliza modelos nativos para diferentes arquiteturas e famílias de processadores, incluindo núcleos da linha ARM Cortex-M. Nesta pesquisa, foi utilizada a implementação nativa do Cortex-M4, o que permite executar binários compilados para essa arquitetura diretamente no simulador, sem a necessidade de reimplementar o núcleo do processador. De modo geral, esses modelos nativos fornecem a base de execução do sistema, cobrindo o comportamento essencial da CPU (execução do conjunto de instruções e estados internos) e sua integração com a infraestrutura mínima da máquina simulada (como o acesso ao mapa de memória e mecanismos de exceção/interrupção). Com isso, o esforço de desenvolvimento se concentrou nos elementos específicos da plataforma alvo. (Renode Project, 2026g).

Uma característica central para teste e validação é o uso de tempo virtual determinístico, que permite sincronizar componentes e reproduzir cenários com as mesmas condições em múltiplas execuções, favorecendo depuração e regressão automatizada (Renode Project, 2026e,g). O Renode também se integra a fluxos de teste automatizados (por exemplo, via scripts Python) e pode ser incorporado em

pipelines de integração contínua para executar validações de *firmware* em hardware virtual (Renode Project, 2026d; Antmicro, 2024).

No contexto de *digital twins* para sistemas embarcados, o Renode pode atuar como o núcleo do “gêmeo digital em nível de *firmware*”, pois viabiliza a execução do *firmware* real contra uma réplica executável do microcontrolador e de seus periféricos, mantendo interfaces de hardware (registradores, interrupções e barramentos) e suportando extensões por integração com componentes externos. Em estudos aplicados, o Renode tem sido explorado como base para simulação e validação de sistemas embarcados em diferentes domínios, reforçando sua adequação a estratégias *software-in-the-loop* e verificação antecipada (Speiser et al., 2024).

Além de servir como núcleo de um gêmeo digital em nível de *firmware*, o Renode se destaca por permitir que a plataforma virtual seja descrita e automatizada por artefatos textuais (arquivos `.repl` e `.resc`) e por possibilitar a extensão do ambiente com novos modelos e configurações de hardware. Em especial, a descrição da plataforma (`.repl`) explicita CPU e periféricos e suas conexões, enquanto scripts (`.resc`) suportam a automação do fluxo de execução (criação da máquina, carregamento da plataforma e imagens binárias e habilitação de *logging*), favorecendo reprodutibilidade e integração com rotinas de teste e depuração (Speiser et al., 2024; Renode Project, 2026h).

Mesmo com os recursos disponíveis no Renode, é comum que a distribuição principal não inclua, de forma nativa, a descrição de *hardware* requerida para um *firmware* real. No caso deste trabalho, o Renode não oferecia no momento do desenvolvimento, suporte completo para uso do controlador que se pretende simular (família STM32 do projeto), o que exigiu a construção manual da plataforma e a implementação/compatibilização de modelos de periféricos. Isso inclui definir o mapeamento de memória, instanciação de blocos no *sysbus*, implementação de periféricos e ajuste de conexões para refletir o comportamento esperado pelo *firmware* (Speiser et al., 2024).

Dessa forma, o propósito deste trabalho não se limita a utilizar o Renode como ambiente de execução, mas a investigar e demonstrar a sua capacidade de expansão como infraestrutura para gêmeos digitais de *firmware*, estendendo o *framework* com modelos adicionais necessários ao microcontrolador alvo e analisando de modo sistemático, a capacidade de adicionar novos controladores da família STM32. Essa perspectiva é coerente com a própria evolução do Renode, que frequentemente incorpora novas plataformas STM32 a partir de elementos reutilizáveis e da criação de periféricos específicos, além de evidenciar a demanda recorrente por suporte a variantes não cobertas integralmente pela distribuição.

3.2 Componentes do Renode

Nesta pesquisa, a implementação do gêmeo digital do STM32L496 no Renode se baseou em um conjunto de artefatos que estruturam a plataforma simulada, codificam o comportamento dos periféricos e garantem a reprodutibilidade dos ex-

perimentos. Assim, apresenta-se os componentes/arquivos que precisaram ser implementados no decorrer do desenvolvimento, juntamente com a função de cada um no ambiente de simulação. São descritos os componentes do Renode empregados no trabalho, com suas respectivas características e funções.

3.2.1 Arquivos de descrição de plataforma

Arquivos de descrição de plataforma do Renode (com extensão `.repl`), usados para a descrição do microcontrolador (MCU) e para a descrição de uma placa que utiliza o novo microcontrolador, de forma similar a um projeto de *hardware* real, com ligações de componentes externos nos respectivos I/Os do microcontrolador.

No Renode, um arquivo `.repl` específico de um MCU descreve a “montagem” da plataforma simulada no nível de sistema, definindo como os blocos do microcontrolador são instanciados e conectados dentro do ambiente de emulação. Em geral, ele declara o mapa de memória e os endereços-base de cada periférico, associa cada região a um modelo correspondente (implementado em `.cs`), configura parâmetros relevantes (por exemplo, frequências de clock, linhas de interrupção, endereços e tamanhos de memória) e explicita as conexões de barramento e ligações lógicas entre componentes. Assim, o `.repl` do MCU funciona como uma especificação estrutural do microprocessador emulado, garantindo que acessos a registradores e eventos (como IRQs) ocorram nas posições e interconexões esperadas pelo *firmware*.

Já um arquivo `.repl` específico de uma *board* que utiliza um determinado MCU eleva essa descrição para o nível da placa, instanciando o microcontrolador e adicionando os componentes externos que compõem o sistema (por exemplo, *cristais*, memórias externas, sensores, rádios, *flash*, *EEPROM*, *LEDs*, botões e conectores). Esse `.repl` estabelece como esses dispositivos se ligam aos periféricos do MCU (por exemplo, quais pinos de *GPIO* controlam *LEDs*, qual *UART* está conectada a um console, qual *SPI* se comunica com um rádio e quais linhas de interrupção são utilizadas), além de definir ajustes específicos da placa, como endereços, *chip select* e topologia de barramentos. Dessa forma, o `.repl` da placa reproduz a integração entre MCU e *hardware* externo.

3.2.2 Arquivos de descrição de periféricos

Arquivos de implementação de periféricos específicos (com extensão `.cs`): arquivos `.cs`, associados a um microcontrolador específico ou genéricos, implementam, em `C#`, o modelo comportamental (modelo de simulação) dos periféricos e blocos internos do MCU. Em vez de serem apenas descrições estáticas, eles codificam como cada periférico deve reagir a leituras/escritas de registradores, gerar interrupções, atualizar *flags* e estados internos, interagir com barramentos e com o mapeamento de memória e, quando aplicável, sincronizar eventos no tempo simulado (por exemplo, *timers*, *UART*, *SPI*, *ADC*, *DMA* e *RCC*).

3.2.3 Arquivos de descrição de simulação

Arquivos de *scripts* do Renode (com extensão `.resc`) são utilizados para controlar a reprodutibilidade e a agilidade dos testes realizados durante o trabalho: um arquivo `.resc` organiza e automatiza a execução de um experimento de simulação, atuando como um roteiro para configurar e operar o ambiente. Tipicamente, ele carrega a plataforma descrita em arquivos `.repl`, define parâmetros de execução (por exemplo, frequência do sistema, uso e nível de *logging*), prepara a memória e os periféricos conforme necessário e carrega os binários do software a ser testado (como *ELF* ou *BIN*) no endereço apropriado. Além disso, o `.resc` pode estabelecer pontos de controle, como iniciar/parar a simulação, observar registradores e periféricos, redirecionar consoles seriais, conectar analisadores virtuais e executar sequências repetíveis de comandos para validação e depuração. Dessa forma, o `.resc` concentra a reprodutibilidade do cenário experimental, permitindo que a mesma configuração e os mesmos passos de teste sejam executados de maneira consistente e rápida entre diferentes execuções e ambientes.

3.2.4 Arquivos executáveis `.ELF`

Arquivos executáveis `.ELF` não são necessários para descrever os MCUs ou placas, porém são necessários para serem carregados nos conjuntos de *hardware* simulados e serem testados. Neste trabalho eles foram gerados no STM32CubeIDE e são os artefatos de saída do processo de compilação e *linkagem* do projeto, contendo o código de máquina do *firmware* para a arquitetura do microcontrolador (ARM Cortex-M), bem como informações estruturadas necessárias para o carregamento e a depuração. Diferentemente de um binário “cru” (`.bin`), o `.ELF` inclui metadados como seções de memória (`.text`, `.data`, `.bss`), tabela de símbolos e, quando compilado em modo de depuração, dados *debug* (por exemplo, mapeamento entre endereços e funções/linhas de código). No contexto do Renode, esses arquivos permitem carregar o *firmware* no mapa de memória simulado de forma consistente com o *linker script* do projeto, além de viabilizar a observação do comportamento do sistema com maior rastreabilidade (como inspeção de símbolos, pontos de parada e análise de execução) durante os testes de validação da plataforma STM32L496.

3.2.5 Arquivos de descrição de sistema (*System View Description*)

Os arquivos `.svd` (CMSIS-SVD) são utilizados no Renode para referenciar na descrição da plataforma, de forma estruturada, o conjunto de periféricos do microcontrolador e o respectivo mapa de registradores (endereços-base, *offsets*, campos de bits, nomes e permissões de acesso). Dentro do Renode, a referência a um `.svd` é feita em arquivos `.repl` e é utilizada como uma fonte de metadados para tornar a plataforma mais observável e depurável, permitindo que ferramentas do simulador apresentem os registradores e seus campos de maneira legível e consistente com a

documentação do fabricante. Assim, o `.svd` complementa os modelos comportamentais implementados em `.cs` ao fornecer a “descrição formal” dos registradores, sem substituir a lógica de simulação: ele não implementa comportamento, mas apoia a inspeção, a validação e a rastreabilidade de acessos a registradores durante os testes do *firmware*.

3.2.6 Arquivos Python

Assim como os arquivos `.ELF`, os *scripts* em `Python` não são componentes essenciais para descrever MCUs ou placas no Renode. Ainda assim, eles desempenham um papel importante de apoio, pois ampliam a usabilidade da ferramenta e a capacidade de conduzir simulações, sem exigir alterações no *firmware* em teste. Em particular, *scripts* em `Python` podem ser empregados para automatizar rotinas de verificação e coleta de evidências, instrumentar interfaces e barramentos para aumentar observabilidade, integrar a simulação com ferramentas externas (por exemplo, analisadores e validadores) e realizar processamento de eventos e *logs* durante a execução.

No Renode, esses *scripts* podem ser associados a componentes específicos por meio de comandos nos arquivos `.resc` como `SetPacketHookFromFile`) usado no trabalho, passando a executar eventos da simulação (transmissão/recepção em um *medium* IEEE 802.15.4).

3.3 Detalhamento dos arquivos `.repl`

Os arquivos de descrição de plataforma do Renode (extensão `.repl`) são o mecanismo utilizado para declarar, de forma textual e reproduzível, como uma plataforma simulada é montada a partir de componentes (CPU, memórias e periféricos) e como esses componentes se conectam no nível de sistema. Conforme discutido no início desta seção, há dois usos recorrentes, arquivos `.repl` voltados ao microcontrolador (MCU) e `.repl` voltados à *boards*.

Neste detalhamento, foram utilizados como exemplos arquivos `.repl` nativos do Renode: um arquivo de MCU que descreve um STM32F4 e um arquivo de *board* baseado nesse microcontrolador. O objetivo foi evidenciar, na prática, como o Renode organiza a declaração do mapa de memória, a anexação de modelos de periféricos (implementados em C#) e o roteamento de sinais (como linhas de interrupção e conexões de *GPIO*), além de mostrar como a descrição em nível de placa complementa o MCU com componentes externos.

Este detalhamento também descreve a sintaxe empregada nos `.repl`. Em particular, serão apresentados os padrões textuais mais frequentes: a instanciação de componentes e seu posicionamento no `SystemBus` (por exemplo, declarações do tipo `<nome>: <tipo> @ sysbus <endereço>`), a atribuição de parâmetros e propriedades, a declaração de *ranges*/tamanhos quando necessário e a forma como o arquivo expressa conexões lógicas entre blocos (como ligações de IRQ e associações

de pinos).

A Figura 3.1 apresenta um trecho do arquivo nativo `stm32f4.repl`, no qual se observa a declaração da memória de *flash* como uma região `MappedMemory` e, em seguida, a instancição do periférico `STM32F4_FlashController` com múltiplos mapeamentos no `SystemBus` e com a injeção explícita da referência para a região de *flash* previamente definida.

Figura 3.1: Seção do `(.repl)` nativo STM32F4 do Renode.



```
flash: Memory.MappedMemory @ sysbus 0x08000000
      size: 0x200000

flash_controller: MTD.STM32F4_FlashController @ {
  sysbus 0x40023C00;
  sysbus new Bus.BusMultiRegistration { address: 0x1FFFC000; size: 0x100; region: "optionBytes" }
}
flash: flash
```

Fonte: Autor.

No primeiro bloco, a linha `flash: Memory.MappedMemory @ sysbus 0x08000000` cria uma região de memória linear no barramento, associada ao endereço-base típico da *flash* em microcontroladores STM32. O parâmetro `size: 0x200000` define o tamanho dessa região (neste exemplo, 2 MiB), delimitando o *range* endereçável que armazenará o binário e os dados persistentes durante a simulação.

No segundo bloco, `flash_controller: MTD.STM32F4_FlashController @ {...}` instancia o modelo do controlador de *flash* e o registra no `SystemBus`. A primeira linha dentro do bloco, `sysbus 0x40023C00;`, mapeia a janela principal de registradores do periférico no endereço-base do controlador (região de registradores `FLASH_ACR`, `FLASH_KEYR`, `FLASH_SR`, `FLASH_CR`, etc.). Em seguida, a declaração `sysbus new Bus.BusMultiRegistration { address: 0x1FFFC000; size: 0x100; region: "optionBytes" }` adiciona um segundo mapeamento para o *mesmo* periférico, criando uma janela adicional no barramento (com tamanho `0x100`) associada à região lógica `"optionBytes"`. Esse padrão de *multi-registration* é usado quando um único modelo precisa responder por mais de uma faixa de endereços, como ocorre no caso dos *option bytes*, que são acessados em um endereço distinto da janela principal de registradores.

Por fim, a linha `flash: flash` estabelece uma ligação por propriedade (*property injection*) entre o controlador e a memória definida anteriormente. Nesse caso, o atributo `flash` do modelo `STM32F4_FlashController` recebe a instância nomeada `flash` criada no início do `.repl`. Assim, o periférico não apenas expõe registradores de controle, mas também pode aplicar operações (por exemplo, apagamento e programação) sobre o conteúdo efetivo da memória simulada, conectando a semântica do controlador ao armazenamento onde o *firmware* está carregado.

Já a Figura 3.2 apresenta um trecho de o arquivo `.repl` em nível de *board*, no qual a plataforma do microcontrolador é incorporada junto a componentes externos simples, botões e LEDs são instanciados e conectados a pinos de *GPIO*. Esse padrão

é representativo da função do `.repl` de placa, reutilizar um (MCU) já descrito e amarrar a ele a topologia de *hardware* externo do sistema.

Figura 3.2: Arquivo (`.repl`) nativo utilizando o STM32F4 em uma *board* do Renode.



```
using "platforms/cpus/stm32f4.repl"

UserButton: Miscellaneous.Button @ gpioPortA
-> gpioPortA@0

UserLED: Miscellaneous.LED @ gpioPortD

gpioPortD:
  12 -> UserLED@0
```

Fonte: Autor.

A diretiva `using "platforms/cpus/stm32f4.repl"` importa a descrição do MCU STM32F4 previamente definida no Renode, tornando disponíveis suas instâncias internas (por exemplo, controladores de *GPIO*) e seus nomes simbólicos. Na prática, isso permite que o `.repl` da placa trate o microcontrolador como um “bloco” já montado e foque apenas em como a placa conecta seus periféricos externos aos recursos do MCU.

No exemplo, `UserButton: Miscellaneous.Button @ gpioPortA` cria um dispositivo externo do tipo `Button` e o anexa ao controlador de *GPIO* `gpioPortA` onde operador `@` indica o ponto de anexação do componente. Em seguida, a linha `-> gpioPortA@0` expressa a ligação lógica do botão ao pino 0 desse *GPIO*. Aqui, a notação `@0` funciona como um seletor de índice: `gpioPortA@0` refere-se explicitamente ao pino 0 do `gpioPortA`.

De forma análoga, `UserLED: Miscellaneous.LED @ gpioPortD` instancia um componente `LED` e o anexa ao controlador `gpioPortD`. A conexão entre o *GPIO* e o `LED` é expressa no bloco nomeado `gpioPortD:`, em que a linha `12 -> UserLED@0` mapeia o pino 12 do *GPIO Port D* para a entrada 0 do dispositivo `UserLED`. Novamente, o `@` seleciona a linha/pino de conexão em cada extremidade: 12 indica o pino do *GPIO*, enquanto `UserLED@0` refere-se ao terminal de índice 0 do `LED` (descrito na classe `LED`). Assim, quando o *firmware* altera o estado do pino 12, o `LED` simulado reflete esse estado.

3.4 Scripts de Execução (`.resc`) e Automação da Simulação no Renode

Os scripts `.resc` para automatização, inicialização e execução de simulações contém no arquivos uma sequência de comandos equivalentes aos do *Monitor* do Renode, permitindo encapsular etapas repetitivas (criação da máquina, carregamento

do `.repl`, carregamento do binário, configuração de logs e execução) e, assim, garantir reprodutibilidade nos experimentos e validações.

Apresentado na figura 3.3, foi apresentado um arquivo `.resc` genérico de exemplo do Renode para depois ser explicado:

Figura 3.3: Exemplo de script `.resc` nativo do Renode.

A screenshot of a terminal window showing a Renode script. The script starts with a title bar (red, yellow, green dots) and a title 'STM32F746'. It contains several lines of code: ':name: STM32F746', ':description: This script runs MBED hello world on STM32F7 Discovery.', 'using sysbus', '\$name?="STM32F746"', 'mach create \$name', 'machine LoadPlatformDescription @platforms/boards/stm32f7_discovery-bb.repl', 'showAnalyzer usart1', 'showAnalyzer ltdc', '\$bin ?= @https://dl.antmicro.com/projects/renode/renode-mbed-pipeline-helloworld.elf-ga2ede71-s_2466384-6e3635e4ed159bc847cf1deb3dc7f24b10d26b41', 'macro reset', '"""', 'sysbus LoadELF \$bin', '"""', and 'runMacro \$reset'.

```
:name: STM32F746
:description: This script runs MBED hello world on STM32F7 Discovery.

using sysbus
$name?="STM32F746"
mach create $name

machine LoadPlatformDescription @platforms/boards/stm32f7_discovery-bb.repl

showAnalyzer usart1
showAnalyzer ltdc

$bin ?= @https://dl.antmicro.com/projects/renode/renode-mbed-pipeline-helloworld.elf-ga2ede71-
s_2466384-6e3635e4ed159bc847cf1deb3dc7f24b10d26b41

macro reset
"""
sysbus LoadELF $bin
"""

runMacro $reset
```

Fonte: Autor.

O script `.resc` da Figura 3.3 exemplifica um fluxo de execução no Renode, ele cria uma máquina, carrega a descrição de plataforma (`.repl`), habilita instrumentos de observação e carrega um binário `.ELF` para iniciar a simulação. As linhas iniciadas por `:name:` e `:description:` são metadados opcionais do próprio script e servem como documentação (identificação do exemplo e descrição do propósito). Em seguida, `using sysbus` expõe o objeto `sysbus` no contexto do script, permitindo invocar comandos relacionados ao barramento do sistema e ao carregamento de imagens na memória. Logo após, a linha `$name?="STM32F746"` define uma variável `$name` com valor padrão, o operador `?` realiza uma atribuição condicional, (o valor só é aplicado se a variável ainda não tiver sido definida externamente). Com `mach create $name`, o Renode cria uma nova instância de máquina com o nome indicado. Na sequência, `machine LoadPlatformDescription @platforms/boards/stm32f7_discovery-bb.repl` carrega a plataforma descrita em um arquivo `.repl` nativo do Renode; o prefixo `@` indica um recurso referenciado pelo ambiente (relativo ao diretório de instalação do Renode). Esse carregamento instancia a CPU, as memórias e os periféricos, registra seus endereços-base no mapa de memória e configura conexões essenciais, como barramentos e roteamento de interrupções.

Após a plataforma estar montada, o script ativa ferramentas de observação com `showAnalyzer usart1` e `showAnalyzer ltdc`. Esses comandos habilitam analisadores associados, respectivamente, à *USART* e ao controlador de *display LTDC*. Em seguida, o binário a ser executado é parametrizado em `$bin ?= @https://...`, novamente com atribuição condicional (`?`), permitindo substituir a imagem do

firmware sem modificar o arquivo. Nesse exemplo, o `$bin` referencia um executável `.ELF` hospedado remotamente.

A definição `macro reset` encapsula uma sequência de comandos sob um nome único. O corpo da macro é delimitado por aspas triplas e contém o comando `sysbus LoadELF $bin`, responsável por carregar o `.ELF` na memória simulada, respeitando os endereços e seções definidos no próprio arquivo (como `.text`, `.data` e `.bss`), além de preservar símbolos que podem ser usados em depuração e inspeção. Por fim, `runMacro $reset` executa a macro definida, efetivando o carregamento do *firmware* e deixando a simulação pronta para ser iniciada pelos comandos do Monitor.

Em síntese, o exemplo demonstra como um `.resc` funciona como um roteiro executável do experimento e também ilustra a sintaxe do arquivo.

Enquanto o `.repl` especifica *onde* cada periférico existe no mapa de memória e *como* ele se conecta a outros componentes (barramentos, *GPIOs* e interrupções), os arquivos `.cs` definem *o que* ocorre quando o *firmware* interage com esses endereços e quais efeitos funcionais e temporais emergem das leituras e escritas em registradores.

3.5 Detalhamento dos arquivos `.cs`

No Renode, uma classe em C# é considerada um modelo de periférico quando implementa a interface `IPeripheral`. Para que esse modelo possa ser anexado ao barramento do sistema e receba acessos do processador, ele deve implementar pelo menos uma interface de largura de acesso (`IBytePeripheral`, `IWordPeripheral`, `IDoubleWordPeripheral` ou `IQuadWordPeripheral`), compatibilizando transações de 8, 16, 32 ou 64 bits, respectivamente. No caso comum de periféricos STM32, que são mapeados em palavras de 32 bits, um modelo `IDoubleWordPeripheral` deve prover métodos de leitura, escrita e *reset* (por exemplo, `ReadDoubleWord`, `WriteDoubleWord` e `Reset`), que são acionados pelo `SystemBus` durante a execução do binário (Renode Project, 2026c).

Do ponto de vista do fluxo de execução, o Renode encaminha para o `SystemBus` os acessos a endereços que não pertencem a regiões de memória simples (como `MappedMemory`), e esses acessos são então tratados pelos modelos C# associados aos endereços declarados na plataforma. Isso estabelece a fronteira entre regiões de memória lineares e regiões que possuem funcionalidade de periférico, nas quais um acesso não é apenas leitura/escrita de bytes, mas pode causar efeitos colaterais como atualização de *flags*, mudança de estado interno, disparo de interrupções e geração de eventos (Renode Project, 2026c).

Para reduzir complexidade e melhorar rastreabilidade, o Renode recomenda o uso do *Register Framework*, no qual os registradores do periférico são definidos como um conjunto de campos (*flags*, *value fields*, *enum fields*) com modos de acesso (leitura, escrita, *read-to-clear*, *write-one-to-clear*, etc.) e *callbacks* associados a eventos de leitura/escrita. Esse padrão facilita a implementação de semânticas típicas de registradores (por exemplo, bits que limpam ao ler, bits que setam ao escrever 1) e permite acoplar lógica de simulação a acessos específicos, aproximando o

comportamento do modelo ao esperado pelo *firmware* (Renode Project, 2026c).

Ainda no nível de integração com o barramento, é comum que o modelo declare o tamanho de sua janela de endereçamento por meio da interface `IKnownSize`, tornando explícito (em bytes) o espaço ocupado pelo periférico. Quando o periférico não declara esse tamanho, o `.repl` precisa informar o *range* ao registrar o dispositivo (Renode Project, 2026c).

Em síntese, os arquivos `.cs` funcionam como a camada de comportamento do gêmeo digital, traduzindo acessos do *firmware* em mudanças de estado internas, regras de registradores e sinalizações (IRQ/*GPIO*), conectando a execução do binário a uma dinâmica coerente com a arquitetura do microcontrolador.

3.5.1 Estrutura interna típica de um modelo (.cs) no Renode

Além de definir a interface do periférico com o `SystemBus`, os arquivos `.cs` na distribuição do Renode tendem a seguir uma organização interna recorrente, que facilita manutenção, rastreabilidade e evolução incremental do modelo. De forma geral, observa-se uma separação clara em três partes: cabeçalho e estrutura da classe, descrição dos registradores e seus campos, e rotinas que consolidam o comportamento do periférico. Essa divisão aparece de maneira consistente em modelos nativos do Renode (por exemplo, controladores seriais, temporizadores, controladores de barramento e módulos de comunicação), variando apenas o conjunto de registradores e a lógica específica de cada bloco.

A seguir, apresenta-se a descrição das partes típicas de um arquivo `.cs` e exemplos extraídos de um modelo nativo do Renode, utilizando como referência o arquivo `stm32f4_FlashController.cs`. Esses trechos são apresentados para evidenciar, a sintaxe e a estrutura recorrente dos modelos em `C#`, ilustrando como o Renode organiza o cabeçalho/classe, a definição de registradores e a implementação das rotinas funcionais do periférico.

A primeira parte corresponde ao cabeçalho do arquivo e à estrutura da classe. Nessa seção, o arquivo declara as diretivas `using` necessárias para acessar *namespaces* do Renode (núcleo da simulação, periféricos, barramento e *logging*) e define a classe do periférico. É comum que a classe herde de uma base como `BasicDoubleWordPeripheral` (quando o mapeamento é majoritariamente de 32 bits) e implemente interfaces como `IKnownSize`, para explicitar o tamanho da janela do periférico no mapa de memória. Ainda nessa parte, encontram-se o construtor (normalmente recebendo `IMachine`), a inicialização do conjunto de registradores e o método `Reset()`, que estabelece um estado inicial coerente com o *power-on reset*. Também é onde ficam declarados os campos privados que representam o estado interno do periférico (por exemplo, sinais de estado, valores de configuração, *buffers* de TX/RX, habilitações de interrupção e referências a linhas de IRQ), os quais serão consumidos tanto pela descrição dos registradores quanto pelas rotinas de comportamento. A Figura 3.4 ilustra esse padrão no modelo nativo `STM32F4_FlashController` do Renode.

