

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

RAFAEL GUSTAVO NAGEL

**IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE *SOFTWARE*
DO PROTOCOLO DA COMUNICAÇÃO CONVENCIONAL
LEITOR-MEDIDOR DA NORMA ABNT 14522**

Florianópolis, 2022

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

RAFAEL GUSTAVO NAGEL

**IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE *SOFTWARE*
DO PROTOCOLO DA COMUNICAÇÃO CONVENCIONAL
LEITOR-MEDIDOR DA NORMA ABNT 14522**

Trabalho de conclusão de curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
engenheiro eletrônico

Orientador:
Prof. Me. Hugo Marcondes

Florianópolis, 2022

Ficha de identificação da obra elaborada pelo autor.

Nagel, Rafael Gustavo

Implementação de uma biblioteca de software do protocolo da comunicação convencional leitor-medidor da norma ABNT 14522 / Rafael Gustavo Nagel; orientação de Hugo Marcondes. - Florianópolis, SC, 2022.
80 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. Bacharelado em Engenharia Eletrônica. Departamento Acadêmico de Eletrônica.
Inclui Referências.

1. NBR 14522. 2. Protocolo de comunicação. 3. Comunicação convencional. 4. Medidores eletrônicos. 5. Biblioteca de software. I. Marcondes, Hugo. II. Instituto Federal de Santa Catarina. III. Implementação de uma biblioteca de software do protocolo da comunicação convencional leitor-medidor da norma ABNT 14522.

IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE *SOFTWARE* DO PROTOCOLO DA COMUNICAÇÃO CONVENCIONAL LEITOR-MEDIDOR DA NORMA ABNT 14522

RAFAEL GUSTAVO NAGEL

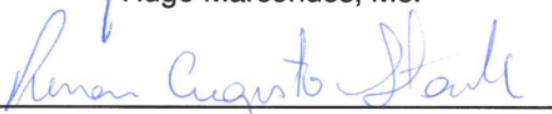
Este trabalho foi julgado adequado para obtenção do título de Engenheiro Eletrônico em julho de 2022 e aprovado na sua forma final pela banca examinadora do curso de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 25 de julho, 2022.

Banca Examinadora:



Hugo Marcondes, Me.



Renan Augusto Starke, Dr.



Samir Bonho, Me.

RESUMO

No Brasil, os medidores eletrônicos de energia devem seguir a norma NBR 14522 para o intercâmbio de informações com sistemas computadorizados, possibilitando uma compatibilidade entre os diversos sistemas e medidores de diferentes procedências. Atualmente há trabalhos que utilizam a comunicação convencional leitor-medidor contida nessa norma, contudo, tais trabalhos não disponibilizam uma implementação pública e de livre acesso dessa comunicação. Nesse contexto, este trabalho apresenta uma biblioteca contendo uma implementação de código aberto e portátil da comunicação convencional da norma NBR 14522 para ser utilizada por aplicações variadas que desejam ler medidores compatíveis com a norma. Este trabalho também contém uma aplicação em linha de comando que utiliza a biblioteca implementada para demonstrar a comunicação com um medidor real a partir de um *Raspberry*. Os resultados do trabalho atingiram seu objetivo dado que a biblioteca apresentou alto grau de portabilidade durante a escrita da aplicação e dos testes, bem como executou com sucesso todas as leituras com um medidor existente compatível com a norma.

Palavras-chave: NBR 14522. Protocolo de comunicação. Comunicação convencional leitor-medidor. Medidores eletrônicos de energia. Biblioteca de *software*.

ABSTRACT

In Brazil, electronic energy meters should follow the NBR 14522 standard for data exchange with computerized systems, enabling compatibility between different systems and meters from different vendors. Currently, there are projects using the conventional reader-meter communication contained in this standard, however, such projects do not provide an open-source implementation of that communication. In this context, this work presents a library containing an open source and portable implementation of the conventional communication of the NBR 14522 standard to be used by varied applications that wish to read meters compatible with the standard. This work also contains a command-line application that uses the implemented library to demonstrate communication with a real meter from a Raspberry computer. The results of this work reached its objective as the library presented a high degree of portability during the writing of the application and tests, as well as successfully performing all readings with an existing meter compatible with the standard.

Keywords: NBR 14522. Communication protocol. Reader-meter conventional communication. Electronic energy meters. Software library.

LISTA DE ILUSTRAÇÕES

Figura 1 – Medidores de energia elétrica	13
Figura 2 – Composição em <i>bytes</i> das mensagens trocadas entre leitor e medidor	16
Figura 3 – Sequência de comando-resposta(s) sem erros e sem atrasos	18
Figura 4 – Sequências de comando-resposta com e sem erros de <i>Cyclic redundancy check</i> (CRC)	19
Figura 5 – Sequências de comando-resposta com atraso de sequência	20
Figura 6 – Máquina de estados da implementação da comunicação convencional leitor-medidor da NBR 14522	34
Figura 7 – Diagrama de classes da implementação do leitor na linguagem <i>C++</i>	39
Figura 8 – Medidor eletrônico modelo E750 da empresa Landys+Gyr	46

LISTA DE TABELAS

Tabela 1 – Alguns comandos presentes na NBR 14522 (ABNT, 2008)	15
Tabela 2 – Leituras padronizadas presentes na NBR 14522 (ABNT, 2008) . . .	16
Tabela 3 – Exemplos de respostas enviadas ao medidor ao invocar a aplicação por linha de comando	45
Tabela 4 – Alguns valores da resposta do medidor ao comando "leitura das grandezas instantâneas"	47
Tabela 5 – Respostas compostas do medidor	48

LISTA DE CÓDIGOS

Código A.1 – Classe <i>LeitorFSM</i>	56
Código A.2 – Classe <i>Leitor</i>	62
Código A.3 – Aplicação de um leitor com interface via linha de comando	65
Código A.4 – Implementação da interface <i>TimerPolicy</i> usada na aplicação	67
Código A.5 – Implementação da interface <i>SerialPolicy</i> usada na aplicação	67
Código A.6 – Testes unitários da máquina de estados da classe <i>LeitorFSM</i>	68

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Justificativa	11
1.2	Descrição do problema	12
1.3	Objetivo geral	12
1.4	Objetivos específicos	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Medidores eletrônicos de energia elétrica	13
2.2	Norma ABNT NBR 14522	14
2.2.1	Comunicação convencional leitor-medidor	15
2.2.1.1	Definição dos tempos	20
2.2.2	Trabalhos relacionados	22
2.3	Portabilidade	22
2.4	Programação orientada a objetos	23
2.5	Princípios para arquitetura de <i>software</i>	24
2.6	Testes unitários	25
2.7	Ferramentas, <i>frameworks</i> e bibliotecas	26
2.7.1	A linguagem de programação <i>C++</i>	26
2.7.2	CMake	28
2.7.3	Doctest	29
2.7.4	Biblioteca para leitura e escrita da porta serial	30
3	METODOLOGIA	31
4	RESULTADOS	33
4.1	Máquina de estados do protocolo	33
4.2	Classes	38
4.3	Organização do repositório e <i>CMake</i>	42
4.4	Testes unitários	43
4.5	Exemplo de aplicação para leitura de um medidor real	44
4.5.0.1	Cenário de execução	45
4.5.0.2	Resultados da execução da aplicação	46
5	ANÁLISE E DISCUSSÃO	49
	REFERÊNCIAS	52

APÊNDICES	55
APÊNDICE A – CÓDIGO-FONTE PRINCIPAL DA BIBLIOTECA . .	56

1 INTRODUÇÃO

Com o avanço da tecnologia, os ativos residenciais, industriais e comerciais fornecem cada vez mais informações de forma digital e sem necessidade de intervenção humana. Uma classe de ativo são os medidores de energia elétrica. Os medidores fornecem grandezas pertinentes ao consumo de energia elétrica para as concessionárias de energia bem como para as unidades consumidoras. Tais informações de consumo são cruciais para que concessionárias cobrem um valor preciso de consumo de cada unidade.

Outras informações como frequência da rede, valores de pico, acesso ao histórico de consumo, desligamentos da rede, entre outras, são relevantes para fins de monitoramento da qualidade do serviço. Complementarmente, a possibilidade de executar remotamente comandos de faturamento — agendados ou sob demanda — nos medidores proporciona flexibilidade às concessionárias e redução nos custos de visitas presenciais.

Nesse cenário, o Brasil, através da Associação Brasileira de Normas Técnicas (ABNT), lançou a norma Norma Brasileira (NBR) 14522, intitulada *Intercâmbio de informações para sistemas de medição de energia elétrica* (ABNT, 2008). Esta norma tem o objetivo de alcançar a compatibilidade entre os sistemas e equipamentos de medição de energia elétrica ao definir um padrão para o intercâmbio de informações. A seção 2.2 descreve em maiores detalhes essa norma.

Este trabalho propõe a implementação de uma biblioteca de *software* de código aberto e de livre acesso do padrão contido na NBR 14522 para intercâmbio de informações com medidores.

1.1 Justificativa

Com o aumento da adoção dos medidores eletrônicos e da coleta de informações automatizadas, uma consequência é a ampliação do suporte de medidores à NBR 14522 pelos fabricantes, tornando-se inclusive um diferencial competitivo no mercado. A norma facilita a integração dos medidores com soluções de automação e integração, pois a comunicação com os diferentes equipamentos dependeria apenas da compatibilidade com a norma.

De maneira geral, a adoção de padrões e protocolos formalmente estabelecidos na indústria tendem a promover produtos e serviços mais competitivos e funcionais. A reutilização de soluções para problemas conhecidos permite que as empresas concentrem recursos em outros aspectos do produto ou serviço que trazem um maior valor agregado para seus clientes.

1.2 Descrição do problema

Apesar de existirem projetos e/ou soluções de leitura de medidores compatíveis com a NBR 14522, como por exemplos os trabalhos e/ou projetos comentados na subseção 2.2.2, ainda assim não foram encontradas pelo autor deste trabalho implementações de livre acesso das especificações contidas na norma. Pesquisas em repositórios públicos em plataformas como o *Github* também não levam a nenhuma implementação existente e documentada. Além disso, o acesso ao conteúdo da norma não é gratuito, o que pode ser mais uma barreira para interessados na leitura dos medidores via sistemas computacionais. Desta maneira, este trabalho pergunta-se se é possível desenvolver uma biblioteca aberta de *software* que implemente funções de leitura segundo a norma NBR14522.

1.3 Objetivo geral

O objeto deste trabalho é implementar uma biblioteca em *software* de livre acesso que execute a comunicação com medidores eletrônicos de energia elétrica compatíveis com a NBR 14522, a partir da comunicação convencional leitor-medidor presente na norma.

1.4 Objetivos específicos

A implementação desta biblioteca pode ser dividida nos seguintes objetivos específicos:

- a) compreensão do protocolo da comunicação convencional leitor-medidor presente na norma ABNT NBR 14522;
- b) contextualização dos cenários de uso do protocolo e trabalhos relacionados;
- c) definição dos requisitos de *software* da biblioteca;
- d) implementação de *software* contendo o comportamento do leitor da comunicação convencional;
- e) demonstração de resultados da comunicação com um medidor existente utilizando a biblioteca implementada.

2 FUNDAMENTAÇÃO TEÓRICA

O tema abordado por este trabalho relaciona dois principais assuntos: a norma NBR 14522 (ABNT, 2008) de medidores eletrônicos de energia e o desenvolvimento de uma biblioteca de *software*.

2.1 Medidores eletrônicos de energia elétrica

É possível classificar os medidores de energia elétrica em dois tipos: eletromecânicos e eletrônicos (Figura 1). Os eletromecânicos funcionam a partir do princípio da indução eletromagnética e a medição da energia é realizada pela contagem do número de rotações que um disco de alumínio realiza em um determinado período. São de menor custo e, conforme a Agência Nacional de Energia Elétrica (ANEEL), ainda compõem a maioria do mercado brasileiro. Já os medidores eletrônicos executam as medições a partir de transdutores de corrente e tensão, eliminando a necessidade de componentes eletromecânicos. As principais vantagens dos medidores eletrônicos em relação ao anterior são a exatidão das medições e baixa manutenção necessária ao longo dos anos. (AFONSO, 2017).

Figura 1 – Medidores de energia elétrica

(a) Medidor eletromecânico



(b) Medidor eletrônico



Fonte: (Nansen S/A, 2022)

Mais especificamente, a norma ABNT NBR 14519 (ABNT, 2000) define medidor eletrônico de energia elétrica como "medidor estático no qual a corrente e a tensão agem sobre elementos de estado sólido (componentes eletrônicos) para produzir uma informação de saída proporcional à quantidade de energia elétrica medida."

A mesma norma especifica e determina os requisitos para medidores eletrônicos de energia elétrica de corrente alternada. A norma descreve requisitos mecânicos

e características construtivas, níveis de tensão e correntes que devem ser suportados, quais informações devem ser armazenadas e por quanto tempo em caso de falta de energia, entre outros detalhes. A norma aplica-se para as classes de consumo A, B, C e D (Veja (ANEEL, 2010) para mais informações sobre as classes de consumo). Pela norma, a classificação dos medidores eletrônicos deve considerar características como: quais grandezas são medidas (energia ativa, reativa, demanda, etc.), valor máximo de corrente, tipo de sensor da medição de corrente e tensão, tecnologia do mostrador, entre outras.

Na especificação de características construtivas da NBR 14519, inclui-se a especificação de uma porta óptica. É uma interface de comunicação a partir da qual se obtém uma transmissão serial assíncrona bidirecional com o medidor. A norma também permite que os fabricantes forneçam outras interfaces de comunicação para um medidor eletrônico. Alguns exemplos de interfaces utilizadas por alguns fabricantes são RS232 e RS485, conforme descrito na subseção 2.2.2.

É através de uma dessas interfaces de comunicação que o protocolo de comunicação convencional (subseção 2.2.1) da ABNT NBR 14522 ocorre.

2.2 Norma ABNT NBR 14522

Esta norma tem o objetivo de padronizar o intercâmbio de informações entre sistemas e medidores de energia elétrica de diversos fabricantes. O documento é dividido em três principais partes: comunicação de dados, formato de dados e informações gerais. A comunicação de dados define os seguintes protocolos para comunicação entre leitor¹ e medidor:

- **Comunicação direcional** leitor-medidor: considerada obsoleta pela norma;
- **Comunicação remota síncrona**: baseia-se no protocolo *Binary Synchronous Communication* (BSC) criado pela *International Business Machines Corporation* (IBM), atualmente considerado um protocolo legado que tem sido amplamente substituído por outros protocolos mais eficientes. Além disso, nenhum trabalho relacionado à norma utilizou-se desta comunicação;
- **Saída serial de usuário**: comunicação assíncrona unidirecional cujo sentido é do medidor para o leitor, onde a cada segundo cheio, o medidor envia um bloco de dados pela saída serial do usuário. Os dados fornecidos são mais limitados comparados à comunicação convencional. Por exemplo, não há informações sobre interrupções no fornecimento de energia elétrica, que, conforme (ROSA,

¹ no contexto da norma NBR 14522, o leitor pode ser entendido como um sistema que realiza a leitura de informações dos medidores

Tabela 1 – Alguns comandos presentes na NBR 14522 (ABNT, 2008)

Código	Descrição
0x14	Leitura das grandezas instantâneas
0x20	Leitura de parâmetros com reposição de demanda
0x21	Leitura de parâmetros sem reposição de demanda atuais
0x22	Leitura de parâmetros sem reposição de demanda anteriores
0x23	Leitura de registradores dos canais visíveis após a última reposição de demanda
0x24	Leitura de registradores dos canais visíveis relativos à última reposição de demanda
0x25	Leitura dos períodos de falta de energia
0x26	Leitura dos contadores da memória de massa desde a última reposição de demanda
0x27	Leitura dos contadores da memória de massa anteriores à última reposição de demanda
0x28	Leitura dos registros de alterações
0x29	Alteração da data
0x30	Alteração da hora
0x31	Alteração do intervalo de demanda
0x32	Alteração dos feriados nacionais
0x41	Leitura de registradores parciais anteriores do primeiro canal visível
0x42	Leitura de registradores parciais anteriores do segundo canal visível
0x43	Leitura de registradores parciais anteriores do terceiro canal visível
0x51	Leitura de parâmetros sem reposição de demanda, com leitura de memória de massa
0x52	Leitura dos contadores da memória de massa
0x80	Leitura de parâmetros de medição

2007), "são essenciais no acompanhamento do desempenho da concessionária local em relação aos índices de continuidades definidos pelo agente regulador."

- **Comunicação convencional leitor-medidor:** é um protocolo de comunicação serial de velocidade de 9.600 *bits* por segundo, bidirecional não simultâneo e assíncrona. Baseia-se no envio de comandos para o medidor e no recebimento das respectivas respostas pelo leitor. Baseado em pesquisas do autor, a maioria dos trabalhos relacionados à norma NBR 14522 utilizam-se da comunicação convencional, bem como o presente trabalho.

2.2.1 Comunicação convencional leitor-medidor

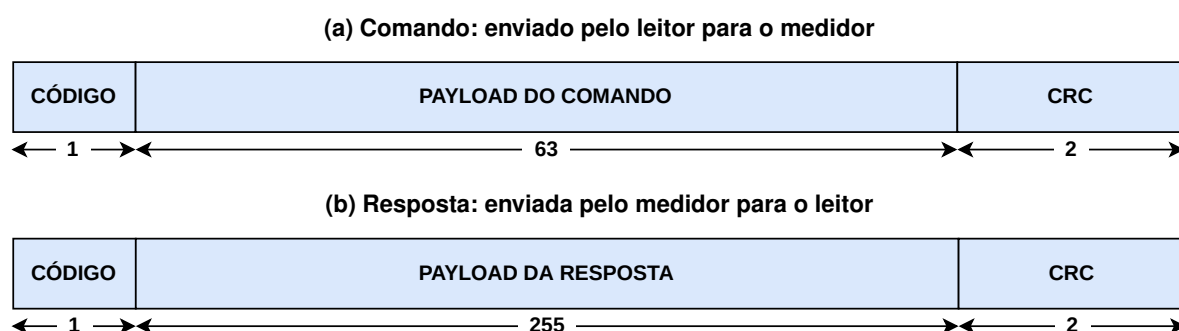
O protocolo da comunicação convencional leitor-medidor define as mensagens trocadas entre leitor e medidor: comandos, respostas e sinalizadores. O leitor envia comandos ao medidor e este envia respostas aos comandos recebidos. Ambos utilizam-se de sinalizadores para auxiliar o fluxo de comandos e respostas. A norma especifica um total de 67 comandos, dos quais alguns estão listados na Tabela 1. A lista completa pode ser conferida na NBR 14522.

A norma também descreve algumas leituras padronizadas, compostas de sequências de comandos e suas respectivas respostas. Essas leituras são os comandos mais frequentemente utilizados pelas concessionárias. A Tabela 2 lista as leituras padronizadas da norma.

