Implementação de driver para linha serial utilizando os protocolos RS422/RS485 no EPOSMote III

Rudimar Baesso Althof

Resumo

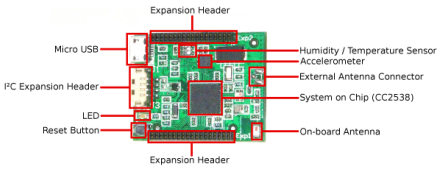
Este trabalho apresenta a modelagem do driver para o circuito integrado (CI) ISL83483, que implementa os padrões de comunicação RS422/485, no dispositivo EPOSMote III. Os detalhes da modelagem e funcionamento das máquinas de estados são apresentados. Antes da implementação realizou-se testes de unidade no componente Ordered_Queue do EPOS, verificando-se o correto funcionamento dos seus métodos. Após esta etapa criou-se testes de unidade para o driver de comunicação, permitindo o desenvolvimento orientado por testes. Após esta etapa, implementou-se os testes.

I. Introdução

A. Hardware

1) EPOSMote III: EPOSMote III é uma das principais plataformas para o EPOS 2 (Embedded Parallel Operating System).

Figura 1: Esquemático do CI.(EMBEDDED..., 2017)



Ligado ao EPOSMote III encontra-se a placa SerialCom Board que possui circuitos integrados que permitem a implementação dos protocolos CAN, RS232, RS422/485 e LIN. O circuito utilizado neste trabalho será o CI que implementa os protocolos RS422/485 e é descrito na próxima subseção.

2) CI de Comunicação: Os circuitos integrados ISL83483 da Intersil atuam como transceptores para os padrões RS-485 e RS-422. Estes dispositivos são alimentados com uma tensão de 3.3V, podendo tolerar uma variação de até 10% deste valor. Este dispositivo utiliza drivers com taxas de variação limitadas com o objetivo de minimizar interferências eletromagnéticas provindas da linha de comunicação.

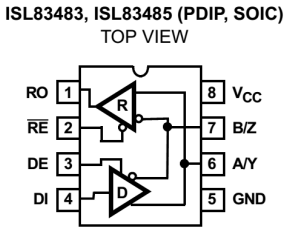
As características deste circuito são:

- Modelo: ISL83483;
- Comunicação: Half-Duplex;
- Taxa de Dados: 0.25 Mbps;
- Taxa de Variação: Limitada;
- Corrente de Fuga: 0.25 mA;
- Número de Pinos: 8;

A Figura 2 apresenta o diagrama esquemático do circuito integrado, que possui os seguintes pinos:

1) **RO:** Saida do receptor.
   Se A>B no mínimo 0.2V, RO é considerado alto. Se A<B no mínimo 0.2V, então RO é considerado baixo. RO também é alto se A e B estiverem desconectados.
2) $\overline{\mathbf{RE}}$: Habilitação da saída do receptor.
   RO é ativo quando $\overline{RE}$ está em nível lógico baixo. Quando $\overline{RE}$ é alto, RO possui alta impedância.
3) **DE:** Habilitação do driver de saída
   As saídas do driver, Y e Z, estão ativas quando o pino DE possui nivel alto. Se DE estiver com nível baixo, Y e Z possuem alta impedância.
4) **DI:** Entrada do driver.
   A inserção de nível baixo em DI força a saída Y como nível baixo e Z como nível alto. Da mesma forma, DI como nível alto força a saída Y como nível alto e Z como nível baixo.
5) **GND:** Aterramento.
6) **A/Y:** Entrada não inversora do driver.
   O pino comporta-se como entrada quando DE=0 e como saída quando DE=1.
7) **B/Z:** Entrada inversora do driver.
   O pino comporta-se como entrada quando DE=0 e como saída quando DE=1.
8) $\mathbf{V_{cc}}$: Pino de alimentação.

Figura 2: Esquemático do CI.