Na Figura 3.4, o *namespace* identifica o módulo do Renode ao qual o pe-

Figura 3.4: Cabeçalho e estrutura da classe no arquivo nativo STM32F4_FlashController do Renode.

```

namespace Antmicro.Renode.Peripherals.MTD
{
    [AllowedTranslations(AllowedTranslation.ByteToDoubleWord | AllowedTranslation.WordToDoubleWord)]
    public class STM32F4_FlashController : STM32_FlashController, IKnownSize
    {
        public STM32F4_FlashController(IMachine machine, MappedMemory flash) : base(machine)
        {
            this.flash = flash;

            controlLock = new LockRegister(this, nameof(controlLock), ControlLockKey);
            optionControlLock = new LockRegister(this, nameof(optionControlLock), OptionLockKey);

            optionBytesRegisters = new DoubleWordRegisterCollection(this);

            DefineRegisters();
            Reset();
        }
    }
}

```

Fonte: Autor.

refêrico pertence e define o nome completo usado para instanciá-lo a partir do `.repl`. O atributo `[AllowedTranslations(...)]` indica que o modelo aceita que acessos de 8 e 16 bits sejam traduzidos pelo barramento para operações de 32 bits, o que é útil porque muitos registradores STM32 são organizados em *double words*. A classe `STM32F4_FlashController` especializa um controlador genérico (`STM32_FlashController`) e implementa `IKnownSize` para explicitar o tamanho da janela de endereçamento do periférico.

No construtor, o modelo recebe a máquina (`IMachine`) e uma referência à região de *flash* (fornecida pela própria plataforma descrita no `.repl`), armazenando-a para que operações do controlador possam atuar sobre essa memória simulada. Em seguida, o modelo inicializa os mecanismos de travamento (`LockRegister`) e a estrutura dedicada aos *option bytes* (`DoubleWordRegisterCollection`), que agrupa os registradores associados a essa configuração persistente. Por fim, `DefineRegisters()` declara offsets/campos no *Register Framework* e `Reset()` coloca o periférico em um estado inicial.

A segunda parte concentra a descrição de registradores por meio do *Register Framework*. Tipicamente, o arquivo consolida os *offsets* de registradores em um `enum`, mostrado na figura 3.5, de modo a manter correspondência direta com o mapa de memória do manual do fabricante e com o `.repl`. Em seguida, um método `DefineRegisters()`, evidenciado na figura 3.6, registra cada *offset* em uma coleção de registradores e descreve seus campos com chamadas como `WithFlag`, `(WithTaggedFlag)`, `WithReservedBits`, `WithValueField` e `WithEnumField`, além de marcar bits reservados e especificar semânticas de acesso quando necessário. Esse padrão torna explícita a intenção de cada bit, facilita a inspeção no Monitor e viabiliza efeitos colaterais associados a leituras/escritas por meio de *callbacks*. Na prática, especialmente em desenvolvimento incremental, é comum que apenas os

campos exigidos pelos *firmwares* em teste sejam modelados com maior fidelidade, enquanto demais campos permanecem como *placeholders* (mantendo o registrador acessível, mas sem lógica completa), o que preserva compatibilidade sem elevar desnecessariamente a complexidade.

Figura 3.5: Enum dos registradores do STM32F4_FlashController do Renode.

```
private enum Registers
{
    AccessControl = 0x00, // FLASH_ACR
    Key = 0x04,           // FLASH_KEYR
    OptionKey = 0x08,     // FLASH_OPTKEYR
    Status = 0x0C,        // FLASH_SR
    Control = 0x10,       // FLASH_CR
    OptionControl = 0x14, // FLASH_OPTCR
}
```

Fonte: Autor.

Figura 3.6: Função DefineRegisters do STM32F4_FlashController do Renode.

```
private void DefineRegisters()
{
    Registers.AccessControl.Define(this)
        .WithValueField(0, 4, name: "LATENCY")
        .WithReservedBits(4, 4)
        .WithTaggedFlag("PRFTEN", 8)
        .WithTaggedFlag("ICEN", 9)
        .WithTaggedFlag("DCEN", 10)
        .WithTaggedFlag("ICRST", 11)
        .WithTaggedFlag("DCRST", 12)
        .WithReservedBits(13, 19);

    Registers.Key.Define(this)
        .WithValueField(0, 32, FieldMode.Write, name: "FLASH_KEYR",
            writeCallback: (_, value) => controlLock.ConsumeValue((uint)value));

    Registers.OptionKey.Define(this)
        .WithValueField(0, 32, FieldMode.Write, name: "FLASH_OPTKEYR",
            writeCallback: (_, value) => optionControlLock.ConsumeValue((uint)value));
}
```

Fonte: Autor.

Na definição dos registradores, o Renode descreve cada campo por meio de métodos auxiliares que especificam *tipo*, *largura* e *semântica* de acesso. O método `WithFlag` declara um campo de 1 bit com estado associado, podendo opcionalmente expor uma referência ao campo via parâmetro `out` para que o modelo consulte ou altere seu valor em rotinas internas, além de permitir *callbacks* em leituras/escritas. Já `WithTaggedFlag` também descreve um campo de 1 bit, porém como um marcação que aparece de forma legível em logs e no Monitor.

O método `WithReservedBits` marca intervalos de bits como reservados, esses bits não são expostos como campos editáveis e servem para manter compatibilidade com o mapa de registradores real. Para campos numéricos com mais de 1 bit, `WithValueField` define uma faixa de bits que codifica um valor inteiro, podendo restringir o modo de acesso e associar *callbacks* para efeitos colaterais. Por fim, `WithEnumField` descreve campos multi-bit cujo valor representa um conjunto discreto de estados, mapeando padrões de bits para um tipo enumerado (*enum*).

A terceira parte corresponde à descrição das funções e rotinas principais do periférico, isto é, o código que traduz mudanças em registradores em comportamento observável durante a simulação. Em modelos nativos do Renode, essa lógica costuma ser organizada em poucas rotinas centrais: procedimentos de atualização de estado (por exemplo, recalcular *flags* de status a partir de *buffers* e habilitações), rotinas de transmissão/recepção ou processamento de eventos (por exemplo, enfileirar dados, consumir dados e atualizar *status registers*), rotinas de interrupção, que avaliam habilitações e *flags* para acionar ou desacionar linhas de IRQ, e rotinas de temporização quando o periférico depende do tempo virtual (por exemplo, *timers* e periféricos com cadência de eventos). Assim, enquanto a descrição de registradores define o que existe no espaço de endereçamento e como cada campo é exposto ao *firmware*, as rotinas do periférico definem o que acontece quando esses campos são alterados, garantindo que o modelo produza efeitos coerentes (eventos, estados internos e sinalizações) no ambiente simulado.

3.6 Tempo no Renode

O Renode distingue o tempo virtual (tempo interno da emulação) do tempo real (ou *host time/wall-clock time*, isto é, o tempo medido no computador que executa o simulador). O tempo virtual é expresso em segundos virtuais e é utilizado para ordenar eventos, disparos de interrupção, temporizações e comunicação entre componentes simulados. Assim, o tempo virtual representa quanto tempo passou no sistema embarcado emulado, enquanto o tempo real representa quanto tempo foi necessário para o simulador executar essa emulação no computador (Renode Project, 2026e).

A menor parcela de tempo virtual representável no Renode é de 1 nanossegundo virtual (10^{-9} s), que corresponde à resolução mínima do *time framework* (Renode Project, 2026e). Na prática, isso define a fração mínima de tempo com que instantes e intervalos de tempo podem ser expressos na emulação.

O avanço do tempo virtual ocorre em fatias chamadas *quantum*, um *quantum* é o intervalo de tempo virtual entre duas fases de sincronização e pode ser configurado. Ao conceder um *quantum* aos componentes que executam no domínio de tempo, o Renode permite que eles avancem (executem) até completar esse intervalo e, então, realiza uma fase de sincronização na qual são processados comunicações e eventos entre nós (Renode Project, 2026e).

Além da granularidade de tempo, o Renode modela o desempenho da CPU

em MIPS (*Millions of Instructions Per Second*), isto é, milhões de instruções por segundo virtual. Esse parâmetro também é configurável e influencia quantas instruções a CPU pode executar por unidade de tempo virtual, por padrão, o Renode utiliza 100 MIPS (Renode Project, 2026e). Consequentemente, ao aumentar ou reduzir o valor de MIPS, altera-se o ritmo de execução de instruções em relação ao tempo virtual, impactando tanto o desempenho quanto a correspondência entre cadências temporais e a execução do *firmware*.

O controle do fluxo de tempo virtual é organizado em torno dos conceitos de *time source* e *time sinks*. Um *time source* é o objeto que controla o avanço do tempo virtual, gerando *quantums* e coordenando as fases de sincronização. No Renode existe um único *time source* chamado *master time source*. (Renode Project, 2026e).

Um *sink* é qualquer componente que utiliza *quantums* e que participa do avanço temporal, eles se conectam ao *time source* e aguardam a concessão de um *quantum* para executar e avançar seu estado interno naquele intervalo de tempo virtual (Renode Project, 2026e). Em termos operacionais, o ciclo é: o *time source* concede um *quantum* aos *sinks*, cada *sink* executa até consumir o intervalo concedido, após todos reportarem conclusão, o Renode entra na fase de sincronização, na qual, como nenhum nó está executando, processa-se comunicação entre nós e demais eventos pendentes, somente após isso um novo *quantum* é concedido (Renode Project, 2026e). Esse mecanismo garante que todos os componentes conectados ao mesmo *time source* permaneçam sincronizados dentro da granularidade definida pelo *quantum*.

Por fim, no comportamento padrão, o Renode sincroniza o avanço do tempo virtual ao tempo real sempre que possível. Quando a propriedade **AdvanceImmediately** do *time source* não está habilitada, o Renode automaticamente desacelera a execução caso detecte que o tempo virtual está passando mais rápido do que o tempo real (Renode Project, 2026e). Quando **AdvanceImmediately** é habilitado, essa limitação é removida e a emulação passa a avançar o mais rápido possível no *host*. Neste modo, o tempo virtual pode passar mais rapidamente do que o tempo real, ficando o ritmo efetivo de execução dependente principalmente ao desempenho disponível no computador e aos parâmetros de execução, como o *MIPS* (Renode Project, 2026e).

Como os *firmwares* desenvolvidos neste trabalho, assim como o *firmware* de produção, utilizam o *SysTick* como temporizador principal, a configuração mais relevante para manter a execução temporizada de forma coerente na simulação foi ajustar o parâmetro **sysTickFrequency** do periférico NVIC (nativo do Renode) no arquivo **.repl** de descrição do MCU com a mesma frequência do *firmware*. Dessa forma, a cadência do *SysTick* no tempo virtual passa a refletir a frequência esperada pelo *firmware*, preservando o comportamento de rotinas dependentes do temporizador.

Capítulo 4

Metodologia

4.1 Cenário de aplicação do *firmware* de produção

Para auxiliar o entendimento da pesquisa e das escolhas realizadas ao longo do desenvolvimento do trabalho, faz-se necessário descrever o funcionamento e o cenário de aplicação do *firmware* de produção cuja simulação constituiu uma das metas deste estudo. Esse *firmware* foi desenvolvido no laboratório LMEST em um projeto conjunto com a Petrobras, no qual foi implementado um sistema de monitoramento de máquinas alocadas em plataformas de petróleo, utilizando módulos de sensores distribuídos em campo.

O subsistema de aquisição de dados das máquinas consiste em dois componentes principais instalados em campo: o módulo sensor e o dispositivo denominado Hub de sensores. O Hub concentra dados de múltiplos módulos sensores e atua como mestre no controle das rotinas de aquisição desses módulos e realiza a conexão com o servidor da Petrobras para envio dos dados coletados.

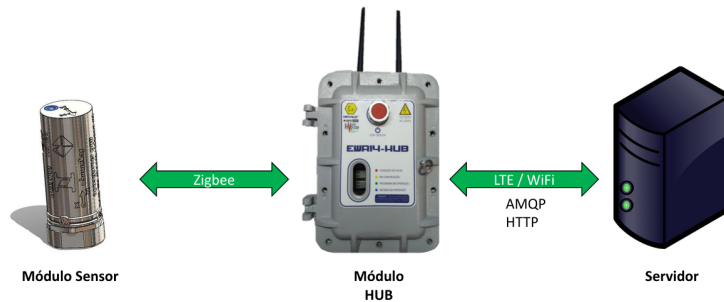
O módulo sensor teve seu projeto integralmente desenvolvido pelo laboratório LMEST e utiliza como conjunto de *hardware* um microcontrolador STM32L496 e um módulo de rádio Zigbee AT86RF233, que são simulados por meio da nova implementação de *digital twin* criada neste trabalho. O Hub, por sua vez, é controlado por um *single-board computer* Raspberry Pi 3 em conjunto com um AT86RF233.

A interação entre esses dois componentes ocorre por meio de comunicação bidirecional via rádio Zigbee, na qual o módulo sensor inicia o contato com o Hub transmitindo leituras básicas dos sensores. Em seguida, o Hub, de acordo com a programação recebida do servidor, responde ao módulo sensor para que seja colocado em etapas de aquisição mais complexas ou em modo de *sleep* (espera). O diagrama de comunicação do sistema de monitoramento é apresentado na Figura 4.1.

Nesse sistema, a extensão do simulador Renode desenvolvida neste trabalho foi utilizada para simular o conjunto do módulo sensor. No *digital twin*, foi carregado o mesmo arquivo binário empregado no hardware real, de modo a reproduzir, com fidelidade, as rotinas de aquisição de sensores, transmissão e recepção de *frames* via rádio Zigbee e os modos de *sleep* (espera).

O Hub, por sua vez, teve seu funcionamento parcialmente simulado por meio

Figura 4.1: Diagrama de comunicação do sistema de monitoramento.



Fonte: Autor.

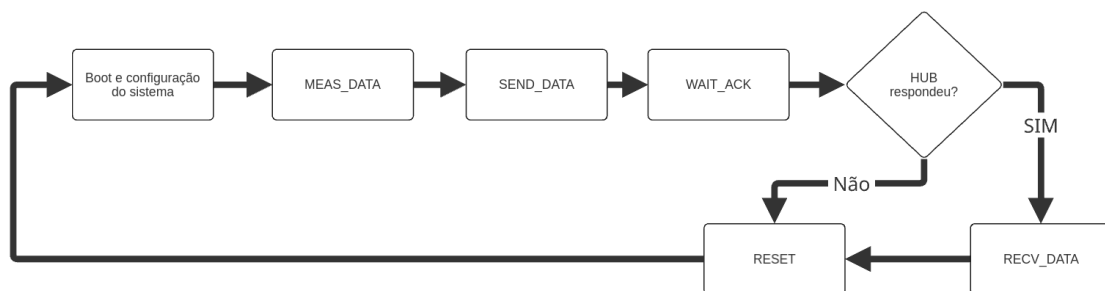
de *scripts* em Python: as tarefas de escuta de *frames*, *parsing* dos dados de sensores e envio de respostas de controle foram representadas em um nível de abstração mais alto.

4.1.1 Firmware de produção

Ainda na contextualização do cenário de aplicação, será necessário descrever o funcionamento detalhado de execução do *firmware* de produção que é carregado nos módulos sensores, que também é o alvo da simulação do gêmeo digital criado a partir desta pesquisa.

Em alto nível, o *firmware* foi desenvolvido para inicializar e manter a configuração do rádio, adquirir medições por ADC (incluindo aquisições via DMA), transmitir dados e/ou segmentos de dados ao nó concentrador (*hub*) e receber comandos do *hub* para controlar medições, confirmar/retransmitir dados e executar comissionamento. O comportamento é organizado como uma máquina de estados, intercalando estados ativos (medição, transmissão e recepção) com períodos de baixo consumo (*sleep*) e com habilitação/desabilitação seletiva de clocks de periféricos.

Na figura 4.2, é apresentado um fluxograma que representa os estágios do *firmware* do módulo sensor:



Fonte: Autor.

Durante o *boot*, o sistema inicializa GPIOs, timers, RTC e realiza a configuração do rádio com parâmetros padrão (PAN ID, canal e parâmetros MAC). Em seguida, o AT86RF233 é configurado e associado a *callbacks* de TX/RX, de forma que eventos (recepção de *frames*, fim de transmissão) disparem processamento na camada de aplicação.

A próxima ação é entrar no estágio de realização de medidas (MEAS_DATA), e dependendo do comando recebido do *hub*, uma determinada amostra de leituras é realizada. Caso o sensor ainda não tenha recebido nenhuma resposta do *hub* ele prepara as medições chamadas globais. Caso o *hub* já tenha respondido a prévia comunicação do sensor, no lugar das medições globais ele pode realizar as medições de formas de onda *waveform* no estágio (MEAS_DATA).

Quando uma medição é realizada (MEAS_DATA), o *firmware* habilita apenas os clocks dos periféricos estritamente necessários (ADC, DMA e timers) e executa aquisições conforme o tipo solicitado. Para formar o *frame* de dados referenciado como “global”, lê canais auxiliares (temperatura e bateria). Para medições de forma de onda, realiza calibração e captura um conjunto de amostras temporizadas, transferidas via DMA, com taxa de amostragem e número de amostras definidos por comandos do *hub*. Ao final, prepara o rádio para transmissão.

No estado de envio (SEND_DATA), o nó monta e transmite *frames* sobre IEEE 802.15.4, após a transmissão, entra em espera por confirmação (WAIT_ACK), com *timeout*. Caso tenha recepção (RECV_DATA), decodifica comandos do *hub* e atualiza *flags* internas que direcionam a próxima leitura após o *reset*, e caso não receba nenhuma resposta do *hub*, apenas entra em modo de baixo consumo, esperando o sinal de *hardware* para acordar.

4.2 Visão geral do método

Esta pesquisa adota uma abordagem de desenvolvimento incremental orientado a testes, na qual a plataforma de simulação é construída em camadas sucessivas:

1. familiarização e instrumentação do ambiente de simulação;
2. criação e integração da plataforma do microcontrolador;
3. validação unitária de periféricos por meio de *firmwares* simples;
4. integração e validação com o chip de rádio;
5. execução e depuração do *firmware* real do projeto-alvo.

Essa estratégia foi escolhida para facilitar o desenvolvimento, uma vez que os periféricos foram criados e testados individualmente, em cenários de uso controlados. Assim, tornou-se possível atribuir falhas a causas específicas antes de avançar para integrações mais complexas.

O objeto do trabalho consiste em estender o Renode para suportar uma nova plataforma baseada no STM32L496 (família STM32L4), por meio da criação

de modelos de periféricos em C# (arquivos `.cs`) e de descrições de controladores e placas (arquivos `.repl`), além da elaboração de *scripts* de simulação (arquivos `.resc`). Adicionalmente, são realizados ajustes no modelo já existente do rádio AT86RF233, de modo a permitir uma simulação consistente da comunicação via SPI, da composição de *frames*, de rotinas de teste e da sinalização por interrupção necessárias.

A estratégia de validação é organizada em três níveis:

1. testes de periférico isolado, com *firmwares* simples;
2. testes de integração, envolvendo múltiplos periféricos;
3. execução do *firmware* final-alvo, na qual envolve requisitos de uso mais complexos do microcontrolador.

4.3 Ambiente e ferramentas

Para garantir a reprodutibilidade, todo o desenvolvimento e os experimentos foram conduzidos em um ambiente controlado, com registro explícito das versões de ferramentas e dependências. Em particular, foram considerados:

1. **Sistema operacional e arquitetura:** Linux Mint 21.2 x86_64.
2. **Hardware da máquina:** processador AMD Ryzen 7 4800H, GPU dedicada NVIDIA GeForce GTX 1650 Ti Mobile, GPU integrada AMD Renoir, 16 GB de memória RAM e SSD NVMe 4.0 de 512 GB.
3. **Renode:** Renode v1.15.3.18516 e desenvolvimento iniciado a partir do *commit* fca3b9a2.
4. **Toolchain ARM:** STM32CubeIDE versão 1.19.0 com arm-none-eabi-gcc 10.2.1 20201103 (*release*).
5. **Depuração:** arm-none-eabi-gdb 10.1.90.20201028-git, com suporte a conexão remota ao Renode.

O repositório foi estruturado seguindo a separação original do Renode, distinguindo descrição de plataforma, modelos de periféricos e *scripts* de teste, além de evidências de execução (*logs*). A estrutura abaixo ilustra uma organização típica adotada ao longo do trabalho:

```
renode/
platforms/cpus/      # .repl de microcontroladores
platforms/boards/    # .repl de placas (boards)
scripts/             # .resc e scripts do Monitor para inicialização
python/              # criado para conter scripts Python
src/Infrastructure/  # modelos .cs (periféricos STM32L4 e extensões)
output_logs/         # criado para conter os logs de depuração
```

A execução dos experimentos foi padronizada por meio de *scripts* de inicialização (.resc) e comandos do Monitor, apresentado na figura 4.3, permitindo repetir cenários com o mesmo conjunto de parâmetros (carregamento do ELF, configuração de analisadores como UART e SPI, habilitação de *logs* de periféricos e controle da simulação). Quando aplicável, procedimentos automatizados foram utilizados para reduzir a variabilidade e facilitar o controle dos testes.

Figura 4.3: Tela inicial do Monitor do Renode.



Fonte: Autor.

4.4 Critérios de validação

Nesta pesquisa, considera-se que a plataforma “funcionou” quando atende, de forma verificável, a dois requisitos:

1. executar corretamente testes individuais e controlados, criados para validar periféricos e mecanismos básicos do microcontrolador;
2. sustentar a execução do *firmware* mais complexo do sensor do laboratório, utilizando o mesmo binário produzido para o hardware real, sem modificações no código.

No primeiro requisito, a validação é caracterizada pela capacidade de inicialização e execução estável dos *firmwares* mínimos, com evidências observáveis de operação (por exemplo, comunicação, eventos de interrupção e efeitos esperados de registradores), permitindo isolar e corrigir inconsistências de mapeamento e de comportamento em cada controlador simulado.

No segundo requisito, a validação é caracterizada pela execução contínua do *firmware* de produção, incluindo seus fluxos orientados a eventos e o uso combinado de periféricos, com preservação do comportamento esperado e produção de evidências equivalentes às observadas no ambiente real (tais como transmissão/recepção de *frames* e acionamento de rotinas por interrupção), sem travamentos ou falhas imediatas.

Como critério prático adicional de maturidade dos controladores simulados, adota-se a redução progressiva de alertas (*warnings*) reportados durante as simulações. Em particular, a diminuição de avisos relacionados a acessos indevidos à memória e a tentativas de operação não implementadas. Outro critério identificador de maturidade, é a redução de alertas de acessos a periféricos não anexados/criados na plataforma do microcontrolador, indicando maior completude do modelo e maior eficácia ao comportamento esperado do sistema.

4.5 Testes de verificação

Ao longo do desenvolvimento incremental do controlador simulado STM32L496 no Renode, foram realizados testes progressivos com o objetivo de verificar a consistência mínima do ambiente de simulação e dos modelos de periféricos implementados em C#. Esses testes tiveram caráter instrumental, sendo empregados para reduzir incertezas nas fases iniciais e intermediárias do desenvolvimento, orientar correções e apoiar a validação do fluxo de execução do *firmware*. A estratégia adotada foi evoluir gradualmente a complexidade dos ensaios: nas fases iniciais, priorizaram-se cenários simples e isolados (como alternância de GPIO e temporização); em seguida, foram incorporados periféricos de comunicação (UART e SPI) para exercitar múltiplas atividades concorrentes; posteriormente, a terceira fase passou a avaliar a integração do controlador com o transceptor IEEE 802.15.4 AT86RF233, envolvendo SPI bidirecional, sinalização por interrupção e propagação de quadros no meio de rádio simulado; por fim, a quarta fase consistiu na execução do *firmware* alvo do laboratório (sensor), que integra aquisição por ADC com DMA e temporizadores, comunicação via rádio e uma máquina de estados com modos de baixo consumo. Nesta seção descreve-se o procedimento adotado e os critérios de verificação.

4.5.1 Primeira fase de testes: alternância de GPIO (*toggle*)

A primeira fase de testes foi conduzida no início da pesquisa, quando a plataforma ainda não possuía suporte completo a periféricos de comunicação (como UART, SPI e I²C) e antes da integração com o modelo do rádio AT86RF233. O objetivo foi:

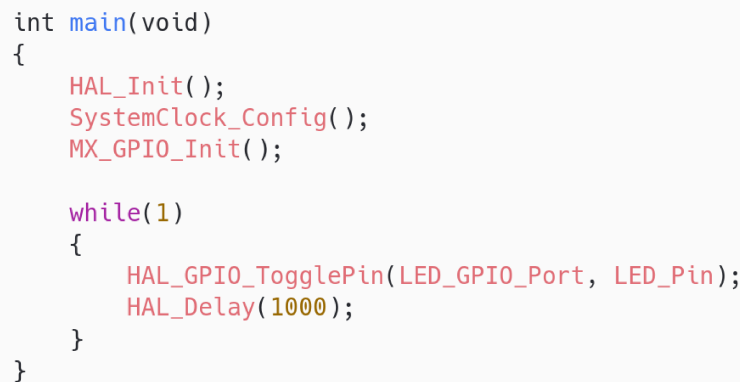
- verificar o funcionamento básico de E/S digitais (GPIO) no controlador simulado;
- validar se referências de memória utilizadas pelo *firmware* (acessos a registradores envolvidos na configuração/uso do GPIO e temporização) estavam coerentes com o mapeamento implementado;
- observar a coerência temporal da simulação, comparando a taxa de alternância do pino com o atraso programado no `.elf`.

Essa fase também serviu para familiarização com os comandos do *Monitor* do Renode, com a instrumentação via *logging* e com o ciclo de desenvolvimento de periféricos.

Firmware de referência

Foi desenvolvido um *firmware* mínimo no qual, a cada iteração do laço principal, o programa alterna o estado de um pino e aplica um atraso fixo. A seguir, apresenta-se o trecho relevante (omitindo o código padrão gerado pelo STM32CubeIDE), mostrado na figura 4.4:

Figura 4.4: Trecho do *firmware* da primeira fase de testes.

A imagem mostra um bloco de código C com uma barra decorativa no topo contendo três círculos (vermelho, amarelo e verde). O código define a função main, inicializa o HAL, configura o relógio do sistema e o GPIO, e entra em um loop infinito onde alterna o estado de um pino de LED a cada 1000 ms.