Tabela 2 – Leituras padronizadas presentes na NBR 14522 (ABNT, 2008)

Nome	Códigos dos comandos da sequência
Reposição de demanda	0x20, 0x80, 0x24, 0x25, 0x28, 0x27
Verificação	0x21, 0x80, 0x23, 0x25, 0x28, 0x26
Recuperação	0x22, 0x80, 0x24, 0x25, 0x28, 0x27
Reposição de demanda resumida	0x20, 0x80, 0x24, 0x41, 0x42, 0x43, 0x25, 0x28
Verificação resumida	0x21, 0x80, 0x23, 0x44, 0x45, 0x46, 0x25, 0x28
Recuperação resumida	0x22, 0x80, 0x24, 0x41, 0x42, 0x43, 0x25, 0x28
Verificação da memória de massa	0x51, 0x80, 0x52

Todo comando enviado ao medidor tem exatamente 66 *bytes*, nesta ordem: 1 *byte* com o código do comando, 63 de conteúdo (*payload*) e 2 finais de checagem de erro usando CRC padrão *American National Standards Institute* (ANSI) de 16 *bits*. Já a resposta ao comando, enviada pelo medidor, tem 258 *bytes*, nesta ordem: 1 *byte* com o código do comando enviado pelo leitor, 255 de *payload* e 2 de CRC ANSI 16 *bits* (Figura 2). A checagem CRC é calculada sobre o código e *payload*.

Figura 2 – Composição em bytes das mensagens trocadas entre leitor e medidor

Fonte: Elaboração do autor baseada na NBR 14522 (ABNT, 2008)

O código e o *payload* utilizam a codificação *Binary-coded decimal* (BCD). Nesta codificação, o resultado numérico das informações deve ser interpretado no sistema decimal a cada 4 *bits* de informação (*nibble*), considerando a ordem de leitura dos números decimais (MOECKE, 2020).

O código de uma resposta deve ser igual ao código do respectivo comando, salvo duas respostas que são exceções: quando o leitor solicita um comando não implementado pelo medidor, então este retorna a resposta "Informação de comando não implementado" (código 0x39). A outra exceção é a resposta "Informação de ocorrência no Medidor" (código 0x40), enviada pelo medidor quando ocorreu algum problema e este não pode executar o comando solicitado.

Um comando pode ainda ser dividido em duas categorias: simples e composto. Um comando simples enviado pelo leitor recebe do medidor uma única resposta.

Já um comando composto pode ter inúmeras respostas do medidor. A Figura 3 ilustra as duas categorias. Os comandos compostos referem-se a pedidos de leitura dos contadores salvos na memória de massa² do medidor. Existem apenas três comandos compostos na norma: 0x26, 0x27 e 0x52 (Veja Tabela 1).

Existem quatro sinalizadores definidos na norma para auxiliar o fluxo de comandos e respostas entre medidor e leitor. Cada sinalizador consiste em 1 *byte*. São eles:

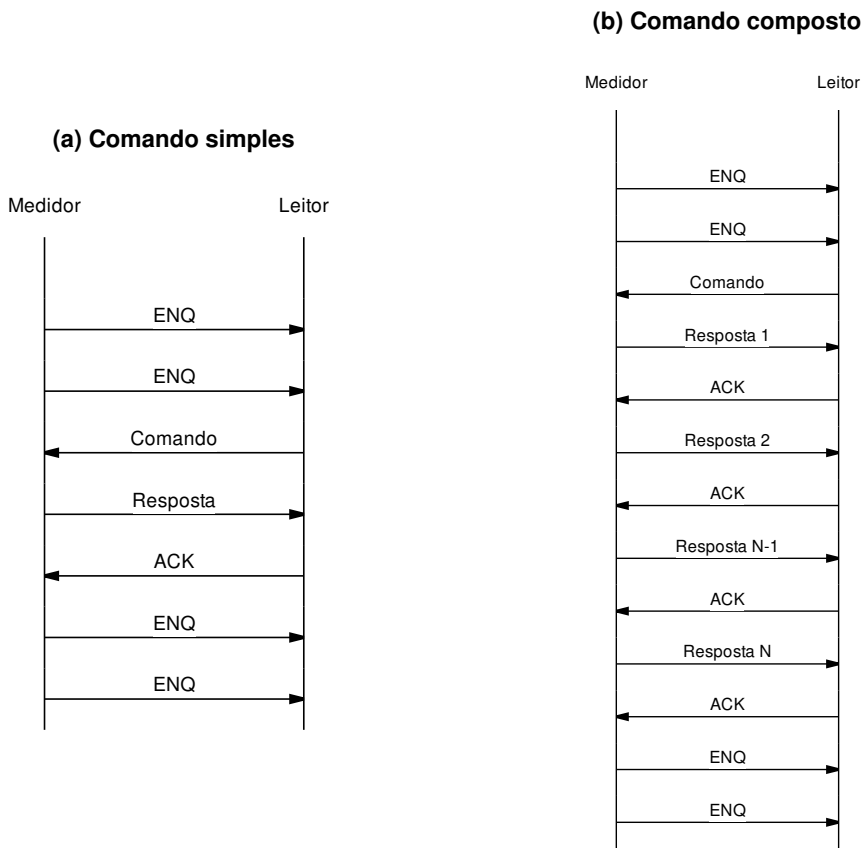
- **ENQ** (0x05): transmitido apenas pelo medidor para sincronizar a comunicação com o leitor antes que algum comando seja transmitido pelo leitor. Esse sinalizador é transmitido periodicamente conforme tempos definidos pela norma (Equações 2.3 e 2.4).
- **ACK** (0x06): transmitido apenas pelo leitor para indicar que uma resposta originada do medidor foi recebida sem erros de CRC.
- **NAK** (0x15): leitor e medidor podem transmitir. É transmitido quando a resposta recebida (no caso do medidor), ou o comando recebido (no caso do leitor) contém erro de CRC (Veja Figura 4).
- **WAIT** (0x10): leitor e medidor podem transmitir. É usado quando o medidor ou o leitor querem atrasar a sequência comando-resposta, mantendo a sincronização entre os dois. Por exemplo, esse sinalizador poderia ser utilizado pelo medidor, após receber um comando, para solicitar ao leitor que aguarde até o medidor conseguir processar o comando e respondê-lo (Veja Figura 5).

Antes da sincronização pelo medidor com envios periódicos do sinalizador *ENQ*, é necessário ocorrer o estabelecimento da conexão entre medidor e leitor na porta serial. Assim, para que o medidor detecte a presença do leitor, este deve colocar o nível lógico 1 em sua porta serial conectada no medidor por pelo menos 1 segundo. Após o medidor perceber o nível lógico 1 estável durante 1 segundo, este inicia o envio do *ENQ* periodicamente.

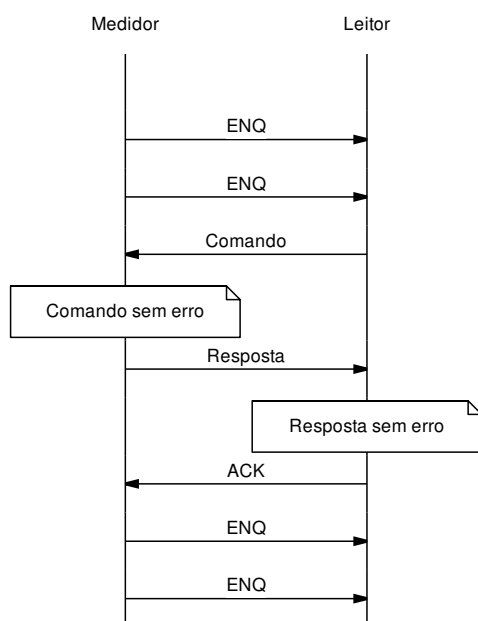
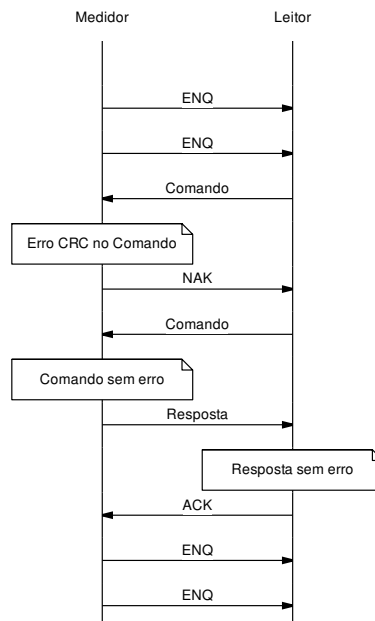
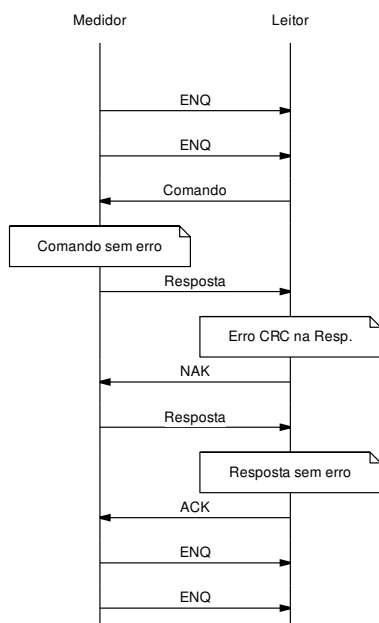
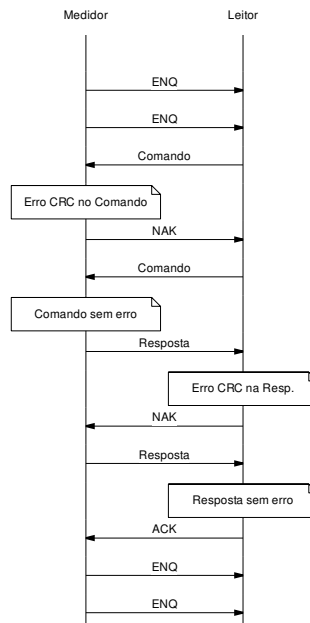
A norma define apenas três elementos: comando, resposta e sinalizadores. O medidor envia apenas respostas e o leitor apenas comandos. Quando o leitor recebe um dado que não seja sinalizador nem resposta, ou o medidor recebe um dado que não seja sinalizador nem comando, então deve haver interrupção da troca de informações, mas a conexão entre os dois é mantida. Após essa interrupção, é preciso ocorrer uma nova sincronização entre leitor e medidor.

² A memória de massa é um dispositivo eletrônico que faz parte integrante do medidor, onde são armazenados pulsos para posterior visualização e/ou recuperação (ABNT, 2000)

Figura 3 – Sequência de comando-resposta(s) sem erros e sem atrasos



Fonte: Elaboração do autor baseada na NBR 14522 (ABNT, 2008)

Figura 4 – Sequências de comando-resposta com e sem erros de CRC**(a) Sem erros de CRC****(b) Com erros de CRC no comando****(c) Com erros de CRC na resposta****(d) Com erros de CRC no comando e na resposta**

Fonte: Elaboração do autor baseada na NBR 14522 (ABNT, 2008)

de T_{rev} ($T_{min,rev}$) a ser cumprido pelo medidor e leitor, no envio de comandos, respostas e sinalizações é dado pela Equação 2.2:

$$T_{min,rev} = T_{car} + 1 \text{ ms} \quad (2.2)$$

Conforme explicado anteriormente, o medidor envia sinalizadores *ENQ* periodicamente para o leitor, após a conexão ter sido estabelecida. O tempo mínimo e máximo entre o envio de dois *ENQs* subsequentes são dados pelas Equações 2.3 e 2.4, respectivamente.

$$T_{min,enq} = T_{min,rev} + 20 \text{ ms} \quad (2.3)$$

$$T_{max,enq} = T_{min,rev} + 500 \text{ ms} \quad (2.4)$$

O tempo de sincronização é definido como o período entre o envio de um *ENQ* pelo medidor e o envio do subsequente caractere pelo leitor. Seu tempo máximo é dado pela Equação 2.5:

$$T_{max,sinc} = T_{min,rev} + 10 \text{ ms} \quad (2.5)$$

O tempo máximo entre dois caracteres consecutivos de um mesmo comando ou resposta é dado pela equação Equação 2.6:

$$T_{max,car} = T_{car} + 5 \text{ ms} \quad (2.6)$$

O tempo máximo de resposta (Equação 2.7) é definido como o tempo máximo entre o último caractere transmitido de um comando, resposta ou sinalizador e o caractere recebido subsequente. O tempo máximo de sincronização (Equação 2.5) é uma exceção ao tempo máximo de resposta.

$$T_{max,rsp} = T_{min,rev} + 500 \text{ ms} \quad (2.7)$$

Ainda, em relação ao sinalizador de atraso de sequência, a norma define o tempo máximo que o medidor possui para enviar um *ENQ* após envio de um *WAIT* e o tempo máximo entre dois *WAITs* subsequentes. Ambos valores são 305 segundos.

A norma não explicita qual deve ser o comportamento do leitor nem do medidor quando um dos lados não cumpre os tempos definidos. Subentende-se, então, que nesses casos o outro lado pode optar por ser tolerante ou interromper a troca de informações.

2.2.2 Trabalhos relacionados

Os trabalhos relacionados investigados pelo autor não tratam da norma NBR 14522 como tema principal, mas a utilizam como o meio para obtenção de dados de medidores eletrônicos. Tais trabalhos utilizaram estes dois protocolos de comunicação contidos na norma: **Saída serial de usuário** e **Comunicação convencional leitor-medidor**.

Os trabalhos em (LOUSADA, 2009) e (LOUSADA; MINI, 2016) executam as leituras de medição a partir dos comandos descritos na subseção 2.2.1 no medidor eletrônico Spectrum S (Nansen S/A, 2011), fabricado pela empresa Nansen S/A. O medidor disponibiliza diferentes interfaces de comunicação: porta óptica, RS232, RS485, RJ45 e serial do usuário. É a partir da interface RS232 que a comunicação convencional (subseção 2.2.1) ocorre. O leitor do medidor é implementado para um microcontrolador em um *kit* de desenvolvimento embarcado XBee Série 2 (XB24BPKD). Posteriormente os dados são encaminhados por uma rede ZigBee.

O trabalho (ANJOS, 2013) também utiliza-se de uma rede ZigBee como infraestrutura de medição e da comunicação convencional. A partir desta, os principais comandos de leitura executados são para obter as grandezas instantâneas (página fiscal), consumo e períodos de falta de energia de cinco medidores eletrônicos da *Eletra Energy solutions*.

O projeto em (BRAGA *et al.*, 2016) lê o comando de grandezas instantâneas (Veja Tabela 1) a partir da comunicação convencional com o medidor CRONOS 7023 (Eletra Energy Solutions, 2010) fabricado pela empresa *Eletra Energy solutions*. O medidor fornece diferentes interfaces de comunicação: RS232, RS485, saída de pulsos e saída de usuário, das quais as duas primeiras são compatíveis com o protocolo da comunicação convencional. Esse projeto em questão utilizou a RS485. A plataforma de *hardware* do leitor é baseada em um sistema embarcado com processador ARM. O projeto integra essas leituras com uma rede de sensores sem fio.

Os autores em (AMARAL; SANTOS; MENDONÇA, 2016) também recorrem ao protocolo da comunicação convencional para leituras de medidores a partir de um aplicativo em um *smartphone*. O mesmo protocolo é usado na tese (MONTE, 2019), onde o autor utiliza-se de um dispositivo *Raspberry Pi* (Raspberry Pi Foundation, 2012) que executa a leitura de dois medidores diferentes, cada um conectado em uma porta USB do dispositivo.

2.3 Portabilidade

Portabilidade é um atributo desejável para a maioria dos *softwares*. Conforme (MOONEY, 2000), portabilidade pode ser definida como a capacidade de produzir uma

versão executável de uma unidade de *software*³ em um novo ambiente computacional, baseado em uma versão existente. Mais especificamente, um *software* apresenta o atributo de portabilidade em uma gama de ambientes conforme o custo de transporte e adaptação dele para um novo ambiente for menor que o custo de desenvolver um novo do zero.

Assim, deve-se enxergar a portabilidade como uma alternativa à reimplantação de um componente já existente. Uma unidade de *software* é considerada perfeitamente portátil quando o custo da portabilidade é zero. Tal custo não é trivial de mensurar, visto que a portabilidade em diferentes ambientes computacionais pode ocorrer em diferentes níveis. Portanto, é mais preciso caracterizar um *software* por seu grau de portabilidade.

Ao iniciar um novo projeto, torna-se relevante uma análise dos requisitos em relação ao grau de portabilidade desejado e escolher um ferramental adequado para atingir tais requisitos. Neste trabalho, considerou-se desejável para a biblioteca um alto grau de portabilidade para os ambientes computacionais embarcados e os para os principais sistemas operacionais atualmente utilizados, dado o contexto de utilização da norma.

2.4 Programação orientada a objetos

A Programação orientada a objetos (POO) é um paradigma de desenvolvimento de *software* baseado no conceito de agrupar dados e comportamentos relacionados a esses dados em uma estrutura denominada objetos. Os objetos são descritos e definidos por modelos denominados classes (SHVETS, 2019).

Uma classe define os dados de seus objetos através de variáveis e o comportamento através de funções. Assim, diferentes objetos contendo diferentes valores para suas variáveis podem se referir a uma mesma classe. A classe de um objeto pode ser considerada seu tipo.

O que torna esse paradigma vantajoso na engenharia de *software* em muitos contextos em relação a outros paradigmas (e.g. procedural) são quatro pilares: abstração, polimorfismo, encapsulamento e herança.

- **Abstração:** é um modelo de um objeto real ou fenômeno, limitado a um contexto específico, que representa todos os detalhes relevantes ao contexto e omite o restante dos detalhes.
- **Encapsulamento:** é a habilidade de um objeto esconder sua implementação, expondo ao restante dos componentes do *software* somente uma interface limitada.

³ o autor (MOONEY, 2000) define uma unidade de *software* como programa de aplicação, programa de sistema ou componente de um programa de aplicação ou sistema.

- **Herança:** mecanismo que permite especializar um comportamento mais genérico para um mais especializado através de uma hierarquia entre objetos que fornece a relação "é um tipo de". Assim, permite construir novas classes a partir de classes existentes. Essas novas classes (denominadas *derivadas*) devem respeitar o conjunto de funções da classe herdada (denominada *base*). Tal mecanismo promove a reutilização e extensão de códigos existentes.
- **Polimorfismo:** o polimorfismo depende do mecanismo de herança e permite ao programador a manipulação de uma referência a um objeto de uma classe base, mas que possui o comportamento de uma classe derivada. Assim, por exemplo, um objeto do tipo Cachorro (derivada) pode ser manipulado a partir da interface de uma classe Animal (base), mas mantendo o comportamento de Cachorro.

A maneira que esses quatro pilares são implementados e utilizados varia conforme a linguagem de programação utilizada. Neste trabalho escolheu-se a linguagem *C++*, conforme explicado e justificado na seção 2.7.