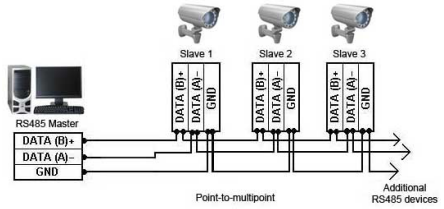


3) Os padrões RS232 e RS485: Os padrões RS422/485 estabelecem comunicação serial diferencial, permitindo maiores taxas de dados e maiores distâncias em ambientes reais. Os sinais diferenciais ajudam a anular os efeitos de deslocamento de terra e os sinais de ruídos que podem aparecer nas tensões de modo comum das redes. (RS485..., 2017)

Nas redes RS422/485 apenas um escravo e um mestre podem se comunicar por vez transmitindo

ou recebendo, mas não ao mesmo tempo. Cada escravo, ou nó, possui um endereço que é geralmente atribuido e controlado por software, podendo ser atribuido pelo software do computador mestre ou pelo software programado em cada dispositivo. Os endereços também podem ser atribuidos por hardware ao invés de software. A Figura 3 apresenta uma configuração *point-to-multipoint* na qual um computador mestre está conectado a um barramento com três câmeras escravas, possibilitando-se adicionar ainda mais dispositivos ao barramento. (RS232..., 2011)

Figura 3: Esquemático de comunicação para o padrão RS485 com um mestre e três escravos. (RS232..., 2011)



Os padrões RS422 e RS485 definem as características elétricas do driver. Como as linhas de transmissão de dados operam e modo diferencial, duas linhas são necessárias para a transmissão de um sinal.

A Tabela **??** apresenta a configuração para o envio de dados através do padrão RS485, na qual quando B está em alto e A em baixo, têm-se o sinal lógico alto. Da mesma forma, com B em baixo e A em alto tem-se o sinal lógico 0. Os padrões RS422 e RS485 têm operação similar, a maior diferença é que o padrão RS485 permite até 32 pares de dispositivos receptores/drivers enquanto que o padrão RS422 permite apenas 1 driver e 10 receptores.

Tabela I: Tabela verdade dos padrões RS422/485

A	B	Valor
1	0	0
0	1	1

A proxima seção apresenta os requerimentos do driver de comunicação RS422/485.

B. *Protocolo Serial*

Este protocolo é altamente configurável para sua correta utilização ambos os dispositivos em um barramento serial devem ser configurados com o mesmo protocolo.

Os mecanismos que permitem a remoção do clock externo e a transferência sem erros de dados são:

- Bits de dados;
- Bits de sincronização;
- Bits de paridade;
- Taxa de transmissão.

Cada bloco de dados é enviado em um pacote de bits que englobam os bits de dados, paridade e sincorização. A quantidade de bits da dados em cada pacote pode ser de 5 a 9 bits, transmitidos a partir do bit menos significativo. Os bits de sincronização são os bit de início e o(s) bit(s) de parada. Há sempre um bit de início e o número de bits de parada pode ser configurado como um ou dois. O bit de início é sempre indicado pela linha de dados ociosa em alto indo para baixo enquanto que o bit de parada recoloca a linha em alto. Os bits de paridade são um parâmetro opcional, não comumente usados, que oferecem uma forma simples de verificação, podendo ser par ou ímpar.

O protocolo mais utilizado é o 9600 8N1 que implica na taxa de transmissão de 9600 bits por segundo, sem paridade e um bit de parada. (SERIAL..., 2017)

II. Requerimentos

Esta seção descreve a partir de um conjunto de regras as respostas que devem ser forncecidas quando uma dada ação é executada. As características comportamentais esperadas do sistema são:

- O usuário deve explicitamente ativar o barramento serial para que possa realizar a comunicação nos padrões RS422/485;
- Após o início, a linha serial deve permanecer em estado ocioso (SERIAL_IDLE) até que algum comando seja emitido, seja ele de envio ou recebimento de dados;
- Operações de envio e recebimento podem apenas ser iniciados a partir do estado idle;
- Ao receber o comando de enviar dados, a linha serial passa para o estado SERIAL_SENDING até que o envio seja terminado, retornando posteriormente ao estado oscioso;
- Com o comando de receber dados, a linha serial passa para o estado de SERIAL_RECEIVING até que os dados sejam recebidos, retornando ao estado ocioso após ter recebido todos os dados ou ter atingido o timeout definido pelo usuário;

Definido as características de acesso ao barramento serial, deve-se também definir as características do protoco utilizado. Portanto, a máquina de estados também deve prever as seguintes condições:

- Permitir a configuração dos parâmetros *bits de dados*, *bits de sincronização*, *bits de paridade* e *taxa de transmissão*;
- Recepção de cada bit escrito na linha serial por um escravo e montagem de uma palavra de dados;
- Verificação da paridade, caso a mesma seja definida;

- Escrita de palavra na linha serial, bit a bit;
- Verificação de timeout na resposta de um escravo e retorno ao estado ocioso;