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    while(1)
    {
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
        HAL_Delay(1000);
    }
}
```

Fonte: Autor.

Critério de verificação

O critério adotado foi a observação, no log, de eventos sucessivos de alternância do pino com espaçamento coerente com o atraso programado no *firmware* (1000 ms entre alternâncias). Esse confronto fornece um indicativo de que o *.elf* está executando corretamente no processador simulado, que os acessos a registradores necessários à configuração e ao uso do GPIO estão funcionais e que o ambiente de simulação sustenta uma base temporal consistente para o comportamento temporizado do laço. As evidências e saídas do *logger* são apresentadas na seção de resultados.

4.5.2 Segunda fase de testes: comunicação via UART e SPI

A segunda fase de testes foi realizada em um estágio intermediário do desenvolvimento do controlador simulado e teve como objetivo verificar a implementação dos periféricos de comunicação UART e SPI, já com seus modelos em C# integrados ao arquivo *.repl* da plataforma. Nesta fase, o aumento da complexidade do

firmware de teste passou a exercitar novos caminhos de execução, o que exigiu ajustes e aprimoramentos em periféricos previamente implementados e revelou avisos (*warnings*) durante a preparação do ambiente de simulação e a execução.

O objetivo planejado foi observar, no próprio Renode, os dados transmitidos pelo STM32L496 via UART e via SPI, utilizando *logging* de depuração em tempo de execução e ferramentas nativas de análise/visualização de comunicação, em especial o analisador associado ao periférico UART. Adicionalmente, buscou-se avaliar a coerência temporal do fluxo de transmissão (no nível de periodicidade das mensagens), comparando a cadência configurada no *firmware* com aquela observada no ambiente simulado. Para manter comparabilidade com a primeira fase de testes e avaliar simultaneidade de atividades, o *toggle* de GPIO foi mantido no laço principal juntamente com as transmissões por UART e SPI.

Firmware de referência

O *firmware* foi estruturado no mesmo padrão da primeira fase de testes, porém incluindo a inicialização de USART1 e SPI1 e o envio periódico de uma mensagem fixa por ambos os canais. O trecho mínimo relevante é apresentado na figura 4.5:



```
MX_GPIO_Init();
MX_USART1_UART_Init();
MX_SPI1_Init();

while (1)
{
    char* frame = "RENODE TEST\n";

    HAL_SPI_Transmit(&hspi1, (uint8_t*)frame, strlen(frame), 1000);
    HAL_UART_Transmit(&huart1, (uint8_t*)frame, strlen(frame), 1000);
    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    HAL_Delay(1000);
}
```

Figura 4.5: Trecho mínimo do *firmware* da segunda fase de testes: transmissões por SPI e UART com *toggle* e atraso.

Além do caso básico (todas as ações com o mesmo período), também foram realizados ensaios variando a frequência relativa das tarefas (adotando atrasos distintos para transmissões e *toggle*), com o intuito de observar o comportamento do modelo quando múltiplas atividades concorrem com cadências diferentes.

Procedimento no Renode e observabilidade

A execução foi automatizada por um script `.resc` específico do teste de comunicação, com ajustes de *logging* para os periféricos `usart1` e `spi1` e ativação do analisador de UART. Os níveis de log foram configurados para que o terminal do Renode exibisse mensagens de depuração relevantes durante o *runtime*, enquanto

o comando `showAnalyzer` foi utilizado para visualizar a saída serial do periférico UART em uma interface dedicada.

Critérios de verificação

Os critérios adotados foram: a visualização, no analisador do Renode, da mensagem transmitida pela UART, confirmando o *payload* esperado; evidências consistentes de atividade de transmissão via SPI registradas no log de depuração, coerentes com as chamadas de envio realizadas no laço; e coerência de periodicidade entre transmissões e *toggle* de GPIO com os intervalos programados no *firmware*, como indicativo de que o ambiente simulado sustenta múltiplas atividades concorrentes com cadências controladas. As saídas observadas no analisador e nos logs, bem como a discussão dos comportamentos obtidos, são apresentadas na seção de resultados.

4.5.3 Terceira fase de testes: integração com o rádio AT86RF233

A terceira fase de testes teve como objetivo avaliar a integração do controlador simulado com o transceptor IEEE 802.15.4 AT86RF233, componente essencial para os cenários de comunicação sem fio exigidos pelo *firmware* alvo do laboratório. Em comparação às fases anteriores, a complexidade destes ensaios foi maior, pois envolveu comunicação SPI bidirecional (leitura e escrita de registradores do rádio, troca de quadros e controle de estados), bem como o uso de UART como canal auxiliar de depuração para expor informações de execução e facilitar a instrumentação via analisadores do Renode.

Embora o Renode disponibilize uma implementação nativa do AT86RF233, foram necessários ajustes no modelo para compatibilizá-lo com os acessos e fluxos empregados pelos *firmwares* utilizados nesta fase. As extensões realizadas no modelo do rádio são detalhadas no capítulo desenvolvimento.

Como parte do processo de validação, os *firmwares* desta etapa foram desenvolvidos sobre uma biblioteca autoral específica de testes, e não sobre a mesma biblioteca utilizada no *firmware* de produção (definido como objetivo principal da simulação). Essa decisão foi tomada para simplificar iterações, permitir modificações rápidas e tornar mais clara a rastreabilidade das alterações, uma vez que o código de teste (dedicado e controlado) facilita a identificação de seções e padrões relevantes durante a depuração. Além disso, a adoção de um *firmware* mais complexo também expôs diferenças de uso de periféricos em relação às fases anteriores, levando à correção de pontos adicionais em modelos já implementados.

Para esta fase de testes, foram criados múltiplos *firmwares* de referência, pois a validação do rádio exigiu diferentes casos de uso (injeção controlada, recepção, transmissão e troca de quadros entre nós) para exercitar o modelo de forma incremental e verificável.

Firmware A (recepção controlada e encaminhamento via UART)

O primeiro *firmware* desta etapa foi projetado para testar a lógica de recepção (RX) do rádio: o dispositivo permanece em modo de escuta e, ao receber um *frame*, encaminha o conteúdo para a UART (observável no `showAnalyzer`); após a impressão, retorna ao estado de escuta, aguardando o próximo quadro. Esse *firmware* teve como finalidade validar a cadeia de eventos e *triggers* necessários para concluir o RX de forma compatível com o hardware real (por exemplo, sinalização de eventos, atualização de *status* e disponibilização do *frame* no microcontrolador).

Uma particularidade desse primeiro *firmware* é que ele foi testado, inicialmente, sem um transmissor externo de rádio na simulação. O objetivo foi isolar e validar o fluxo de recepção no Renode antes da troca de quadros entre nós. Para isso, foi utilizada uma rotina auxiliar no modelo do rádio para injeção manual de um *frame* de teste, permitindo exercitar o caminho de RX a partir de uma chamada controlada.

Firmware B (ping-pong com dois nós simulados e meio IEEE 802.15.4)

O segundo *firmware* foi planejado como um teste de integração mais completo, envolvendo dois conjuntos independentes de hardware simulados dentro da mesma emulação, sendo cada conjunto composto por um STM32L496 conectado a um AT86RF233 pela linha `spi1`. Ambos executam essencialmente o mesmo *firmware*, diferindo apenas por uma *flag* de inicialização para definir qual nó inicia a troca (mestre/escravo). A aplicação realiza um *ping-pong* de mensagens: um nó transmite um quadro contendo uma string fixa e um valor numérico incremental, e o nó receptor responde com um novo quadro cujo contador é atualizado a partir do último valor recebido. Esse arranjo permite validar que, na mesma simulação, um nó simulado consegue transmitir, propagar pelo meio e ser recebido por outro nó também simulado, exercitando transmissão (TX), recepção (RX), controle de estado do rádio e interação via SPI em ambos os lados.

Para viabilizar a troca de quadros entre os dois rádios, foi necessário utilizar o mecanismo nativo do Renode de *media* sem fio. Para a utilização do AT86, foi criado um meio IEEE 802.15.4 (“**Medium**”) e ambos os rádios foram conectados a esse meio. Consequentemente, o script `.resc` deste teste também precisou ser estendido para criar o meio, instanciar duas máquinas, carregar descrições de plataforma e *firmwares* distintos e conectar cada rádio ao mesmo meio.

Critérios de verificação

Os critérios adotados nesta fase foram:

- **Firmware A:** confirmar que o quadro injetado manualmente percorre o caminho de RX no modelo do rádio e que o conteúdo recebido é encaminhado e observado na UART via analisador.

- **Firmware B:** validar a troca *ping-pong* entre dois nós simulados conectados ao mesmo meio IEEE 802.15.4, com progressão coerente do contador e observação de eventos relevantes via UART/*logs*.
- **Completude do modelo:** reduzir/eliminar *warnings* associados a barramentos/periféricos não conectados e a acessos não implementados, como indicativo de maior cobertura do modelo.

Tal como nas fases anteriores, as evidências e resultados completos (saídas do analisador e *logs*) são apresentados na seção de resultados.

4.5.4 Quarta fase de testes: execução do *firmware* de produção

A quarta fase de testes foi realizada com o *firmware* de produção já descrito no início da metodologia, sendo, diretamente associado a um dos objetivos principais do trabalho: executar, no Renode, o mesmo binário compilado para o STM32L496 de forma equivalente ao hardware real. Para isso, foi carregado na plataforma simulada o arquivo *.elf* compilado a partir do *source code* do projeto, sem qualquer alteração no código (isto é, o mesmo *firmware* utilizado no projeto real foi executado no hardware simulado).

Como implicação prática, reproduzir esse comportamento em simulação exigiram sustentar:

1. temporizações e *sleep*;
2. aquisição por ADC com DMA e timers;
3. integração STM32L496↔AT86RF233 via SPI (com RX/TX e IRQ);
4. geração e observação coerentes dos quadros IEEE 802.15.4.

Simulação parcial do *hub*

Como falado anteriormente nesta seção, o *hub* foi parcialmente simulado utilizando a capacidade do renode de utilizar scripts em python durante o processo de simulação, com ativações e chamadas com *triggers* específicos, assim o *hub* teve como cumprir suas funções principais de resposta ao módulo sensor e *parse* de dados.

Integração e correção de avisos (*warnings*) em execução

Por exercitar múltiplos periféricos de forma integrada e em sequências mais realistas do que os *firmwares* de verificação anteriores, esta fase foi a que mais demandou resolução *warnings* e ajustes finos nos modelos. Em particular, foram endereçados pontos relacionados à utilização conjunta de ADC e DMA, à formatação/encapsulamento de *frames* no rádio, às ligações de interrupções (por exemplo, timers, rádio e DMA) e às configurações e temporização, de modo a permitir a execução estável da máquina de estados.

Critérios de verificação

Os critérios adotados nesta fase foram:

1. execução do `.elf` do *firmware* de produção sem modificações, com estabilidade da máquina de estados;
2. redução/eliminação de *warnings* relevantes à medida que ajustes eram aplicados aos modelos (especialmente nos caminhos integrados ADC+DMA+timers e nas interrupções associadas);
3. observação e encaminhamento dos quadros transmitidos para fora da simulação via *packet hook* e UDP;
4. capacidade de provocar um fluxo mínimo de interação (requisição/resposta) por meio da resposta forçada do nó *dummy*, exercitando o caminho de RX no nó principal.

Os resultados e evidências experimentais (logs, capturas e saídas do coletor UDP) são apresentados na seção de resultados.

Capítulo 5

Desenvolvimento

5.1 Criação e compilação de *firmwares* para serem utilizados no Renode

Um dos objetivos centrais deste trabalho é viabilizar a simulação, no Renode, de um *firmware* real destinado ao microcontrolador STM32L496, utilizado em um projeto já desenvolvido e em operação no ambiente físico. Para isso, a ferramenta STM32CubeIDE foi empregada como ambiente de desenvolvimento e compilação, tanto para gerar as imagens executáveis do *firmware* alvo (a partir do seu *source code* original) quanto para criar e compilar múltiplos *firmwares* auxiliares de teste, utilizados ao longo do processo de implementação e validação incremental da plataforma.

O STM32CubeIDE é uma IDE voltada ao desenvolvimento embarcado para a família STM32, integrando editor, compilador e *toolchain* para ARM (GCC), mecanismos de *build* e *link*, suporte a depuração (*debug*) e integração com o ecossistema STM32Cube (incluindo bibliotecas e código de inicialização típicos, como HAL/LL e *startup code*). Na prática, a ferramenta automatiza e padroniza a geração de projetos e o processo de compilação, produzindo artefatos como `.elf` (com seções de memória, símbolos e informações de depuração) e, quando necessário, binários `.bin/.hex`, coerentes com a organização de memória definida pelo *linker script* do microcontrolador.

No contexto deste trabalho, além de compilar o *firmware* final do projeto real, o STM32CubeIDE foi utilizado para criar projetos mínimos e testes pontuais (por exemplo, exercitando periféricos específicos, interrupções e configurações de clock), com o objetivo de isolar funcionalidades e produzir cargas de trabalho controladas para a simulação. Esses projetos de teste não são componentes diretamente necessários para o funcionamento do Renode; entretanto, tais binários tiveram papel metodológico relevante, pois forneceram `.elfs` de referência para validar, de maneira incremental e reproduzível, se a plataforma STM32L496 implementada no simulador respondia conforme o esperado para cada subsistema avaliado, até atingir o nível de estabilidade requerido para a execução do *firmware* alvo.

5.2 Criação dos arquivos `.repl` do STM32L496

A criação de múltiplos arquivos `.repl` segue o padrão adotado pelos desenvolvedores do Renode nas plataformas STM32 já disponibilizadas nativamente. Nesse padrão, utiliza-se um arquivo para descrever o microcontrolador, contendo o mapa de memória, CPU, periféricos internos e interconexões, e arquivos adicionais de *kits/boards* para representar placas específicas que utilizam o MCU e adicionam componentes externos (por exemplo, rádios e sensores) e suas conexões. Essa organização foi amplamente observada nas implementações nativas da ferramenta e foi mantida neste trabalho para aumentar a modularidade, facilitar a manutenção e permitir a criação de múltiplas variantes de placa a partir do mesmo microcontrolador.

A descrição de plataformas no Renode foi realizada por meio de arquivos `.repl`, os quais definem a composição do sistema simulado (memórias, CPU, periféricos, endereços-base, mapeamento de interrupções e interconexões internas). Neste trabalho, foram criados dois arquivos principais: `stm32l496.repl`, que descreve o microcontrolador de forma genérica, e `stm32l496_kit.repl`, que descreve uma configuração de placa (*board*) baseada nesse microcontrolador e adiciona a conexão com o rádio AT86RF233.

Criação da placa (*board*) personalizada a partir de um MCU

A criação de dois arquivos foi adotada para aumentar a modularidade e reduzir a duplicação:

- **`stm32l496.repl` (MCU):** concentra a definição do microcontrolador, incluindo mapa de memória, CPU/NVIC, periféricos internos e roteamento de interrupções.
- **`stm32l496_kit.repl` (*board*):** reutiliza o microcontrolador e adiciona componentes externos e conexões físicas típicas de placa, como o rádio conectado a uma instância de SPI e a pinos de *GPIO*.

Essa estratégia também facilita evoluções futuras, uma vez que diferentes placas podem ser descritas reaproveitando o mesmo MCU e variando apenas os módulos externos e suas respectivas conexões, como será apresentado neste trabalho.

5.2.1 Base utilizada e customizações a partir do `stm32f4.repl`

Durante a criação do arquivo `stm32l496.repl`, foi utilizado como referência principal o arquivo `stm32f4.repl`, disponibilizado nativamente na distribuição do Renode. Esse arquivo foi utilizado por já representar, de forma consolidada, o padrão de descrição de um microcontrolador STM32 dentro da ferramenta (mapa de memórias, CPU/NVIC, roteamento de interrupções e anexação de periféricos). A partir desse ponto inicial, o `stm32l496.repl` foi criado para refletir os endereços-base, a composição de periféricos e as interconexões específicas do STM32L496.

Para a criação do novo `.repl`, foi necessário criar novas implementações de modelos de periféricos em C# para acomodar as particularidades da família STM32L4, visto que nativamente não havia os periféricos implementados para esta linha de microcontroladores. O arquivo `.repl` template STM32F4 utiliza um modelo específico do F4 descrito no arquivo `.cs` (`STM32F4_RCC.cs`) e, neste trabalho, foi criada uma versão específica para o STM32L4 (`STM32L4_RCC.cs`).

No Renode, quando um periférico é específico para uma determinada família de MCU, fica explícito em seu nome para qual é, como apresentado na figura 5.1.

Diferente do periférico RCC, os blocos de memória SRAM e *flash* no `stm32f4.repl` são descritos por meio de uma classe genérica de memória mapeada (`Memory.MappedMemory`), sem especialização exclusiva para a família STM32F4, como mostra a figura 5.2, assim, o mesmo mecanismo foi mantido e reutilizado no `stm32l496.repl`, por permanecer adequado ao propósito de descrever regiões de memória simples no nível de plataforma.

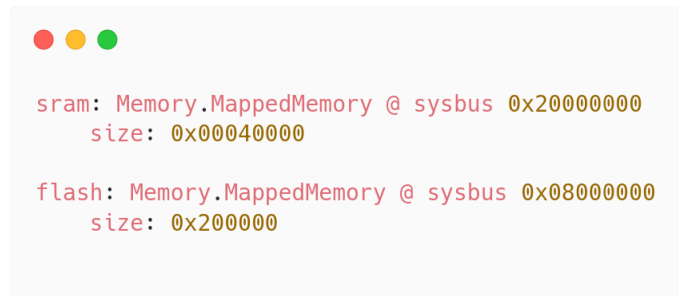
Figura 5.1: Periférico RCC do STM32F4 nativo do Renode.



```
● ● ●  
rcc: Miscellaneous.STM32F4_RCC @ sysbus 0x40023800  
rtcPeripheral: rtc  
HclkFrequencyHz: 8000000
```

Fonte: Autor.

Figura 5.2: Periférico SRAM e FLASH do STM32F4 nativo do Renode.



```
● ● ●  
sram: Memory.MappedMemory @ sysbus 0x20000000  
size: 0x00040000  
  
flash: Memory.MappedMemory @ sysbus 0x08000000  
size: 0x200000
```

Fonte: Autor.

5.3 Organização do arquivo `stm32l496.repl` por grupos funcionais de periféricos

A descrição do microcontrolador STM32L496 no Renode foi organizada em grupos, agrupando os periféricos em conjuntos funcionais coerentes com a arquitetura do dispositivo real. Essa estratégia facilita tanto a compreensão do modelo

quanto o desenvolvimento incremental e a validação isolada de cada subsistema, além de refletir a separação conceitual adotada na documentação oficial do fabricante.

Cada grupo representa um conjunto de blocos do microcontrolador que desempenham funções específicas no sistema e que, no contexto da simulação, possuem interdependências bem definidas. O arquivo completo foi anexado no apêndice A (o apêndice é composto por somente arquivos para amostra, os demais arquivos criados estão no repositório citado anteriormente), e a seguir, descrevem-se os grupos adotados no arquivo `stm321496.repl` e o papel de cada um na plataforma simulada.

5.3.1 Núcleo de Processamento e Controle de Execução

Este grupo representa o núcleo computacional do sistema e a infraestrutura mínima necessária para a execução do *firmware*.

Ele é composto pelo modelo da CPU ARM Cortex-M4 e pelo controlador de interrupções NVIC, além do temporizador de sistema (SysTick), que é implicitamente associado ao NVIC. Esses componentes são responsáveis pela execução das instruções do *firmware*, pelo tratamento de interrupções, pela temporização básica do sistema.

No arquivo `stm321496.repl`, a CPU é configurada como ARM Cortex-M4 correspondente ao STM32L4, e o NVIC é instanciado e ligado diretamente à CPU, estabelecendo o mecanismo central de entrega de interrupções provenientes dos periféricos, apresentado na figura 5.3. Esse conjunto garante que o fluxo de execução do *firmware* e o tratamento de eventos assíncronos ocorram de forma compatível com o comportamento esperado do *hardware* real.

Figura 5.3: Descrição CPU e NVIC do STM32L4 no `.repl`.



```
cpu: CPU.CortexM @ sysbus
  cpuType: "cortex-m4"
  nvic: nvic

nvic: IRQControllers.NVIC @ sysbus 0xE000E000
  priorityMask: 0xF0
  systickFrequency: 8000000
  IRQ -> cpu@0
```

Fonte: Autor.

A frequência do temporizador de sistema (*SysTick*) é configurada de forma explícita por meio do parâmetro `systickFrequency`, associado ao controlador NVIC. Essa configuração define a cadência de incremento do contador do SysTick e, consequentemente, a geração de interrupções periódicas utilizadas pelo *firmware* para temporização básica, *delays* e serviços de sistema. A definição do valor de 8 MHz para o SysTick garante consistência com o *clock* configurado no *firmware*.

de produção, assegurando que eventos temporizados, como interrupções periódicas e funções de atraso, apresentem comportamento compatível com o observado no *hardware* real.

No modelo do controlador de interrupções NVIC nativo do Renode, o parâmetro `priorityMask` define quais bits do campo de prioridade de interrupção são efetivamente considerados durante a simulação. Na arquiteturas ARM Cortex-M4, embora o campo de prioridade possua largura de 8 bits, apenas 3 bits precisam ser necessariamente implementados, os demais ficam a critério de utilização do fabricante, sendo utilizados, em geral, apenas os bits mais significativos, enquanto os demais são ignorados pelo *hardware* (Samek, 2014). No caso do MCU STM32L496, foi configurado 4 bits de prioridade, como descrito no manual de referência (STMicroelectronics, 2021), ao configurar o valor `priorityMask` como `0xF0`, indica-se que somente os quatro bits mais altos do campo de prioridade devem ser considerados, o que resulta em 16 níveis distintos de prioridade. Essa configuração força o modelo do NVIC no Renode a desconsiderar os bits não implementados, evitando discrepâncias entre os valores de prioridade escritos pelo *firmware* e o comportamento efetivo do sistema simulado.

Ainda sobre o controlador de interrupções nativo do Renode, o *index* da numerações do vetor se inicia após as entradas das interrupções de sistema, portando, no arquivo `.repl`, a referência do NVIC descrita com `NVIC@0` é correspondente a primeira interrupção de periférico.

5.3.2 Memória e Armazenamento Não Volátil

Este grupo reúne os elementos responsáveis pela organização do espaço de memória do microcontrolador e pela persistência de dados e código.

Incluem-se nesse conjunto a memória *flash* interna, a SRAM, o controlador de *flash* (com suporte à região de *option bytes*) e a área de identificação única do dispositivo (UID). No `stm32l496.repl`, cada uma dessas regiões é mapeada explicitamente no barramento do sistema (`sysbus`), respeitando os endereços-base e tamanhos definidos no manual de referência do STM32L496 (STMicroelectronics, 2021).

A *flash* armazena o código do *firmware* carregado no formato *ELF*, enquanto a SRAM é utilizada para dados voláteis e pilha de execução. O controlador de *flash* é responsável por mediar acessos à memória não volátil e por expor registradores necessários para inicialização e configuração do sistema. Já a região de UID permite que *firmwares* que dependem de identificação única do dispositivo inicializem corretamente, mesmo em ambiente simulado. A especificação destes periféricos pode ser vista na figura 5.4.

No Renode, regiões de memória são descritas por instâncias de `Memory.MappedMemory`, que representam blocos lineares de endereçamento no `sysbus`. No trecho apresentado, a *flash* interna é mapeada a partir do endereço `0x08000000` com tamanho de `0x00100000` (1 MiB) (STMicroelectronics, 2021), refletindo a região padrão de memória de programa utilizada pelo STM32L496. Esse

Figura 5.4: Descrição da *flash*, *flash_controller*, SRAM e UID do STM32L4 no *.repl*.

```
flash: Memory.MappedMemory @ sysbus 0x08000000
      size: 0x00100000

flash_controller: MTD.STM32L4_FlashController @ {
  sysbus 0x40022000;
  sysbus new Bus.BusMultiRegistration {
    address: 0x1FFF7800;
    size: 0x0F;
    region: "optionBytes" }
}
flash: flash

sram: Memory.MappedMemory @ sysbus 0x20000000
     size: 0x00050000

uid: Memory.MappedMemory @ sysbus 0x1FFF7400
   size: 0x400
```

Fonte: Autor.

mapeamento é essencial para que o carregamento do executável (*.elf*) ocorra nos endereços definidos pelo *linker script* do projeto, garantindo que as seções de código (*.text*) e de dados inicializados (*.data*) sejam posicionadas corretamente no espaço de endereços da simulação.

De forma análoga, a SRAM é declarada como um bloco de memória mapeada em *0x20000000* com tamanho de *0x00050000* (320 KiB) (STMicroelectronics, 2021), consistente com a memória volátil disponível no dispositivo. Essa região é utilizada pelo *runtime* do *firmware* para alocação de variáveis globais não inicializadas (*.bss*), pilha (*stack*) e estruturas dinâmicas, quando aplicável. Na prática, a presença de uma SRAM com tamanho coerente é requisito para a execução estável do *firmware*, uma vez que inconsistências no tamanho ou no endereço-base resultam em falhas precoces, como corrupção de pilha ou acesso inválido durante a inicialização do sistema.

Além dos blocos de memória, o trecho também explicita a instanciação do controlador de *flash* (*STM32L4_FlashController*) criado neste trabalho, no endereço *0x40022000* (STMicroelectronics, 2021), que corresponde à região de registradores do periférico FLASH no STM32L4. Esse periférico representa o comportamento de controle de memória não volátil, expondo registradores necessários para fluxos comuns do *firmware*, como configuração de latência, desbloqueio, sinalização de *busy/error* e sequências de escrita/apagamento, quando utilizadas. Ainda que a pesquisa não exija a emulação completa de todas as operações de programação da *flash*, a presença desse controlador é importante para compatibilidade binária, pois bibliotecas do ecossistema STM32 (rotinas HAL/LL e código de inicialização) acessam esses registradores durante o *boot* e em rotinas de manutenção do sistema.

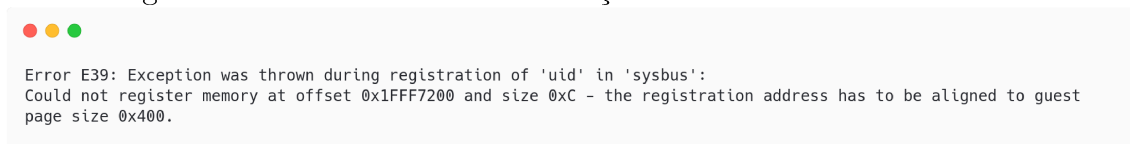
Um aspecto relevante do modelo adotado é a criação de uma se-

gunda região de mapeamento associada ao controlador de *flash* por meio de `BusMultiRegistration`, registrando o intervalo `0x1FFF7800` com tamanho de `0x0F` sob o rótulo "`optionBytes`" (STMicroelectronics, 2021). No STM32L4, os *option bytes* armazenam configurações persistentes do dispositivo, como opções de proteção, *boot configuration* e parâmetros de segurança. Na simulação, a disponibilização desse intervalo permite que o *firmware* leia esses valores em endereços coerentes com o dispositivo real, reduzindo *warnings* e falhas decorrentes de acessos a regiões não mapeadas durante rotinas de inicialização.

A linha `flash`: `flash` no bloco do `flash_controller` estabelece o vínculo entre o controlador e a memória *flash* definida anteriormente, conectando a camada de controle ao armazenamento que efetivamente contém os bytes do *firmware*. Essa ligação é importante no Renode porque separa explicitamente o *backend* de armazenamento (o bloco de memória) do periférico que controla seu uso, reproduzindo a distinção existente no microcontrolador real entre a matriz de *flash* e o periférico FLASH responsável por sua operação.

Por fim, a região `uid` foi mapeada em `0x1FFF7400` com tamanho `0x400`, diferindo do tamanho `0xC` (96 bits) especificado no manual de referência do fabricante (STMicroelectronics, 2021). Essa diferença ocorre devido a uma limitação do Renode, que exige que regiões de memória mapeada sejam registradas com tamanho alinhado ao *guest page size* da máquina (neste caso, `0x400`), resultando no erro apresentado na Figura 5.5.

Figura 5.5: Erro ao iniciar a simulação com uid com exatos 96 bits.



Fonte: Autor.

Como o `uid` não é essencial para a fidelidade da simulação do *firmware* de produção, sendo utilizado apenas como identificador, adotou-se o tamanho mínimo permitido pelo simulador. Não foi encontrada, nas documentações consultada, uma explicação para a existência dessa limitação na plataforma, o que também contribuiu para a escolha da solução adotada.

5.3.3 Gerenciamento de *clock*, energia e *reset* do sistema

Este grupo descreve os periféricos responsáveis pelo controle temporal, gerenciamento de energia e estados globais do microcontrolador.

São incluídos nesse conjunto o controlador de *reset* e *clock* (RCC), o módulo de controle de energia (PWR) e o relógio de tempo real (RTC). No modelo implementado, o RCC desempenha papel central ao controlar a habilitação de *clocks* dos barramentos e periféricos, influenciando diretamente o comportamento funcional dos demais blocos do sistema.

O periférico PWR permite que o *firmware* configure estados de energia e modos de operação compatíveis com a família STM32L4 e o RTC fornece suporte a temporização de longa duração e dependências comuns em código de inicialização. A presença desses periféricos é fundamental para satisfazer dependências do *firmware* real e garantir uma inicialização correta e livre de falhas.

Segue a seção referente a esses periféricos, figura 5.6:

Figura 5.6: Descrição RCC, PWR e RTC STM32L4 no `.rep1`.