Por fim, esses pilares da orientação a objetos se tornam úteis quando sua utilização proporciona benefícios ao projeto em custo e tempo. Redução no tempo de desenvolvimento proporciona vantagens competitivas como entrar em um determinado mercado antes que outras empresas. E a redução nos custos de desenvolvimento pode tornar o produto ou serviço mais barato.

2.5 Princípios para arquitetura de *software*

A orientação a objetos (seção 2.4) fornece ferramentas para o desenvolvedor que possibilitam escrever códigos mais reutilizáveis e extensíveis, duas características desejáveis para qualquer projeto que visa redução de custos e de tempo. Contudo, para atingir tais características, torna-se necessário utilizar os recursos disponíveis de uma linguagem de programação da maneira e no contexto apropriado.

Nesse sentido, existem muitos livros que abordam o tema da engenharia de *software* e descrevem padrões e boas práticas para os mais variados contextos. O autor em (MARTIN, 2002) foi um dos pioneiros no assunto a formalizar alguns princípios gerais que podem ser aplicados a maioria dos projetos de *software* visando proporcioná-los maior flexibilidade, compreensão e manutenibilidade. Os princípios são:

1. **Princípio da responsabilidade única:** cada classe deve ser responsável por uma funcionalidade única do *software*. Por exemplo, uma classe que calcula o preço final de um produto e também calcula os impostos embutidos no preço final, não respeita este princípio. Neste caso, algumas consequências são: alto

acoplamento (geração de maiores dependências entre componentes/módulos do sistema), dificuldade de reaproveitar uma classe existente e de realizar testes unitários.

2. **Princípio aberto-fechado:** classes devem ser abertas para extensão e fechadas para modificação. Isto é, quando novas funcionalidades precisam ser adicionadas a um componente do sistema, não se deve alterar códigos existentes do componente, mas sim adicionar novos códigos.
3. **Princípio da substituição de Liskov:** conforme descrito na seção 2.4, a orientação a objetos fornece o conceito de polimorfismo e herança, que permitem que uma classe derivada herde a interface de uma classe base, e a derivada consiga ser manipulada através de uma referência à classe base. Este princípio diz que em qualquer circunstância um objeto da classe derivada deve ser substituível por um objeto da classe base sem necessidade de alterações no código existente.
4. **Princípio da segregação da interface:** no contexto da POO uma interface é um conjunto de funções declaradas⁴ de uma classe. Quando pelo menos uma das funções da classe não contém uma implementação, então tal classe denomina-se abstrata. Tais classes só são utilizadas como base, onde as derivadas são obrigadas a implementar as funções não implementadas da base. Nesse contexto, este princípio afirma que uma derivada não deve ser forçada a implementar funções que não irá utilizar. Em outras palavras, é mais adequado criar mais classes abstratas com suas especificidades ao invés de ter uma classe abstrata mais genérica.
5. **Princípio da inversão da dependência:** afirma que as dependências de uma classe, quando existirem, devem ser compostas de classes abstratas. Esse princípio reduz o acoplamento entre módulos do sistema, facilita os testes unitários de um componente bem como aumenta o seu potencial de reutilização futuro.

Tais princípios mostram-se úteis especialmente em etapas iniciais de um novo projeto, situação em que é difícil ter clareza do grau de flexibilidade e manutenibilidade que um projeto terá. Por esse motivo, durante a implementação da biblioteca deste trabalho buscou-se seguir tais princípios.

2.6 Testes unitários

Testes unitários significam testar os componentes ou módulos de um sistema de maneira individual. O que define um componente e seu limite normalmente é uma

⁴ a declaração de uma função é apenas o seu cabeçalho, isto é, seu nome, seus parâmetros e seu valor de retorno, sem incluir a implementação (corpo) da função.

interface de programação ou *Application Program Interface* (API) a partir de qual é possível verificar a saída, isto é, o resultado gerado pelo componente dado uma ou mais entradas.

Em geral, padrões e princípios de projetos de *software* buscam a escrita de componentes que são facilmente testáveis em um ambiente controlado.

A adoção de testes unitários em um projeto tem vantagens e desvantagens. Exige um custo de tempo adicional para a escrita dos testes. Além disso, existe uma curva de aprendizagem para utilizar um *framework* de testes para a linguagem de programação que está sendo utilizada. Por outro lado, a testagem individual dos componentes permite que o programador tenha *feedbacks* em estágios iniciais e intermediários do processo de desenvolvimento do sistema. Também reduz a complexidade de depuração e resolução de problemas durante o desenvolvimento. Outro ponto é que os testes podem ajudar a documentar o comportamento esperado dos componentes.

As ferramentas e *frameworks* de testes buscam automatizar a testagem de todos os componentes a partir da execução de poucos comandos ou botões. Isso permite que todos os componentes sejam testados regularmente durante o desenvolvimento sem comprometer a produtividade do programador. Uma consequência desse processo é que novas modificações ao código-fonte podem ser verificadas de maneira a garantir que os componentes do sistema continuam funcionando conforme o esperado.

2.7 Ferramentas, *frameworks* e bibliotecas

2.7.1 A linguagem de programação C++

Conforme (STROUSTRUP, 2013), C++ é uma linguagem de programação de propósito geral que fornece um modelo de *hardware* direto e eficiente combinado com recursos que permitem definir abstrações de baixo ou nenhum custo. Mais especificamente, a linguagem é baseada na ideia de fornecer ao programador:

- mapeamento direto de operações e tipos nativos da linguagem ao *hardware* e
- mecanismos de abstração de baixo custo e flexíveis para fornecer tipos definidos pelos usuários que possuem mesmas características que os tipos nativos.

A linguagem foi projetada com foco em técnicas de programação que lidam com noções fundamentais tais como memória, mutabilidade, abstração, gerenciamento de recursos, expressão de algoritmos, tratamento de erros e modularidade. Essas são as principais preocupações dos programadores de sistemas de recursos limitados e/ou de alto desempenho (STROUSTRUP, 2013).

A biblioteca deste trabalho tem como premissa ser utilizável em aplicações de sistemas de recursos limitados bem como de sistemas onde o *hardware* e detalhes de *hardware* são abstraídos. O C++ mostra-se uma escolha adequada para tanto.

Alguns recursos do C++ relacionados a POO que podem promover o uso de bons princípios de um projeto de *software* (conforme a seção 2.5) serão apresentados.

Um desses princípios (inversão da dependência) está relacionado à programação de módulos do sistema que não dependem da implementação de outros módulos, mas sim de classes abstratas que servem como interface comum para diferentes implementações. O autor em (MEYERS, 2012) denomina esse método de *programação baseada em interface*. Tal método utiliza-se do polimorfismo (Veja a seção 2.4).

O polimorfismo no C++ pode ser explorado de três maneiras distintas, que se diferenciam quanto ao momento que o polimorfismo ocorre durante a construção ou execução do programa: em tempo de execução, ligação (também denominado *linkedição*) ou compilação (MEYERS, 2012). Tempo de execução são eventos que ocorrem enquanto o programa é executado. Já o tempo de ligação se refere ao momento que o *linker* (ou *linkeditor*) - programa que une funções compiladas em arquivos separados em um programa executável - está sendo executado. Por fim, o tempo de compilação são os eventos que ocorrem enquanto um ou mais arquivos estão sendo compilados (SCHILDT, 1997).

- **polimorfismo em tempo de execução:** é o mais flexível, pois um objeto de uma classe derivada pode ser tratado como uma referência à classe base durante a execução do programa. É apropriado quando precisa-se escolher qual implementação (derivada) de uma interface (base) deve ser usada em tempo de execução. Esse método de polimorfismo é o mais custoso e acontece através do uso da palavra-chave *virtual* em pelo menos uma das funções da classe abstrata (interface).
- **polimorfismo em tempo de ligação:** útil quando a implementação a ser usada é conhecida durante a ligação do programa. Durante a compilação diferentes implementações são compiladas em seus respectivos arquivos, mas ainda não se sabe qual destes arquivos compilados será usado no programa final. Neste método, os arquivos que contêm a implementação desta interface devem incluir um arquivo cabeçalho comum que contém a classe interface cujas funções são declaradas, mas não implementadas. A palavra-chave *virtual* não é usada. A implementação dessas funções estará nos arquivos de implementação.
- **polimorfismo em tempo de compilação:** qual implementação escolher é determinável em tempo de compilação ao usar informações acessíveis nesse momento.

Esse método é baseado em uma interface implícita, isto é, que não é definida por uma classe via sintaxe e regras da linguagem. É um polimorfismo semântico. Assim, esta técnica exige que as diferentes classes que implementam a interface sigam uma convenção definida pelo programador. Um dos métodos utiliza-se do recurso *typedef* do *C++* para escolher a implementação (Veja em (MEYERS, 2012)).

Outro recurso do *C++* explorado na biblioteca deste trabalho foi o *template*. Os *templates* permitem gerar classes e funções em tempo de compilação a partir de uma classe ou função genérica que pode ser parametrizada com tipos ou valores. Tal recurso promove maior generalização e reutilização de código. Um aspecto essencial dos *templates* é que eles são interpretados em tempo de compilação, não trazendo nenhum custo em processamento para a aplicação final. (STROUSTRUP, 2013) Por outro lado, tornam o código relativamente mais complexo, pois novos elementos sintáticos são adicionados e este recurso não é comum em todas as linguagens de programação. Através de *templates* também é possível implementar o polimorfismo em tempo de compilação.

2.7.2 CMake

Os sistemas de compilação buscam facilitar o processo de compilar e empacotar *software* a partir de um código-fonte existente. A escolha de um sistema de compilação entre os muitos existentes é influenciada por alguns fatores como ambiente de desenvolvimento utilizado pelo programador, plataformas nas quais o *software* vai ser executado e as facilidades — ou redução das complexidades — obtidas.

O sistema de compilação *CMake* é um conjunto de ferramentas multiplataformas de código aberto planejadas para construir, testar e empacotar *software*. É usado para controlar o processo de compilação do *software* usando arquivos de configuração independentes da plataforma e do compilador (CEDILNIK *et al.*, 2022a).

O *CMake* é organizado como um conjunto de *targets* lógicos de alto nível. Cada *target* por padrão corresponde a uma biblioteca ou a um executável, mas também pode-se ter um *target* customizado que contém comandos. As dependências entre os *targets* são expressas no próprio sistema de compilação do *CMake*. Isso permite determinar a ordem de compilação e as regras de reconstrução dos *targets* quando ocorre mudanças no código-fonte (CEDILNIK *et al.*, 2022b).

Outra característica do *CMake* é que ele possui mecanismos que dão suporte completo à cross-compilação, isto é, quando um *software* é compilado e empacotado em um sistema (plataforma hospedeira) com o objetivo de ser executado em outro sistema diferente (plataforma alvo). Para tanto, o *CMake* exige que um arquivo que

descreva e especifique detalhes da plataforma alvo seja criado (arquivo *toolchain*) e passado como parâmetro na invocação do *CMake*. Algumas informações contidas em um arquivo *toolchain* são:

- nome da plataforma alvo;
- quais compiladores usar para as linguagens *C* e *C++*;
- caminho para uma pasta que contém as dependências do sistema da plataforma alvo necessárias para compilação e ligação de um mais *targets*. Tais dependências incluem arquivos cabeçalhos específicos da plataforma, bibliotecas estáticas, bibliotecas dinâmicas (caso a plataforma contenha um sistema operacional de alto nível), etc.;
- se deve ou não procurar pelas dependências também na plataforma hospedeira.

Pelas características descritas e pela experiência do autor, o *CMake* se mostra uma escolha adequada para gerenciar os processos de compilação e empacotamento de uma biblioteca escrita em *C++* com premissas de portabilidade e multiplataforma.

2.7.3 Doctest

A relevância de testes no contexto de *softwares* foi discutida na seção 2.6. É possível escrever testes para os componentes de um sistema apenas com a utilização de funções nativas do *C++* e com comandos de compilação. Contudo, atualmente existem dezenas de *frameworks* para *C++* (Wikipedia contributors, 2022) que automatizam e facilitam o processo de escrita de testes, execução e visualização dos resultados. Um deles é o *Doctest* (KIRILOV, 2022).

O *Doctest* é um dos mais recentes *frameworks* e se diferencia de outros com recursos similares por ser mais rápido em tempo de compilação e de execução. Ele permite que funções de testes sejam escritas nos próprios arquivos do código-fonte existente, uma característica pouco comum entre tais *frameworks*. Seu uso exige apenas a inclusão de um arquivo cabeçalho (*header*) no código-fonte onde os testes estão escritos. Outro ponto é que a ferramenta fornece uma interface de programação que contém apenas a diretiva de assertividade "*CHECK(condição)*".

O *Doctest* também tem um processo de integração com o *CMake* (subseção 2.7.2) documentado e, na experiência do autor, de fácil configuração e utilização.

Essas características o tornam uma ferramenta com uma curva de aprendizagem baixa e adequada para este projeto.

2.7.4 Biblioteca para leitura e escrita da porta serial

O protocolo da norma NBR14522 descrito na subseção 2.2.1 ocorre em cima de um protocolo de comunicação serial entre leitor e medidor. Assim, a comunicação serial torna-se uma dependência para a biblioteca deste trabalho poder funcionar. Essa comunicação serial é amplamente utilizada em outros contextos e aplicações, desde sistemas operacionais de uso geral até sistemas embarcados.

Para sistemas embarcados, é comum que os fabricantes de microcontroladores forneçam os componentes de *software* com uma interface pronta para se utilizar as portas seriais, em cima da qual a aplicação é escrita. Para sistemas operacionais de uso geral como Windows e Linux, o acesso às portas seriais depende da instalação de *drivers* genéricos ou de fabricantes, mas o componente da aplicação que acessa as portas seriais costuma ser similar entre os sistemas operacionais.

Há algumas bibliotecas em *C/C++* que implementam a leitura e escrita de porta serial e são compatíveis com os sistemas operacionais mais utilizados (*Windows*, *Linux* e *macOS*): *Boost.Asio* (KOHLHOFF, 2022), *Libserialport* (SIGROK, 2020) e *Seriallib* (LUCIDARME, 2021).

A primeira faz parte do módulo *Asio* da biblioteca *Boost*, sendo amplamente utilizada e reconhecida por sua eficiência e portabilidade. Contudo, a utilização do submódulo de porta serial do *Asio* não pode ser utilizada isoladamente do *Asio* devido às dependências existentes (KOHLHOFF, 2021). Já a biblioteca *Libserialport* é escrita em *C* e a *Seriallib* em *C++*. Apesar disso, ambas possuem uma interface de programação similar e de fácil utilização.

A preferência pela utilização da biblioteca *Seriallib* ao invés da *Libserialport* neste projeto é devido à facilidade de integração com outros projetos: a *Seriallib* necessita de apenas dois arquivos para funcionar.

3 METODOLOGIA

Este trabalho constitui-se de implementar um protocolo de comunicação em *software* para ser utilizado por outras aplicações. Para tanto, uma compreensão do protocolo, sua norma e contextos de uso, bem como de técnicas e conceitos de programação são abordados com o fim de alcançar os objetivos definidos na seção 1.4. Assim, é possível classificar parte deste trabalho como uma pesquisa exploratória, pois, conforme (GIL, 2002): "[...] estas pesquisas têm como objetivo principal o aprimoramento de ideias ou a descoberta de intuições. Seu planejamento é, portanto, bastante flexível, de modo que possibilite a consideração dos mais variados aspectos relativos ao fato estudado."

Após compreensão do protocolo, estudo de trabalhos relacionados e também baseado na experiência do autor, alguns requisitos de *software* da biblioteca foram definidos tendo em vista ampliar sua utilização para diferentes cenários de uso. Tais requisitos incluem:

- Gerenciar de forma transparente ao usuário da biblioteca a retransmissão de comandos e a solicitação de retransmissão de respostas a comandos, quando for aplicável conforme a norma;
- Suportar a comunicação com mais de um medidor;
- Ser portátil para os principais sistemas operacionais do mercado: *Windows*, *Linux* e *macOS*;
- Ser portátil para sistemas embarcados e para sistemas de recursos de *hardware* limitados.

Para a compilação e organização deste projeto, composto de uma biblioteca, uma aplicação e testes unitários, utilizou-se o sistema *CMake*. Os testes utilizaram-se do *framework Doctest* integrado ao *CMake* e o projeto foi escrito na linguagem de programação *C++*. A biblioteca e a aplicação são compatíveis com a versão 11 do *C++* e os testes unitários com a versão 14.

Primeiramente, compreendeu-se detalhadamente o protocolo da comunicação convencional leitor-medidor da norma NBR 14522 a ser implementado. Desenhou-se uma máquina de estados contendo todos os possíveis estados do leitor, suas condições e eventos. Com essa máquina de estados em mãos, iniciou-se a implementação da biblioteca, cuja principal classe implementa essa máquina de estados, que define a lógica e comportamento do leitor. Então, escreveu-se os testes unitários para validar todos os possíveis estados e comportamentos desse leitor. Durante esse

processo, iterações aconteceram envolvendo ajustes no desenho da máquina de estados, na arquitetura da classe e na sua lógica. Com isso validado, uma aplicação em linha de comando que utiliza a biblioteca foi escrita para fins de demonstração. Com essa aplicação pronta, compilou-se a aplicação para outra plataforma conectada em um medidor existente. A partir disso, executou-se a aplicação nessa plataforma para realizar a leitura de comandos desse medidor.

Todo essa implementação foi escrita em um sistema operacional *Linux* com distribuição *Debian* e está disponível publicamente no repositório (NAGEL, 2022) (*tag* v1.0.0).

4 RESULTADOS

4.1 Máquina de estados do protocolo

A implementação do protocolo da leitura convencional da norma, explicada na subseção 2.2.1, é descrita a partir de uma máquina de estados do tipo *Mealy*¹ da Figura 6.

Incluiu-se nessa máquina de estado eventos para quando limites de transmissões de sinalizadores (*NAK* e *WAIT*) e tempos máximos (*TMAXRSP*, *TSEMWAIT*, *TMAXCAR*, etc.) são atingidos. Tais valores são especificados na norma, contudo, a norma não explicita qual deve ser o comportamento do leitor nessas situações.

A máquina de estados começa quando o usuário solicita à entidade leitor que um comando seja transmitido. A sequência que ocorre para uma comunicação sem falhas é a seguinte. Após comando solicitado, o leitor vai para o estado *DESSINCRONIZADO*, este que aguarda um sinalizador *ENQ* (que indica conexão do medidor) para ir ao estado *SINCRONIZADO*. A partir deste, após receber mais um *ENQ*, o leitor então transmite o comando ao medidor e vai para o estado *COMANDO TRANSMITIDO*.

Após o leitor receber um *byte* de dados da porta serial e verificar que esse *byte* é igual ao código do comando solicitado ou ao código de uma das exceções ("informação de ocorrência no medidor" ou "informação de comando não implementado"), vai então para o estado *CÓDIGO RECEBIDO*. Quando receber os restantes dos *bytes* da resposta vinda da porta serial, o leitor calcula o CRC da resposta recebida e verifica se é igual ao CRC recebido com a resposta. Se o CRC for válido, o leitor transmite um sinalizador *ACK* ao medidor e salva a resposta recebida. Após transmitir esse sinalizador, o leitor verifica se a resposta recebida é referente a um comando simples (única resposta) ou comando composto (múltiplas respostas).