III. Modelagem

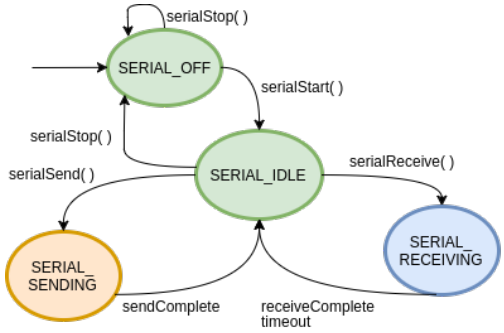
A. *Versão Simplificada*

A primeira modelagem visou definir os estados básicos do sistema, que são:

- SERIAL_IDLE: A linha serial encontra-se disponível para utilização;
- SERIAL_SENDING: A linha serial está sendo utilizada para envio de dados;
- SERIAL_RECEIVING: A linha serial está sendo utilizada para recebimento de dados.

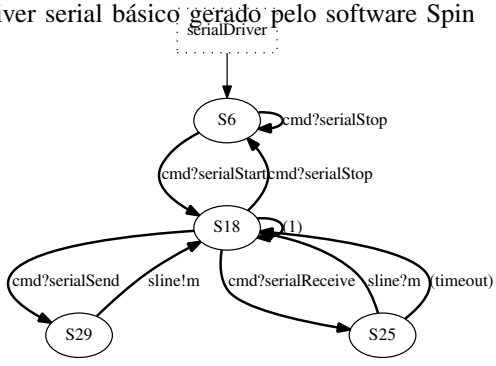
A Figura 4 apresenta o diagrama de estados que representa as regras citadas acima.

Figura 4: Diagrama de estados do funcionamento do driver serial



O código implementando o teste deste modelo foi escrito na linguagem *Promela* e pode ser visto no Apêncice A. O diagrama de estados gerado pelo código de teste criado pode ser visto na Figura 5 e condiz com o diagrama de estados criado.

Figura 5: Diagrama de estados do funcionamento do driver serial básico gerado pelo software Spin



No código em anexo criou-se dois processos adicionais, o processo *slave* representa um escravo ouvindo a linha serial, recebendo a mensagem e a retornando. Já o processo *application* representa os comandos dados pela aplicação que farão o driver alternar entre estados.

Além disso, utilizou-se dois canais, o canal *sline* representa a linha serial onde palavras *m* (representando uma mensagem) são escritas, já o canal *cmd* é utilizado para o envio de comandos da aplicação para o driver, representando a chamada de métodos do driver serial. Todos os canais possuem comunicação half-duplex, isto é, apenas um dispositivo pode enviar dados para o outro por vez.

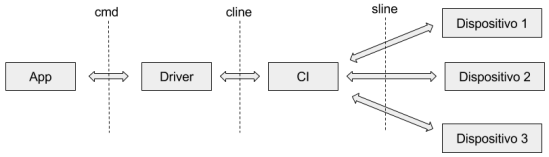
B. *Versão Completa*

Após a verificação do funcionamento do modelo simples, implementou-se o modelo completo simulando o padrão serial de comunicação.

A Figura 6 apresenta o esquema de comunicação modelado. Observa-se três canais de comunicação, operando da seguinte maneira:

- **cmd**: Canal utilizado para a troca de informações entre a aplicação e o driver serial;
- **cline**: Canal de comunicação para a escrita de bits provindos do EposMote III no CI de comunicação;
- **sline**: Canal para a troca de bits entre os dispositivos conectados ao barramento serial e o CI.

Figura 6: Esquema de comunicação do sistema.



O Apêncice B apresenta o código Promela completo desenvolvido para a verificação deste modelo.

As principais modificações feitas a partir da versão simplificada foram:

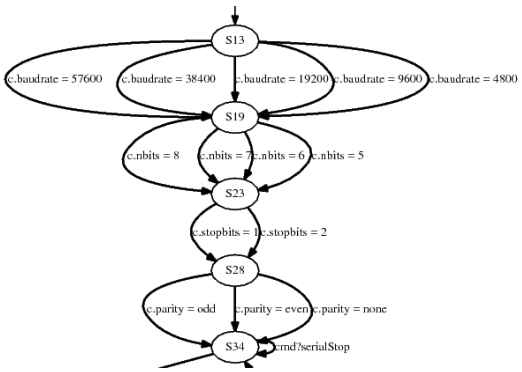
- Implementação da máquina de estados para o CI que implementa o padrão RS422/485.
- Criação de transições representando a definição dos parâmetros da comunicação serial, escolhidos de maneira não deterministica;
- O canal de comunicação *cline* representado a comuncação entre o EposMote III e o CI ISL83483 na qual ambos os dispositivos podem trocar bits entre si;
- O canal de comunicação *sline* passa a receber bits 0 e 1, representando o canal real de comunicação, a linha serial entre o CI e os dispositivos conectados à linha serial;
- Rotina representando o envio de palavra de dados bit a bit, comportando-se de forma diferenciada de acordo com as configurações de paridade, tamanho da mensagem e número de bits de parada;
- Rotina representando o recebimento de palavra de dados bit a bit, também respeitando os parâmetros citados no ítem anterior;

1) Modelo do Driver Serial:

- Múltiplas opções de taxas de dados;
- Tamanhos de palavra entre 5 a 8 bits;
- Um ou dois bits de parada;
- Paridades ímpar, par ou nula.

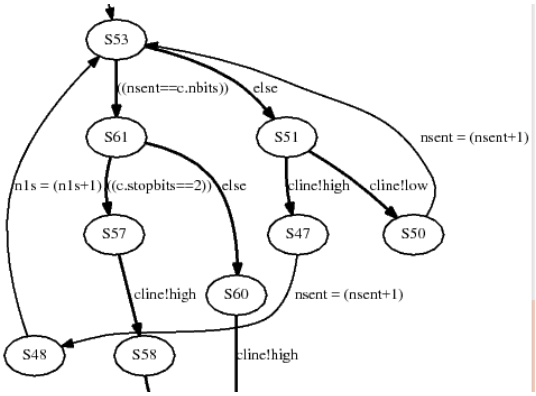
A Figura 7 apresenta esta etapa do diagrama.

Figura 7: Parte do diagrama referente a configuração.



A Figura 8 apresenta parte do diagrama de estados que modela o envio dos bits da mensagem e os bits de parada.

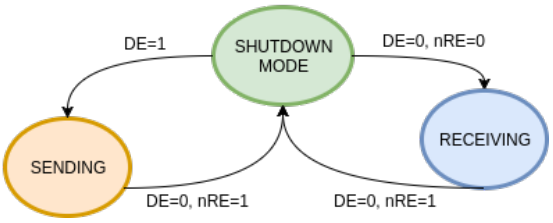
Figura 8: Parte do diagrama referente ao enviodos bits da mensage e bits de parada.



Observa-se através da Figura 25, no Apêndice C, que ao adicionar apenas estas características ao sistema obteve-se um aumento substancial no número de estados.

2) Modelo do CI ISL83483: Três estados foram modelados para o CI de comunicação e os mesmos podem ser vistos na Figura 9. Embora um quarto estado seja possível de se implementar para o CI, este não foi considerado pois possui o mesmo comportamento que o estado de desligado.

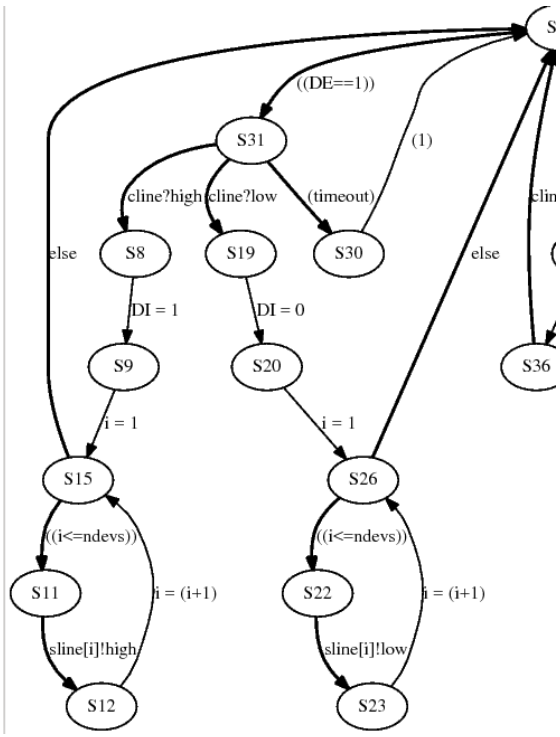
Figura 9: Diagrama de estados do CI.