```
rcc: Miscellaneous.STM32L4_RCC @ sysbus 0x40021000
rtcPeripheral: rtc

pwr: Miscellaneous.STM32L4_PWR @ sysbus 0x40007000
Voltage: 3.0
UseExternalSmps: false

rtc: Timers.STM32L4_RTC @ sysbus 0x40002800
AlarmIRQ -> nvic@41
WakeupIRQ -> exti@20
```

Fonte: Autor.

No PWR, os parâmetros `Voltage` e `UseExternalSmps` são utilizados para selecionar o perfil de consumo adotado pelo estimador, permitindo ajustar a estimativa de corrente e energia em função da tensão de alimentação e da presença de um regulador externo. Dessa forma, a plataforma passa a oferecer uma estimativa do consumo acumulado com base no tempo virtual decorrido e no estado global do microcontrolador, mantendo compatibilidade com a execução de binários reais sem modificações.

No STM32L4, os periféricos RCC, PWR e RTC constituem a base sobre a qual todos os demais periféricos operam, uma vez que o funcionamento de cada bloco depende da habilitação de *clocks* em seus respectivos barramentos e, frequentemente, de sequências de inicialização que envolvem estados globais do sistema. Por esse motivo, no modelo implementado no Renode, são instanciados como parte do conjunto mínimo necessário para permitir que o *firmware* inicie corretamente e avance para a configuração dos demais subsistemas.

O controlador de *Reset and Clock Control* criado neste trabalho (RCC) é mapeado no endereço-base `0x40021000` (STMicroelectronics, 2021), correspondente à região de registradores RCC no STM32L496. Na prática, o RCC atua como ponto central de habilitação de *clock*, controlando quais periféricos recebem sinal de *clock* nos barramentos AHB/APB e, conseqüentemente, determinando se eles podem ou não responder a acessos e evoluir seu estado interno. Essa característica é especialmente relevante em simulação, pois evita que periféricos operem de forma irrealista quando o *firmware* ainda não realizou a habilitação apropriada.

No trecho apresentado, o RCC também estabelece uma ligação explícita com o periférico RTC por meio do parâmetro `rtcPeripheral: rtc`. Essa associação

é importante porque, no STM32L4, o RTC é alimentado por domínios de *clock* e *reset* parcialmente separados do restante do sistema. No Renode, a referência direta do RCC ao objeto do RTC permite coordenar, de forma estruturada, a habilitação do *clock* do RTC e eventuais dependências de *reset*, garantindo que o *firmware* enxergue um caminho consistente de inicialização entre RCC e RTC, como ocorre no microcontrolador real.

O periférico de controle de energia (PWR), instanciado em `0x40007000` (ST-Microelectronics, 2021), complementa o RCC ao fornecer registradores e estados necessários para configuração de domínios de energia, modos de operação e permissões de acesso a regiões sensíveis (por exemplo, domínio *backup* e configuração de reguladores internos). A presença do PWR é relevante porque fluxos de inicialização e bibliotecas do ecossistema STM32 acessam seus registradores para liberar configurações do sistema, e porque o STM32L4 depende desse periférico em rotinas comuns de habilitação e configuração de domínios. Dessa forma, incluir o PWR no `.repl` reduz falhas de inicialização e aumenta a compatibilidade com binários não modificados. Como extensão do modelo, o PWR também foi utilizado como ponto de instrumentação para estimar consumo, integrando uma corrente típica ao longo do tempo virtual de execução e permitindo obter métricas de energia estimada ao final de um teste.

Por fim, o RTC é instanciado em `0x40002800` (STMicroelectronics, 2021) e conectado ao NVIC por meio da linha de interrupção `AlarmIRQ -> nvic@41`. Além disso, a linha `WakeupIRQ -> exti@20` conecta o evento de *wakeup* do RTC ao controlador EXTI, reproduzindo o caminho típico de sinalização do STM32L4, no qual a interrupção de *wakeup* do RTC é roteada via EXTI antes de chegar ao NVIC. Essa conexão é importante para *firmwares* que utilizam o temporizador de *wakeup* do RTC para retomar a execução após períodos de espera, pois assegura que o encadeamento de interrupções (RTC → EXTI → NVIC) esteja descrito e funcional na plataforma simulada.

Em conjunto, RCC, PWR e RTC estabelecem a infraestrutura de inicialização, habilitação de periféricos e temporização de longa duração da plataforma simulada. A inclusão explícita desses blocos no `stm32l496.repl` contribui para a execução de *firmwares* reais sem modificações, pois reproduz dependências típicas do STM32L4 relacionadas a domínios de *clock*, permissões de configuração e sinalização por interrupções em subsistemas globais. Além disso, os parâmetros adicionados ao `.repl` permitem que a plataforma seja configurada com uma referência de frequência e alimentação coerente com os testes, habilitando a coleta de métricas de consumo estimadas de forma reprodutível ao longo dos experimentos.

5.3.4 Infraestrutura de Interrupções e Eventos Externos

O principal componente desse conjunto é o controlador de interrupções externas (EXTI), que recebe sinais provenientes de pinos *GPIO*, do RTC e de outros periféricos. No `stm32l496.repl`, o EXTI é configurado com suas linhas de saída e conectado ao NVIC diretamente ou por meio de entradas combinadas (`CombinedInput`),

reproduzindo a multiplexação de interrupções existente no *hardware* real. Ao descrever endereços base e roteamentos de linhas, foi utilizado o manual de referência do microcontrolador (STMicroelectronics, 2021).

Essa infraestrutura permite que eventos externos, como mudanças de nível em pinos digitais ou sinais de dispositivos conectados, interrompam o fluxo normal de execução do *firmware*. A modelagem correta desse grupo é essencial para simular comportamento *real-time*, principalmente em aplicações orientadas a eventos, e é apresentada na figura 5.7.

Figura 5.7: Descrição EXTI do STM32L4 no `.repl`.



```
exti: IRQControllers.STM32L4_EXTI @ sysbus 0x40010400
numberOfOutputLines: 40
[0-4] -> nvic@[6-10]
[5-9] -> nvicInput23@[0-4]
[10-15] -> nvicInput40@[0-5]

16 -> nvic@1
18 -> nvic@41
19 -> nvic@2
20 -> nvic@3
21 -> nvic@64
22 -> nvic@64

nvicInput23: Miscellaneous.CombinedInput @ none
numberOfInputs: 5
-> nvic@23

nvicInput40: Miscellaneous.CombinedInput @ none
numberOfInputs: 6
-> nvic@40
```

Fonte: Autor.

No Renode, o periférico `IRQControllers.STM32L4_EXTI` criado é utilizado para modelar o controlador de interrupções e eventos externos do STM32L4, sendo mapeado no endereço `0x40010400`, coerente com a região de registradores do EXTI no microcontrolador real. Esse bloco atua como ponte entre sinais do domínio de *GPIO* (linhas externas) e o vetor de interrupções (NVIC), permitindo que alterações de nível ou borda em pinos digitais se traduzam em requisições de interrupção tratáveis pelo *firmware*. No trecho apresentado, o parâmetro `numberOfOutputLines: 40` define a quantidade de linhas de saída do EXTI consideradas no modelo, abrangendo tanto as linhas associadas diretamente aos pinos (tipicamente 0–15) quanto linhas internas reservadas para fontes como RTC, PVD e outros eventos do sistema (STMicroelectronics, 2021).

A ligação entre o EXTI e o NVIC é explicitada por meio do mapeamento de intervalos de linhas para vetores de interrupção, refletindo a forma como o STM32 agrupa e multiplexa fontes externas (STMicroelectronics, 2021). As linhas `[0-4] -> nvic@[6-10]` indicam que as interrupções EXTI0 a EXTI4 possuem vetores dedicados no NVIC (um para cada linha), permitindo tratamento individu-

alizado no *firmware*. Já as linhas [5-9] -> `nvicInput23@[0-4]` e [10-15] -> `nvicInput40@[0-5]` representam o agrupamento típico do *hardware* real, no qual múltiplas linhas EXTI compartilham um único vetor de interrupção (EXTI9_5 e EXTI15_10, respectivamente). Esse agrupamento é relevante porque, quando o *firmware* recebe a interrupção, ele deve consultar os registradores de *pending* do EXTI para identificar qual linha específica disparou o evento, reproduzindo o fluxo real de *dispatch* de interrupções em STM32. Como comentado anteriormente, a numeração do *index* do NVIC tem como início as interrupções de usuários, pulando as interrupções de sistema.

Para representar esse mecanismo de compartilhamento de vetores no Renode, foram instanciados dois componentes `Miscellaneous.CombinedInput`: `nvicInput23` e `nvicInput40`. Esses blocos atuam como multiplexadores de sinais, agregando múltiplas entradas (uma por linha EXTI do grupo) em uma única saída conectada ao NVIC. No exemplo, `nvicInput23` possui `numberOfInputs`: 5 e encaminha seu sinal agregado para `nvic@23`, enquanto `nvicInput40` possui `numberOfInputs`: 6 e encaminha para `nvic@40`. Com isso, o modelo permite que qualquer uma das linhas de um grupo dispare a interrupção compartilhada correspondente, preservando o comportamento de multiplexação observado no microcontrolador físico.

Além das linhas externas associadas a pinos, o trecho também mapeia linhas internas específicas do EXTI para vetores do NVIC, como 16 -> `nvic@1` e 18 -> `nvic@41`. No STM32L4, essas linhas são utilizadas por eventos do sistema e periféricos internos (por exemplo, RTC *alarm/wakeup* e outras fontes internas), tornando necessário que a plataforma simule o caminho completo de sinalização para que o *firmware* inicialize e opere sem modificações (STMicroelectronics, 2021). Esse mapeamento explícito reduz discrepâncias entre o comportamento da simulação e do *hardware* real, especialmente em inicializações que habilitam eventos internos do EXTI como parte do *startup*.

5.3.5 Interfaces de Comunicação Serial

Nesta seção foram descritos os periféricos responsáveis pela comunicação digital do microcontrolador com dispositivos externos.

Incluem-se aqui as interfaces UART/USART (utilizada para depuração durante o desenvolvimento) e SPI. No modelo implementado, as USARTs são utilizadas principalmente para saída de mensagens de depuração e interação com analisadores do Renode, enquanto a interface SPI permite a comunicação com módulos externos, como o rádio AT86RF233.

Cada periférico é instanciado com seus endereços base e linhas de interrupção correspondentes, garantindo que operações de transmissão, recepção e sinalização por IRQ ocorram de forma consistente, ver figura 5.8. Esse grupo é central para os testes de integração realizados no trabalho, pois permitem validar cenários reais de comunicação entre o microcontrolador e dispositivos externos.

No `stm32l496.repl`, as interfaces de comunicação serial são instanciadas di-

Figura 5.8: Descrição da UART e SPI do STM32L4 no `.repl`.

```
uart1: UART.STM32L4_UART @ sysbus <0x40013800, +0x100>
      -> nvic@37

uart2: UART.STM32L4_UART @ sysbus <0x40004400, +0x100>
      -> nvic@38

spi1: SPI.STM32L4_SPI @ sysbus 0x40013000
      IRQ -> nvic@35

spi2: SPI.STM32L4_SPI @ sysbus 0x40003800
      IRQ -> nvic@36

spi3: SPI.STM32L4_SPI @ sysbus 0x40003C00
      IRQ -> nvic@51
```

Fonte: Autor.

retamente no `sysbus`, com seus endereços-base e com parâmetros de temporização e conexões explícitas às linhas de interrupção do NVIC. Essa descrição é fundamental para garantir que o *firmware* execute sequências de inicialização e rotinas de comunicação sem qualquer adaptação, uma vez que bibliotecas STM32 (HAL/LL) assumem offsets e vetores de interrupção específicos para cada instância de periférico (STMicroelectronics, 2021).

As interfaces USART são descritas por instâncias de periféricos `STM32L4_UART` criados para este trabalho e mapeadas nos endereços `0x40013800` (USART1) e `0x40004400` (USART2), com tamanho de região `+0x100`. A notação `<base, +size>` explicita a janela de endereçamento reservada ao periférico no barramento e reforça que o modelo ocupa apenas o intervalo correspondente aos registradores relevantes. Cada USART é conectada a uma linha de interrupção dedicada no NVIC (`nvic@37` e `nvic@38`), de modo que eventos, condições de erro e sinalizações de estado possam ser tratados por interrupção pelo *firmware* (STMicroelectronics, 2021). Neste trabalho, a USART é utilizada principalmente como canal de *logging*, sendo utilizada nas simulações para melhorar a depuração dos testes.

As interfaces SPI são instanciadas com os periféricos implementados `STM32L4_SPI` nos endereços `0x40013000` (SPI1), `0x40003800` (SPI2) e `0x40003C00` (SPI3), cada uma conectada à sua linha correspondente no NVIC (`nvic@35`, `nvic@36` e `nvic@51`) (STMicroelectronics, 2021). A SPI possui papel central nesta pesquisa porque é o barramento utilizado para comunicação com o rádio AT86RF233. A conexão por interrupção é particularmente relevante em integrações realistas, pois *firmwares* frequentemente utilizam IRQs para sinalizar conclusão de transferência e demais eventos, reduzindo *polling* e aproximando o comportamento do sistema simulado ao sistema físico.

O trecho apresentado garante que as principais interfaces seriais do STM32L496 para a simulação do *firmware* de produção estejam disponíveis com endereçamento correto e com rotas de interrupção consistentes, estabelecendo a base

para os testes unitários e de integração do trabalho. Essas configurações permitem observar, evidências objetivas de comunicação (como, mensagens em analisadores UART e transações no barramento SPI) e avaliar, de forma reproduzível, o comportamento do *firmware* em cenários representativos de operação real.

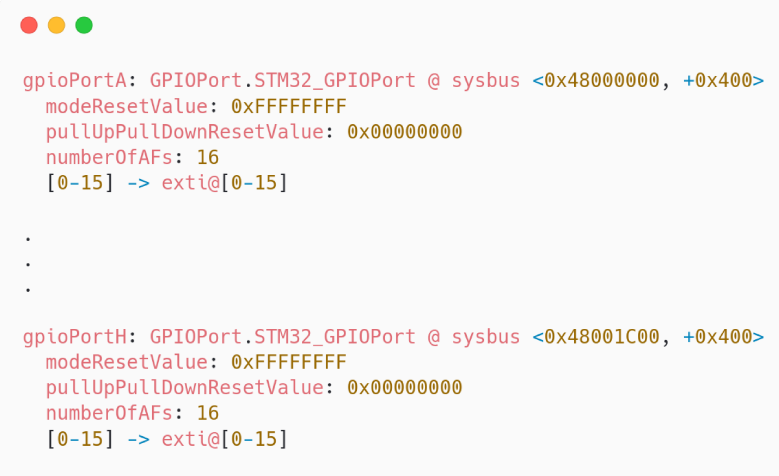
5.3.6 Grupo de Entrada e Saída Digital (GPIO)

O grupo de GPIO agrupa os controladores de portas digitais do microcontrolador.

No `stm32l496.rep1`, cada porta GPIO (A até H) é instanciada individualmente utilizando o periférico genérico de GPIOs do Renode, com configuração de número de pinos, modos de operação e suporte a funções alternativas (AF), como apresentado na figura 5.9. As linhas de cada porta são conectadas ao controlador EXTI, permitindo que eventos nos pinos digitais gerem interrupções externas. Todas as descrições foram realizadas utilizando o manual de referência (STMicroelectronics, 2021), para certificar que o endereços de memória estão corretos.

Esse grupo fornece a base para a simulação de sinais discretos, controle de *chip select*, LEDs, botões virtuais e linhas de interrupção de dispositivos externos.

Figura 5.9: Descrição dos GPIO do STM32L4 no `.rep1`.



```
gpioPortA: GPIOPort.STM32_GPIOPort @ sysbus <0x48000000, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

.
.
.

gpioPortH: GPIOPort.STM32_GPIOPort @ sysbus <0x48001C00, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]
```

Fonte: Autor.

No modelo do STM32L496, as portas *General-Purpose Input/Output* (GPIO) constituem a principal interface de entrada e saída digital do microcontrolador, sendo responsáveis tanto pelo controle de sinais discretos quanto pela multiplexação de funções alternativas associadas a periféricos como USART, SPI e *timers*. No `stm32l496.rep1`, cada porta é instanciada como `GPIOPort.STM32_GPIOPort` e registrada no `sysbus` em sua faixa de endereçamento correspondente. No trecho apresentado, observa-se a porta A mapeada em `0x48000000` e a porta H em `0x48001C00`, ambas com janela de `+0x400`, compatível com o mapa de registradores das portas GPIO na família STM32L4 (STMicroelectronics, 2021).

Os parâmetros `modeResetValue` e `pullUpPullDownResetValue` definem os valores de *reset* dos registradores de configuração de modo e resistores internos, tornando o estado inicial do periférico coerente com a condição após *reset* do sistema. Já o parâmetro `numberOfAFs: 16` indica suporte a 16 funções alternativas por pino, refletindo a capacidade típica de multiplexação dos pinos STM32 (STMicroelectronics, 2021).

Um ponto central do trecho é a ligação das linhas de pino para o controlador EXTI: `[0-15] -> exti@[0-15]`. Essa ligação estabelece que cada um dos 16 pinos de uma porta GPIO pode acionar a linha EXTI correspondente, permitindo que transições de sinal em pinos digitais gerem eventos e interrupções externas conforme configurado pelo *firmware* (STMicroelectronics, 2021). Essa conexão é particularmente relevante na integração com dispositivos externos na simulação em nível de placa, como o rádio AT86RF233.

Como as portas GPIOB e GPIOG seguem o mesmo padrão estrutural e de mapeamento, sua listagem é omitida por repetição.

Em síntese, o grupo de GPIO descrito no `stm32l496.repl` fornece a base para simular sinais discretos e multiplexação de pinos, permitindo que periféricos internos e componentes externos conectados na descrição da placa operem sobre o mesmo modelo de pinos esperado no *hardware* real.

5.3.7 Conversão Analógica e Aquisição de Dados

Os periféricos ADC representam o domínio analógico do microcontrolador e sua interface com sinais do mundo físico.

Incluem-se na figura 5.10 o bloco comum dos ADCs e os conversores individuais (ADC1, ADC2 e ADC3). Esses periféricos permitem a aquisição de sinais analógicos e sua conversão para valores digitais acessíveis ao *firmware*. No modelo implementado, os ADCs são integrados com temporizadores e com o subsistema de DMA, possibilitando cenários de amostragem acionada por eventos temporais e transferência eficiente de dados.

Figura 5.10: Descrição do ADC do STM32L4 no `.repl`.



```

adc_common: Analog.STM32L4_ADC_Common @ sysbus 0x50040300

adc1: Analog.STM32L4_ADC @ sysbus <0x50040000, +0x100>
IRQ-> nvic@18
DMARequest -> dma1@0

adc2: Analog.STM32L4_ADC @ sysbus <0x50040100, +0x100>
IRQ -> nvic@18

adc3: Analog.STM32L4_ADC @ sysbus <0x50040200, +0x100>
IRQ -> nvic@47

```

Fonte: Autor.

No `.repl`, o de ADC é representado por um bloco comum (`adc_common`),

descrito pelo periférico criado `STM32L4_ADC_Common.cs` e por três instâncias (`adc1`, `adc2` e `adc3`), descritas pelo novo periférico `STM32L4_ADC.cs`, cada uma mapeada em sua respectiva faixa de endereços no `sysbus`, conforme o mapa de memória do STM32L496 (STMicroelectronics, 2021). As linhas `IRQ -> nvic@...` associam as interrupções geradas por cada conversor ao NVIC. Observa-se que `adc1` e `adc2` compartilham o mesmo vetor (`nvic@18`), enquanto `adc3` utiliza um vetor dedicado (`nvic@47`), conforme a organização de interrupções da família STM32L4 (STMicroelectronics, 2021). Adicionalmente, a linha `DMARequest -> dma1@0` em `adc1` conecta a requisição de DMA do conversor ao canal 0 do DMA1, viabilizando a transferência automática de amostras convertidas para a memória, sendo necessário para a simulação do *firmware* de produção.

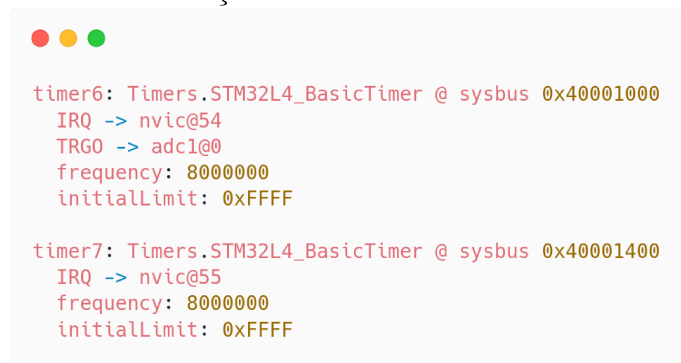
5.3.8 Temporizadores e Geração de Eventos Temporais

Este grupo é composto pelos temporizadores do microcontrolador, responsáveis pela geração de eventos periódicos e sincronização temporal.

No `stm32l496.repl`, os *timers* (TIM6 e TIM7) descritos com o periférico `STM32L4_BasicTimer.cs` criados para este trabalho, e são configurados com frequência e limites iniciais compatíveis com a configuração de *clock* do sistema, como apresentado na figura 5.11. Esses periféricos geram interrupções periódicas e podem disparar eventos internos, como sinais de *trigger* para os ADCs, no caso do *firmware* de produção.

Apesar dos *timers* 6 e 7 serem genéricos entre as famílias STM32, foi criado um periférico *timer* novo para este trabalho pois o *firmware* de produção utiliza uma característica não implementada no periférico nativo do Renode, a integração com o ADC. Como descrito na seção 4, o funcionamento da aquisição do módulo sensor é temporizada através da utilização de *timers*, e para manter a padronização do repositório, foi escolhido implementar um novo arquivo `.cs` ao invés de modificar o nativo.

Figura 5.11: Descrição dos *timers* do STM32L4 no `.repl`.



```

timer6: Timers.STM32L4_BasicTimer @ sysbus 0x40001000
  IRQ -> nvic@54
  TRG0 -> adc1@0
  frequency: 8000000
  initialLimit: 0xFFFF

timer7: Timers.STM32L4_BasicTimer @ sysbus 0x40001400
  IRQ -> nvic@55
  frequency: 8000000
  initialLimit: 0xFFFF

```

Fonte: Autor.

No contexto do STM32L4, temporizadores como TIM6 e TIM7 são frequentemente utilizados como base de tempo para rotinas periódicas, geração de *timeouts*

e disparo de eventos internos para outros periféricos. Por esse motivo, embora o STM32L496 possua diversos temporizadores, nesta pesquisa foram instanciados apenas TIM6 e TIM7, uma vez que foram os únicos efetivamente utilizados pelo *firmware* de produção.

No trecho apresentado, ambos os temporizadores são modelados como `Timers.STM32L4_BasicTimer` e registrados no `sysbus` em seus endereços-base (`0x40001000` para TIM6 e `0x40001400` para TIM7), compatíveis com o mapa de memória do STM32L496 (STMicroelectronics, 2021). Cada instância é conectada a uma linha do NVIC (`nvic@54` e `nvic@55`), permitindo que gerem interrupções periódicas e sejam tratados pelo *firmware*. A ligação do sinal `TRG0` do TIM6 ao ADC1 (`TRG0 -> adc1@0`) habilita o disparo de conversões do ADC a partir de eventos do temporizador.

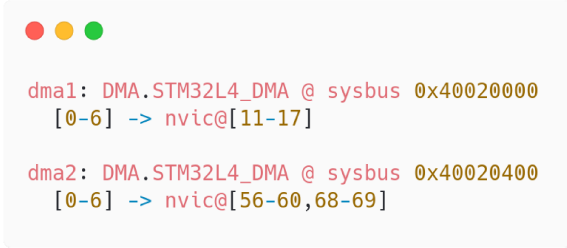
Os parâmetros `frequency: 8000000` e `initialLimit: 0xFFFF` definem, respectivamente, a frequência de referência do modelo e um limite inicial do *auto reload* coerente com a largura típica do contador em temporizadores básicos. Ao definir 8000000 Hz, a simulação mantém consistência com o *SystemCoreClock* de 8 MHz adotado nos *firmwares* de teste, reduzindo discrepâncias em rotinas de atraso e *timeouts*.

5.3.9 Acesso Direto à Memória (DMA)

Este grupo é dedicado à transferência eficiente de dados entre periféricos e memória, sem intervenção direta da CPU.

Os controladores DMA1 e DMA2 são instanciados e conectados ao NVIC, ver figura 5.12, permitindo que periféricos como ADCs, SPI e UART utilizem transferências automatizadas. No contexto da simulação, o DMA viabiliza testes de integração mais próximos de cenários reais, em que o uso intensivo de DMA é comum para otimização de desempenho.

Figura 5.12: Descrição do DMA do STM32L4 no `.repl`.