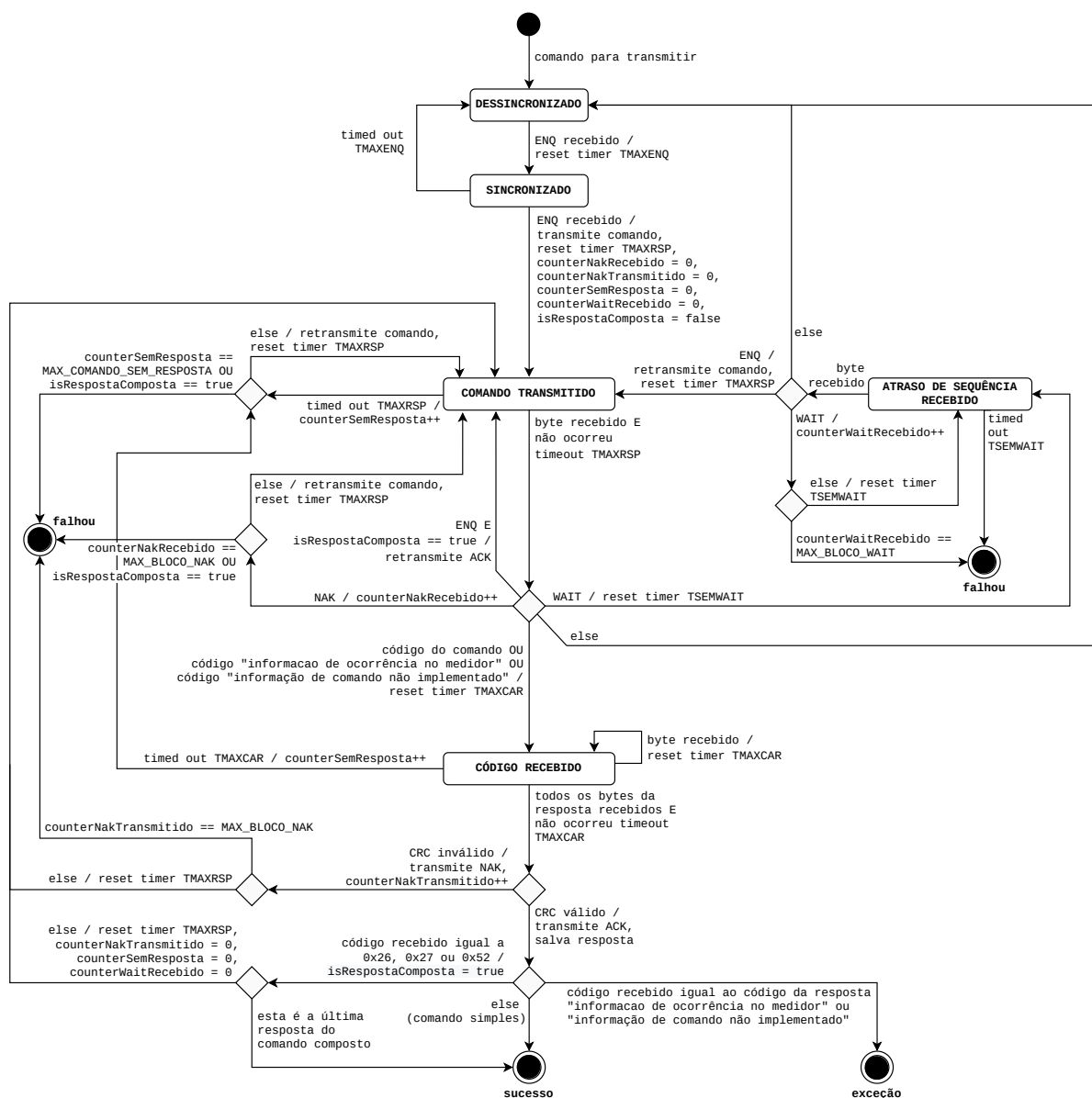
Atualmente os únicos comandos compostos contidos na norma são os que possuem os códigos 0x26, 0x27 e 0x52. Assim, se a resposta não for de um comando composto, o leitor encerra a lógica com sucesso. Caso contrário, verifica se a resposta que acabou de ser recebida é a última resposta do comando composto. As respostas aos comandos compostos da norma possuem um *nibble* que indica se a resposta é a última referente a um comando composto. Se não for, muda para o estado *COMANDO RECEBIDO*. Neste, aguarda o recebimento do primeiro *byte* da próxima resposta, igual ao código do comando transmitido inicialmente ao medidor. Ao recebê-lo, repete o fluxo descrito acima até a última resposta ser recebida.

A descrição detalhada de cada estado é feita a seguir.

- **DESSINCRONIZADO**: estado que representa uma dessincronização do leitor

¹ Uma máquina de estados do tipo *Mealy* produz saídas a partir das transições entre os estados.

Figura 6 – Máquina de estados da implementação da comunicação convencional leitor-medidor da NBR 14522



Fonte: Elaboração do autor

com o medidor. Conforme subseção 2.2.1, um medidor conectado a uma porta serial deve enviar o sinalizador *ENQ* periodicamente. Assim, o recebimento de um *ENQ* é a condição para o leitor sair deste estado. Este estado representa uma etapa posterior à conexão entre leitor e medidor, definida na norma.

- **SINCRONIZADO:** o leitor está sincronizado com o medidor e este deve enviar *ENQs* periodicamente para o leitor através da porta serial. Caso o envio de *ENQs* seja interrompido, esse interrompimento é percebido por um alarme (ou *timer*) sendo disparado após o período *TMAXENQ* (Equação 2.4) sem receber o *ENQ*. Esse alarme é zerado em todas as transições de estado para *SINCRONIZADO*. Por outro lado, um segundo recebimento de um sinalizador *ENQ* dispara a transmissão do comando para o medidor. Essa condição também zera os seguintes contadores:

- contador de *NAKs* recebidos pelo leitor (*counterNakRecebido*)
- contador de *NAKs* transmitidos pelo leitor (*counterNakTransmitido*)
- contador de falta de respostas para um comando transmitido (*counterSemResposta*)
- contador de *WAITs* recebidos pelo leitor (*counterWaitRecebido*)

E também atribui o valor padrão *falso* para a variável *isRespostaComposta*. Essa variável informa se a resposta recebida pelo leitor é uma resposta composta. A *isRespostaComposta* é somente reatribuída após o leitor receber uma resposta sem erros e verificar que se trata de uma resposta composta.

- **COMANDO TRANSMITIDO:** neste estado o leitor aguarda uma das seguintes condições:
- disparo do alarme de tempo *TMAXRSP* esgotado: caso nenhum *byte* seja recebido no tempo máximo de resposta *TMAXRSP* (Equação 2.7) esse alarme é disparado e o contador *counterSemResposta* é incrementado. Caso o contador seja menor que o número máximo sem respostas a um comando transmitido (*MAX_COMANDO_SEM_RESPOSTA*), definido na norma, o comando é retransmitido e o leitor é transicionado para este mesmo estado. Observe que o alarme *TMAXRSP* é sempre zerado ao transicionar para o estado *COMANDO TRANSMITIDO*. Por outro lado, caso o contador seja igual a *MAX_COMANDO_SEM_RESPOSTA*, o leitor termina a operação com falha. Também termina com falha caso o leitor já tenha feito a leitura de alguma resposta sem erros (variável *isRespostaComposta* igual a *verdadeiro*). Este término acontece, pois, se o leitor já recebeu pelo menos uma resposta com sucesso, significa que o medidor já recebeu o comando com

sucesso antes da primeira resposta e então não faria sentido retransmitir esse comando.

- *byte* recebido antes de *TMAXRSP*: o leitor aguarda pelo recebimento do primeiro *byte* da resposta (código do comando) referente ao comando transmitido. O leitor também deve aceitar o código de "informação de ocorrência no medidor" e "informação de comando não implementado" (exceções). Nesses dois casos o leitor vai para o estado *CÓDIGO RECEBIDO*. Outros *bytes* recebidos diferentes de um código de comando devem ser tratados conforme o seguinte. Se for um sinalizador *NAK*, o leitor incrementa o contador *counterNakRecebido*. Se o contador for menor que o número máximo de *NAKs* para um mesmo bloco (*MAX_BLOCO_NAK*) definido na norma, então o comando é retransmitido e o leitor permanece neste estado. Caso o contador seja igual a *MAX_BLOCO_NAK*, a operação é encerrada com falha. Já se o *byte* for um sinalizador *WAIT* recebido, o leitor muda para o estado *ATRASO DE SEQUÊNCIA RECEBIDO*. Se o *byte* for igual ao sinalizador *ENQ* e já tiver recebido uma resposta com sucesso do medidor, o leitor deve retransmitir o *ACK*. Essa condição é obrigatória, pois a norma afirma que, se o medidor não receber um *ACK* do leitor no tempo permitido, o medidor deve enviar um *ENQ* aguardando o recebimento do *ACK*. Isto só aconteceria caso o *ACK* transmitido pelo leitor não chegasse ao medidor por qualquer motivo. Por fim, se o *byte* recebido não for nenhuma das possibilidades anteriores, a norma define que deve haver interrupção na troca de informações e aguardar uma nova sincronização, portanto, o leitor retorna ao estado *DESSINCRONIZADO*.
- **ATRASO DE SEQUÊNCIA RECEBIDO**: inicia quando o medidor, após receber um comando, solicita um atraso de sequência (sinalizador *WAIT*). Neste estado o leitor aguarda uma das seguintes condições:
 - disparo do alarme de tempo *TSEMWAIT* esgotado: caso nenhum *byte* seja recebido no tempo máximo sem *WAIT* (tempo máximo entre dois *WAITs* subsequentes, sendo 305 segundos), então o leitor termina a operação com falha. Observe que o alarme *TSEMWAIT* é sempre zerado quando o leitor transiciona para o estado *ATRASO DE SEQUÊNCIA RECEBIDO*.
 - *byte* recebido antes de *TSEMWAIT*: em função deste *byte* o leitor pode transicionar para três estados distintos. Caso o *byte* seja o sinalizador *ENQ*, significa que o medidor deseja retornar à sequência comando-resposta, então o leitor retransmite o comando e retorna ao estado *COMANDO TRANSMITIDO*. Se o *byte* for igual ao sinalizador *WAIT*, significa que o medidor quer continuar com o atraso de sequência por mais tempo. Neste caso,

se incrementa o contador *counterWaitRecebido*. Caso o contador seja menor que o número máximo de *WAITs* recebidos para um mesmo comando (*MAX_BLOCO_WAIT*), o leitor retorna para este mesmo estado. Caso contrário, o leitor termina a operação com falha. Por fim, se o *byte* recebido não for um *ENQ* nem um *WAIT*, a norma afirma que deve acontecer a interrupção da troca de informações até uma nova sincronização. Portanto, o leitor muda para o estado *DESSINCRONIZADO*.

- **CÓDIGO RECEBIDO:** o leitor muda para este estado quando recebe o código de uma resposta (primeiro *byte* da resposta) referente a um comando transmitido ou, excepcionalmente, recebe o código "informação de ocorrência no medidor" ou "informação de comando não implementado" após qualquer comando transmitido. Neste estado o leitor aguarda pelas condições:
 - disparo do alarme de tempo *TMAXCAR* esgotado: caso nenhum *byte* seja recebido no tempo máximo entre dois *bytes* (*TMAXCAR*), definido na norma pela Equação 2.6, então incrementa-se o contador *counterSemResposta*. Caso o contador seja menor que *MAX_COMANDO_SEM_RESPOSTA*, o comando é retransmitido e o leitor é transicionado para o estado *COMANDO TRANSMITIDO*. Por outro lado, caso o contador seja igual a *MAX_COMANDO_SEM_RESPOSTA*, o leitor termina a operação com falha. Observe que o alarme *TMAXCAR* é sempre zerado ao transicionar para o estado *CÓDIGO RECEBIDO*.
 - todos os *bytes* da resposta recebidos: no estado *CÓDIGO RECEBIDO*, toda vez que um *byte* é recebido, o alarme *TMAXCAR* é zerado. Quando o tamanho total de uma resposta é recebido (258 *bytes*, sendo 1 *byte* de código, 255 de *payload* e 2 de CRC), o CRC sobre o código e *payload* é calculado pelo leitor. Caso o CRC calculado pelo leitor difira do CRC contido na resposta, o leitor transmite o sinalizador *NAK* ao medidor e incrementa o contador *counterNakTransmitido*. Caso este contador seja menor que *MAX_BLOCO_NAK*, o leitor retorna ao estado *COMANDO TRANSMITIDO* para aguardar novamente a resposta vinda do medidor. Caso o contador seja igual a *MAX_BLOCO_NAK*, o leitor encerra o processo com falha. Também encerra com falha caso o leitor já tenha recebido uma resposta (*isResposta-Composta* como verdadeiro), pois não é esperado que o medidor transmita *NAKs* após já ter recebido um comando íntegro. Por outro lado, se o CRC calculado é igual ao CRC contido na resposta, então a resposta é considerada íntegra e o leitor transmite um *ACK* ao medidor e salva a resposta recebida. Em seguida, o leitor verifica se a resposta recebida refere-se a uma resposta simples ou composta. Se for uma simples, isto é, o código

da resposta não é igual aos códigos 0x26, 0x27 ou 0x52, então a operação termina com sucesso ou com exceção. Caso seja uma resposta composta, a variável *isRespostaComposta* é atribuída como verdadeira e o leitor verifica se esta resposta é a última. Se sim, termina a operação com sucesso, pois todas as respostas foram recebidas com sucesso. Caso não seja última, o leitor deve retornar ao estado *COMANDO TRANSMITIDO* para receber a próxima resposta. Ao fazê-lo, zera os contadores *counterNakTransmitido*, *counterSemResposta* e *counterWaitRecebido*. Esses contadores são zerados porque são incrementados para cada resposta individual a ser recebida, mesmo que seja uma resposta composta, isto é, *n* respostas a um único comando. Observe que o contador *counterNakRecebido* não é zerado, pois um *NAK* é somente recebido pelo leitor quando o medidor não recebe um comando íntegro. Contudo, nesta etapa o comando já foi recebido integralmente pelo medidor, caso contrário o leitor não teria recebido a resposta.

4.2 Classes

A máquina de estados da Figura 6, descrita na seção 4.1, é implementada em *C++* em uma classe *LeitorFSM*. O início da execução da máquina de estados, representada na Figura 6 como a condição *comando para transmitir*, representa a invocação do método *setComando* desta classe. Essa classe é ilustrada na Figura 7 e será explicada ao longo desta seção.

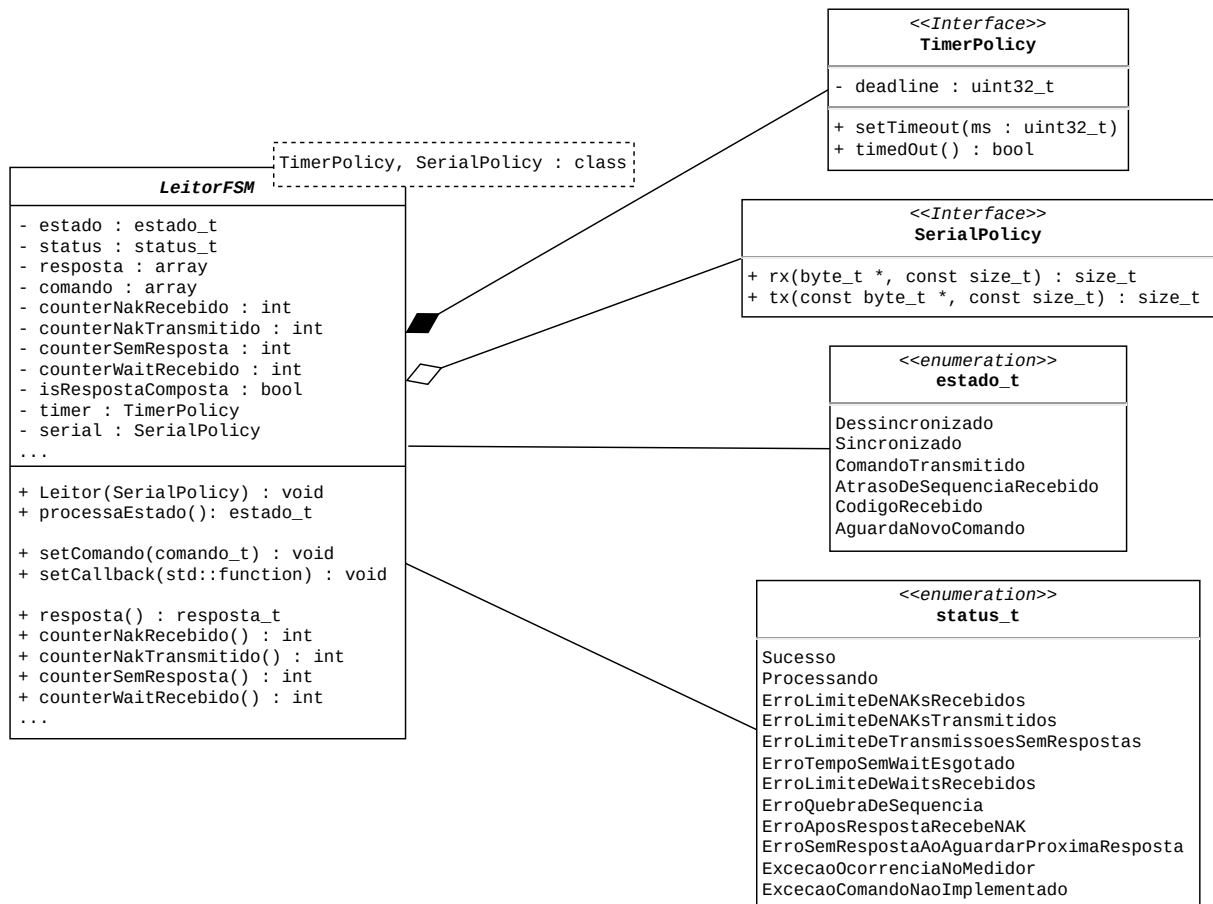
Durante a modelagem das classes, sua modularização baseou-se em separar:

1. Regras, ações e condições invariáveis que pertence à implementação do protocolo NBR 14522, assim reutilizáveis em qualquer cenário que a norma esteja inserida;
2. Elementos dos quais o protocolo depende para ser executado, mas que variam conforme o ambiente ou a plataforma onde a implementação do protocolo é executada.

Nesse sentido, concentrou-se a implementação da máquina de estados (Figura 6) na classe *LeitorFSM*. Suas dependências, que variam conforme a plataforma ou ambiente onde o *software* será executado, foram transformadas em interfaces que a classe *LeitorFSM* utiliza. As duas dependências da implementação do protocolo são:

1. transmissão de dados via porta serial (classe *SerialPolicy*);
2. temporização e alarmes (classe *TimerPolicy*).