O CI funciona como uma ponte entre os dispositivos conectados ao barramento serial e o computador que possui gerencia o próprio CI.

Quando em estado de transmissão, o modelo do CI na máquina de estados faz com que ao receber um bit na entrada no canal *cline*, o CI retransmita este sinal para todos os dispositivos conectados ao canal *sline*. Esta característica pode ser observada na Figura 10.

Figura 10: Parte do diagrama de estados do CI que representa a escrita no canal sline.



Para o estado de recebimento de dados, os bits recebidos no canal *sline* são retransmitidos para o canal *cline* indicando a tradução dos sinais nos terminais A e B para o pino RO, que se comunica com o EPOSMote.

Já para o estado de shutdown qualquer sinal provindo da linha serial é ignorado. O modelo da máquina de estados recebe este sinal e não realiza nada com esta informação.

O diagrama da máquina de estados completo para o CI pode ser visto na Figura 26, no Apêndice C.

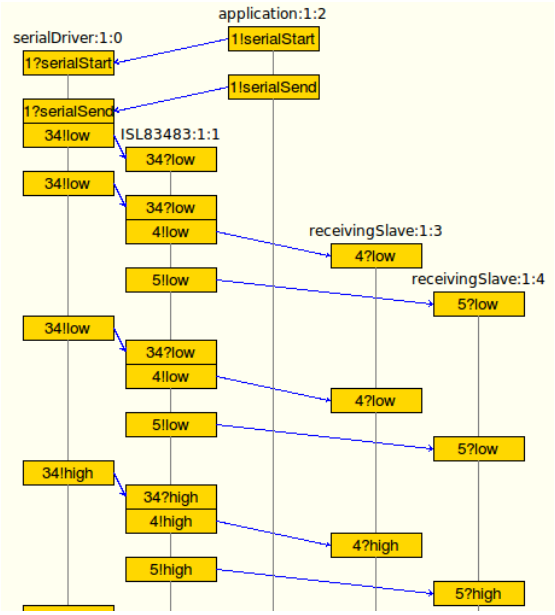
IV. Teste dos Modelos

Para o teste dos modelos de estados do driver e do CI, criou-se três processos adicionais, sendo eles:

- **Dispositivo de recepção (2x):** dois processos representando os dispositivos que recebem os dados na linha serial foram criados, sua única função é receber as mensagens da linha serial. A Figura 28 apresenta o diagrama de estados deste processo.
- **Dispositivo de envio:** este processo representa um disposito enviando dados na linha serial representada pelo canal *sline*, de acordo com os parâmetros de configuração definidos. A Figura 27 apresenta o diagrama de estados deste processo.

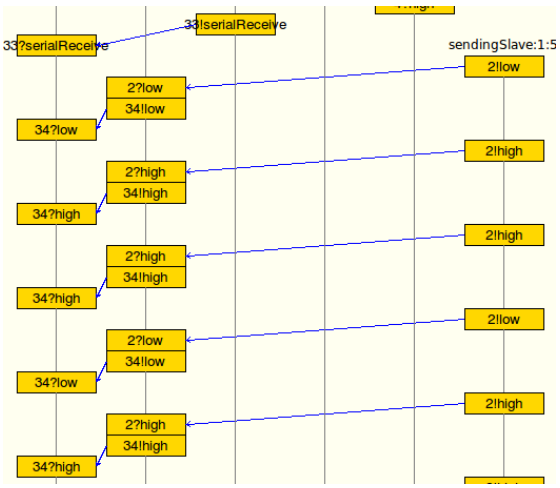
A Figura 11 apresenta um caso de execução na qual o driver é iniciado e uma requisição de envio de mensagem para a linha serial é feita. O driver então envia os bits para o CI através do canal *cline* que repassa estes bits para os dois dispositivos conectados a linha serial através do canal *sline*.

Figura 11: Envio de mensagem para os dispositivos de recepção.