```

dma1: DMA.STM32L4_DMA @ sysbus 0x40020000
[0-6] -> nvic@[11-17]

dma2: DMA.STM32L4_DMA @ sysbus 0x40020400
[0-6] -> nvic@[56-60,68-69]
```

Fonte: Autor.

No STM32L4, os controladores de *Direct Memory Access* (DMA) realizam transferências de dados entre periféricos e memória de forma autônoma, reduzindo a necessidade de intervenção da CPU e aumentando a eficiência em operações de E/S. No `stm32l496.repl`, DMA1 e DMA2 são instanciados como periféricos `STM32L4_DMA.cs`, criados para este trabalho, e mapeados no `sysbus` em `0x40020000`

e 0x40020400, coerentes com o mapa de memória do STM32L496 (STMicroelectronics, 2021).

A ligação entre canais de DMA e o NVIC é descrita explicitamente no trecho por meio do mapeamento [0-6] -> nvic@[...]. No caso do DMA1, as interrupções associadas aos canais 0 a 6 são roteadas para nvic@11 até nvic@17. No DMA2, os canais 0 a 6 são roteados para o conjunto de vetores nvic@[56-60,68-69], conforme a distribuição de interrupções do dispositivo (STMicroelectronics, 2021).

Do ponto de vista da simulação, a inclusão do DMA permite reproduzir cenários de integração mais realistas, principalmente em ADC+DMA (utilizado no *firmware* de produção), em que o periférico emite requisições de transferência e o *firmware* sincroniza o processamento com base em interrupções de conclusão.

5.3.10 Periféricos de Suporte, Diagnóstico e Segurança

Este grupo reúne periféricos auxiliares frequentemente utilizados em inicialização, configuração do sistema e rotinas de integridade/aleatoriedade e foram exibidos na figura 5.13.

Incluem-se nesse conjunto o módulo de depuração (DBGMCU), o controlador de configuração do sistema (SYSCFG), a unidade de CRC e o gerador de números aleatórios (RNG). Esses periféricos ajudam a satisfazer dependências comuns de bibliotecas e do código de inicialização, porém foram implementados as variantes STM32L496 principalmente para reduzir *warnings* associados a acessos a registradores não mapeados.

Figura 5.13: Descrição dos periféricos DBGMCU, SYSCFG, CRC e RNG do STM32L4 no `.repl`.



```
dbg: Miscellaneous.STM32L4_DBGMCU @ sysbus 0xE0042000
syscfg: Miscellaneous.STM32L4_SYSCFG @ sysbus 0x40010000
crc: CRC.STM32_CRC @ sysbus 0x40023000
    series: STM32Series.L4
rng: Miscellaneous.STM32L4_RNG @ sysbus 0x50060800
    -> nvic@80
```

Fonte: Autor.

O DBGMCU, mapeado em 0xE0042000, é utilizado principalmente para identificação do dispositivo e ajustes de comportamento em modo *debug*, evitando falhas em rotinas que consultam esses registradores (STMicroelectronics, 2021). O SYSCFG, em 0x40010000, concentra configurações globais do sistema, incluindo o roteamento de linhas EXTI para portas GPIO, sendo comumente programado durante a habilitação de interrupções por pino (STMicroelectronics, 2021). A unidade CRC, em 0x40023000, fornece cálculo de CRC por *hardware* e é referenciada por

bibliotecas e rotinas de integridade (STMicroelectronics, 2021). Por fim, o RNG, em 0x50060800 e conectado a `nvic@80`, fornece entropia (*True Random Number Generator*) e pode sinalizar disponibilidade/estado por interrupção, mantendo compatibilidade com *firmwares* que utilizam esse recurso (STMicroelectronics, 2021).

5.3.11 Infraestrutura de Simulação e Integração com Ferramentas

Por fim, esta seção apresenta os elementos específicos do ambiente Renode que aumentam a fidelidade e a observabilidade da simulação.

Incluem-se aqui o barramento do sistema (`sysbus`), a aplicação do arquivo SVD por meio do comando `ApplySVD` e o uso de *tags* para mapeamento explícito de registradores e estados de reset, ver figura 5.14. Esses mecanismos não implementam comportamento funcional do *hardware*, mas facilitam a inspeção de registradores, a integração com ferramentas como STM32CubeMX e a análise detalhada do comportamento do *firmware* durante a execução.

Esse grupo é fundamental para tornar a plataforma simulada utilizável como ferramenta de desenvolvimento, depuração e validação, além de reforçar o caráter de gêmeo digital do microcontrolador STM32L496.

Figura 5.14: Utilização do SVD e *tags* do STM32L4 no `.repl`.



```
sysbus:
init:
  ApplySVD @platforms/svd/STM32L496.svd
  Tag <0x40022010, 0x4002203F> "FLASH:RESET" 0x40000000
  Tag <0x50000010, 0x5000003F> "USB:RESET" 0x40000000
```

Fonte: Autor.

O comando `ApplySVD @platforms/svd/STM32L496.svd` instrui o Renode a carregar um arquivo CMSIS-SVD associado ao STM32L496. Esse arquivo descreve, de forma estruturada, os periféricos e seus registradores (endereços base, offsets, campos e nomes), permitindo que ferramentas internas do Renode apresentem os registradores de maneira legível e hierarquizada durante a depuração. É importante destacar que o SVD não implementa comportamento, ele atua como um artefato de metadados que auxiliam a inspeção e a rastreabilidade de acessos do *firmware*. Na prática, ao se utilizar o SVD, torna-se possível correlacionar endereços acessados pelo binário com nomes simbólicos (por exemplo, `GPIOx_MODER`, `RCC_AHB2ENR`) e observar *bitfields* com maior clareza, o que facilita diagnósticos e validações durante o desenvolvimento dos modelos.

O trecho também utiliza o mecanismo `Tag` do Renode, que permite rotular intervalos de endereços no barramento com um nome e um valor associado. Nesse contexto, as *tags* foram empregadas com dois propósitos principais, aumentar a legibilidade e a capacidade de inspeção de regiões específicas de registradores e fornecer

valores de referência (por exemplo, valores pós-*reset*) para regiões que o *firmware* e ferramentas do ecossistema STM32 consultam de forma recorrente.

As duas *tags* "FLASH:RESET" e "USB:RESET" são utilizadas como ajustes de compatibilidade, pois certos *firmwares* gerados por STM32CubeMX e bibliotecas associadas realizam leituras ou assumem valores específicos em regiões relacionadas a reset/estado de periféricos, mesmo quando tais periféricos não são o foco do estudo. Ao marcar intervalos como <0x40022010, 0x4002203F> (relacionado ao periférico FLASH) e <0x50000010, 0x5000003F> (relacionado a USB) e associar o valor 0x4000000, a plataforma fornece um estado inicial coerente para essas regiões, reduzindo a ocorrência de *warnings* e falhas por acesso a registradores não inicializados. Esse tipo de ajuste é útil quando o objetivo é executar *firmware* não modificado: ainda que o periférico USB não seja utilizado no experimento, o *startup code* necessita em registradores dessa região durante a inicialização.

Em síntese, o bloco `sysbus:init` atua como uma camada de instrumentação e compatibilidade da plataforma: o `ApplySVD` melhora a observabilidade ao fornecer nomes e estrutura para registradores, enquanto as *tags* adicionam pontos de referência e valores iniciais para endereços críticos, auxiliando depuração e reduzindo discrepâncias durante o *boot*. Esses mecanismos não substituem os modelos comportamentais em `.cs`, mas fortalecem a usabilidade do gêmeo digital como ambiente de desenvolvimento e validação, tornando a execução e inspeção do *firmware* no Renode mais próxima do fluxo observado em plataformas físicas.

5.4 Arquivo `stm32l496_kit.repl`: composição de placa e anexação do rádio

O arquivo `stm32l496_kit.repl` descreve uma placa baseada no microcontrolador, reutilizando o microcontrolador previamente definido:

```
using "platforms/cpus/stm32l496.repl"
```

A partir dessa base, foram adicionadas conexões para anexar o rádio AT86RF233 a uma instância SPI e integrar sinais de controle/IRQ por *GPIO*, mantendo aderência ao mapeamento de pinos definido no arquivo `.ioc` do projeto.

5.4.1 Instanciação do rádio no barramento SPI

O rádio foi instanciado como `Wireless.AT86RF233 @ spi3`, definindo que sua comunicação principal ocorre via SPI3. Esse padrão reflete o uso típico do AT86RF233 em *hardware* real, no qual comandos, acesso a registradores e transferência de quadros são realizados via SPI.

5.4.2 Arquivo `stm32l496_kit.repl` completo

A seguir, na figura 5.15, apresenta-se o trecho completo utilizado para a composição da placa e para a anexação do rádio, incluindo as conexões no barramento

SPI e as ligações de controle e interrupção por *GPIO*:

Figura 5.15: Trecho do arquivo `stm32l496_kit.repl`: anexação do rádio AT86RF233 e conexões por *GPIO*.

```
using "platforms/cpus/stm32l496.repl"

radio: Wireless.AT86RF233 @ spi3
  IRQ -> gpioPortA@9

gpioPortA:
  15 -> radio@0

gpioPortB:
  6 -> radio@1
  7 -> radio@2
```

Fonte: Autor.

5.4.3 Conexões por *GPIO* (controle e interrupção)

Além do SPI, o rádio requer linhas adicionais de controle e sinalização. No `.repl`, essas ligações foram representadas conforme o mapeamento de pinos do projeto:

- **CS (pino A15):** `gpioPortA: 15 -> radio@0`.
Conecta o sinal `RF_CS` do microcontrolador (*chip select*), utilizado para seleção do dispositivo no barramento SPI.
- **IRQ (pino A9):** `radio: IRQ -> gpioPortA@9`.
Conecta o sinal `RF_IRQ` do rádio ao microcontrolador, permitindo que o *firmware* reaja a eventos de RX/TX via EXTI/NVIC.
- **SLP_TR (pino B6):** `gpioPortB: 6 -> radio@1`.
Conecta o sinal `SLP_TR`, utilizado para controle de estado do transceptor.
- **RESET (pino B7):** `gpioPortB: 7 -> radio@2`.
Conecta o sinal de *reset* do rádio, utilizado para reinicialização do dispositivo.

5.5 Arquivos de descrição de periférico (`.cs`)

Após a definição estrutural da plataforma por meio dos arquivos `.repl`, o próximo passo consistiu em implementar os arquivos de descrição de periférico (`.cs`), responsáveis por materializar o comportamento dos blocos internos do microcontrolador no ambiente do Renode.

5.5.1 O papel dos `.cs` na simulação e utilização do STM32L496

No contexto desta pesquisa, os arquivos de descrição de periférico (`.cs`) foram o mecanismo central para viabilizar a execução de *firmwares* do STM32L496 sem modificações no Renode. Como a família STM32L4 não possuía suporte nativo suficiente para o cenário alvo, foi necessário implementar modelos específicos de periféricos, cada um codificado individualmente como um novo `.cs` e referenciado pelo `stm32l496.repl` no endereço-base correspondente.

A função desses arquivos na simulação do STM32L496 pode ser organizada em quatro responsabilidades principais:

1. **Compatibilidade por mapa de registradores:** definir offsets, campos e valores de *reset* mínimos para que rotinas de inicialização (HAL/LL, *startup code*) consigam configurar *clocks*, *GPIOs* e interfaces seriais sem acessar endereços inexistentes. Seguindo as diretrizes do Renode, não é necessário implementar todos os registradores, mas sim priorizar aqueles efetivamente exercitados pelo software, mantendo a modelagem validável por testes (Renode Project, 2026c).
2. **Semântica funcional e efeitos colaterais:** associar a cada registrador os efeitos esperados pelo *firmware* (bits de *enable*, *busy/ready*, *status flags* e registros de dados), de modo que leituras e escritas produzam respostas coerentes.
3. **Interrupções e *GPIO* como mecanismo de eventos:** expor linhas de interrupção e sinais digitais como propriedades/portas do periférico e acioná-las quando condições internas ocorrerem. Na prática, isso permite que o caminho *periférico* → NVIC (ou *GPIO* → EXTI → NVIC) opere como no *hardware* real, habilitando testes orientados a evento (como o IRQ do AT86RF233 chegando ao STM32 via EXTI). A própria descrição de plataforma do Renode trata conexões de *GPIO*/IRQ de forma uniforme via o operador `->`, tornando essa integração explícita na topologia da simulação (Renode Project, 2026a).
4. **Coerência temporal:** para periféricos que evoluem no tempo (como *timers*, RTC e fluxos de comunicação), a lógica do `.cs` deve produzir eventos e estados em cadência consistente com a configuração do sistema simulado, de forma que *timeouts*, interrupções periódicas e sequências de inicialização não se tornem incorretas.

A implementação dos `.cs` seguiu a mesma filosofia incremental descrita na visão geral do método: cada periférico foi inicialmente implementado com um núcleo mínimo (registradores essenciais e comportamento necessário para avançar o *boot*) e, em seguida, expandido conforme os *firmwares* de teste revelavam novas dependências (novos campos, semânticas de *flag*, interrupções adicionais e integrações com DMA/EXTI). Esse processo facilita atribuir falhas a causas específicas, pois cada incremento é validado por evidências objetivas (mensagens UART, comunicação SPI observadas, interrupções disparadas e redução de *warnings* de acesso no *sysbus*).

Assim, os arquivos de descrição de periférico (`.cs`) cumprem o papel de transformar a descrição estrutural (`.repl`) em um microcontrolador funcionalmente simulável: são eles que garantem que o binário enxergue registradores e comportamentos compatíveis com o STM32L4, que as integrações ocorram com roteamento correto e que o ambiente do Renode possa ser utilizado como base para depuração e validação do *firmware* alvo.

5.5.2 Base utilizada de modelos STM32 já existentes

De forma análoga ao que foi realizado na criação dos arquivos `.repl`, a implementação dos modelos `.cs` partiu da utilização de modelos já disponíveis no ecossistema do Renode como referência, especialmente os referentes a microcontroladores da ST já suportados nativamente, com destaque para a família STM32F4. Essa decisão foi adotada por dois motivos principais: os modelos existentes refletem padrões consolidados de implementação (estrutura de registradores, uso do *Register Framework*, estilo de *logging* e integração com o *sysbus*), e periféricos homólogos entre famílias STM32 compartilham conceitos arquiteturais e fluxos de uso (como habilitação de *clock* via RCC, geração de IRQs e semântica de registradores de status/controle), o que torna os arquivos nativos bases sólidas para referência na criação dos modelos de periféricos do trabalho.

No repositório do Renode, os modelos em C# são mantidos no submódulo `src/Infrastructure`, que referencia o repositório `renode-infrastructure`. Essa organização separa o núcleo do simulador das implementações de periféricos e da infraestrutura de emulação, permitindo evolução modular dos modelos e simplificando a manutenção do conjunto de periféricos suportados.

Como exemplo desse processo, o arquivo `STM32F4_RCC.cs` foi utilizado como referência para a criação do modelo `STM32L4_RCC.cs`. Embora ambos representem o periférico *Reset and Clock Control*, a família STM32L4 apresenta diferenças relevantes no mapa de registradores, em campos de habilitação de *clock* (AHB/APB), em opções de osciladores e em aspectos de inicialização associados a domínios específicos. Assim, o novo modelo do RCC foi criado utilizando o conjunto de registradores e *bitfields* coerentes com o STM32L4, e das rotinas internas e dos efeitos colaterais relevantes para a simulação.

5.5.3 Relação entre os `.cs` implementados e o *firmware* alvo

A seleção de periféricos implementados e o nível de detalhe de cada modelo foram guiados por requisitos observados no *firmware* e nos *firmwares* auxiliares de teste utilizados na validação incremental.

Essa escolha implica que determinados periféricos podem apresentar implementação parcial, ou ainda que certos registradores/campos existam apenas como *stubs* (como campos reservados, ignorados ou tratados apenas para evitar acessos inválidos), caso não sejam exercitados pelos cenários de teste e pelo *firmware* alvo. Em outras palavras, a modelagem foi orientada por evidências: funcionalidades foram

implementadas quando sua ausência gerava falhas, *warnings* persistentes, divergências observáveis de comportamento ou impedia a progressão do *boot* e dos fluxos de execução do sistema.

Do ponto de vista de engenharia de simulação, essa abordagem é compatível com o fluxo recomendado para modelagem de periféricos: em vez de buscar uma implementação completa e difícil de validar, implementa-se inicialmente o subconjunto de registradores e comportamentos efetivamente utilizados e passíveis de teste, expandindo o modelo conforme novas dependências surgem ao executar binários mais complexos.

5.5.4 Estrutura de um modelo de periférico em C# no Renode

Como citado anteriormente, no Renode, um periférico escrito em C# é representado por uma classe que implementa a interface `IPeripheral` e, para ser mapeável no barramento do sistema, deve implementar ao menos uma interface de largura de acesso, como `IBytePeripheral`, `IWordPeripheral` ou `IDoubleWordPeripheral`, correspondendo a acessos de 8, 16 e 32 bits, respectivamente (Renode Project, 2026c).

Embora seja possível implementar a lógica de leitura e escrita manualmente, o padrão recomendado é utilizar o *Register Framework*, no qual o periférico mantém uma coleção de registradores (como `DoubleWordRegisterCollection`) e define os campos por meio de primitivas como `WithFlag`, `WithValueField` e `WithEnumField`, incluindo modos de acesso (*Read*, *WriteOneToClear*, *ReadToClear*) e *callbacks* para refletir o comportamento de *hardware* (Renode Project, 2026c).

Uma diretriz central é a criação de um `enum` (`Registers`) contendo os registradores suportados e seus offsets relativos ao início do bloco do periférico, pois o mesmo tipo de periférico pode existir em múltiplas instâncias na plataforma (Renode Project, 2026c). Campos não implementados foram marcados como reservados, ignorados ou anotados com *tags* para melhorar a rastreabilidade em logs (Renode Project, 2026c).

5.5.5 Detalhamento do modelo (.cs)

Nesta parte apresenta-se um exemplo completo de modelagem de periférico `STM32L4_SPI.cs` em C# no Renode, por ser um bloco compacto, amplamente utilizado no *firmware* de produção. Como o arquivo criado é extenso, foi apresentado somente as partes relevantes para as explicações. O arquivo completo foi adicionado no apêndice B (o apêndice é composto por somente arquivos para amostra, os demais arquivos criados estão no repositório citado anteriormente).

Mapa de memória e endereços do periférico

No STM32L4, as instâncias do SPI são mapeadas no barramento APB. No modelo adotado, consideram-se as três instâncias utilizadas na plataforma simulada:

- **SPI1:** 0x40013000
- **SPI2:** 0x40003800
- **SPI3:** 0x40003C00

A Tabela 5.1 lista o conjunto de registradores do bloco SPI (offsets) e seus endereços absolutos para o SPI1. Para SPI2 e SPI3, os endereços são obtidos substituindo-se o endereço-base (STMicroelectronics, 2021).

Tabela 5.1: Mapa de registradores do SPI (endereços calculados para SPI1 em 0x40013000).

Registrador	Offset	Endereço (SPI1)
CR1	0x00	0x40013000
CR2	0x04	0x40013004
SR	0x08	0x40013008
DR	0x0C	0x4001300C
CRCPR	0x10	0x40013010
RXCRCR	0x14	0x40013014
TXCRCR	0x18	0x40013018
I2SCFGR	0x1C	0x4001301C
I2SPR	0x20	0x40013020

No código, figura 5.16, esses offsets aparecem consolidados em um `enum` (ou constantes), garantindo rastreabilidade direta com o mapa de memória do manual:

```
private enum Registers : long
{
    CR1      = 0x00,
    CR2      = 0x04,
    SR       = 0x08,
    DR       = 0x0C,
    CRCPR    = 0x10,
    RXCRCR   = 0x14,
    TXCRCR   = 0x18,
    I2SCFGR  = 0x1C,
    I2SPR    = 0x20,
}
```

Figura 5.16: Offsets de registradores do SPI no arquivo STM32L4_SPI.cs.

Cabeçalho do arquivo e estrutura da classe STM32L4_SPI

O arquivo `STM32L4_SPI.cs` segue o padrão dos periféricos do Renode: diretivas `using` para os *namespaces* do simulador, seguidas da declaração da classe do periférico, integração ao barramento (`sysbus`) e inicialização de uma coleção de registradores com seus *callbacks* de leitura/escrita, apresentado na figura 5.17.

```
using Antmicro.Renode.Core;
using Antmicro.Renode.Peripherals.Bus;
using Antmicro.Renode.Peripherals;
using Antmicro.Renode.Logging;

public sealed class STM32L4_SPI : BasicDoubleWordPeripheral, IKnownSize
{
    public STM32L4_SPI(IMachine machine) : base(machine)
    {
        DefineRegisters();
        Reset();
    }

    public long Size => 0x400; //janela do bloco no sysbus

    public override void Reset()
    {
        //reset de flags
        base.Reset();
    }

    private void DefineRegisters()
    {
        //CR1/CR2/SR/DR...
    }
}
```

Figura 5.17: Estrutura típica de classe de periférico no Renode para o SPI (trecho reduzido).

Descrição de registradores

A seguir, na figura 5.18, apresenta-se um trecho mostrando como campos relevantes foram descritos. Na prática, selecionam-se os bits utilizados pelos *firmwares* testados. E para identificação com o manual de referência da STM32L496 (STMicroelectronics, 2021), a figura 5.19 mostra no *reference manual* o mesmo registrador, no recorte também é possível verificar o *offset* previamente mostrado.

Figura 5.18: Definição de registradores no `STM32L4_SPI.cs` (trecho reduzido).