Figura 7 – Diagrama de classes da implementação do leitor na linguagem C++



Fonte: Elaboração do autor

A definição da interface dessas dependências deve ser abstrata o suficiente para ser independente da plataforma e do ambiente de execução e também para esconder os detalhes de sua implementação.

Para a interface de transmissão de dados (*SerialPolicy*), a classe *LeitorFSM* a utiliza a partir de dois métodos: um para transmissão (do leitor para o medidor) e outro para recepção de dados (do medidor para o leitor). Convencionou-se que a abertura e configuração da porta serial não deve ser responsabilidade da entidade leitor, e sim da aplicação que utiliza o leitor. Por isso tais comportamentos não estão incluídos na interface. Essa decisão fundamenta-se na possibilidade de o leitor poder utilizar outros meios de comunicação que não uma porta serial, e assim a inicialização e configuração deste meio podem variar para cada implementação e possuir diferentes parâmetros.

Para a interface de temporização e alarmes (*TimerPolicy*), o leitor necessita apenas do seguinte recurso: inicializar uma contagem de tempo relativa e verificar, posteriormente, se essa contagem ultrapassou determinado valor em unidade de tempo. Esta unidade é definida pela interface em milissegundos, pois é granular o suficiente para todas as verificações do protocolo. Assim, para esta interface apenas dois métodos

são necessários: uma que inicializa a contagem e configura a quantidade de tempo em milissegundos, e outro método que checa se a quantidade de tempo foi excedida.

Conforme descrito na subseção 2.7.1, a linguagem *C++* possui diferentes abordagens para trabalhar com abstrações, interfaces e polimorfismo. Na implementação desta biblioteca optou-se pelo polimorfismo em tempo de compilação através de *templates*. Essa abordagem possui algumas propriedades conforme discutidas na subseção 2.7.1. A seguir estão listados os fundamentos que direcionaram esta escolha, tendo em vista o contexto e requisitos da biblioteca de *software* deste trabalho:

1. **flexibilidade**: qualquer classe pode ser compatível com uma ou mais interfaces, desde que implemente os métodos definidos pela(s) interface(s). É uma flexibilidade similar ao polimorfismo em tempo de execução, porém sem obrigação sintática.
2. **especialização**: com o uso de *templates*, a escrita de uma classe genérica (parametrizável via *templates*) pode gerar n classes concretas e distintas, em tempo de compilação. Isso possibilita, por exemplo, parametrizar a classe genérica *LeitorFSM* com diferentes implementações da porta serial para se comunicar com medidores que utilizam meios de comunicação diferentes, gerando uma classe concreta para cada meio de comunicação. Essa especialização também é uma característica similar ao polimorfismo em tempo de execução.
3. **otimização**: permite que o compilador faça otimizações com a invocação de funções da interface implementada. Essa abordagem permite escrever o corpo das funções (implementação) no próprio arquivo cabeçalho na declaração da classe (recurso denominado *inline*), passível de otimização pelo compilador. O polimorfismo em tempo de execução também permite esse recurso, mas tem outros custos de processamento na invocação das funções. Já o polimorfismo em tempo de ligação não permite o *inline* pois o corpo das funções precisa ser escrito em um arquivo separado do arquivo cabeçalho que define a classe.

A flexibilidade e a especialização descritas acima também facilitam a escrita de testes que objetivam validar a lógica das classes, pois é trivial parametrizar um objeto simulado que imita o comportamento de uma dependência.

Os testes unitários, ou por classe, são facilitados ou mesmo tornam-se possíveis quando as classes são desenhadas de maneira a possibilitar a verificação da sua lógica interna. A classe *LeitorFSM* foi projetada visando possibilitar a verificação do seu comportamento, isto é, a implementação do protocolo do ponto de vista do leitor.

Para isso, utilizou-se o método *processaEstado* que, quando invocado, executa a máquina de estados da Figura 6 e retorna o estado atual do protocolo. Em

uma aplicação final, esse método deve estar contido em um laço de repetição que o invoca com frequência suficiente para verificar as condições da máquina de estados: recebimento de dados do medidor, limite de tempos ultrapassados, etc. Essa é uma abordagem conhecida como *polling*, em que o processador verifica o estado de determinada condição de maneira síncrona e periódica.

A frequência mínima necessária desse *polling* pode ser definida considerando o menor tempo especificado pela norma, sendo este a Equação 2.1. Assim, sugere-se o período máximo de $T_{car}/10$ para o *polling*.

A decisão de tornar o processo de *polling* da classe *LeitorFSM* responsabilidade de seu usuário por meio do método *processaEstado* baseou-se no princípio da responsabilidade única (Veja seção 2.5), tornando a classe *LeitorFSM* mais flexível e testável. Complementarmente, expor alguns estados dessa classe, como os contadores da máquina de estado, a tornam mais testável ainda. Por outro lado, tais métodos expostos comprometem a abstração da classe ao seu usuário. Este precisa saber de informações extras relacionadas à implementação para usar a classe de maneira efetiva.

Assim, a classe *LeitorFSM* pode ser utilizada por outras classes cujos métodos serão expostos ao usuário final. Isso permite que a biblioteca possa ter diferentes interfaces de aplicação aos usuários finais, podendo atender necessidades e contextos específicos, a partir de uma única classe de mais baixo nível, *LeitorFSM*, que implementa o protocolo.

Um exemplo de interface de aplicação exposta ao usuário final da biblioteca está contida na aplicação da seção 4.5. Trata-se da classe *Leitor* (Código A.2) que expõe somente o método *leitura* à aplicação final. O parâmetro *callback* informa uma função que será invocada pelo *Leitor* quando uma resposta tiver sido recebida. No caso de uma resposta composta com n respostas, essa função será invocada n vezes, cada uma com a respectiva resposta passada como parâmetro. Ou seja, a aplicação deve lidar com o que será feito com cada resposta individual.

Como ilustração, outra possibilidade seria uma classe com um método similar, porém, ao invés de utilizar-se de um *callback*, retornar um vetor contendo todas as respostas, caso tenham sido recebidas com sucesso pelo leitor.

Por fim, a classe *LeitorFSM* utiliza-se dos tipos *estado_t* e *status_t*. O primeiro representa os estados da Figura 6 e do estado *AguardaNovoComando*, este que representa o momento que a máquina de estados ainda não foi inicializada (sem comando para transmitir). A máquina de estados retorna ao *AguardaNovoComando* após sua conclusão, seja com sucesso, falha ou exceção. Já o segundo tipo representa a situação em que a máquina de estados se encontra em relação a uma leitura solicitada. Todos os erros e exceções possíveis da máquina de estado estão contidos no *status_t*,

podendo ser recuperado pelo usuário da classe *LeitorFSM*. A situação *Processando* representa uma solicitação de leitura que ainda não foi concluída ou iniciada.

4.3 Organização do repositório e *CMake*

O repositório da biblioteca deste trabalho encontra-se em (NAGEL, 2022) (*tag* v1.0.0) e utiliza o *CMake* versão 3.16 (Veja a subseção 2.7.2 para mais informações sobre o *CMake*) como ferramenta para compilação e empacotamento de *software* e execução de testes unitários. Com essa ferramenta, organizou-se a implementação deste trabalho em uma biblioteca e dois executáveis, denominados *targets* na nomenclatura do *CMake*:

1. biblioteca que contém a implementação do protocolo da leitura convencional (Veja subseção 2.2.1) descrito pela máquina de estados da seção 4.1.
2. executável que testa essa biblioteca. A biblioteca é uma dependência deste *target*. Os testes escritos para este *target* são descritos na seção 4.4.
3. executável que representa uma aplicação que utiliza a biblioteca acima. Assim, a biblioteca também é uma dependência deste *target*. Essa aplicação é descrita na seção 4.5.

A biblioteca não pode ser compilada sozinha, pois possui dependências (temporizador e comunicação serial) em tempo de compilação que precisam ser definidas por um *software* que a utiliza. Caso a biblioteca seja usada fora do sistema de compilação *CMake* deste repositório, por exemplo, para integrar em outro código-fonte já existente, é possível fazê-lo ao incluir os arquivos cabeçalhos da biblioteca nesse outro código-fonte.

Já o *target* executável de testes deve ser compilado com a biblioteca e desacoplado de outros executáveis. Este *target* também pode ser executado na própria plataforma de desenvolvimento onde foi compilado (normalmente um sistema operacional de propósito geral como *Windows*, *Linux* ou *macOS*).

O *target* executável da aplicação deve ser compilado com o *target* da biblioteca, mas é independente de outros *targets*. Cada executável também pode ser compilado para outra plataforma onde será executado. Neste trabalho compilou-se este executável em uma plataforma *Linux*, arquitetura *x86*, para ser executado em outra plataforma *Linux* com arquitetura *Arm*, distribuição *Raspbian* (Raspberry Pi Foundation, 2012). Essa compilação foi compreendida e executada com sucesso pelo *CMake*. Para tanto, foi necessário parametrizar um arquivo *toolchain* (explicado na subseção 2.7.2) na invocação do programa *CMake* e a plataforma hospedeira ter acesso aos arquivos

de *toolchain*, cabeçalhos e bibliotecas da plataforma alvo, obtidos em (PROFANTER; CONTRIBUTORS, 2022).

Novas aplicações (*targets*) podem ser estendidos ao repositório de maneira desacoplada das aplicações nele existentes.

4.4 Testes unitários

Os testes unitários da biblioteca (Código A.6) foram escritos utilizando o *framework Doctest*. Em relação à classe *LeitorFSM*, que contém a implementação da máquina de estados da Figura 6 do protocolo, os itens listados a seguir foram testados de forma individual, isto é, cada item foi testado a partir de um novo estado de execução, o que significa que o estado remanescente de um teste executado não influencia nos testes seguintes.

- No estado *SINCRONIZADO*, alarme *TMAXENQ* é excedido;
- No estado *COMANDO TRANSMITIDO*, alarme *TMAXRSP* excedido várias vezes até *counterSemResposta* ser igual a *MAX_COMANDO_SEM_RESPOSTA* e o processo falhar;
- No estado *COMANDO TRANSMITIDO*, o leitor recebe consecutivamente *NAKs* do medidor respectivos aos comandos (re)transmitidos, até *counterNakRecebido* ser igual a *MAX_BLOCO_NAK* e o processo falhar;
- No estado *CÓDIGO RECEBIDO*, o alarme *TMAXCAR* é excedido várias vezes até o *counterSemResposta* ser igual a *MAX_COMANDO_SEM_RESPOSTA* e o processo falhar;
- No estado *CÓDIGO RECEBIDO*, recebe-se todos os *bytes* de uma resposta com CRC inválido e então transmite-se o *NAK*. Isso repete-se até o *counterSemResposta* ser igual a *MAX_COMANDO_SEM_RESPOSTA* e o processo falhar;
- No estado *ATRASO DE SEQUÊNCIA RECEBIDO*, aguarda-se o alarme *TSEMWAIT* ser excedido e o processo falhar;
- No mesmo estado, retorna-se ao estado *COMANDO TRANSMITIDO* após receber um *ENQ*;
- No mesmo estado, recebe-se *WAIT* consecutivos até *counterWaitRecebido* ser igual a *MAX_BLOCO_WAIT*;
- Execução de uma sequência comando-resposta simples com sucesso;

- Execução de uma sequência comando-resposta composta com sucesso;
- Execução de uma sequência comando-resposta, mas com o recebimento da resposta de exceção "informação de ocorrência no medidor";
- Execução de uma sequência comando-resposta, mas com o recebimento da resposta de exceção "informação de comando não implementado".

Devido à flexibilidade da classe quanto as suas dependências, nos testes simulou-se uma porta serial que se comunica com um medidor fictício. Isso possibilitou fazer a verificação de todos dados transmitidos pelo leitor e inserir dados supostamente vindos do medidor. Assim, sem ter um medidor real, foi possível validar a lógica da máquina de estados da Figura 6.

Quanto à dependência de temporização da classe *LeitorFSM*, para a execução dos testes listados acima, com exceção dos testes do estado *ATRASSO DE SEQUÊNCIA*, utilizou-se a implementação de um temporizador baseado na biblioteca *std::chrono*, que faz parte da biblioteca padrão do C++. Isso torna tal implementação compatível com a maioria dos sistemas operacionais de propósito geral, nos quais testes unitários são geralmente executados.

Somente para os testes que envolveram o estado *ATRASSO DE SEQUÊNCIA* utilizou-se outro temporizador adaptado, cuja única diferença em relação ao anterior é que ele, quando configurado para disparar o alarme de tempo igual a *TSEMWAIT*, reduz esse tempo para apenas 1 segundo, ao invés dos 305 segundos.

4.5 Exemplo de aplicação para leitura de um medidor real

Criou-se um programa com interface para linha de comando para validar a interface de aplicação da biblioteca e também validar a implementação do protocolo comunicando-se com um medidor real.

O programa possui dois parâmetros obrigatórios:

1. **nome da porta serial do medidor:** para sistemas baseados em *unix* como *Linux* esse parâmetro é um arquivo localizado na pasta */dev/* e.g. */dev/ttyUSB0*, */dev/ttyUSB1*, etc. Para *Windows*, será *COM1*, *COM2*, etc.
2. **comando a ser transmitido ao medidor:** comando da comunicação convencional leitor-medidor conforme subseção 2.2.1. O programa interpreta este parâmetro na codificação BCD. Caso o comando não seja passado por parâmetro contendo todos os 64 *bytes* referentes ao código e *payload*, o programa preenche o restante com zeros. Os dois últimos *bytes* de CRC são sempre adicionados ou sobrescritos pela implementação desta biblioteca.

Tabela 3 – Exemplos de respostas enviadas ao medidor ao invocar a aplicação por linha de comando

Código do comando	Invocação do programa via linha de comando	Comando enviado ao medidor (codificado em BCD)
0x14	<i>leitor-cli /dev/ttyUSB0 14</i> <i>leitor-cli /dev/ttyUSB0 14112233</i>	14...050F 14112233...0E38
0x20	<i>leitor-cli /dev/ttyUSB0 20445566</i> <i>leitor-cli /dev/ttyUSB0 204455660101</i>	20445566...F77E 204455660101...A643
0x25	<i>leitor-cli /dev/ttyUSB0 25</i> <i>leitor-cli /dev/ttyUSB0 2577889901</i>	25...C8EB 2577889901...DA34

A Tabela 3 contém exemplos de invocação desta aplicação por linha de comando. As reticências representam uma sequência de zeros.

A dependência de temporização para este programa foi a mesma usada para os testes unitários da seção 4.4 (com exceção dos testes do estado *ATRASSO DE SEQUÊNCIA*) baseada na biblioteca *std::chrono*. Quanto à dependência da porta serial, escreveu-se uma implementação que utiliza a biblioteca *Serialib*² (LUCIDARME, 2021). A *std::chrono* e a *Serialib* são compatíveis os principais sistemas operacionais de propósito geral tais como *Windows*, *Linux* e *macOS*, tornando este programa igualmente compatível.

4.5.0.1 Cenário de execução

O sistema operacional utilizado para execução do programa foi *Raspbian GNU/Linux 10 (buster)* específico para a plataforma *Raspberry Pi 3 Model B* (Raspberry Pi Foundation, 2012).

Já o medidor eletrônico de energia utilizado trata-se do modelo *E750* da empresa *Landis+Gyr* (Landis+Gyr, 2022) da Figura 8. Esse modelo oferece, opcionalmente, uma das seguintes interfaces de comunicação de dados com um leitor: RS232, RS485 ou RS232 com acoplamento óptico. O medidor utilizado neste trabalho possui a interface RS232 com acoplamento óptico. Esta interface do medidor conecta-se em um cabo que possui um conector de acoplamento óptico compatível. Esse cabo, na sua outra extremidade, possui um conector do tipo *DB9* fêmea. Para conectar-se ao *Raspberry*, então, outro cabo é utilizado com conector *DB9* macho. No outro lado deste cabo há um conector do tipo USB-A macho, que se conecta na porta USB-A fêmea do dispositivo. Do ponto de vista do dispositivo, a conexão é vista como uma porta serial */dev/ttyUSB0*.

Os equipamentos tais como medidor, cabos, dispositivo e toda infraestrutura

² O código-fonte da biblioteca *Serialib* foi incluído no repositório deste trabalho, respeitando sua licença.

Figura 8 – Medidor eletrônico modelo E750 da empresa Landys+Gyr

Fonte: (Landys+Gyr, 2022)

utilizados neste cenário pertencem à empresa *Telespazio Brasil S/A*, localizada no Rio de Janeiro. O acesso remoto ao dispositivo para execução do programa ocorreu por meio do protocolo *Secure Shell* (SSH). Os equipamentos estão contidos em um ambiente controlado para testes em que se tem apenas tensão monofásica conectada ao medidor e sem cargas consumindo energia.

4.5.0.2 Resultados da execução da aplicação

Executou-se a leitura dos comandos da Tabela 1 utilizando o cenário e o programa descritos nesta seção. Todos os comandos enviados pelo leitor foram respondidos pelo medidor conforme o protocolo e de maneira íntegra. Também, solicitou-se ao medidor um comando de código inexistente (código 0x01), e o medidor respondeu com a resposta "informação de comando não implementado" conforme esperado.

A Tabela 4 mostra alguns campos contidos na resposta enviada pelo medidor ao comando "leitura de grandezas instantâneas". Observe que os valores mostram haver tensão apenas na fase A e os valores de potência estão zerados. Isso corresponde com o cenário descrito na subseção 4.5.0.1, em que se tem apenas uma fase conectada ao medidor e nenhuma carga consumindo energia.

Os três comandos que possuem respostas compostas contidas na norma (códigos 0x26, 0x27 e 0x52) foram lidos do medidor e estão ilustrados na Tabela 5. Tais respostas contêm os contadores salvos na memória do medidor temporariamente. Os contadores são proporcionais a energia consumida dentro de um período que é,

Tabela 4 – Alguns valores da resposta do medidor ao comando "leitura das grandezas instantâneas"

Campo	Valor	Unidade
Tensão de fase A	126.350	<i>volt</i>
Tensão de fase B	0.000	
Tensão de fase C	0.000	
Tensão de linha AB	124.553	
Tensão de linha BC	0.000	
Tensão de linha CA	124.567	<i>ampere</i>
Corrente de fase A	0.000	
Corrente de fase B	0.000	
Corrente de fase C	0.000	
Corrente de neutro	0.001	
Potência ativa fase A	0.000	<i>watt</i>
Potência ativa fase B	0.000	
Potência ativa fase C	0.000	
Potência ativa trifásica	0.000	
Potência reativa fase A	0.000	
Potência reativa fase B	0.000	
Potência reativa fase C	0.000	
Potência reativa trifásica	0.000	

normalmente, de 5, 10 ou 15 minutos (configurável). Dentro de cada bloco de resposta de uma resposta composta comporta-se 56 contadores.