```

private void DefineRegisters()
{
    Registers.CR1.Define(registers)
        .WithFlag(15, name: "BIDIMODE")
        .WithFlag(14, name: "BIDIOE")
        .WithFlag(13, name: "CRCEN")
        .WithFlag(12, name: "CRCNEXT")
        .WithFlag(11, name: "CRCL")
        .WithFlag(10, name: "RXONLY")
        .WithFlag(9, name: "SSM")
        .WithFlag(8, name: "SSI")
        .WithFlag(7, name: "LSBFIRST")
        .WithFlag(6, out spiEnable, name: "SPE")
        .WithValueField(3, 3, name: "BR")
        .WithFlag(2, name: "MSTR")
        .WithFlag(1, name: "CPOL")
        .WithFlag(0, name: "CPHA");
}

```

Fonte: Autor.

Figura 5.19: Recorte dos detalhes do registrador no manual de referência.

41.6.1 SPI control register 1 (SPIx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDIOE	CRC EN	CRCN EXT	CRCL	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR[2:0]			MSTR	CPOL	CPHA
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Fonte: Recorte do manual de referência da STMicroelectronics.

Funções principais da classe SPI (visão breve)

Após o mapeamento de registradores, o comportamento do periférico é consolidado em poucas rotinas, em geral:

- **Reset()**: aplica estados de *reset* coerentes com o *power-on reset* (flags, buffers e registradores).
- **DefineRegisters()**: descreve offsets, *bitfields* e ações colaterais de leitura/escrita.
- Rotinas auxiliares de **transmissão/recepção** (ex.: enfileirar TX, produzir RX, atualizar TXE/RXNE/BSY).
- Rotinas de **interrupção** (quando implementadas): atualização de linha/IRQ a partir de TXEIE/RXNEIE e flags do SR.

5.5.6 Demais periféricos implementados (apenas visão das funções principais)

Após o exemplo completo do SPI, os demais arquivos `.cs` foram descritos de forma objetiva na tabela 5.2, destacando somente as rotinas que concentram o comportamento funcional empregado nos testes.

Tabela 5.2: Periféricos STM32L4 implementados no Re-node

Periférico (.cs)	Implementações no modelo
STM32L4_RCC.cs	<code>DefineRegisters()</code> , <code>Reset()</code> , rotinas de atualização do estado de <i>clock</i> (habilitação de periféricos e seleção de fontes) e <i>stubs</i> para bits de <i>enable</i> de blocos não utilizados.
STM32L4_PWR.cs	<code>DefineRegisters()</code> para controle de energia (modos/ <i>flags</i>) e <i>stubs</i> suficientes para não bloquear inicializações típicas do <i>firmware</i> . Criação do estimador de consumo/energia ao longo do tempo virtual de simulação, selecionando perfis típicos de corrente conforme o modo global (por exemplo, <i>run/sleep</i>) e parâmetros de alimentação configurados, com base em dados do datasheet (STMicroelectronics, 2025).
STM32L4_UART.cs	<code>DefineRegisters()</code> , rotinas de TX/RX (buffers) e atualização de <i>flags</i> (TXE/RXNE/TC) e interrupções associadas.
STM32L4_ADC_Common.cs	<code>DefineRegisters()</code> do bloco comum (como <i>clock</i> /prescaler/recursos compartilhados) e <i>stubs</i> para campos não usados nos cenários de teste.
STM32L4_ADC.cs	<code>DefineRegisters()</code> , rotina de início de conversão e alimentação de DR, além de integração com DMA quando necessária.
STM32L4_DMA.cs	<code>DefineRegisters()</code> , configuração de canal, disparo de transferências (memória e periférico) e sinalização de <i>flags</i> /interrupções de término.
STM32L4_BasicTimer.cs	<code>DefineRegisters()</code> , atualização do contador e eventos (<i>overflow/update</i>), e de geração de sinais internos (<i>triggers</i>).
STM32L4_DBGMCU.cs	<code>DefineRegisters()</code> e bits de congelamento de periféricos (timers, RTC, IWDG/WWDG, I2C, CAN, LPTIM) para compatibilidade com rotinas de inicialização.

Periférico (.cs)	Implementações no modelo
STM32L4_SYSCFG.cs	DefineRegisters() para roteamento de EXTI e configurações de sistema usadas na inicialização (como seleção de portas para linhas EXTI).
STM32L4_RNG.cs	DefineRegisters(), habilita o gerador por RNGEN e controla interrupção por IE, e retorna valor pseudoaleatório de 32 bits quando habilitado
STM32L4_FlashController.cs	DefineRegisters() (por exemplo, ACR/SR/CR) e stubs para latência e flags utilizadas em rotinas de startup.
STM32L4_EXTI.cs	DefineRegisters() (IMR/EMR/RTSR/FTSR/PR) e rotinas para pendência e limpeza de interrupções externas.

Como o desenvolvimento foi orientado aos requisitos observados no *firmware* do laboratório e nos *firmwares* de teste, algumas funcionalidades do microcontrolador podem estar implementadas apenas parcialmente ou ausentes quando não foram necessárias para os cenários avaliados, preservando o escopo verificável e incremental adotado na metodologia.

5.6 Extensões no modelo AT86RF233.cs

O Renode possui uma implementação nativa do AT86RF233, entretanto, no momento de desenvolvimento deste trabalho, o modelo existente não era suficientemente completo para reproduzir todas as tarefas esperadas pelo *firmware* de produção. Por esse motivo, foi necessário aprimorar o arquivo .cs do módulo de rádio, adicionando registradores e rotinas de comportamento com base no *datasheet* do AT86RF233 (Atmel Corporation, 2014). O objetivo dessas alterações foi compatibilizar o modelo com os padrões de acesso SPI empregados pelo *firmware* e reduzir falhas decorrentes de leituras/escritas em endereços não modelados.

Do ponto de vista arquitetural, o AT86RF233 opera como transceptor IEEE 802.15.4 e disponibiliza ao microcontrolador um *frame buffer* acessado via SPI, quadros recebidos pelo de rádio são armazenados internamente e, em seguida, lidos pelo MCU por meio de transações SPI específicas. O formato adotado nesse caminho segue a organização típica do PHY/MAC do IEEE 802.15.4, em que um campo de comprimento (*PHR*) antecede os bytes do quadro (*PSDU*/MPDU) e, ao final, encontra-se o campo de verificação de integridade (*FCS*) com 16 bits (CRC) (Atmel Corporation, 2014).

5.6.1 Implementações no AT86RF233.cs

O modelo AT86RF233 disponível nativamente no Renode foi modificado para incluir registradores não modelados. As principais mudanças foram:

1. ***Shadow registers (0x00–0x3F)***

Foi criado `shadowRegisters = new byte[0x40]` para armazenar valores escritos pelo *firmware* e permitir leituras coerentes (reduzindo *warnings*).

2. ***Reset também zera o estado dos registradores***

Além de resetar o estado interno, passou a executar `Array.Clear(shadowRegisters, ...)`.

3. ***Escritas adicionais passaram a ser aceitas e armazenadas***

Foram adicionados casos que salvam em *shadow registers* para: `PHY_TX_PWR`, `CCA_THRES`, `IRQ_MASK`, `SHORT_ADDRO/1`, `PAN_IDO/1`, `XAH_CTRL1`, `CSMA_SEED0`, `XAH_CTRL2`.

4. ***enum Registers ampliado***

Incluiu os endereços usados pelo *firmware*: `0x05`, `0x09`, `0x0D`, `0x0E`, `0x20–0x23`, `0x2C`, `0x2D`, `0x2F`.

5. ***Frame buffer mais robusto***

Proteção para leitura além do final do *frame*: retorna 0 e reseta `accessMode/context`.

O PHR passou a alocar `data + 2 bytes` (reserva para CRC).

6. ***IRQ também após TX***

Após `SendPacket()`, a implementação também chama `IRQ.Set()`.

7. ***Funções auxiliares***

Adicionados `SendTestFrame()` e `SendFwlDataRequestEmpty()`.

5.7 Estimativa de consumo energético durante a simulação

Além da validação funcional dos periféricos, a plataforma foi estendida com um mecanismo de estimativa de consumo energético aplicado ao microcontrolador STM32L496 durante a execução no Renode. A estimativa é baseada em métricas típicas de corrente extraídas da folha de dados (*datasheet*) do dispositivo, considerando a dependência do consumo com frequência de *clock*, temperatura e modo de operação (por exemplo, estados de execução e modos de baixo consumo) (STMicroelectronics, 2025). No modelo, essas referências de consumo são associadas ao estado global do controlador observado via registradores e *flags* relevantes (principalmente de RCC e PWR), permitindo inferir, ao longo da simulação, em qual regime o microcontrolador se encontra em cada trecho de tempo.

A partir do tempo virtual decorrido na simulação e da identificação do regime ativo (frequência configurada e modo de energia corrente), o estimador acumula o consumo ao longo da execução, produzindo métricas agregadas (por exemplo, carga

e energia estimadas) associadas a cada experimento. Assim, torna-se possível relacionar a evolução do consumo estimado com o estágio de execução do *firmware* (inicialização, aquisição, transmissão, espera/*sleep*), mantendo o processo reproduzível e observável. Os resultados de consumo estimado obtidos com esse mecanismo são apresentados e discutidos na seção de resultados.

Por escopo, o estimador foi aplicado somente ao controlador STM32L496 (microcontrolador) e não abrangeu o consumo simulado/estimado do módulo de rádio AT86RF233, uma vez que esta implementação teve como objetivo principal validar o conceito de estimação de consumo no Renode e aprofundar a precisão no nível do microcontrolador.

5.8 Scripts de Execução (.resc) criados

Além dos arquivos `.repl`, que descrevem estruturalmente a plataforma (instâncias de periféricos, endereços base e conexões), o Renode utiliza scripts `.resc` para automatizar a inicialização e a execução de uma simulação. Um arquivo `.resc` contém uma sequência de comandos equivalentes aos do *Monitor* do Renode, permitindo encapsular etapas repetitivas (criação da máquina, carregamento do `.repl`, carregamento do binário, configuração de logs e execução) e, assim, garantir reprodutibilidade nos experimentos e validações.

No contexto deste trabalho, scripts `.resc` foram utilizados para instanciar a máquina virtual correspondente ao “kit” descrito em `stm32l4-kit_v2.repl`, parametrizar o caminho do *firmware* em formato `.elf`, ajustar níveis de *logging* por periférico (reduzindo ruído em módulos não relevantes a um cenário específico) e definir macros de *reset*/carga para agilizar ciclos iterativos de depuração e validação.

Apresentado na figura 5.20, foi apresentado um arquivo `.resc` padrão:

Figura 5.20: Exemplo de script `.resc` para inicialização da simulação e carga do `.elf`.

```
using sysbus

$bin?=@/home/narimatsu/Documents/lmest/fw-sens-v3-full/sensor/Debug/sensor.elf

mach add "board_vulkan" machine
machine LoadPlatformDescription @platforms/boards/stm32l4-kit_v2.repl

logFile @/home/narimatsu/Documents/renode_git/renode_logs/output.log
logLevel 1
sysbus.spi3.radio logLevel -1
sysbus.adc1 logLevel -1
sysbus.dma1 logLevel -1

macro reset_vulkan ""
sysbus LoadELF $bin
""
runMacro $reset_vulkan
```

Fonte: Autor.

Declaração de contexto e parametrização do binário

O comando `using sysbus` estabelece um *prefixo* padrão para acesso aos objetos conectados ao barramento do sistema, reduzindo a verbosidade ao interagir com periféricos e com o próprio `sysbus`. Em seguida, a variável `$bin` armazena o caminho do *firmware* alvo (arquivo `.elf`). A forma `$bin?=` define um valor *default* apenas se a variável ainda não tiver sido definida anteriormente, o que permite reutilizar o mesmo script com diferentes binários (por exemplo, sobrescrevendo `$bin` via linha de comando ou por um script que inclua este arquivo).

Criação da máquina e carregamento da descrição de plataforma

O bloco `mach add "board_vulkan"machine` cria uma nova máquina de simulação e a seleciona como contexto corrente. Em seguida, `machine LoadPlatformDescription` carrega a descrição estrutural da plataforma definida no arquivo `.repl`. Nesta etapa, são instanciados CPU, memórias e periféricos e são aplicados os endereços base e as conexões conforme o mapa de memória do microcontrolador e a integração de placa definida (por exemplo, MCU + periféricos externos). Assim, o `.resc` atua como camada operacional que “liga” a plataforma `.repl` e prepara o ambiente para execução do *firmware*.

Configuração de *logging* global e por periférico

O comando `logFile` direciona a saída de logs também para um arquivo, permitindo análise posterior de execuções longas e a preservação de evidências de validação. O comando `logLevel` ajusta o nível mínimo de prioridade, enquanto invocações do tipo `<objeto> logLevel <valor>` permitem refinar o nível de log por periférico. No exemplo, periféricos como `spi3.radio`, `adc1` e `dma1` são configurados para exibirem todas as mensagens de *debug* ao usar `logLevel -1`, abrangendo todos os níveis de mensagem de *log* quando esses módulos são foco do cenário em análise. Essa prática é útil para depuração incremental, pois facilita isolar interações relevantes (por exemplo concentrar logs no barramento SPI ou em uma UART específica).

Macros e ciclo de carga/reset

Scripts `.resc` suportam a definição de macros por meio do comando `macro`, armazenando um bloco de comandos entre aspas triplas (`"... "`). No exemplo, a macro `reset_vulkan` encapsula a etapa de carga do binário no barramento do sistema com `sysbus LoadELF $bin`. Em fluxos iterativos, isso simplifica o reinício do experimento: o pesquisador pode recarregar rapidamente o `.elf` (ou variar `$bin`) e reiniciar a execução sem repetir manualmente toda a sequência de comandos. Por fim, `runMacro` executa o conteúdo previamente definido, colocando a simulação em estado pronto para iniciar/retomar a execução (comumente seguido do comando `start` quando se deseja iniciar automaticamente a emulação no fim do script).

5.9 Desenvolvimento e geração dos *firmwares* com STM32CubeIDE

O desenvolvimento dos *firmwares* utilizados na validação da plataforma simulada foi realizado com o STM32CubeIDE. Essa ferramenta foi empregada em quatro frentes principais:

1. como ambiente de desenvolvimento integrado (edição, organização e gerenciamento do projeto);
2. como interface de configuração e geração automática de código a partir de arquivos `.ioc` (STM32CubeMX integrado);
3. como ferramenta de compilação e *build* dos binários executáveis;
4. como ferramenta de *debugging*.

A partir da configuração no `.ioc` (seleção do microcontrolador, clocks, pinos e periféricos), o STM32CubeIDE gera a estrutura do projeto e os arquivos de inicialização correspondentes, permitindo que o *firmware* seja desenvolvido de forma incremental, mantendo rastreabilidade entre a configuração de *hardware* e o código produzido. Ao final do processo de compilação, o ambiente produz o artefato `.elf`, que inclui tanto o código executável quanto as informações de depuração (por exemplo, símbolos), sendo este o formato utilizado como entrada na simulação.

Os arquivos `.elf` gerados pelo STM32CubeIDE foram carregados no Renode para execução na máquina virtual descrita pelos arquivos `.rep1` e pelos modelos de periféricos em C#. Dessa forma, o mesmo *firmware* compilado para o alvo STM32L4 pode ser executado sem alterações no ambiente de simulação, preservando o fluxo de desenvolvimento típico do microcontrolador real.

5.9.1 Depuração com GDB integrada ao Renode

Além do *build* dos *firmwares*, o STM32CubeIDE também foi utilizado como ferramenta de depuração por meio do protocolo GDB. O Renode provê uma interface de *debug* compatível com GDB, permitindo a conexão do depurador ao processador simulado e a realização de análises mais detalhadas durante a execução do *firmware*.

Essa integração viabilizou depurações mais complexas, incluindo a utilização de *breakpoints*, execução passo a passo, inspeção de registradores da CPU, leitura/escrita de memória e observação do fluxo de execução em trechos críticos do código. Como o `.elf` contém símbolos de depuração, a análise pôde ser realizada em nível de código-fonte, facilitando a identificação de comportamentos inesperados no *firmware* durante a modelagem dos periféricos e servindo como apoio direto ao processo de validação.

5.10 Fases de teste

5.10.1 Primeira fase de testes

Ajustes na plataforma (.repl) para suporte ao *toggle pin*

Para facilitar a visualização dos *GPIOs* manipulados pelo *firmware* de teste descrito na metodologia, foi criado um arquivo `.repl` específico para a *board* desta fase, conectando o MCU implementado neste trabalho a um periférico *LED* nativo do Renode. Dessa forma, no `.resc` correspondente, foi adicionada uma linha para habilitar o *logging* desse *LED*, permitindo observar no monitor do Renode as alterações de estado geradas pelo *toggle* do pino. O arquivo `.repl` criado é apresentado na Figura 5.21.

Figura 5.21: Arquivo `.repl` da *board* da primeira fase de testes.



```
using "platforms/cpus/stm32l496.repl"

toggle_pin: Miscellaneous.LED @gpioPortB

gpioPortB:
  15 -> toggle_pin@0
```

Fonte: Autor.

O arquivo começa importando a plataforma do microcontrolador por meio da diretiva `using "platforms/cpus/stm32l496.repl"`, reutilizando o mapeamento de memória e a instanciação dos periféricos básicos já definidos para o STM32L496. Em seguida, é criado o periférico `toggle_pin`, do tipo `Miscellaneous.LED`, anexado ao bloco `@gpioPortB`. Por fim, a seção `gpioPortB` estabelece a conexão física entre o pino 15 da porta B e a entrada do *LED* (`15 -> toggle_pin@0`). Assim, sempre que o *firmware* alterna o estado do pino PB15, o *LED* virtual reflete essa mudança, permitindo a validação do *toggle pin* de forma direta por meio do *logging* no monitor do Renode.

5.10.2 Segunda fase de testes

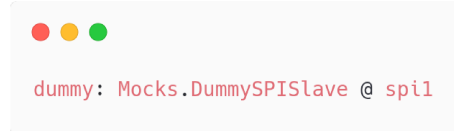
Ajustes na plataforma (.repl) para suporte ao barramento SPI

A segunda fase de testes manteve a estrutura definida na etapa anterior (incluindo o *LED* virtual para observar o *toggle* em *GPIO*), porém adicionou à plataforma os periféricos `spi1` e `uart` para viabilizar ensaios de comunicação. Como o SPI é um barramento do tipo mestre-escravo, a simples presença do controlador SPI no microcontrolador não é suficiente para caracterizar um fluxo de comunicação completo, então foi necessário anexar ao sistema um dispositivo escravo conectado

ao barramento para que as transações do mestre tenham um destino e para evitar avisos do simulador sobre um barramento sem conexões.

A adição realizada no arquivo `.repl` foi descrita na figura 5.22:

Figura 5.22: Adição feita ao `.repl` da *board* para a segunda fase de testes.

A terminal window with a light gray background and three colored dots (red, yellow, green) in the top left corner. The text `dummy: Mocks.DummySPISlave @ spi1` is displayed in a red monospace font.

Fonte: Autor.

O arquivo `.repl` desta fase foi estendido com a instanciação de um escravo *dummy* nativo do Renode, suficiente para “terminar” o barramento e permitir a observação controlada das transmissões realizadas pelo *firmware*. A linha `dummy: Mocks.DummySPISlave @ spi1` cria o periférico *dummy* (um escravo SPI simplificado, sem papel funcional no sistema final) e o anexa diretamente ao barramento `spi1`. Essa adição tem caráter estritamente instrumental, sem implementar o comportamento do rádio ou de qualquer dispositivo real do projeto, mas elimina *warnings* associados a um SPI sem conexão e garante que o tráfego gerado pelo STM32L496 seja consumido.

5.10.3 Terceira fase de testes

Ajustes no `.repl`: substituição do *dummy* por um rádio AT86RF233 e linha de interrupção

Para suportar este teste, o arquivo `.repl` da placa foi modificado para substituir o dispositivo SPI *dummy* utilizado no conjunto de testes 2 por uma instância real do transceptor AT86RF233 conectada ao barramento `spi1`. Além disso, foi configurada a conexão da linha de interrupção do rádio para um pino GPIO do microcontrolador (por exemplo, `gpioPortB@10`), permitindo que eventos do rádio sejam sinalizados ao STM32L496 conforme o modelo de integração esperado.

Com essa alteração, o *dummy* conectado ao `spi1` (utilizado apenas para evitar avisos de barramento sem *slave*) foi removido, uma vez que o rádio passou a desempenhar o papel de dispositivo escravo SPI no cenário de teste, como pode ser visto na figura 5.23.

Após a substituição do *dummy* pela instância do transceptor `Wireless.AT86RF233`, o arquivo `.repl` descreve duas ligações essenciais para a integração rádio-microcontrolador, a seleção do dispositivo no barramento SPI utilizando o (*chip select*) e a sinalização de eventos via interrupção (*IRQ*).

A linha `radio: Wireless.AT86RF233 @ spi1` instancia o AT86RF233 como um periférico escravo conectado ao barramento `spi1` do STM32L496. Em seguida, o mapeamento em `gpioPortB` define o sinal de *chip select*. Ao declarar `12 -> radio@0`, o pino PB12 é conectado à entrada `@0` do periférico `radio`, que repre-

Figura 5.23: Arquivo `.repl` da *board* da terceira fase de testes.

```
using "platforms/cpus/stm32l496.repl"

gpioPortB:
  12 -> radio@0

radio: Wireless.AT86RF233 @ spi1
  -> gpioPortB@10
```

Fonte: Autor.

senta o sinal de *chip select*. Assim, quando o *firmware* controla PB12, a simulação reflete a seleção do AT86RF233 no barramento SPI.

Por fim, a ligação `-> gpioPortB@10` conecta a saída de interrupção do rádio a um pino do microcontrolador, neste caso PB10. Dessa forma, eventos internos do AT86RF233 podem acionar a linha de *IRQ* e serem entregues ao STM32L496 via entrada `gpioPortB@10`, permitindo que o *firmware* trate essas ocorrências por interrupção.

Função auxiliar para injeção manual de *frame* (RX)

Durante a validação inicial do fluxo de recepção, foi adicionada ao modelo uma função auxiliar para injetar manualmente um *frame* de teste: a rotina monta um *payload* com, tamanho do frame, texto fixo e um campo numérico incremental, calcula CRC e invoca o caminho de recepção (`ReceiveFrame`), seguindo os detalhes do AT86RF233 citados anteriormente, permitindo exercitar a lógica de RX a partir de uma chamada controlada. A função é apresentada na Figura 5.24.

5.10.4 Quarta fase de testes

Ajustes na plataforma (`.repl`) para suporte ao *firmware* de produção

Para a última seção de testes, foi criado o arquivo `.repl` da placa que descreve o mesmo *hardware* utilizado no módulo sensor descrito no início da metodologia, isto é, com as mesmas ligações do AT86RF233 no MCU STM32L496.

O último `.repl` já foi apresentado na figura 5.15.

Arranjo de simulação: nó ativo, nó *dummy* e propagação no *Medium*

Como o sensor do laboratório se comunica com um equipamento externo (*hub*) cuja simulação completa não fazia parte dos objetivos deste trabalho, não foi utilizado um segundo nó ativo executando *firmware*. Entretanto, na configuração adotada para o meio IEEE 802.15.4 no Renode, a propagação de *frames* no *Medium* (e a observabilidade do tráfego via *hook*) depende da existência de mais de um dispositivo conectado ao meio. Assim, foi anexada à emulação uma segunda placa

Figura 5.24: Rotina adicionada ao modelo do AT86RF233 para injetar manualmente um *frame* de teste (RX).

```

public void SendTestFrame()
{
    var message = $"Radio transmission {transmissionCount}";
    var messageBytes = System.Text.Encoding.ASCII.GetBytes(message);
    var payload = new byte[1 + messageBytes.Length];

    payload[0] = (byte)messageBytes.Length;
    Array.Copy(messageBytes, 0, payload, 1, messageBytes.Length);

    var crc = Frame.CalculateCRC(payload);
    var fullFrame = new byte[payload.Length + 2];

    Array.Copy(payload, fullFrame, payload.Length);
    fullFrame[fullFrame.Length - 2] = crc[0];
    fullFrame[fullFrame.Length - 1] = crc[1];

    this.Log(LogLevel.Info, "Sending test frame: {0}", message);

    ReceiveFrame(fullFrame, null);

    transmissionCount++;
}

```

Fonte: Autor.

com o mesmo hardware (STM32L496 + AT86RF233), porém sem carregamento de *.elf*, operando apenas como nó estático (*dummy*). Seu papel foi participar do *Medium* para viabilizar a passagem dos quadros transmitidos pelo nó principal, sem executar qualquer lógica de aplicação.

A configuração do nó *dummy* no *.resc* consistiu no carregamento do *.repl* da plataforma e na seleção manual do canal do rádio (canal 24), sem carregar *firmware*, essa configuração foi adicionada ao *.resc* referente a esta seção de simulação e pode ser visto na figura 5.25:

Figura 5.25: Configurações adicionadas no *.resc* da quarta seção de testes.

```

mach add "board_dummy"
mach set "board_dummy"
machine LoadPlatformDescription @platforms/boards/stm32l4-kit.repl
sysbus.spi3.radio Channel 24

```

Fonte: Autor.

Aquisição e variabilidade dos dados (ADC/DMA)

Para evitar leituras constantes o modelo do ADC foi configurado para retornar valores aleatórios. Essa estratégia foi suficiente para validar o caminho funcional de aquisição e transferência via DMA: o *firmware* executa amostragem, preenche telemetria e produz quadros com valores distintos em execuções sucessivas.

Interceptação e repasse de quadros: *packet hook* e UDP

Para externar os quadros gerados pelo *firmware* de produção para fora do ambiente Renode (permitindo salvar e/ou consumir esses dados por ferramentas externas), foi realizada a integração com um script Python executado pelo Renode. Esse script foi configurado para ser acionado por um *packet hook* associado ao tráfego no *Medium*, ao interceptar um quadro, ele encaminha os dados via UDP para um serviço rodando na máquina local no endereço 127.0.0.1:9000. Assim, o tráfego produzido pelo *firmware* pode ser coletado e analisado sem instrumentar o código do sensor.

Emulação parcial do *hub*: resposta pela placa *dummy*

Além do encaminhamento de quadros, o script Python também implementou uma resposta mínima ao nó principal. Como os objetivos desta fase não incluíam a implementação completa do hardware e do *firmware* do *hub*, o script atua como um “orquestrador” externo: ao detectar determinados quadros no *Medium*, ele comanda a placa *dummy* a responder com um quadro fixo, chamando diretamente uma função do periférico de rádio da placa *dummy*. Esse mecanismo permitiu exercitar o caminho de recepção e o fluxo de estados do *firmware* do sensor mesmo na ausência de um *hub* totalmente simulado.

O *script python* que realiza o repasse dos quadros e força a resposta pela placa *dummy* foi configurado no arquivo *.resc* referente a esta seção de simulação e pode ser visto na figura 5.26:

Figura 5.26: Configurações do *script python* adicionadas no *.resc* da quarta seção de testes.



```
radio_medium SetPacketHookFromFile sysbus.spi3.radio @python/zigbee_packet_forward_request_empty.py
```

Fonte: Autor.

Leitura externa e *parse* dos dados (script Python fora do Renode)

Além do *packet hook* e do encaminhamento via UDP, foi desenvolvido um script Python executado fora do Renode com a finalidade de receber os quadros externados no endereço 127.0.0.1:9000 e realizar a leitura do *payload*. Esse script também implementa o *parse* dos dados no mesmo formato adotado pelo *hub* do laboratório, traduzindo os campos do quadro (por exemplo, identificadores, medições e metadados) para uma representação mais direta e interpretável. Com isso, tornou-se possível observar, em tempo de execução, não apenas os bytes brutos transmitidos pelo nó simulado, mas também os valores já decodificados, o que facilitou a verificação funcional e a inspeção dos resultados de medição e telemetria.

Capítulo 6

Resultados

Esta seção apresenta os resultados obtidos a partir das fases de teste executadas ao longo do desenvolvimento incremental do controlador STM32L496 no Renode e de sua integração com o transceptor IEEE 802.15.4 AT86RF233. Os experimentos foram organizados em quatro subseções, seguindo a mesma estrutura adotada na metodologia, de modo a evidenciar a evolução do ambiente simulado desde validações iniciais de execução e temporização até cenários de comunicação e a execução do *firmware* alvo do laboratório.

Em cada subseção, são apresentadas as evidências experimentais coletadas (saídas de *logs*, capturas e resultados de analisadores do Renode). Essa organização permite relacionar diretamente os comportamentos observados durante a simulação com as decisões técnicas tomadas em cada etapa, destacando como os resultados orientaram o refinamento do gêmeo digital até atingir condições compatíveis com a execução de *firmwares* reais sem modificações.

6.1 Fase 1: Validação inicial de temporização e execução básica (*GPIO toggle*)

Nesta primeira fase de resultados, ainda no início da pesquisa e antes da implementação completa de todos os periféricos necessários ao trabalho, foi executado um teste mínimo de funcionamento do microcontrolador simulado: a alternância (*toggle*) de um pino GPIO, acompanhada por mensagens de depuração no terminal do Renode.

O objetivo deste ensaio foi validar dois pontos fundamentais para a continuidade do desenvolvimento do gêmeo digital, se a execução do *firmware* e a modelagem básica do GPIO estavam operacionais e se a temporização programada no *firmware* estava sendo respeitada no ambiente simulado.

Após os ajustes de configuração do ambiente, o comportamento temporal do teste passou a corresponder ao esperado: a inversão do estado do pino ocorreu em intervalos de aproximadamente 1 s, compatível com o atraso programado no *firmware*. A figura 6.1 a seguir apresenta o resultado obtido no *log*:

Figura 6.1: *Log* obtido na fase 1 de testes.A terminal window with a light gray background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a log of LED state changes for 'gpioPortB.toggle_pin'. The log consists of eight lines, each showing a timestamp, the log level '[NOISY]', and the message 'gpioPortB.toggle_pin: LED state changed to True' or 'False'. The timestamps are: 09:16:03.2826, 09:16:04.2835, 09:16:05.2844, 09:16:06.2854, 09:16:07.2861, 09:16:08.2869, 09:16:09.2878, and 09:16:10.2887. The states alternate between True and False.