O comando 0x26 obteve somente uma resposta, pois se trata do período desde a última reposição de demanda, executada minutos antes de solicitar-se ao medidor o comando "leitura de parâmetros com reposição de demanda" (código 0x20). Já o comando 0x27 obteve 21 respostas, pois o período solicitado refere-se ao anterior à última reposição de demanda. Por fim, o comando 0x52 obteve 235 respostas, pois se trata de todos os valores contidos na memória de massa do medidor.

Por motivos de legibilidade, removeu-se da Tabela 5 as respostas intermediárias aos comandos 0x27 e 0x52, mostrando-se apenas as primeiras e as últimas respostas. Observe que a última resposta sempre contém o 11º dígito (em BCD) igual a "1" (o que equivale ao segundo *nibble* do 6º *byte* igual a "0001"). Esse valor indica ser a última resposta ao comando. Note também que a sequência "33363638" está sempre presente nas respostas após o código do comando, pois se trata do identificador de 4 *bytes* do medidor, e deve ser igual em todas as respostas originadas de um mesmo medidor.

Os três comandos compostos lidos do medidor têm todos os contadores com valor zero, pois tais contadores se referem a energia consumida temporalmente, e como explicado em subseção 4.5.0.1, o medidor estava sem cargas. Os valores desses contadores foram omitidos: na Tabela 5, as reticências representam os contadores

Tabela 5 – Respostas compostas do medidor

Comando de leitura dos contadores da memória de massa	Resposta composta ao comando (codificada em BCD)
0x26 - desde a última reposição de demanda	26333636381001...5257
0x27 - anteriores a última reposição de demanda	27333636380001...3804 27333636380002...C34B 27333636380003...6BB1 ⋮ 27333636380019...E6FB 27333636380020...BBEB 27333636381021...B8BE
0x52 - toda memória de massa	52333636380001...E22F 52333636380002...1960 52333636380003...B19A ⋮ 52333636380233...FB18 52333636380234...A57D 52333636381235...A628

zerados. Cada reticências equivale a 249 *bytes* igual a zero.

5 ANÁLISE E DISCUSSÃO

Após uma contextualização da norma NBR 14522 e do estudo de técnicas, ferramentas e práticas de engenharia de *software* para construção de projetos modulares e reutilizáveis, implementou-se um conjunto de classes na linguagem *C++* (biblioteca) para serem usadas por outras aplicações que desejam se comunicar com medidores eletrônicos de energia a partir da comunicação convencional presente na norma. Para demonstração da utilização dessas classes, implementou-se também uma aplicação em linha de comando que foi utilizada para a leitura de comandos de um medidor existente.

Em relação à arquitetura das classes desta biblioteca, uma ferramenta explorada que permite alcançar maior abstração e reutilização de código é o polimorfismo. Na linguagem *C++*, esse recurso pode ser implementado por diferentes meios. Aqui, optou-se pelo uso de *templates* para implementá-lo. *Templates* permitem que as implementações de certos comportamentos de uma classe possam ser escolhidas por parâmetro em tempo de compilação. Essa abordagem proporciona flexibilidade em tempo de compilação e eficiência comparado às outras técnicas de polimorfismo. Por outro lado, torna a sintaxe do *C++* relativamente mais complexa para o programador, dificulta a implementação de uma classe parametrizada por *templates* fora de um arquivo cabeçalho e fundamenta-se em interfaces semânticas e não sintáticas.

As duas dependências da classe que implementa o leitor da comunicação convencional são: a (1) temporização e a (2) comunicação serial com o medidor. Essas duas dependências são completamente definíveis em tempo de compilação pelo programador. Assim, a utilização de *templates* é viável e justificada.

Uma maneira qualitativa e indireta de mensurar a modularidade e abstração das classes deste trabalho foi por meio da escrita dos testes unitários. Tais testes precisam validar a lógica de uma classe em um ambiente controlado e replicável, de forma automatizada. Os testes unitários deste trabalho validaram o comportamento esperado do leitor conforme a comunicação convencional contida na norma. Para tanto, variações da sequência comando-resposta entre leitor e medidor foram programadas, incluindo situações de leitura com sucesso, com falhas e com exceções, abrangendo todos os estados possíveis do leitor.

Isso foi possível porque se simulou os comportamentos das duas dependências sem que a lógica da classe que implementa o leitor fosse alterada. Ainda, como a escrita desses testes conseguiu testar todo o comportamento da classe sem a necessidade de alterações em sua arquitetura, então tem-se um indicativo qualitativo que sua arquitetura possui flexibilidade e um alto grau de abstração.

O projeto exige a utilização de pelo menos a versão 14 do *C++* para compilar

os testes da biblioteca. Contudo, a implementação da biblioteca não utiliza recursos que exijam uma versão maior que a 11. Portanto, é possível compilar aplicações que utilizam a biblioteca com a versão 11, desde que a aplicação seja compatível.

Finalizada a escrita dos testes, o próximo passo foi o desenvolvimento de uma pequena aplicação para demonstrar e validar o uso da biblioteca. A aplicação foi projetada para ser executada em sistemas operacionais de propósito geral para leitura de comandos passados por parâmetro via linha de comando. Para demonstração, compilou-se essa aplicação para um sistema *Linux* baseado na plataforma *Raspberry*. A escolha da plataforma deu-se em função dos equipamentos que estavam disponíveis em um cenário montado com um medidor existente, fornecido pela empresa *Telespazio S/A* e acessado remotamente via *SSH* pelo autor.

A partir desse cenário e com o binário da aplicação copiado para a plataforma, executou-se a leitura de diferentes comandos no medidor modelo *E750*. Todas as leituras foram realizadas com sucesso e conforme a norma, assim validando o protocolo implementado com um medidor existente.

Essa mesma aplicação poderia ser compilada, a princípio sem alterações no código da aplicação, para outro sistema operacional de propósito geral como *Windows* e *macOS*, e então testada, desde que houvesse um medidor conectado a tais plataformas. Por limitações do cenário existente, não foi possível testar nessas outras plataformas.

A aplicação escrita para essa demonstração não deve funcionar para sistemas embarcados, pois as dependências de temporização e comunicação serial não são compatíveis a tais sistemas. Também, é comum tais sistemas não possuírem uma interface por linha de comando para execução de programas de forma interativa. Apesar disso, aplicações de sistemas embarcados conseguem utilizar a biblioteca deste trabalho sem a necessidade de modificar seu código-fonte, desde que as duas dependências sejam implementadas pela aplicação.

Essa extensibilidade da biblioteca é possível porque o código da classe que implementa o protocolo não contém funções e/ou recursos do *C++* específicos de alguma plataforma ou fabricante. Os recursos variáveis conforme a plataforma e/ou ambiente de execução são transformados em dependências que devem ser implementadas pela aplicação que utiliza a biblioteca. Também, esta implementação não utiliza alocação dinâmica de memória e mantém o uso de memória *Random Access Memory* (RAM) reduzido ao salvar somente a última resposta recebida do medidor, cabendo a aplicação gerenciar o conjunto de respostas recebidas.

Por fim, é possível afirmar que os objetivos deste trabalho foram atingidos. A implementação do protocolo do ponto de vista do leitor foi executada considerando todos os sinalizadores, exceções e fluxos de comando-resposta contidos na comunicação convencional da norma NBR 14522. O leitor aqui implementado não exige que o

usuário da biblioteca lide com casos de retransmissões de comandos e/ou respostas. Ao usuário cabe ser informado se a operação de leitura foi executada e qual é a respectiva resposta, e se ocorreu algum erro, qual a natureza deste erro. A biblioteca suporta a leitura com n medidores a partir de uma única aplicação mesmo que os medidores utilizem diferentes interfaces de comunicação.

A biblioteca apresentou um alto grau de portabilidade e extensibilidade durante a escrita dos testes unitários e da aplicação, contudo, mais aplicações, especialmente no contexto de sistemas embarcados, devem ser escritas e testadas com a biblioteca para validar outros cenários e plataformas. Complementarmente, mais medidores existentes de outros fabricantes devem ser testados com esta biblioteca para consolidá-la.

REFERÊNCIAS

- AFONSO, P. *Protótipo de medidor inteligente para consumidores grupo B, com base no cenário brasileiro e normas vigentes*. 2017. Disponível em: <https://portal.ifba.edu.br/paulo-afonso/anexos/anexos-cursos/graduacao/Engenharia_Eletrica/TCC-EE/2017/tcc-jose-elias-carvalho-marcelino-2017.pdf>. Acesso em: 15 mai. 2022.
- AMARAL, A. R. do; SANTOS, M. F. C.; MENDONÇA, D. R. Aplicativo para leitura de medidor de energia elétrica digital. In: *OpenCon*. São Paulo: [s.n.], 2016.
- ANEEL. *Resolução Normativa ANEEL Nº 414*. 2010.
- ANJOS, R. S. dos. *SGMlee - Software de gestão da medição inteligente de energia elétrica*. Dissertação (Mestrado) — Centro de Tecnologia, Universidade Federal do Ceará, Fortaleza, 2013.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 14519*: Medidores eletrônicos de energia elétrica (estáticos) - especificação. Rio de Janeiro, 2000. 23 p.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 14522*: Intercâmbio de informações para sistemas de medição de energia elétrica. Rio de Janeiro, 2008. 135 p.
- BRAGA, S. A. *et al.* Integração de medidores de energia elétrica residenciais a uma estrutura de *Smart Grid* usando redes de sensores sem fio. *XXXIV Simpósio Brasileiro de Telecomunicações*, 2016.
- CEDILNIK, A. *et al.* *CMake*. 2022. Disponível em: <<https://cmake.org/>>. Acesso em: 5 mai. 2022.
- CEDILNIK, A. *et al.* *CMake build system*. 2022. Disponível em: <<https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html>>. Acesso em: 5 mai. 2022.
- Eletra Energy Solutions. *CRONOS 7023*. 2010. Disponível em: <<http://www.eletraenergy.com/images/catalogos/Catlogo-CRONOS-7023.pdf>>. Acesso em: 1 mai. 2022.
- GIL, A. C. *Como Elaborar Projetos de Pesquisa*. 4ª. ed. [S.l.: s.n.], 2002.
- KIRILOV, V. *Doctest C++ testing framework*. 2022. Disponível em: <<https://github.com/doctest/doctest>>. Acesso em: 15 mai. 2022.
- KOHLHOFF, C. M. *Using, Building, and Configuring Boost.Asio*. 2021. Disponível em: <https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio/using.html>. Acesso em: 15 mai. 2022.
- KOHLHOFF, C. M. *Serial Ports*. 2022. Disponível em: <https://www.boost.org/doc/libs/1_79_0/doc/html/boost_asio/overview/serial_ports.html>. Acesso em: 15 mai. 2022.
- Landis+Gyr. *Medidor E750*. 2022. Disponível em: <<https://www.landisgyr.com.br/product/e750/>>. Acesso em: 5 jul. 2022.

LOUSADA, E. E. O. *Ciclo de trabalho dinâmico em uma rede para medição de energia elétrica*. Dissertação (Mestrado) — Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, 2009.

LOUSADA, E. E. O.; MINI, R. A. F. Proposta de solução para aplicação de internet das coisas nas redes elétricas inteligentes brasileiras considerando as limitações das normas vigentes. *XXXVI Congresso da Sociedade Brasileira de Computação*, p. 1715–1726, 2016.

LUCIDARME, P. *C++ cross-platform RS232 serial communication library*. 2021. Disponível em: <<https://lucidar.me/en/serialib/cross-plateform-rs232-serial-library/>>. Acesso em: 15 mai. 2022.

MARTIN, R. C. *Agile Software Development, Principles, Patterns and Practices*. [S.l.: s.n.], 2002.

MEYERS, S. *Effective C++ in an Embedded Environment*. 2012. Presentation Materials.

MOECKE, M. *BCD - Binary-coded decimal*. 2020. Disponível em: <https://wiki.sj.ifsc.edu.br/index.php/BCD_-_Binary-coded_decimal>. Acesso em: 5 jul. 2022.

MONTE, J. V. L. do. *IOTALKER: Aprimorando a integração de dispositivos IoT*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, 2019.

MOONEY, J. D. *Bringing Portability to the Software Process*. 2000. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.9762&rep=rep1&type=pdf>>. Acesso em: 5 jul. 2022.

NAGEL, R. G. *Biblioteca NBR14522*. 2022. Disponível em: <<https://github.com/RGNagel/lib-NBR14522/tree/v1.0.0>>. Acesso em: 5 jul. 2022.

Nansen S/A. *Manual SPECTRUM S*. 2011. Disponível em: <http://ruthmann.com.br/cms/assets/pdf/arquivos/1972012_manual-medidor-de-energia-nansen-spectrum-s.pdf>. Acesso em: 30 jul. 2022.

Nansen S/A. *Medidores*. 2022. Disponível em: <<http://nansen.com.br>>. Acesso em: 30 jul. 2022.

PROFANTER, S.; CONTRIBUTORS. *Raspberry Pi Toolchains*. 2022. Disponível em: <<https://github.com/Pro/raspi-toolchain>>. Acesso em: 5 jul. 2022.

Raspberry Pi Foundation. *Raspberry Pi*. 2012. Disponível em: <<https://www.raspberrypi.org/>>. Acesso em: 5 jul. 2022.

ROSA, L. H. L. da. *Sistema de apoio à gestão de utilidades e energia: aplicação de conceitos de sistemas de informação e de apoio à tomada de decisão*. Dissertação (Mestrado) — Escola Politécnica, Universidade de São Paulo, São Paulo, 2007.

SCHILDT, H. *C Completo e Total*. 3ª. ed. [S.l.]: MAKRON Books Ltda, 1997.

SHVETS, A. *Dive Into Design Patterns*. [S.l.]: Refactoring.Guru, 2019.

SIGROK. *Libserialport*. 2020. Disponível em: <<https://sigrok.org/wiki/Libserialport>>. Acesso em: 15 mai. 2022.

STROUSTRUP, B. *The C++ Programming Language*. 4ª. ed. [S.l.]: Addison-Wesley, 2013.

Wikipedia contributors. *List of unit testing frameworks*. Wikipedia, The Free Encyclopedia, 2022. Disponível em: <https://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=1097219947>. Acesso em: 15 mai. 2022.

APÊNDICES

APÊNDICE A – CÓDIGO-FONTE PRINCIPAL DA BIBLIOTECA

Código A.1 – Classe *LeitorFSM*

```

1 #pragma once
2 #include <CRC.h>
3 #include <NBR14522.h>
4 #include <functional>
5 #include <memory>
6
7 template <typename T> using sptr = std::shared_ptr<T>;
8
9 template <class TimerPolicy, class SerialPolicy> class LeitorFSM {
10 public:
11     typedef enum {
12         Dessincronizado,
13         Sincronizado,
14         ComandoTransmitido,
15         AtrasoDeSequenciaRecebido,
16        CodigoRecebido,
17         AguardaNovoComando,
18     } estado_t;
19
20     typedef enum {
21         Sucesso,
22         Processando,
23         ErroLimiteDeNAKsRecebidos,
24         ErroLimiteDeNAKsTransmitidos,
25         ErroLimiteDeTransmissoesSemRespostas,
26         ErroTempoSemWaitEsgotado,
27         ErroLimiteDeWaitsRecebidos,
28         ErroQuebraDeSequencia,
29         ErroAposRespostaRecebeNAK,
30         ErroSemRespostaAoAguardarProximaResposta,
31         ExcecaoOcorrenciaNoMedidor,
32         ExcecaoComandoNaoImplementado
33     } status_t;
34
35     void setComando(const NBR14522::comando_t& comando) {
36         _comando = comando;
37         _estado = Dessincronizado;
38         _status = Processando;
39         _esvaziaPortaSerial();
40     }
41
42     void
43     setCallback(std::function<void(const NBR14522::resposta_t& rsp)> callback) {

```

```
44     _callback = callback;
45 }
46
47 estado_t processaEstado() {
48
49     byte_t byte;
50     size_t bytesLidosSz;
51
52     switch (_estado) {
53     case AguardaNovoComando:
54         // nao faz nada neste estado, aguardando comando ser setado em
55         // setComando()
56         break;
57     case Dessincronizado:
58         if (_porta->rx(&byte, 1) && byte == NBR14522::ENQ) {
59             _estado = Sincronizado;
60             _timer.setTimeout(NBR14522::TMAXENQ_MSEC);
61         }
62         break;
63     case Sincronizado:
64         if (_timer.timedOut()) {
65             _estado = Dessincronizado;
66             _esvaziaPortaSerial();
67         } else if (_porta->rx(&byte, 1) && byte == NBR14522::ENQ) {
68             _transmiteComando();
69             _counterNakRecebido = 0;
70             _counterNakTransmitido = 0;
71             _counterSemResposta = 0;
72             _counterWaitRecebido = 0;
73             _isRespostaComposta = false;
74             _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
75             _estado = ComandoTransmitido;
76         }
77         break;
78     case ComandoTransmitido:
79         if (_timer.timedOut()) {
80             _esvaziaPortaSerial();
81             _counterSemResposta++;
82             if (_counterSemResposta == NBR14522::MAX_COMANDO_SEM_RESPOSTA) {
83                 // falhou
84                 _estado = AguardaNovoComando;
85                 _status = ErroLimiteDeTransmissoesSemRespostas;
86             } else if (_isRespostaComposta) {
87                 // falhou
88                 _estado = AguardaNovoComando;
89                 _status = ErroSemRespostaAoAguardarProximaResposta;
90             } else {
```

```

91         _transmiteComando();
92         _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
93     }
94 } else if (_porta->rx(&byte, 1)) {
95     // byte recebido
96     if (byte == NBR14522::NAK) {
97         _counterNakRecebido++;
98         if (_counterNakRecebido == NBR14522::MAX_BLOCO_NAK) {
99             // falha
100             _status = ErroLimiteDeNAKsRecebidos;
101             _estado = AguardaNovoComando;
102         } else if (_isRespostaComposta) {
103             // falha
104             _status = ErroAposRespostaRecebeNAK;
105             _estado = AguardaNovoComando;
106         } else {
107             _transmiteComando();
108             _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
109         }
110     } else if (byte == NBR14522::WAIT) {
111         _estado = AtrasoDeSequenciaRecebido;
112         _timer.setTimeout(NBR14522::TSEMWAIT_SEC * 1000);
113     } else if (
114         byte == _comando.at(0) ||
115         byte == NBR14522::CodigoInformacaoDeOcorrenciaNoMedidor ||
116         byte ==
117             NBR14522::CodigoInformacaoDeComandoNaoImplementado) {
118         // código do comando
119         _resposta.at(0) = byte;
120         _respostaBytesLidos = 1;
121         _timer.setTimeout(NBR14522::TMAXCAR_MSEC);
122         _estado = CodigoRecebido;
123     } else if (byte == NBR14522::ENQ && _isRespostaComposta) {
124         // "se após o tempo permitido para a leitora enviar ACK
125         // este ainda não foi enviado, o medidor deve enviar ENQ
126         // aguardando o recebimento do ACK"
127
128         // retransmite ACK
129         byte = NBR14522::ACK;
130         _porta->tx(&byte, 1);
131     } else {
132         // "a recepção de algo que não seja SINALIZADOR ou
133         // BLOCO DE DADOS [resposta ou comando] deve provocar
134         // uma QUEBRA DE SEQUÊNCIA"
135         _estado = Dessincronizado;
136         _status = ErroQuebraDeSequencia;
137         _esvaziaPortaSerial();