```
09:16:03.2826 [NOISY] gpioPortB.toggle_pin: LED state changed to True
09:16:04.2835 [NOISY] gpioPortB.toggle_pin: LED state changed to False
09:16:05.2844 [NOISY] gpioPortB.toggle_pin: LED state changed to True
09:16:06.2854 [NOISY] gpioPortB.toggle_pin: LED state changed to False
09:16:07.2861 [NOISY] gpioPortB.toggle_pin: LED state changed to True
09:16:08.2869 [NOISY] gpioPortB.toggle_pin: LED state changed to False
09:16:09.2878 [NOISY] gpioPortB.toggle_pin: LED state changed to True
09:16:10.2887 [NOISY] gpioPortB.toggle_pin: LED state changed to False
```

Fonte: Autor.

Como resultado, esta fase validou a execução básica do *firmware* e a operacionalidade do periférico de GPIO, além de estabelecer um critério essencial de consistência temporal para as próximas fases de teste.

6.2 Fase 2: Validação de comunicação serial via UART e SPI

A segunda fase de testes foi conduzida em um estágio intermediário do desenvolvimento da plataforma, quando o controlador simulado já apresentava execução estável de *firmwares* simples e parte dos periféricos essenciais havia sido integrada. O objetivo principal desta fase foi validar a implementação dos periféricos de comunicação serial USART1 e SPI1, verificando se as rotinas de transmissão do *firmware* eram corretamente refletidas no ambiente simulado.

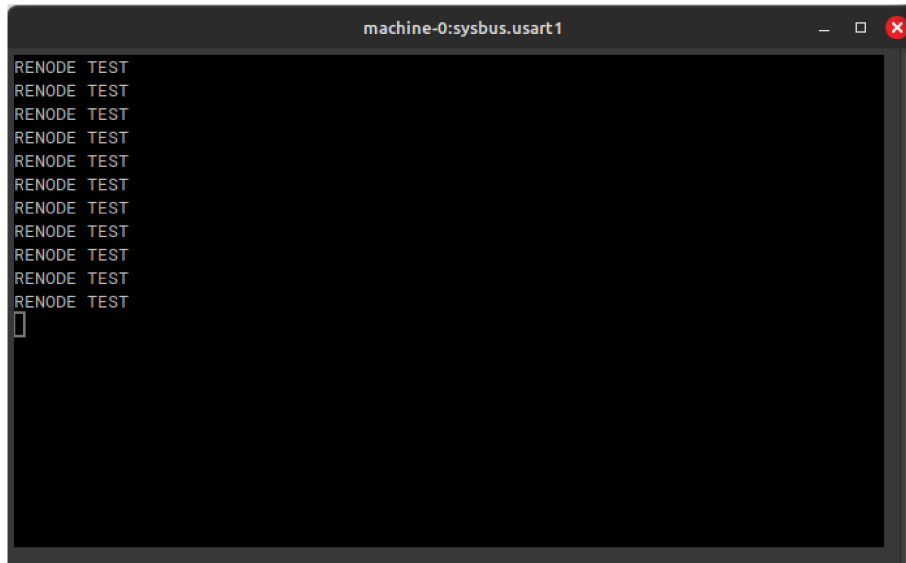
Para manter comparabilidade com a fase anterior e observar a coexistência de atividades periódicas no mesmo laço de execução, o *firmware* desta etapa manteve o *toggle* de GPIO, mas adicionou inicialização e uso dos periféricos de comunicação. No *loop* principal, foi temporizado o envio do mesmo *frame* por SPI e por UART, seguido da alternância do GPIO e do *delay* fixo.

Do ponto de vista de observabilidade, o Renode fornece um analisador nativo para UART, permitindo visualizar o conteúdo transmitido em uma janela dedicada. Para SPI, as evidências foram obtidas por *logging* no terminal, configurado no script `.resc`. As figuras 6.2 e 6.3 seguir apresentam o analisador de UART e um trecho do *output* observado no terminal durante a execução:

Observa-se que as mensagens de SPI aparecem agrupadas em pares de bytes. Esse comportamento decorre do fato de que, no modelo adotado, as escritas no registrador DR do SPI foram registradas em transações de 16 bits, enquanto o *firmware* estava configurado para operar com *data size* de 8 bits. Ainda assim, o conteúdo transmitido é preservado e pode ser interpretado corretamente, evidenciando o envio do mesmo *frame* por UART e SPI. Adicionalmente, o analisador serial do Renode confirmou a saída pela UART em formato de caracteres.

Os testes realizados com diferentes temporizações entre os envios de dados por UART e SPI, e também do *toggle* do GPIO também tiveram resultados posi-

Figura 6.2: Analisador UART nativo do Renode com o resultado da transmissão.



Fonte: Autor.

Figura 6.3: Log obtido na fase 2 de testes.



Fonte: Autor.

tivos e indicaram que a simulação respeitou as diferentes temporizações, conforme o programado no *firmware*.

6.3 Fase 3: Integração com o rádio AT86RF233 e validação de RX/TX no meio IEEE 802.15.4

A terceira fase de testes foi conduzida após a inclusão e o aprimoramento do transceptor IEEE 802.15.4 AT86RF233 na plataforma simulada. Nesta etapa, o objetivo foi validar, de forma controlada, que o modelo AT86RF233.cs é funcional para os fluxos mínimos necessários à comunicação sem fio, incluindo, recepção de quadros e disponibilização do *payload* ao *firmware*, sinalização de eventos e retorno ao estado de escuta e transmissão de quadros pelo meio e recepção por um segundo nó simulado.

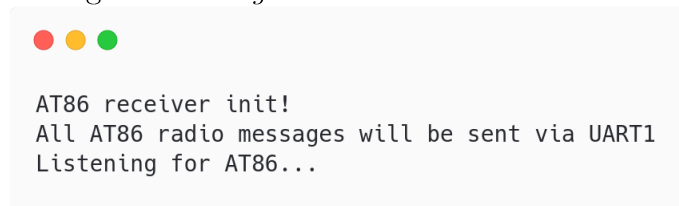
Os testes foram realizados com dois *firmwares* distintos, ambos executando em módulos STM32L496 conectados ao AT86RF233. Em todos os casos, a UART foi utilizada como canal de depuração e observação, explorando o analisador serial do Renode para visualização direta das mensagens.

6.3.1 Firmware A: validação de recepção (RX) com injeção manual de quadros

O primeiro *firmware* foi projetado para operar de forma isolada (sem transmissor externo), validando o processo de RX de maneira determinística por meio do método `SendTestFrame()` implementado no modelo do rádio.

Após iniciar a simulação, as mensagens de inicialização foram exibidas no analisador de UART, indicando que o rádio foi configurado e entrou em modo RX, apresentado na figura 6.4:

Figura 6.4: Log obtido do início da UART.

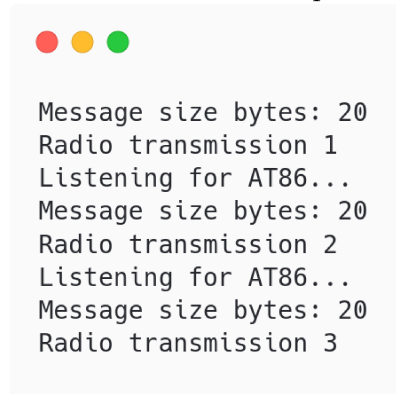


Fonte: Autor.

O método `SendTestFrame()` foi acionado manualmente no terminal do Renode e a cada chamada, o *firmware* recebeu o quadro, reportou via UART e retornou ao estado de escuta, segue o *log* na figura 6.5:

6.3.2 Firmware B: comunicação *ping-pong* entre dois nós (TX e RX)

O segundo *firmware* validou transmissão e recepção no meio IEEE 802.15.4, instanciando duas máquinas na mesma emulação, ambas conectadas ao mesmo *Me-*

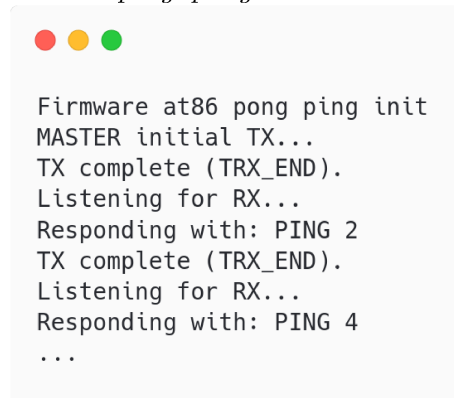
Figura 6.5: *Log* da saída no analisador UART após inicialização do Firmware A.A terminal window with a light gray background and three colored dots (red, yellow, green) at the top left. It displays the following text:

```
Message size bytes: 20
Radio transmission 1
Listening for AT86...
Message size bytes: 20
Radio transmission 2
Listening for AT86...
Message size bytes: 20
Radio transmission 3
```

Fonte: Autor.

dium. Um dos nós foi inicializado como *master*, iniciando a troca de mensagens PING/PONG com incremento do contador.

No nó *master*, observou-se o padrão de respostas PING com valores pares, identificados na figura 6.6:

Figura 6.6: *Log* da saída *ping-pong* no analisador UART do *master*.A terminal window with a light gray background and three colored dots (red, yellow, green) at the top left. It displays the following text:

```
Firmware at86 pong ping init
MASTER initial TX...
TX complete (TRX_END).
Listening for RX...
Responding with: PING 2
TX complete (TRX_END).
Listening for RX...
Responding with: PING 4
...
```

Fonte: Autor.

No nó *slave*, observou-se o padrão complementar com PONG em valores ímpares, identificados na figura 6.7:

O padrão observado confirma que o modelo `AT86RF233.cs` sustentou os fluxos completos de comunicação no meio: transmissão inicial, propagação no *Medium*, recepção do *payload*, resposta e retorno consistente ao modo de escuta.

Figura 6.7: Log da saída *ping-pong* no analisador UART do *slave*.

A screenshot of a UART log window. At the top, there are three colored circles: red, yellow, and green. Below them, the log text reads: "Firmware at86 pong ping init", "Listening for RX...", "Responding with: PONG 1", "TX complete (TRX_END).", "Listening for RX...", "Responding with: PONG 3", and "...".

Fonte: Autor.

6.4 Fase 4: Execução do *firmware* de produção e interceptação/decodificação de dados via UDP

A quarta fase de testes correspondeu à execução do *firmware* alvo do trabalho: o *firmware* de produção, compilado para o STM32L496 e destinado a operar no hardware real. O arquivo `.elf` carregado na simulação foi obtido diretamente do *source code* do projeto do laboratório, sem qualquer alteração no código, preservando o mesmo artefato que seria implantado em um sensor físico.

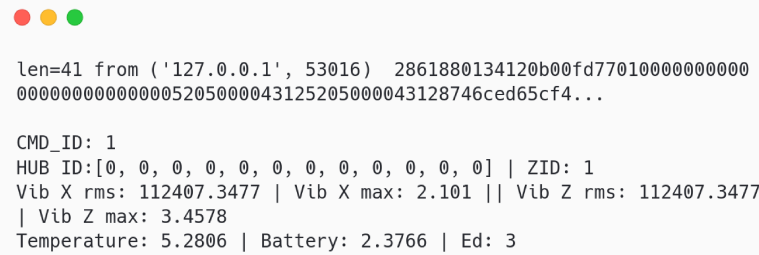
Em termos funcionais, o *firmware* implementa uma aplicação orientada a eventos organizada como uma máquina de estados para um nó IEEE 802.15.4. Após o *boot* e inicializações (GPIO, timers e RTC), o sistema configura o rádio AT86RF233 e estabelece *callbacks* de TX/RX; em seguida, alterna entre estados de decisão/o-cioso, aquisição (com ADC, incluindo transferências via DMA e temporização por timers), transmissão de telemetria e/ou dados segmentados, janelas de recepção/espera por confirmação e períodos de baixo consumo (*sleep*) controlados por temporização. Esse encadeamento torna a execução mais exigente do que os *firmwares* das fases anteriores, pois exercita múltiplos periféricos em conjunto, com dependências temporais e rotinas assíncronas por interrupções.

Como consequência, esta foi a fase em que se observou maior concentração de avisos (*warnings*) e travas provenientes de *bugs* de execução. As validações anteriores confirmaram caminhos básicos, mas não expunham requisitos críticos do *firmware* real (ADC+DMA, integração de interrupções e formatação completa de *frames*). Assim, as correções mais relevantes do ambiente simulado concentraram-se nesta etapa, destacando-se ajustes no fluxo ADC/DMA, correções no caminho de montagem/leitura de *frames* do AT86RF233, ligações de interrupções envolvendo timers, rádio e DMA, e adequações de temporização/RTC para suportar *sleep*.

Para tornar a transmissão do sensor interpretável (além de bytes em hexadecimal), foi estendido o script Python responsável por receber, via UDP, os quadros interceptados no *Medium*. Nesse script, foi implementado um decodificador de *payload* compatível com a lógica de interpretação utilizada no *hub* real, permitindo exibir os campos da mensagem traduzidos.

O script reportou os principais campos do *frame*. Nesta configuração, o *PID* permaneceu com valor zero, o *CMD_ID* manteve-se constante em 1, o *ZID* foi fixado em 1 e o valor *ED* foi fixado como 3 no decodificador/script. Os valores provenientes do ADC variaram entre transmissões, sendo apresentados pelo decodificador em campos como vibração e métricas auxiliares (temperatura e bateria). Um exemplo de saída é mostrado a seguir, na figura 6.8:

Figura 6.8: Log da saída do *hub* parcialmente simulado.



```
len=41 from ('127.0.0.1', 53016) 2861880134120b00fd77010000000000
0000000000000005205000043125205000043128746ced65cf4...

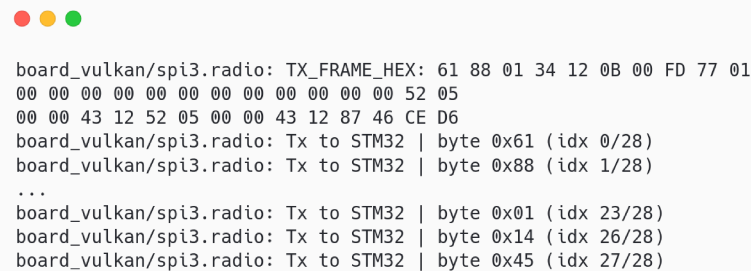
CMD_ID: 1
HUB ID:[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] | ZID: 1
Vib X rms: 112407.3477 | Vib X max: 2.101 || Vib Z rms: 112407.3477
| Vib Z max: 3.4578
Temperature: 5.2806 | Battery: 2.3766 | Ed: 3
```

Fonte: Autor.

6.4.1 Evidência de TX e RX no rádio dentro da máquina de estados

No terminal do Renode, foi possível observar tanto o envio do *frame* pelo nó principal quanto a posterior leitura/recepção de um *frame* proveniente do nó *dummy*, o que indica que o *firmware* executou o ciclo de transmissão e retornou ao fluxo de recepção conforme sua máquina de estados, exibido na figura 6.9:

Figura 6.9: Log da transmissão e leitura do *frame* recebido pelo STM32 via rádio simulado.



```
board_vulkan/spi3.radio: TX_FRAME_HEX: 61 88 01 34 12 0B 00 FD 77 01
00 00 00 00 00 00 00 00 00 00 00 00 00 00 52 05
00 00 43 12 52 05 00 00 43 12 87 46 CE D6
board_vulkan/spi3.radio: Tx to STM32 | byte 0x61 (idx 0/28)
board_vulkan/spi3.radio: Tx to STM32 | byte 0x88 (idx 1/28)
...
board_vulkan/spi3.radio: Tx to STM32 | byte 0x01 (idx 23/28)
board_vulkan/spi3.radio: Tx to STM32 | byte 0x14 (idx 26/28)
board_vulkan/spi3.radio: Tx to STM32 | byte 0x45 (idx 27/28)
```

Fonte: Autor.

Na Figura 6.8, observou-se a recepção de **len=41** bytes, embora o *payload* de aplicação transmitido pelo *firmware* possua 29 bytes. Essa diferença decorre do fato de que a captura via UDP (no *Medium*) expõe o quadro IEEE 802.15.4 no nível MAC/PHY, e não apenas o *payload* da aplicação. Em particular, o AT86RF233 entrega no caminho de recepção o *PSDU* completo, tipicamente precedido pelo byte de comprimento (*PHR*) e incluindo o cabeçalho MAC e o campo de verificação de

integridade (*FCS/CRC*) ao final. No exemplo, o byte inicial `0x28` indica comprimento de 40 bytes subsequentes, compatível com a soma do cabeçalho MAC com endereçamento curto (FCF, número de sequência, PAN ID, endereços de destino e origem), os 29 bytes do *payload* de dados e 2 bytes de *FCS*, totalizando 41 bytes reportados pelo script.

Além da observação por *logs* e *hooks*, a depuração via GDB no STM32CubeIDE (conectado ao Renode) foi utilizada para verificar, no nível de código-fonte, se o sensor simulado interpretava corretamente a resposta do *hub*. Por meio de *breakpoints* em trechos críticos do fluxo de recepção, inspeção do *frame* recebido em memória e execução passo a passo (*step/next*) para acompanhar as chamadas realizadas após a interrupção de RX, foi possível confirmar que o *firmware* decodificou o conteúdo do quadro e aplicou os parâmetros recebidos conforme esperado, evidenciando operação correta do sensor simulado no cenário de testes.

6.4.2 Estimativa de consumo energético durante a execução do *firmware* de produção

Além das evidências funcionais de comunicação e aquisição, nesta fase foi executado o cálculo de consumo energético a partir do estimador implementado no periférico PWR. A medição foi realizada durante a simulação do *firmware* de produção e coletada ao final de um ciclo completo da máquina de estados, após a interpretação do *frame* de resposta do *hub* e imediatamente antes da transição para o estado de *reset*. O cálculo considerou a execução com $HCLK=8$ MHz, $V_{DD} = 3.0$ V e $T = 25^{\circ}\text{C}$, acumulando carga/energia ao longo do tempo de simulação conforme o modo de operação ativo em cada intervalo e levando em conta, adicionalmente, a habilitação de periféricos (clocks/estados) durante a execução do *firmware*.

A saída do Renode, apresentada na figura 6.10, sumariza o tempo total observado, a carga acumulada e a corrente média equivalente, além da decomposição por modos de baixo consumo e execução:

Figura 6.10: Dados de consumo exibidos no Renode Monitor.

```
(board_vulkan) start
Starting emulation...
(board_vulkan) pause
Pausing emulation...
(board_vulkan) sysbus.pwr GetEnergyReport
PWR energy estimator (STM32L4): totalTime=6.501400s, total=0.000425990 mAh, avg=0.235882 mA, @VDD=3.00V
Current: mode=Stop2, I=0.003180 mA (HCLK=8000000 Hz)
- Run: 1.178000s (18.1%), avgI=1.287466 mA
- Sleep: 0.000000s (0.0%), avgI=0.000000 mA
- Stop0: 0.000000s (0.0%), avgI=0.000000 mA
- Stop1: 0.000000s (0.0%), avgI=0.000000 mA
- Stop2: 5.323400s (81.9%), avgI=0.003180 mA
- Standby: 0.000000s (0.0%), avgI=0.000000 mA
- Shutdown: 0.000000s (0.0%), avgI=0.000000 mA
(board_vulkan)
```

Fonte: Autor.

A Tabela 6.1 reorganiza os dados do relatório para facilitar a leitura, incluindo a carga estimada por modo. Observa-se que o *firmware* permaneceu na maior parte

do tempo em modo de baixo consumo (Stop2, 81.9%), acordando apenas para operações pontuais (*Run*, 18.1%), comportamento coerente com as características do projeto (operação orientada a eventos e janelas curtas de atividade). Esse resultado também evidencia que o estimador baseado em modo de operação é capaz de capturar adequadamente a dinâmica *run/sleep* do sistema ao longo do tempo de simulação.

Ressalta-se, contudo, que a estimativa obtida refere-se apenas ao microcontrolador STM32L496, uma vez que o estimador implementado no periférico PWR acumula consumo somente da MCU. Para uma estimativa fiel ao hardware do nó, ainda seria necessário somar o consumo do transceptor AT86RF233.

Tabela 6.1: Estimativa de consumo na Fase 4 (HCLK=8 MHz, $V_{DD} = 3.0$ V, $T = 25$ °C).

Modo	Tempo (s)	Fração (%)	Corrente média (mA)	Carga (mAh)
Run	1.1780	18.1	1.287466	0.000421287
Stop2	5.3234	81.9	0.003180	0.000004702
Total	6.5014	100.0	0.235882	0.000425990

Em síntese, os resultados desta fase demonstram que o ambiente simulado foi capaz de executar o *firmware* real do laboratório (sem modificações), sustentar fluxos integrados de aquisição (ADC+DMA+timers), transmitir quadros coerentes pelo meio IEEE 802.15.4 e permitir a observação externa e decodificação da telemetria via UDP, caracterizando a validação mais representativa do gêmeo digital desenvolvido.

Capítulo 7

Discussão

Este capítulo discute e interpreta os resultados apresentados no Capítulo 6, relacionando-os aos objetivos do trabalho e à estratégia metodológica adotada. Em vez de apenas reiterar as evidências experimentais, busca-se explicitar o que elas indicam sobre a viabilidade de um gêmeo digital em baixo nível no Renode, quais aspectos se mostraram críticos para alcançar execução estável do *firmware* real e quais limitações permanecem em aberto.

7.1 Interpretação dos resultados por fases de teste

Os experimentos foram organizados em fases que evoluem de validações mínimas de execução para cenários de integração com maior complexidade. Em termos de discussão, esse encadeamento é relevante porque evidencia um padrão recorrente em simulação de microcontroladores: os testes com *firmwares* confirmaram caminhos básicos, enquanto o *firmware* real expõe dependências temporais e acoplamentos entre periféricos que não aparecem em cargas de trabalho simplificadas. Assim, o propósito das fases iniciais não foi apenas em funcionar, mas em reduzir incertezas e isolar variáveis antes da etapa mais exigente.

7.1.1 Fase 1: consistência de execução e temporização

A primeira fase sustenta a premissa de que o ambiente simulado possui capacidade mínima de executar um binário e produzir comportamento temporal coerente. A observação de alternância periódica em GPIO com *delay* é um tipo de evidência simples, porém importante, ao reduzir a probabilidade de que falhas posteriores sejam causadas por problemas de *clock*, mapeamento de memória ou inicialização do sistema. Do ponto de vista metodológico, trata-se do ponto inicial a partir do qual falhas nas fases seguintes podem ser atribuídas com maior confiança à integração de periféricos e interrupções, e não à ausência de execução estável de temporização.

7.1.2 Fase 2: periféricos de comunicação

A segunda fase evidencia um ponto central para gêmeos digitais em baixo nível, a modelagem de periféricos de comunicação não é um requisito apenas de conectividade, mas também de viabilidade prática de testes e validação. Interfaces como UART/USART e SPI sustentam o funcionamento comum de aplicações embarcadas (telemetria, interação com módulos externos e sensores), ao mesmo tempo em que fornecem um canal simples e robusto para instrumentação do *firmware* durante a simulação. No ambiente desenvolvido, a UART foi utilizada principalmente para saída de mensagens e interação com analisadores do Renode, enquanto a SPI viabilizou a comunicação com dispositivos externos, como o rádio AT86RF233, compondo a base dos testes de integração do trabalho.

7.1.3 Fase 3: integração com o rádio e validação de fluxo fim a fim no meio

A terceira fase foi uma importante porque passou a validar fluxos fim a fim que atravessam mais de um sistema: MCU + SPI + rádio + meio de comunicação (*Medium*). O padrão de *ping/pong* observado (recepção, resposta, retorno consistente ao modo de escuta) é especialmente informativo, pois depende de encapsulamento correto do *frame*, sincronização entre estados internos do transceptor e sinalização por interrupções/eventos em momentos compatíveis com o *firmware*. Nessa fase, a evidência deixou de ser “um periférico responde” e passa a ser “a cadeia de comunicação se mantém coerente”, aproximando o gêmeo digital de um uso realista.

7.1.4 Fase 4: execução do *firmware* real como critério de fidelidade prática

A quarta fase concentra o principal argumento de fidelidade prática do trabalho: executar no Renode o mesmo arquivo (*.elf*) compilado para o STM32L496 e destinado ao hardware real, sem adaptações no código do projeto. Nessa etapa, a complexidade da máquina de estados do sensor (alternância entre aquisição, transmissão, recepção e períodos de baixo consumo) tornou-se uma vantagem metodológica, pois executou múltiplos periféricos em conjunto que dependiam de rotinas assíncronas por interrupções, o *firmware* real atuou como um “teste de estresse” do gêmeo digital.

Como consequência, a presença de mais *warnings* e erros na simulação nesta fase foram interpretados como um resultado esperado em ambientes que buscam maior fidelidade, foi justamente o aumento de realismo que identificou lacunas de modelagem. A discussão técnica, portanto, deve enfatizar que as correções realizadas (como ajustes no fluxo ADC/DMA, correções no caminho de montagem/leitura de *frames* do rádio, ligações de interrupções e adequações de temporização/RTC para suportar *sleep*) representaram a transição do “funciona em casos simples” para

“sustenta o caminho principal de operação do sistema real”.

Do ponto de vista de evidências, dois elementos fortalecem a interpretação dos resultados:

- Introdução de variabilidade em leituras do ADC no modelo, suficiente para demonstrar que o caminho de aquisição e transferência via DMA estava ativo e refletindo na telemetria.
- Arranjo de interceptação/decodificação de quadros via *packet hook* com encaminhamento UDP, que transforma o tráfego em sinais interpretáveis (campos e medições) e viabiliza validação qualitativa do conteúdo transmitido, não apenas da existência de bytes no barramento.

7.2 Observabilidade, depuração e reprodutibilidade do ambiente simulado

Um resultado de às fases é que o ambiente não foi apenas executável, mas também depurável. A integração com depuração via GDB (em conjunto com o STM32CubeIDE) foram relevantes porque aproxima o fluxo de desenvolvimento do cenário real: permitindo utilizar *breakpoints*, execução passo a passo e inspeção de registradores/memória em nível de código-fonte. Na prática, essa capacidade muda o status do gêmeo digital: ele deixa de ser um simulador e passa a ser uma plataforma de engenharia que suporta ciclos iterativos de correção de modelos e validação do *firmware*.

Além disso, a necessidade de um nó adicional no *Medium* (nó *dummy*) para viabilizar a observabilidade do tráfego via *hook* explicita um detalhe importante: nem toda condição de teste é equivalente ao mundo real, e algumas adaptações do arranjo experimental são necessárias para tornar o fenômeno observável no simulador. Ao discutir isso, é útil deixar claro que tais adaptações não mudam o *firmware* alvo, mas alteram o cenário de teste para permitir instrumentação e coleta de evidências.

7.3 Estimativa de consumo energético: utilidade e limitações

A estimativa de consumo energético adiciona uma dimensão de análise que vai além do funcional, calcular métricas de energia a partir do tempo de simulação e do estado de operação do microcontrolador. O método empregado (estimador no periférico PWR com métricas derivadas de documentação do fabricante, ajustadas por *clock*, temperatura e modo de operação) é adequado como prova de conceito, pois conecta sinais internos do modelo (registros/estados) e uma métrica importante em sistemas embarcados. Nesse contexto, essa métrica fornece subsídios para ava-

liar, ainda na fase de projeto, a viabilidade de alimentação por fontes renováveis e estratégias de aumento de autonomia energética do sistema.

A predominância de modos de baixo consumo ao longo da execução é coerente com o *firmware* do sensor do laboratório organizado como máquina de estados, no qual períodos de atividade curta (aquisição/transmissão) são intercalados com *sleep*. Ao mesmo tempo, é essencial explicitar limites, a estimativa depende de parâmetros típicos e integração manual dos valores de consumo em cada fase direto na descrição do periférico, e neste trabalho foi aplicada ao STM32L496, não abrangendo o consumo estimado do módulo de rádio.

7.4 Limitações do modelo criado

Apesar das evidências de execução estável e de comunicação fim a fim, algumas limitações devem ser consideradas:

- **Validade interna (aproximações de modelo):** a configuração do ADC para retornar valores aleatórios é suficiente para validar o caminho funcional ADC/DMA, mas não representa um sensor físico nem permite avaliar exatidão de conversão. De modo similar, parte das conclusões depende de coerência estrutural do fluxo (estados, interrupções e tráfego), e não de equivalência elétrica/analógica completa.
- **Validade externa (generalização):** os resultados foram obtidos com um *firmware* real específico e com o conjunto de periféricos criados para ele. A generalização para outros projetos (ou para configurações diferentes do mesmo microcontrolador) tende a exigir expansão do conjunto de modelos e novos testes, uma vez que, embora o modelo do STM32L496 desenvolvido inclua a descrição extensiva de diversos periféricos, podem surgir cenários em que seja necessário ampliar ou refinar a modelagem para contemplar funcionalidades específicas que não foram utilizadas pelo *firmware* validado neste trabalho.
- **Completude do cenário:** a simulação completa do *hub* não foi objetivo do trabalho, assim, parte do comportamento de rede foi validada por interceptação e decodificação, em vez de por interação completa entre duas aplicações reais. Isso reduz o escopo de afirmações sobre a aplicação real de mais alto nível (sensor do laboratório intergrado ao *hub*), embora preserve a validade das evidências sobre o conjunto STM32L4–rádio–*Medium*.
- **Estimativa energética:** por depender de parâmetros típicos e de simplificações, a estimativa deve ser tratada como indicativa e comparativa dentro do próprio ambiente simulado, não como afirmação exata do consumo.

Capítulo 8

Conclusões

Esta dissertação investigou a viabilidade de empregar o Renode como base para um *digital twin* funcional voltado a sistemas embarcados, com ênfase na execução e na depuração de *firmwares* em um ambiente virtual reprodutível e instrumentável. A proposta foi implementada a partir da modelagem incremental de periféricos em nível de registradores, eventos e interrupções, de modo a permitir que o *firmware* interagisse com a plataforma simulada de forma compatível com o comportamento esperado no hardware real. Adicionalmente, o ambiente foi estendido com um estimador de consumo no periférico PWR, buscando fornecer subsídios para análises de autonomia e estudos de viabilidade de alimentação por fontes renováveis.

Os resultados demonstraram que a estratégia de desenvolvimento incremental orientada a testes foi adequada para reduzir incertezas e sustentar a evolução da plataforma até a etapa de maior complexidade, a execução do *firmware* real de produção, carregado no microcontrolador simulado sem modificações no binário. Ao longo das fases experimentais, observou-se que a progressão de testes simples para cenários integrados foi essencial para revelar e corrigir dependências que emergem apenas quando múltiplos periféricos operam em conjunto, especialmente em fluxos assíncronos baseados em interrupções, temporização e transferência de dados via DMA. Nesse sentido, o principal indicativo de fidelidade prática foi a execução do código, com sustentação do ciclo funcional do *firmware* em um cenário com comunicação via rádio e coleta de evidências de tráfego e telemetria.

Entre as contribuições técnicas, destaca-se a disponibilização de uma plataforma de simulação capaz de apoiar testes repetíveis e rastreáveis, reduzindo a dependência de *hardware* nas etapas iniciais e intermediárias de integração. A instrumentação por interface de comunicação (UART/USART) e o arranjo de interceptação/decodificação de quadros via *packet hook* com encaminhamento UDP ampliaram a capacidade de validação do comportamento observado, permitindo analisar não apenas a ocorrência de comunicação, mas também a coerência estrutural e qualitativa do conteúdo transmitido. Além disso, o estimador energético incorporado ao modelo acrescentou uma dimensão de análise relevante para sistemas embarcados, ao permitir relacionar o tempo de simulação e os estados de operação do microcontrolador com métricas de energia, oferecendo base inicial para discussões de autonomia

e viabilidade de alimentação por fontes renováveis.

Embora os resultados sustentem a viabilidade do *digital twin* proposto, algumas limitações devem ser reconhecidas. Parte das validações utilizou aproximações (por exemplo, variabilidade aleatória no ADC para evidenciar o caminho funcional ADC/DMA), e a estimativa energética depende de parâmetros típicos do fabricante e de integração manual de valores no modelo, devendo ser interpretada como indicativa no contexto do ambiente simulado. Além disso, o escopo do estimador foi restrito ao STM32L496, não abrangendo o consumo do módulo de rádio, e o cenário de rede foi validado principalmente por interceptação/decodificação de tráfego, sem a implementação completa de um *hub* simulado em nível de aplicação. Ainda assim, tais limites não comprometem a contribuição central do trabalho, uma vez que o objetivo principal consistiu em demonstrar a execução e a depuração do *firmware* real em um ambiente virtual de baixo nível, bem como avaliar a utilidade de mecanismos adicionais (instrumentação e estimativa energética) para ampliar o alcance do gêmeo digital.

8.1 Trabalhos futuros

Os resultados sugerem que a construção incremental de um gêmeo digital em baixo nível no Renode é viável e pode sustentar execução do *firmware* real sem modificações, desde que a modelagem contemple detalhes críticos de integração (interrupções, temporização e caminhos de DMA, por exemplo). Como trabalhos futuros, destacam-se:

- calibrar o estimador energético com medições em hardware para reduzir incertezas e permitir comparação quantitativa;
- estender a estimativa de consumo para abranger também o módulo de rádio e estados de transmissão/recepção;
- ampliar o conjunto de testes (unitários e de integração) para cobrir mais combinações de periféricos, incluindo a integração com sensores simulados, e cenários de erro;
- evoluir o cenário de comunicação com um *hub* simulado mais completo (ou com uma aplicação real em segundo nó), permitindo validar interações de mais alto nível além da telemetria interceptada.
- integrar a plataforma de simulação ao ciclo de *CI/CD*, automatizando a execução de testes de *firmwares* no Renode, com coleta de métricas e geração de relatórios (*logs*, cobertura de testes, desempenho/tempo virtual e indicadores energéticos);

Por fim, conclui-se que a construção incremental de um *digital twin* em baixo nível no Renode constitui uma abordagem viável e útil para apoiar o ciclo de de-

envolvimento de *firmware*, ao oferecer um ambiente controlado para execução, instrumentação e testes repetíveis. Ao aproximar o comportamento do *firmware* ao observado em hardware real e ao incorporar mecanismos de observação e estimativa energética, o trabalho estabelece uma base consistente para evoluções futuras voltadas tanto ao aumento de fidelidade quanto à ampliação de cenários de validação e de análises de autonomia energética.

Referências

- Antmicro (2024). Renode github action for automated testing on simulated hardware. Disponível em: <https://renode.io/news/renode-github-action-for-automated-testing-in-simulation/>. Acesso em: 01 jan. 2026.
- Atmel Corporation (2014). AT86RF233: Low power, 2.4ghz transceiver for zigbee, rf4ce, iee 802.15.4, 6lowpan, and ism applications. Datasheet. Rev. 8351E-MCU Wireless-07/14. Disponível em: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8351-MCU_Wireless-AT86RF233_Datasheet.pdf. Acesso em: 12 fev. 2026.
- Barbie, A. e Hasselbring, W. (2023). Embedded software development with digital twins: Specific requirements for small and medium-sized enterprises. Disponível em: <https://arxiv.org/abs/2309.09216>. Acesso em: 11 fev. 2026.
- Barbie, A. e Hasselbring, W. (2024). From digital twins to digital twin prototypes: Concepts, formalization, and applications. *IEEE Access*, 12:75337–75365.
- Barbie, A., Hasselbring, W., e Hansen, M. (2024). Digital twin prototypes for supporting automated integration testing of smart farming applications. *Symmetry*, 16(2):221.
- Barbie, A., Hasselbring, W., e Pech, N. (2021). Continuous integration testing of embedded software with digital twin prototypes. TechRxiv preprint.
- Barricelli, B. R., Casiraghi, E., e Fogli, D. (2019). A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7:167653–167671.
- Cheng, S.-M., Lui, Y.-C., Tsai, N.-J., e Hong, B.-K. (2024). Toward intelligent IoT endpoint detection and response using digital twins via firmware emulation. *IEEE Internet of Things Magazine*, páginas 20–24.
- do Amaral, J. V. S., dos Santos, C. H., Montevechi, J. A. B., e de Queiroz, A. R. (2023). Energy digital twin applications: A review. *Renewable and Sustainable Energy Reviews*, 188:113891.

- Franco da Silva, A. C., Wagner, S., Lazebnik, E., e Traitel, E. (2021). Using a cyber digital twin for continuous automotive security requirements verification. Disponível em: <https://arxiv.org/abs/2102.00790>. Acesso em: 11 fev. 2026.
- He, Y., Zhang, N., e Wang, A. (2020). Digital twin process and simulation operation control technology for intelligent manufacturing unit. *IOP Conference Series: Materials Science and Engineering*, 836(1):012010.
- Javaid, M., Haleem, A., e Suman, R. (2023). Digital twin applications toward Industry 4.0: A review. *Cognitive Robotics*, 3:71–92.
- Kellermüller, S., Obushevs, A., Gonzalez, M. R., e Korba, P. (2022). Digital twin development of a dynamic hardware emulator. *Energies*, 15(13):4547.
- Kritzing, W., Karner, M., Traar, G., Henjes, J., e Sih, W. (2018). Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022.
- Kutscher, V., Anokhin, O., e Anderl, R. (2020). Enhancing digital twin performance through simulation of computerized numerical control firmware. *Procedia Manufacturing*, 49:69–74. 23rd International Conference on Material Forming (ESAFORM 2020).
- Minerva, R., Lee, G. M., e Crespi, N. (2020). Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models. *Proceedings of the IEEE*, 108(10):1785–1824.
- Parnianifard, S. M., Chen, V. C. P., de Carvalho, L. P. F., e Rodrigues, J. J. P. C. (2022). Digital-twins towards cyber-physical systems: A brief survey. *Engineering Journal*, 26(9).
- Presti, F. N. (2026). Repositório do gêmeo digital stm32l496 no renode: arquivos .repl, modelos .cs e scripts .resc. GitHub. Disponível em: <https://github.com/FelipeNarimatsu/renode>. Acesso em: 12 fev. 2026.
- Presti, F. N., de Araujo Dantas, L., de Freitas, L. C. G., da Cunha, M. J., e de Almeida, M. B. (2025). Creating digital twins in renode: A case study with stm32l4 and at86rf233 zigbee radio. In *2025 16th IEEE International Conference on Industry Applications (INDUSCON)*, páginas 677–682. IEEE.
- Rasheed, A., San, O., e Kvamsdal, T. (2020). Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access*, 8:21980–22012.
- Renode Project (2026a). Renode documentation: Describing platforms (platform description). Disponível em: https://renode.readthedocs.io/en/latest/basic/describing_platforms.html. Acesso em: 01 jan. 2026.

- Renode Project (2026b). Renode documentation: Monitor and script syntax. Disponível em: <https://renode.readthedocs.io/en/latest/basic/monitor-syntax.html>. Acesso em: 01 jan. 2026.
- Renode Project (2026c). Renode documentation: Peripheral modeling guide. Disponível em: <https://renode.readthedocs.io/en/latest/advanced/writing-peripherals.html>. Acesso em: 01 jan. 2026.
- Renode Project (2026d). Renode documentation: Testing with renode. Disponível em: <https://renode.readthedocs.io/en/latest/introduction/testing.html>. Acesso em: 01 jan. 2026.
- Renode Project (2026e). Renode documentation: Time framework. Disponível em: <https://renode.readthedocs.io/en/latest/advanced/time-framework.html>. Acesso em: 01 jan. 2026.
- Renode Project (2026f). Renode: site oficial. Disponível em: <https://renode.io/>. Acesso em: 01 jan. 2026.
- Renode Project (2026g). Renode: What is renode? (about). Disponível em: <https://renode.io/about/>. Acesso em: 01 jan. 2026.
- Renode Project (2026h). renode/renode: repositório oficial do framework. Acesso em: 01 jan. 2026.
- Samek, M. (2014). Cutting through the confusion with arm cortex-m interrupt priorities. Arm Developer (blog). Disponível em: <https://developer.arm.com/community/arm-community-blogs/b/embedded-and-microcontrollers-blog/posts/cutting-through-the-confusion-with-arm-cortex-m-interrupt-priorities>. Acesso em: 01 jan. 2026.
- Speiser, F., Szalay, I., e Fodor, D. (2024). Embedded system simulation using renode. *Engineering Proceedings*, 79(1):52.
- STMicroelectronics (2021). Rm0351: STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced arm-based 32-bit mcus – reference manual. Reference manual.
- STMicroelectronics (2025). STM32L496xx datasheet – production data. DS11585 Rev. 20, October 2025.

Apêndice A

Arquivo stm32l496.repl desenvolvido

Listing A.1: Plataforma STM32L496 no Renode (.repl)

```
sram: Memory.MappedMemory @ sysbus 0x20000000
    size: 0x00050000

flash: Memory.MappedMemory @ sysbus 0x08000000
    size: 0x00100000

flash_controller: MTD.STM32L4_FlashController @ {
    sysbus 0x40022000;
    sysbus new Bus.BusMultiRegistration {
        address: 0x1FFF7800;
        size: 0x0F;
        region: "optionBytes" }
    }
    flash: flash

pwr: Miscellaneous.STM32L4_PWR @ sysbus 0x40007000
    RccPeripheral: rcc
    Voltage: 3.0
    UseExternalSmps: false

dbg: Miscellaneous.STM32L4_DBGMCU @ sysbus 0xE0042000

rtc: Timers.STM32L4_RTC @ sysbus 0x40002800
    AlarmIRQ -> nvic@41
    WakeupIRQ -> exti@20

rcc: Miscellaneous.STM32L4_RCC @ sysbus 0x40021000
    rtcPeripheral: rtc

syscfg: Miscellaneous.STM32L4_SYSCFG @ sysbus 0x40010000

dma1: DMA.STM32L4_DMA @ sysbus 0x40020000
    [0-6] -> nvic@[11-17]

dma2: DMA.STM32L4_DMA @ sysbus 0x40020400
    [0-6] -> nvic@[56-60,68-69]

exti: IRQControllers.STM32L4_EXTI @ sysbus 0x40010400
    numberOfOutputLines: 32
    [0-4] -> nvic@[6-10]
    [5-9] -> nvicInput23@[0-4]
    [10-15] -> nvicInput40@[0-5]

    16 -> nvic@1
    18 -> nvic@41
```

```

19 -> nvic@2
20 -> nvic@3
21 -> nvic@64
22 -> nvic@64

nvicInput23: Miscellaneous.CombinedInput @ none
  numberOfInputs: 5
  -> nvic@23

nvicInput40: Miscellaneous.CombinedInput @ none
  numberOfInputs: 6
  -> nvic@40

rng: Miscellaneous.STM32L4_RNG @ sysbus 0x50060800
  -> nvic@80

usart1: UART.STM32L4_UART @ sysbus <0x40013800, +0x100>
  frequency: 8000000
  -> nvic@37

usart2: UART.STM32L4_UART @ sysbus <0x40004400, +0x100>
  -> nvic@38

nvic: IRQControllers.NVIC @ sysbus 0xE000E000
  priorityMask: 0xF0
  systickFrequency: 8000000
  IRQ -> cpu@0

cpu: CPU.CortexM @ sysbus
  cpuType: "cortex-m4"
  nvic: nvic

crc: CRC.STM32_CRC @ sysbus 0x40023000
  series: STM32Series.L4

adc_common: Analog.STM32L4_ADC_Common @ sysbus 0x50040300

adc1: Analog.STM32L4_ADC @ sysbus <0x50040000, +0x100>
  IRQ-> nvic@18
  DMARequest -> dma1@0

adc2: Analog.STM32L4_ADC @ sysbus <0x50040100, +0x100>
  IRQ -> nvic@18

adc3: Analog.STM32L4_ADC @ sysbus <0x50040200, +0x100>
  IRQ -> nvic@47

gpioPortA: GPIOPort.STM32_GPIOPort @ sysbus <0x48000000, +0x400>
  modeResetValue: 0xFFFFFFFF
  pullUpPullDownResetValue: 0x00000000
  numberOfAFs: 16
  [0-15] -> exti@[0-15]

gpioPortB: GPIOPort.STM32_GPIOPort @ sysbus <0x48000400, +0x400>
  modeResetValue: 0xFFFFFFFF
  pullUpPullDownResetValue: 0x00000000
  numberOfAFs: 16
  [0-15] -> exti@[0-15]

gpioPortC: GPIOPort.STM32_GPIOPort @ sysbus <0x48000800, +0x400>
  modeResetValue: 0xFFFFFFFF
  pullUpPullDownResetValue: 0x00000000
  numberOfAFs: 16
  [0-15] -> exti@[0-15]

gpioPortD: GPIOPort.STM32_GPIOPort @ sysbus <0x48000C00, +0x400>
  modeResetValue: 0xFFFFFFFF

```

```

pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

gpioPortE: GPIOPort.STM32_GPIOPort @ sysbus <0x48001000, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

gpioPortF: GPIOPort.STM32_GPIOPort @ sysbus <0x48001400, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

gpioPortG: GPIOPort.STM32_GPIOPort @ sysbus <0x48001800, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

gpioPortH: GPIOPort.STM32_GPIOPort @ sysbus <0x48001C00, +0x400>
modeResetValue: 0xFFFFFFFF
pullUpPullDownResetValue: 0x00000000
numberOfAFs: 16
[0-15] -> exti@[0-15]

timer6: Timers.STM32_Timer @ sysbus 0x40001000
IRQ -> nvic@54
TRGO -> adc1@0
frequency: 8000000
initialLimit: 0xFFFF

timer7: Timers.STM32_Timer @ sysbus 0x40001400
IRQ -> nvic@55
frequency: 8000000
initialLimit: 0xFFFF

spi1: SPI.STM32L4_SPI @ sysbus 0x40013000
IRQ -> nvic@35

spi2: SPI.STM32L4_SPI @ sysbus 0x40003800
IRQ -> nvic@36

spi3: SPI.STM32L4_SPI @ sysbus 0x40003C00
IRQ -> nvic@51

uid: Memory.MappedMemory @ sysbus 0x1FFF7400
size: 0x400

sysbus:
init:
ApplySVD @platforms/svd/STM32L496.svd
# Required tags for STM32CubeMX-based projects (FLASH, USB)
Tag <0x40022010, 0x4002203F> "FLASH:RESET" 0x4000000
Tag <0x50000010, 0x5000003F> "USB:RESET" 0x4000000

```

Apêndice B

Arquivo stm32l4_spi.cs desenvolvido

Listing B.1: Modelo do periférico SPI (STM32L4) no Renode em C#

```
using Antmicro.Renode.Peripherals.Bus;
using Antmicro.Renode.Logging;
using Antmicro.Renode.Core.Structure;
using Antmicro.Renode.Core;
using Antmicro.Renode.Core.Structure.Registers;
using Antmicro.Renode.Utilities.Collections;

namespace Antmicro.Renode.Peripherals.SPI
{
    public sealed class STM32L4_SPI : NullRegistrationPointPeripheralContainer<ISPIPeripheral>,
        ↪ IWordPeripheral, IDoubleWordPeripheral, IBytePeripheral, IKnownSize
    {
        public STM32L4_SPI(IMachine machine, int bufferCapacity = DefaultBufferCapacity) :
            ↪ base(machine)
        {
            IRQ = new GPIO();
            DMARecieve = new GPIO();
            registers = new DoubleWordRegisterCollection(this);
            receiveBuffer = new CircularBuffer<byte>(DefaultBufferCapacity);
            DefineRegisters();
            Reset();
        }

        public byte ReadByte(long offset)
        {
            if(offset % 4 == 0)
            {
                return (byte)ReadDoubleWord(offset);
            }
            this.LogUnhandledRead(offset);
            return 0;
        }

        public void WriteByte(long offset, byte value)
        {
            this.NoisyLog("WriteByte off=0x{0:X} val=0x{1:X2}", offset, value);
            if(offset % 4 == 0)
            {
                WriteDoubleWord(offset, (uint)value);
            }
            else
            {
                this.LogUnhandledWrite(offset, value);
            }
        }
    }
}
```



```

public ushort ReadWord(long offset)
{
    return (ushort)ReadDoubleWord(offset);
}

public void WriteWord(long offset, ushort value)
{
    this.NoisyLog("WriteWord off=0x{0:X} val=0x{1:X4}", offset, value);
    WriteDoubleWord(offset, (uint)value);
}

public uint ReadDoubleWord(long offset)
{
    return registers.Read(offset);
}

public void WriteDoubleWord(long offset, uint value)
{
    registers.Write(offset, value);
}

public override void Reset()
{
    IRQ.Unset();
    DMARecieve.Unset();
    lock(receiveBuffer)
    {
        receiveBuffer.Clear();
    }
    registers.Reset();
}

public long Size
{
    get
    {
        return 0x400;
    }
}

public GPIO IRQ { get; }

public GPIO DMARecieve { get; }

private uint HandleDataRead()
{
    IRQ.Unset();
    lock(receiveBuffer)
    {
        if(receiveBuffer.TryDequeue(out var value))
        {
            Update();

            this.NoisyLog("Received 0x{0:X}.", value);

            return value;
        }
        return 0;
    }
}

private void HandleDataWrite(uint value)
{
    IRQ.Unset();
    lock(receiveBuffer)
    {

```

```

        var peripheral = RegisteredPeripheral;
        if(peripheral == null)
        {
            this.Log(LogLevel.Warning, "SPI transmission while no SPI peripheral is
↪ connected.");
            receiveBuffer.Enqueue(0x0);
            if((value & 0xFF00) != 0) receiveBuffer.Enqueue(0x0);
            return;
        }

        if((value & 0xFF00) != 0)
        {
            var b0 = (byte)(value & 0xFF);
            var b1 = (byte)((value >> 8) & 0xFF);

            var r0 = peripheral.Transmit(b0);
            var r1 = peripheral.Transmit(b1);

            receiveBuffer.Enqueue(r0);
            receiveBuffer.Enqueue(r1);

            this.NoisyLog("Transmitted 0x{0:X2} 0x{1:X2}, received 0x{2:X2} 0x{3:X2}.",
↪ b0, b1, r0, r1);
        }
        else
        {
            var b = (byte)value;
            var r = peripheral.Transmit(b);
            receiveBuffer.Enqueue(r);
            this.NoisyLog("Transmitted 0x{0:X2}, received 0x{1:X2}.", b, r);
        }

        if(rxDmaEnable?.Value ?? false)
        {
            DMARecieve.Blink();
        }
    }
    Update();
}

private void Update()
{
    var rxBufferNotEmpty = receiveBuffer.Count != 0;

    var txeie = txBufferEmptyInterruptEnable?.Value ?? false;
    var rxneie = rxBufferNotEmptyInterruptEnable?.Value ?? false;

    var rxIrq = rxBufferNotEmpty && rxneie;

    IRQ.Set(txeie || rxIrq);
}

private void DefineRegisters()
{
    Registers.CR1.Define(registers)
        .WithFlag(15, name: "BIDIMODE")
        .WithFlag(14, name: "BIDIOE")
        .WithFlag(13, name: "CRCEN")
        .WithFlag(12, name: "CRCNEXT")
        .WithFlag(11, name: "CRCL")
        .WithFlag(10, name: "RXONLY")
        .WithFlag(9, name: "SSM")
        .WithFlag(8, name: "SSI")
        .WithFlag(7, name: "LSBFIRST")
        .WithFlag(6, out spiEnable, name: "SPE")
        .WithValueField(3, 3, name: "BR")
        .WithFlag(2, name: "MSTR")

```

```

        .WithFlag(1, name: "CPOL")
        .WithFlag(0, name: "CPHA");

Registers.CR2.Define(registers)
    .WithReservedBits(15, 1)
    .WithFlag(14, name: "LDMA_TX")
    .WithFlag(13, name: "LDMA_RX")
    .WithFlag(12, name: "FRXTH")
    .WithValueField(8, 4, out dataSize, name: "DS")
    .WithFlag(7, out txBufferEmptyInterruptEnable, name: "TXEIE")
    .WithFlag(6, out rxBufferNotEmptyInterruptEnable, name: "RXNEIE")
    .WithFlag(5, name: "ERRIE")
    .WithFlag(4, name: "FRF")
    .WithFlag(3, name: "NSSP")
    .WithFlag(2, name: "SSOE")
    .WithFlag(1, name: "TXDMAEN")
    .WithFlag(0, out rxDmaEnable, name: "RXDMAEN");

Registers.SR.Define(registers)
    .WithValueField(11, 2, FieldMode.Read, valueProviderCallback: _ => OUL, name:
↪ "FTLVL")
    .WithValueField(9, 2, FieldMode.Read, valueProviderCallback: _ => OUL, name: "FRLVL")
    .WithFlag(8, FieldMode.Read, valueProviderCallback: _ => false, name: "FRE")
    .WithFlag(7, FieldMode.Read, valueProviderCallback: _ => false, name: "BSY")
    .WithFlag(6, FieldMode.Read, valueProviderCallback: _ => false, name: "OVR")
    .WithFlag(5, FieldMode.Read, valueProviderCallback: _ => false, name: "MODF")
    .WithFlag(4, FieldMode.Read, valueProviderCallback: _ => false, name: "CRCERR")
    .WithFlag(1, FieldMode.Read, valueProviderCallback: _ => true, name: "TXE")
    .WithFlag(0, FieldMode.Read, valueProviderCallback: _ => receiveBuffer.Count > 0,
↪ name: "RXNE");

Registers.DR.Define(registers)
    .WithValueField(0, 16, writeCallback: (_, val) => HandleDataWrite((uint)val),
    valueProviderCallback: _ => HandleDataRead(), name: "DR");

Registers.CRCPR.Define(registers)
    .WithTag("CRCPOLY", 0, 16);

Registers.RXCRCR.Define(registers)
    .WithTag("RXCRC", 0, 16);

Registers.TXCRCR.Define(registers)
    .WithTag("TXCRC", 0, 16);
}

private void HandleTransmit(byte value)
{
    var peripheral = RegisteredPeripheral;
    byte response = peripheral?.Transmit(value) ?? (byte)0;
    receiveBuffer.Enqueue(response);
}

private byte HandleReceive()
{
    return receiveBuffer.TryDequeue(out var val) ? val : (byte)0;
}

private IFlagRegisterField spiEnable;
private DoubleWordRegisterCollection registers;
private IFlagRegisterField txBufferEmptyInterruptEnable,
↪ rxBufferNotEmptyInterruptEnable, rxDmaEnable;
private CircularBuffer<byte> receiveBuffer;

private IValueRegisterField dataSize;

private const int DefaultBufferCapacity = 64;

```

```
private enum Registers
{
    CR1 = 0x00,
    CR2 = 0x04,
    SR = 0x08,
    DR = 0x0C,
    CRCPR = 0x10,
    RXCRCR = 0x14,
    TXCRCR = 0x18,
}
}
```