```

```

138         }
139     }
140     break;
141     case AtrasoDeSequenciaRecebido:
142         if (_timer.timedOut()) {
143             // falhou
144             _estado = AguardaNovoComando;
145             _status = ErroTempoSemWaitEsgotado;
146         } else if (_porta->rx(&byte, 1)) {
147             // byte recebido
148             if (byte == NBR14522::ENQ) {
149                 _estado = ComandoTransmitido;
150                 _transmiteComando();
151                 _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
152             } else if (byte == NBR14522::WAIT) {
153                 _counterWaitRecebido++;
154                 if (_counterWaitRecebido == NBR14522::MAX_BLOCO_WAIT) {
155                     // falhou
156                     _estado = AguardaNovoComando;
157                     _status = ErroLimiteDeWaitsRecebidos;
158                 } else {
159                     _timer.setTimeout(NBR14522::TSEMWAIT_SEC * 1000);
160                 }
161             } else {
162                 // "a recepção de algo que não seja SINALIZADOR ou
163                 // BLOCO DE DADOS [resposta ou comando] deve provocar
164                 // uma QUEBRA DE SEQUÊNCIA"
165                 _estado = Dessincronizado;
166                 _status = ErroQuebraDeSequencia;
167                 _esvaziaPortaSerial();
168             }
169         }
170     break;
171     case CodigoRecebido:
172         bytesLidosSz =
173             _porta->rx(&_resposta[_respostaBytesLidos],
174                     NBR14522::RESPOSTA_SZ - _respostaBytesLidos);
175         _respostaBytesLidos += bytesLidosSz;
176
177         if (bytesLidosSz)
178             _timer.setTimeout(NBR14522::TMAXCAR_MSEC);
179
180         if (_timer.timedOut()) {
181             _counterSemResposta++;
182             _esvaziaPortaSerial();
183             if (_counterSemResposta == NBR14522::MAX_COMANDO_SEM_RESPOSTA) {
184                 // falhou

```

```

185         _estado = AguardaNovoComando;
186         _status = ErroLimiteDeTransmissoesSemRespostas;
187     } else {
188         _transmiteComando();
189         _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
190         _estado = ComandoTransmitido;
191     }
192 } else if (_respostaBytesLidos >= NBR14522::RESPOSTA_SZ) {
193     // resposta completa recebida, verifica CRC
194     if (NBR14522::getCRC(_resposta) ==
195         CRC16(_resposta.data(), _resposta.size() - 2)) {
196         // CRC correto
197         // transmite ACK
198         byte = NBR14522::ACK;
199         _porta->tx(&byte, 1);
200
201         // chama callback caso tenha sido setado
202         if (_callback)
203             _callback(_resposta);
204
205         if (_isComposto(_resposta.at(0))) {
206             _isRespostaComposta = true;
207             if (_isUltimaRespostaDeComandoComposto(_resposta)) {
208                 // resposta composta recebida por completo,
209                 // sucesso
210                 _estado = estado_t::AguardaNovoComando;
211                 _status = Sucesso;
212             } else {
213                 // resetar contadores, pois são referentes a
214                 // cada resposta. Obs.: nao zera contador de NAK
215                 // recebidos pois o comando já foi recebido
216                 // corretamente pelo medidor e a partir de agora
217                 // o medidor nao deve enviar mais NAKs.
218                 _counterNakTransmitido = 0;
219                 _counterSemResposta = 0;
220                 _counterWaitRecebido = 0;
221                 _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
222                 _estado = ComandoTransmitido;
223             }
224         } else {
225             // resposta simples recebida
226
227             if (_resposta.at(0) ==
228                 NBR14522::CodigoInformacaoDeOcorrenciaNoMedidor)
229                 _status = ExcecaoOcorrenciaNoMedidor;
230             else if (_resposta.at(0) ==
231                 NBR14522::

```

```

232                                    CodigoInformacaoDeComandoNaoImplementado)
233         _status = ExcecaoComandoNaoImplementado;
234     else
235         _status = Sucesso;
236
237         _estado = estado_t::AguardaNovoComando;
238     }
239 } else {
240     // CRC incorreto
241     // transmite NAK
242     byte = NBR14522::NAK;
243     _porta->tx(&byte, 1);
244     _counterNakTransmitido++;
245     if (_counterNakTransmitido == NBR14522::MAX_BLOCO_NAK) {
246         // falhou
247         _estado = estado_t::AguardaNovoComando;
248         _status = status_t::ErroLimiteDeNAKsTransmitidos;
249     } else {
250         _estado = estado_t::ComandoTransmitido;
251         _timer.setTimeout(NBR14522::TMAXRSP_MSEC);
252     }
253 }
254 }
255 break;
256 }
257
258 return _estado;
259 }
260
261 LeitorFSM(sptr<SerialPolicy> porta) : _porta(porta) {}
262
263 uint32_t counterNakRecebido() { return _counterNakRecebido; }
264 uint32_t counterNakTransmitido() { return _counterNakTransmitido; }
265 uint32_t counterSemResposta() { return _counterSemResposta; }
266 uint32_t counterWaitRecebido() { return _counterWaitRecebido; }
267 status_t status() { return _status; }
268
269 NBR14522::resposta_t resposta() { return _resposta; }
270
271 private:
272     estado_t _estado = AguardaNovoComando;
273     status_t _status = Processando;
274     sptr<SerialPolicy> _porta;
275     TimerPolicy _timer;
276     NBR14522::comando_t _comando;
277     NBR14522::resposta_t _resposta;
278     size_t _respostaBytesLidos;

```

```

279     uint32_t _counterNakRecebido = 0;
280     uint32_t _counterNakTransmitido = 0;
281     uint32_t _counterSemResposta = 0;
282     uint32_t _counterWaitRecebido = 0;
283     bool _isRespostaComposta = false;
284     std::function<void(const NBR14522::resposta_t& rsp)> _callback = nullptr;
285
286     void _esvaziaPortaSerial() {
287         byte_t buf[32];
288         while (_porta->rx(buf, sizeof(buf)))
289             ;
290     }
291
292     void _transmiteComando() {
293         // nao incluir os dois ultimos bytes de CRC no calculo do CRC
294         NBR14522::setCRC(_comando, CRC16(_comando.data(), _comando.size() - 2));
295         _porta->tx(_comando.data(), _comando.size());
296     }
297
298     bool _isComposto(const byte_t codigo) {
299         return codigo == 0x26 || codigo == 0x27 || codigo == 0x52;
300     }
301
302     bool
303     _isUltimaRespostaDeComandoComposto(const NBR14522::resposta_t& resposta) {
304         return resposta.at(5) & 0x10;
305     }
306 };

```

Código A.2 – Classe *Leitor*

```

1 #pragma once
2
3 #include <functional>
4 #include <leitor_fsm.h>
5 #include <log_policy.h>
6
7 template <class TimerPolicy, class SerialPolicy,
8           class LogPolicy = LogPolicyStdout>
9 class Leitor {
10
11     using FSM = LeitorFSM<TimerPolicy, SerialPolicy>;
12
13 public:
14     Leitor(sptr<SerialPolicy> porta) : _leitor(porta) {}
15
16     bool leitura(NBR14522::comando_t& comando,

```

```

17         std::function<void(const NBR14522::resposta_t& rsp)> callback,
18         uint32_t timeout_resposta_ms = 0) {
19
20     _leitor.setComando(comando);
21     _leitor.setCallback(callback);
22
23     TimerPolicy leituraDeadline;
24     if (timeout_resposta_ms)
25         leituraDeadline.setTimeout(timeout_resposta_ms);
26
27     _leitor.setCallback([&](const NBR14522::resposta_t& rsp) {
28         // reatualiza o timeout informado pelo usuário toda vez que uma
29         // resposta é recebida pelo leitor
30         if (timeout_resposta_ms)
31             leituraDeadline.setTimeout(timeout_resposta_ms);
32
33         callback(rsp);
34     });
35
36     while (true) {
37         typename FSM::estado_t estado = _leitor.processaEstado();
38
39         switch (estado) {
40             case FSM::estado_t::AguardaNovoComando:
41                 switch (_leitor.status()) {
42                     case FSM::status_t::Sucesso:
43                         return true;
44                     case FSM::status_t::Processando:
45                         break;
46                     default:
47                         LogPolicy::log("Processo falhou. Status: %s\n",
48                                     _status2verbose(_leitor.status()));
49                         return false;
50                 }
51                 break;
52             default:
53                 break;
54         }
55
56         if (timeout_resposta_ms && leituraDeadline.timedOut()) {
57             LogPolicy::log(
58                 "O processo de leitura excedeu o tempo informado pelo "
59                 "usuário (%i ms) sem receber nenhuma resposta. Estado: "
60                 "%s\n, Status: %s",
61                 timeout_resposta_ms, _estado2string(estado),
62                 _status2verbose(_leitor.status()));
63             return false;

```

```

64         }
65     }
66 }
67
68 private:
69     const char* _estado2string(const typename FSM::estado_t estado) {
70         switch (estado) {
71             case FSM::estado_t::Dessincronizado:
72                 return "Dessincronizado";
73             case FSM::estado_t::Sincronizado:
74                 return "Sincronizado";
75             case FSM::estado_t::ComandoTransmitido:
76                 return "Comando Transmitido";
77             case FSM::estado_t::AtrasoDeSequenciaRecebido:
78                 return "Atraso de Sequência Recebido";
79             case FSM::estado_t::CodigoRecebido:
80                 return "Código Recebido";
81             case FSM::estado_t::AguardaNovoComando:
82                 return "Aguarda Novo Comando";
83         }
84         return "";
85     }
86     const char* _status2verbose(const typename FSM::status_t status) {
87         switch (status) {
88             case FSM::status_t::Sucesso:
89                 return "Leitura realizada com sucesso";
90             case FSM::status_t::Processando:
91                 return "Leitura em processamento";
92             case FSM::status_t::ErroLimiteDeNAKsRecebidos:
93                 return "Erro: limite excedido de NAKs recebidos pelo leitor";
94             case FSM::status_t::ErroLimiteDeNAKsTransmitidos:
95                 return "Erro: limite excedido de NAKs transmitidos pelo leitor";
96             case FSM::status_t::ErroLimiteDeTransmissoesSemRespostas:
97                 return "Erro: limite excedido de transmissões sem resposta do "
98                     "medidor";
99             case FSM::status_t::ErroTempoSemWaitEsgotado:
100                 return "Erro: limite de tempo excedido sem receber WAIT";
101             case FSM::status_t::ErroLimiteDeWaitsRecebidos:
102                 return "Erro: limite excedido de WAITs recebidos";
103             case FSM::status_t::ErroQuebraDeSequencia:
104                 return "Erro: quebra de sequência (byte diferente de sinalizador "
105                     "ou resposta) pelo medidor";
106             case FSM::status_t::ErroAposRespostaRecebeNAK:
107                 return "Erro: NAK recebido mesmo após medidor já ter recebido "
108                     "comando íntegro";
109             case FSM::status_t::ErroSemRespostaAoAguardarProximaResposta:
110                 return "Erro: Ao aguardar próxima resposta de um comando composto, "

```

```

111         "leitor não recebe nada do medidor";
112     case FSM::status_t::ExcecaoOcorrenciaNoMedidor:
113         return "Exceção: informação de ocorrência no medidor recebida no "
114             "lugar da resposta ao comando solicitado";
115     case FSM::status_t::ExcecaoComandoNaoImplementado:
116         return "Exceção: comando solicitado ao medidor não foi "
117             "implementado";
118     }
119
120     return "";
121 }
122
123 FSM _leitor;
124 };

```

Código A.3 – Aplicação de um leitor com interface via linha de comando

```

1 #include <NBR14522.h>
2 #include <functional>
3 #include <leitor.h>
4 #include <serial/serial_policy_win_unix.h>
5 #include <timer/timer_policy_win_unix.h>
6
7 using namespace NBR14522;
8
9 #define TIMEOUT_SEM_RESPOSTA_MS 5000
10
11 template <typename T, size_t N>
12 void print_arr_hex(const std::array<T, N>& arr) {
13     for (const auto i : arr)
14         printf("%02X", (int)i);
15 }
16
17 void print_usage() {
18     printf(
19         "Este programa recebe dois argumentos obrigatórios:\n\n"
20         "1 - porta serial que o medidor está conectado. Em sistemas unix será\n"
21         "um pseudo-arquivo localizado em /dev/, por exemplo: /dev/ttyUSB0,\n"
22         "/dev/ttyUSB1, etc. No Windows: COM1, COM2, etc\n\n"
23
24         "2 - comando da comunicação convencional leitor-medidor (NBR14522). \n"
25         "Este programa interpreta este parâmetro na codificação BCD. Caso o \n"
26         "comando não seja passado por parâmetro contendo todos os 64 bytes \n"
27         "referentes ao código + payload, o programa preenche o restante dos \n"
28         "bytes com zeros. Os dois últimos bytes referentes ao CRC são sempre \n"
29         "adicionados ou sobrescritos pela biblioteca NBR14522.\n\n"
30

```

```
31     "Exemplos de invocação do programa:\n"
32     "./leitor-cli /dev/ttyUSB0 14\n"
33     "./leitor-cli /dev/ttyUSB0 14123456\n"
34     "./leitor-cli /dev/ttyUSB0 20\n"
35     "./leitor-cli /dev/ttyUSB0 204455660101\n\n");
36 }
37
38 std::vector<byte_t> get_hex_bytes(const std::string& hexbytes) {
39     std::vector<byte_t> retval;
40
41     // should not be odd length
42     if (hexbytes.size() & 1)
43         return retval;
44
45     for (size_t i = 0; i < hexbytes.size(); i += 2) {
46         std::string hexbyte = hexbytes.substr(i, 2);
47         auto byte = strtoul(hexbyte.c_str(), nullptr, 16);
48         retval.push_back(static_cast<byte_t>(byte));
49     }
50
51     return retval;
52 }
53
54 int main(int argc, char* argv[]) {
55     if (argc < 3) {
56         printf("Número de argumentos inválido.\n\n");
57         print_usage();
58         return EXIT_FAILURE;
59     }
60
61     std::shared_ptr<SerialPolicyWinUnix> porta =
62         std::make_shared<SerialPolicyWinUnix>();
63
64     if (!porta->open(argv[1], 9600)) {
65         printf("Não foi possível abrir a porta serial\n\n");
66         return EXIT_FAILURE;
67     }
68
69     Leitor<TimerPolicyWinUnix, SerialPolicyWinUnix> leitor(porta);
70
71     std::vector<comando_t> comandos;
72     for (int i = 2; i < argc; i++) {
73         auto bytes = get_hex_bytes(argv[i]);
74         comando_t cmd;
75         cmd.fill(0x00);
76         std::copy(bytes.begin(), bytes.end(), cmd.begin());
77         comandos.push_back(cmd);
```

```

78     }
79
80     for (auto& comando : comandos) {
81         setCRC(comando, CRC16(comando.data(), comando.size() - 2));
82         printf("\nComando:\n");
83         print_arr_hex(comando);
84
85         printf("\nResposta(s):\n");
86         leitor.leitura(
87             comando,
88             [&](const resposta_t& rsp) {
89                 print_arr_hex(rsp);
90                 printf("\n");
91             },
92             TIMEOUT_SEM_RESPOSTA_MS);
93     }
94
95     printf("\n");
96
97     porta->close();
98     return EXIT_SUCCESS;
99 }

```

Código A.4 – Implementação da interface *TimerPolicy* usada na aplicação

```

1 #pragma once
2 #include <chrono>
3
4 using clock_type = std::chrono::system_clock;
5 using moment = std::chrono::time_point<clock_type>;
6
7 class TimerPolicyWinUnix {
8     public:
9         void setTimeout(unsigned int milliseconds) {
10             _deadline = clock_type::now() + std::chrono::milliseconds(milliseconds);
11         }
12         bool timedOut() { return clock_type::now() >= _deadline; }
13
14     private:
15         moment _deadline;
16 };

```

Código A.5 – Implementação da interface *SerialPolicy* usada na aplicação

```

1 #pragma once
2
3 #include "serialib.h"

```

```

4
5 class SerialPolicyWinUnix {
6     public:
7         ~SerialPolicyWinUnix() { _serial.closeDevice(); }
8         bool open(const char* name, const unsigned int baudrate) {
9             if (_serial.openDevice(name, baudrate) == 1)
10                 return true;
11             else
12                 return false;
13         }
14         void close() { _serial.closeDevice(); }
15
16         size_t tx(const byte_t* data, const size_t data_sz) {
17             if (_serial.writeBytes(data, data_sz) == 1)
18                 return data_sz;
19             else
20                 return 0;
21         }
22
23         size_t rx(byte_t* data, const size_t max_data_sz) {
24             int toread = _serial.available();
25             if (toread <= 0)
26                 return 0;
27
28             if (toread > static_cast<int>(max_data_sz))
29                 toread = max_data_sz;
30
31             int read = _serial.readBytes(data, toread);
32             if (read >= 0)
33                 return static_cast<size_t>(read);
34             else
35                 return 0;
36         }
37
38     private:
39         serialib _serial;
40 };

```

Código A.6 – Testes unitários da máquina de estados da classe *LeitorFSM*

```

1 #include "doctest/doctest.h"
2 #include <CRC.h>
3 #include <NBR14522.h>
4 #include <leitor_fsm.h>
5 #include <memory>
6 #include <ring_buffer.h>
7 #include <timer/timer_policy_win_unix.h>

```

```
8
9 using namespace NBR14522;
10
11 static void waitFor(unsigned int milliseconds) {
12     TimerPolicyWinUnix timer;
13     timer.setTimeout(milliseconds);
14     while (!timer.timedOut())
15         ;
16 }
17
18 // porta serial simulada para testes que permite ler dados do leitor e add
19 // dados como se fossemos o medidor
20
21 class SerialPolicyDummy {
22 public:
23     RingBuffer<byte_t, 300> toLeitor;
24     RingBuffer<byte_t, 300> toMedidor;
25     SerialPolicyDummy() {
26         // esvazia ring buffers
27         while (this->toMedidor.toread())
28             this->toMedidor.read();
29         while (this->toLeitor.toread())
30             this->toLeitor.read();
31     }
32     size_t tx(const byte_t* data, const size_t data_sz) {
33         for (size_t i = 0; i < data_sz; i++)
34             toMedidor.write(data[i]);
35         return data_sz;
36     }
37     size_t rx(byte_t* data, const size_t max_data_sz) {
38         size_t toread = toLeitor.toread();
39         size_t sz = max_data_sz >= toread ? toread : max_data_sz;
40         for (size_t i = 0; i < sz; i++)
41             data[i] = toLeitor.read();
42         return sz;
43     }
44 };
45
46 TEST_CASE("Leitor") {
47
48     sptr<SerialPolicyDummy> porta = std::make_shared<SerialPolicyDummy>();
49
50     using Leitor = LeitorFSM<TimerPolicyWinUnix, SerialPolicyDummy>;
51
52     Leitor leitor(porta);
53
54     // comando a ser transmitido pelo leitor
```

```

55     comando_t cmd_transmitido, cmd_recebido_pelo_medidor;
56     cmd_transmitido.fill(0xAA);
57     cmd_transmitido.at(0) = 0x14;
58     setCRC(cmd_transmitido,
59           CRC16(cmd_transmitido.data(), cmd_transmitido.size() - 2));
60     leitor.setComando(cmd_transmitido);
61
62     CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
63     CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
64     waitFor(1000);
65     CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
66     CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
67
68     SUBCASE("alarme TMAXENQ excedido no estado Sincronizado") {
69         porta->toLeitor.write(ENQ);
70         CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
71         CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
72         waitFor(TMAXENQ_MSEC);
73         CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
74         CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
75     }
76
77     SUBCASE("limite de transmissões sem respostas excedido (alarme TMAXRSP "
78           "excedido)") {
79         porta->toLeitor.write(ENQ);
80         CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
81         porta->toLeitor.write(ENQ);
82         CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
83         CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
84         CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
85
86         for (int i = 1; i < MAX_COMANDO_SEM_RESPOSTA; i++) {
87             // verifica que o comando foi enviado pelo leitor e está correto
88             REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
89             for (size_t j = 0; j < COMANDO_SZ; j++)
90                 cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
91             CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
92
93             // aguarda timeout TMAXRSP
94             waitFor(TMAXRSP_MSEC);
95
96             // processa e retransmite o comando
97             CHECK(leitor.processaEstado() ==
98                   Leitor::estado_t::ComandoTransmitido);
99             CHECK(leitor.processaEstado() ==
100                  Leitor::estado_t::ComandoTransmitido);
101             CHECK(leitor.processaEstado() ==

```

```

102         Leitor::estado_t::ComandoTransmitido);
103
104         // verifica estado e vars. após retransmissão
105         CHECK(leitor.counterSemResposta() == i);
106         CHECK(leitor.status() == Leitor::status_t::Processando);
107     }
108
109     // ultima tentativa
110     // verifica que o comando foi enviado pelo leitor e está correto
111     REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
112     for (size_t j = 0; j < COMANDO_SZ; j++)
113         cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
114     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
115     // aguarda timeout TMAXRSP
116     waitFor(TMAXRSP_MSEC);
117
118     // processa e nao retransmite comando, pois excedeu o limite de
119     // retransmissões
120     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
121     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
122     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
123
124     CHECK(leitor.status() ==
125         Leitor::status_t::ErroLimiteDeTransmissoesSemRespostas);
126 }
127
128 SUBCASE("alarme TMAXCAR excedido e n de retransmissões excedida") {
129     porta->toLeitor.write(ENQ);
130     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
131     porta->toLeitor.write(ENQ);
132     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
133     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
134     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
135
136     for (int i = 1; i < MAX_COMANDO_SEM_RESPOSTA; i++) {
137         // verifica que o comando foi enviado pelo leitor e está correto
138         REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
139         for (size_t j = 0; j < COMANDO_SZ; j++)
140             cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
141         CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
142
143         // envia primeiro byte (código) ao leitor
144         porta->toLeitor.write(cmd_transmitido.at(0));
145         CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
146
147         // aguarda timeout TMAXCAR
148         waitFor(TMAXCAR_MSEC);

```

```

149
150         // processa e retransmite o comando
151         CHECK(leitor.processaEstado() ==
152             Leitor::estado_t::ComandoTransmitido);
153         CHECK(leitor.processaEstado() ==
154             Leitor::estado_t::ComandoTransmitido);
155         CHECK(leitor.processaEstado() ==
156             Leitor::estado_t::ComandoTransmitido);
157
158         // verifica estado e vars. após retransmissão
159         CHECK(leitor.counterSemResposta() == i);
160         CHECK(leitor.status() == Leitor::status_t::Processando);
161     }
162
163     // ultima tentativa
164     // verifica que o comando foi enviado pelo leitor e está correto
165     REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
166     for (size_t j = 0; j < COMANDO_SZ; j++)
167         cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
168     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
169
170     // envia primeiro byte (código) ao leitor
171     porta->toLeitor.write(cmd_transmitido.at(0));
172     CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
173
174     // aguarda timeout TMAXCAR
175     waitFor(TMAXCAR_MSEC);
176
177     // processa e nao retransmite comando, pois excedeu o limite de
178     // retransmissões
179     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
180     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
181     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
182
183     CHECK(leitor.status() ==
184         Leitor::status_t::ErroLimiteDeTransmissoesSemRespostas);
185 }
186
187 SUBCASE("limite de NAKs recebidos excedido") {
188     porta->toLeitor.write(ENQ);
189     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
190     porta->toLeitor.write(ENQ);
191
192     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
193     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
194     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
195

```

```

196     for (int i = 1; i < MAX_BLOCO_NAK; i++) {
197         porta->toLeitor.write(NAK);
198         CHECK(leitor.processaEstado() ==
199             Leitor::estado_t::ComandoTransmitido);
200         CHECK(leitor.counterNakRecebido() == i);
201     }
202
203     porta->toLeitor.write(NAK);
204
205     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
206     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
207     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
208
209     CHECK(leitor.status() == Leitor::status_t::ErroLimiteDeNAKsRecebidos);
210 }
211
212 SUBCASE("limite de NAKs transmitidos excedido (CRC das respostas sempre "
213     "com erro)") {
214     porta->toLeitor.write(ENQ);
215     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
216     porta->toLeitor.write(ENQ);
217
218     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
219     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
220     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
221
222     for (int i = 1; i < MAX_BLOCO_NAK; i++) {
223         // envia 1o byte da resposta (código do comando)
224         porta->toLeitor.write(cmd_transmitido.at(0));
225
226         CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
227         CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
228         CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
229
230         // envia restante do comando (dados errados para o CRC nao bater)
231         for (size_t j = 1; j < RESPOSTA_SZ; j++)
232             porta->toLeitor.write(0xDE);
233
234         CHECK(leitor.processaEstado() ==
235             Leitor::estado_t::ComandoTransmitido);
236         CHECK(leitor.counterNakTransmitido() == i);
237     }
238
239     // envia 1o byte da resposta (código do comando)
240     porta->toLeitor.write(cmd_transmitido.at(0));
241
242     CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);

```

```
243     CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
244     CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
245
246     // envia restante do comando (dados errados para o CRC nao bater)
247     for (size_t j = 1; j < RESPOSTA_SZ; j++)
248         porta->toLeitor.write(0xDE);
249
250     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
251     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
252     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
253
254     CHECK(leitor.status() ==
255           Leitor::status_t::ErroLimiteDeNAKsTransmitidos);
256 }
257
258 SUBCASE("Resposta simples válida recebida") {
259     porta->toLeitor.write(ENQ);
260     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
261     porta->toLeitor.write(ENQ);
262
263     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
264     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
265     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
266
267     // gera resposta válida
268     resposta_t resposta;
269     resposta.at(0) = cmd_transmitido.at(0);
270     NBR14522::setCRC(resposta, CRC16(resposta.data(), resposta.size() - 2));
271
272     for (size_t i = 0; i < (NBR14522::RESPOSTA_SZ - 1); i++) {
273         porta->toLeitor.write(resposta.at(i));
274         CHECK(leitor.processaEstado() == Leitor::estado_t::CodigoRecebido);
275     }
276
277     porta->toLeitor.write(resposta.at(NBR14522::RESPOSTA_SZ - 1));
278     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
279     CHECK(leitor.status() == Leitor::status_t::Sucesso);
280
281     resposta_t resposta_recebida_pelo_leitor = leitor.resposta();
282     CHECK(resposta_recebida_pelo_leitor == resposta);
283 }
284
285 SUBCASE("Resposta composta válida recebida") {
286
287     cmd_transmitido.at(0) = 0x26; // composto
288
289     setCRC(cmd_transmitido,
```

```
290         CRC16(cmd_transmitido.data(), cmd_transmitido.size() - 2));
291     leitor.setComando(cmd_transmitido);
292
293     porta->toLeitor.write(ENQ);
294     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
295     porta->toLeitor.write(ENQ);
296
297     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
298     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
299     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
300
301     // verifica comando transmitido pelo leitor
302     REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
303     for (size_t j = 0; j < COMANDO_SZ; j++)
304         cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
305     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
306
307     resposta_t rsp;
308
309     // resposta intermediaria 1
310
311     rsp.fill(0x00);
312     rsp.at(0) = 0x26;
313     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
314
315     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
316         porta->toLeitor.write(rsp.at(j));
317         leitor.processaEstado();
318     }
319     CHECK(porta->toMedidor.read() == ACK);
320     CHECK(leitor.status() == Leitor::status_t::Processando);
321     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
322     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
323     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
324     CHECK(leitor.resposta() == rsp);
325
326     // resposta intermediaria 2
327
328     rsp.fill(0x00);
329     rsp.at(0) = 0x26;
330     rsp.at(47) = 0xAB; // só pra ser diferente da resposta 1
331     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
332
333     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
334         porta->toLeitor.write(rsp.at(j));
335         leitor.processaEstado();
336     }
```

```

337     CHECK(porta->toMedidor.read() == ACK);
338     CHECK(leitor.status() == Leitor::status_t::Processando);
339     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
340     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
341     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
342     CHECK(leitor.resposta() == rsp);
343
344     // resposta intermediaria 3
345
346     rsp.fill(0x00);
347     rsp.at(0) = 0x26;
348     rsp.at(22) = 0xCD; // só pra ser diferente da resposta 1 e 2
349     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
350
351     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
352         porta->toLeitor.write(rsp.at(j));
353         leitor.processaEstado();
354     }
355     CHECK(porta->toMedidor.read() == ACK);
356     CHECK(leitor.status() == Leitor::status_t::Processando);
357     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
358     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
359     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
360     CHECK(leitor.resposta() == rsp);
361
362     // resposta final
363
364     rsp.fill(0x00);
365     rsp.at(0) = 0x26;
366     rsp.at(5) = 0x10; // indica que é resposta final
367     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
368
369     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
370         porta->toLeitor.write(rsp.at(j));
371         leitor.processaEstado();
372     }
373     CHECK(porta->toMedidor.read() == ACK);
374     CHECK(leitor.status() == Leitor::status_t::Sucesso);
375     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
376     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
377     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
378     CHECK(leitor.resposta() == rsp);
379 }
380
381 SUBCASE("Resposta composta com retransmissão de ACK pelo leitor") {
382
383     cmd_transmitido.at(0) = 0x26; // composto

```

```
384
385     setCRC(cmd_transmitido,
386           CRC16(cmd_transmitido.data(), cmd_transmitido.size() - 2));
387     leitor.setComando(cmd_transmitido);
388
389     porta->toLeitor.write(ENQ);
390     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
391     porta->toLeitor.write(ENQ);
392
393     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
394
395     // verifica comando transmitido pelo leitor
396     REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
397     for (size_t j = 0; j < COMANDO_SZ; j++)
398         cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
399     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
400
401     resposta_t rsp;
402
403     // resposta intermediaria 1
404
405     rsp.fill(0x00);
406     rsp.at(0) = 0x26;
407     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
408
409     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
410         porta->toLeitor.write(rsp.at(j));
411         leitor.processaEstado();
412     }
413
414     CHECK(porta->toMedidor.read() == ACK);
415     CHECK(leitor.status() == Leitor::status_t::Processando);
416     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
417     CHECK(leitor.resposta() == rsp);
418
419     // medidor avisa o leitor que nao recebeu o ultimo ACK
420     porta->toLeitor.write(ENQ);
421
422     // verifica que o leitor reenviou o ACK
423     CHECK(porta->toMedidor.read() == ACK);
424     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
425
426     // resposta final
427
428     rsp.fill(0x00);
429     rsp.at(0) = 0x26;
430     rsp.at(5) = 0x10; // indica que é resposta final
```

```

431     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
432
433     for (size_t j = 0; j < RESPOSTA_SZ; j++) {
434         porta->toLeitor.write(rsp.at(j));
435         leitor.processaEstado();
436     }
437     CHECK(porta->toMedidor.read() == ACK);
438     CHECK(leitor.status() == Leitor::status_t::Sucesso);
439     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
440     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
441     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
442     CHECK(leitor.resposta() == rsp);
443 }
444
445 SUBCASE("Recebimento de exceções") {
446     cmd_transmitido.at(0) = 0x26; // composto
447
448     // seta comando
449     setCRC(cmd_transmitido,
450           CRC16(cmd_transmitido.data(), cmd_transmitido.size() - 2));
451     leitor.setComando(cmd_transmitido);
452
453     porta->toLeitor.write(ENQ);
454     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
455     porta->toLeitor.write(ENQ);
456
457     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
458
459     // verifica comando transmitido pelo leitor
460     REQUIRE(porta->toMedidor.toread() == COMANDO_SZ);
461     for (size_t j = 0; j < COMANDO_SZ; j++)
462         cmd_recebido_pelo_medidor.at(j) = porta->toMedidor.read();
463     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
464
465     resposta_t rsp;
466
467     SUBCASE("informação de ocorrência no medidor") {
468         // medidor responde com a resposta "informacao de ocorrência no
469         // medidor" ao invés da resposta ao comando solicitado
470         rsp.at(0) = NBR14522::CodigoInformacaoDeOcorrenciaNoMedidor;
471         setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
472         for (size_t i = 0; i < RESPOSTA_SZ; i++) {
473             porta->toLeitor.write(rsp.at(i));
474             leitor.processaEstado();
475         }
476
477         CHECK(leitor.processaEstado() ==

```

```

478         Leitor::estado_t::AguardaNovoComando);
479     CHECK(leitor.status() ==
480         Leitor::status_t::ExcecaoOcorrenciaNoMedidor);
481 }
482
483 SUBCASE("informacao de comando não implementado") {
484     // medidor responde com a resposta "informacao de comando não
485     // implementado" ao invés da resposta ao comando solicitado
486     rsp.at(0) = NBR14522::CodigoInformacaoDeComandoNaoImplementado;
487     setCRC(rsp, CRC16(rsp.data(), rsp.size() - 2));
488     for (size_t i = 0; i < RESPOSTA_SZ; i++) {
489         porta->toLeitor.write(rsp.at(i));
490         leitor.processaEstado();
491     }
492
493     CHECK(leitor.processaEstado() ==
494         Leitor::estado_t::AguardaNovoComando);
495     CHECK(leitor.status() ==
496         Leitor::status_t::ExcecaoComandoNaoImplementado);
497 }
498 }
499 }
500
501 class TimerPolicyReduzTempoDeWAIT {
502 public:
503     static const unsigned int TSEMWAIT_REDUZIDO_MSEC;
504     void setTimeout(unsigned int milliseconds) {
505         if (milliseconds == (NBR14522::TSEMWAIT_SEC * 1000))
506             _deadline = clock_type::now() +
507                 std::chrono::milliseconds(TSEMWAIT_REDUZIDO_MSEC);
508         else
509             _deadline =
510                 clock_type::now() + std::chrono::milliseconds(milliseconds);
511     }
512     bool timedOut() { return clock_type::now() >= _deadline; }
513
514 private:
515     moment _deadline;
516 };
517 const unsigned int TimerPolicyReduzTempoDeWAIT::TSEMWAIT_REDUZIDO_MSEC = 1000;
518
519 TEST_CASE("Atraso de sequência (WAIT) com timer simulado") {
520
521     sptr<SerialPolicyDummy> porta = std::make_shared<SerialPolicyDummy>();
522
523     using Leitor = LeitorFSM<TimerPolicyReduzTempoDeWAIT, SerialPolicyDummy>;
524

```

```

525     Leitor leitor(porta);
526
527     // seta comando
528     comando_t cmd_transmitido, cmd_recebido_pelo_medidor;
529     cmd_transmitido.fill(0xAA);
530     cmd_transmitido.at(0) = 0x14;
531     setCRC(cmd_transmitido,
532           CRC16(cmd_transmitido.data(), cmd_transmitido.size() - 2));
533     leitor.setComando(cmd_transmitido);
534
535     CHECK(leitor.processaEstado() == Leitor::estado_t::Dessincronizado);
536     porta->toLeitor.write(ENQ);
537     CHECK(leitor.processaEstado() == Leitor::estado_t::Sincronizado);
538     porta->toLeitor.write(ENQ);
539     CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
540
541     // verifica que comando foi enviado corretamente pelo leitor
542     CHECK(porta->toMedidor.toread() == COMANDO_SZ);
543     for (size_t i = 0; i < COMANDO_SZ; i++)
544         cmd_recebido_pelo_medidor.at(i) = porta->toMedidor.read();
545     CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
546
547     // medidor solicita atraso de sequência
548     porta->toLeitor.write(WAIT);
549     CHECK(leitor.processaEstado() ==
550           Leitor::estado_t::AtrasoDeSequenciaRecebido);
551
552     SUBCASE("com retorno à sequência") {
553         porta->toLeitor.write(ENQ);
554         CHECK(leitor.processaEstado() == Leitor::estado_t::ComandoTransmitido);
555
556         // verifica que comando foi enviado corretamente pelo leitor
557         CHECK(porta->toMedidor.toread() == COMANDO_SZ);
558         for (size_t i = 0; i < COMANDO_SZ; i++)
559             cmd_recebido_pelo_medidor.at(i) = porta->toMedidor.read();
560         CHECK(cmd_transmitido == cmd_recebido_pelo_medidor);
561     }
562
563     SUBCASE("com alarme TSEMWAIT excedido") {
564         waitFor(TimerPolicyReduzTempoDeWAIT::TSEMWAIT_REDUZIDO_MSEC);
565         CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
566         CHECK(leitor.status() == leitor.ErroTempoSemWaitEsgotado);
567     }
568
569     SUBCASE("com alarme MAX_BLOCO_WAIT excedido") {
570
571         for (int i = 1; i < MAX_BLOCO_WAIT; i++) {

```

```
572         porta->toLeitor.write(WAIT);
573         CHECK(leitor.processaEstado() ==
574             Leitor::estado_t::AtrasoDeSequenciaRecebido);
575         CHECK(leitor.counterWaitRecebido() == i);
576     }
577
578     porta->toLeitor.write(WAIT);
579     CHECK(leitor.processaEstado() == Leitor::estado_t::AguardaNovoComando);
580     CHECK(leitor.counterWaitRecebido() == MAX_BLOCO_WAIT);
581     CHECK(leitor.status() == Leitor::status_t::ErroLimiteDeWaitsRecebidos);
582 }
583 }
```