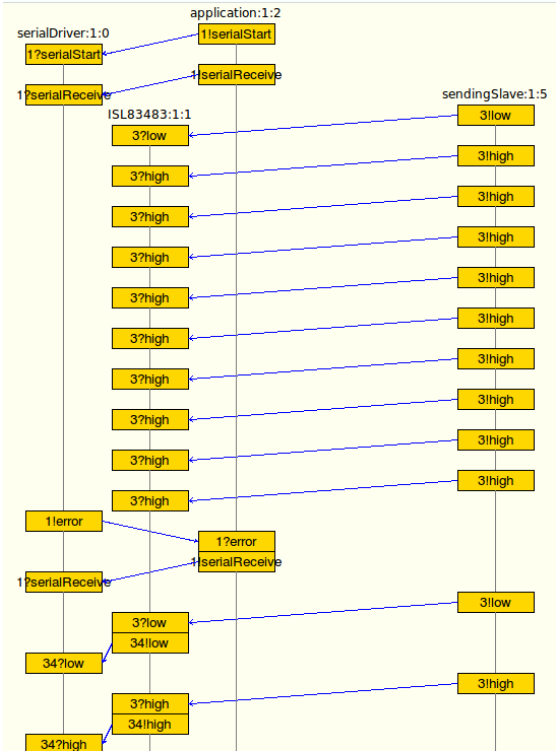
No caso apresentado na Figura 12 há o recebimento de uma mensagem da linha serial. Observa-se que o bit é recebido pelo IC e então repassado para o driver do dispositivo.

Figura 12: Recebimento de mensagem da linha serial.



No terceiro caso apresentado na Figura 13 há uma requisição para recebimento de mensagem e o dispositivo serial começa o envio antes da configuração do CI de comunicação. Como o CI encontra-se desligado, a mensagem não é repassada para o driver que por sua vez retorna um erro a aplicação através do canal *cmd*, realizado para efetuar requisições ao driver. Posteriormente, uma nova requisição de recebimento é feita na qual o CI já está ligado e os dados são entregues ao driver.

Figura 13: Erro de comunicação, CI ainda em estado desligado.



O sistema modelado foi simulado com diferentes padrões aleatórios e não apresentou deadlock, cumprindo portanto o critério de liveness.

V. Teste de Componente do EPOS

A. Classes de Equivalência

A decomposição do problema em classes de equivalência resulta na redução de casos de testes necessários para uma aplicação, evitando a explosão de condições de testes de um sistema. Duas classes de equivalências são consideradas, a classe com entradas válidas (*classe de equivalência válida* e a classe com entradas inválidas (*classe de equivalência inválida).*

B. Valores limite

O uso de valores limite é utilizado para complementar a decomposição das classes de equivalência, visando selecionar casos de testes que se encontram nas "extremidades"da classe. Em outras palavras, se o código foi elaborado para receber variáveis entre os valores a e b, testa-se com valores logo acima e abaixo dos limites.

C. Google Test

A biblioteca googletest é utilizada para o teste de unidades. Esta biblioteca permite o uso de métodos do tipo ASSERT e EXPECT que permitem a verificação de variáveis do sistema, geração de relatório de falhas e interrupção do teste ou apenas alerta de erro.

D. Google Mock

Esta biblioteca permite a simulação de objetos. Estes objetos permitem que seja feita a imitação de comportamento de objetos que seriam utilizados na aplicação e não se encontram disponíveis ou podem vir a causar atrasos como por exemplo interações com bancos de dados, redes ou outros hardwares.

A utilização de GTest e GMock é simples; ao executar a chamada RUN_ALL_TESTS() no maim, os testes são lançados. O código basico para a execução dos testes é apresentado no Algoritmo 1.

Algoritmo 1: Estrutura básica de testes utilizando a biblioteca googletest.

```
1  #include <iostream>
2  #include "gmock\gmock.h"
3  #include "gtest\gtest.h"
4
5  // Testes aqui
6
7  int main( int argc, char *argv[] ) {
8      ::testing::InitGoogleMock( &argc,
           argv );
9      return RUN_ALL_TESTS( );
10 }
```

VI. Teste do componente Ordered_Queue

A fila ordenada é uma implementação meta programada baseada nas listas do EPOS. A classe Ordered_Queue armazena objetos do tipo *Element* * que consiste de um objeto que possui um ponteiro para o seu *rank* e um ponteiro para objeto do tipo sendo armazenado.

A. Definição dos testes

Para a implementação dos testes, declarou-se o objeto do tipo *Inteiro*, apresentado no Algoritmo 2. Este objeto recebe na sua criação os valores *valor* e *rank* que representam o valor do inteiro e seu rank, respectivamente.

Algoritmo 2: Objeto *Inteiro*, definido para a execução dos testes

```
1  struct Inteiro {
2
3      //Construtor
4      Inteiro(int valor, int rank) :
5    e(this, rank), //e._object = this; e.
       rank = rank;
6    i(valor)
7    {}
8
9      int i;
10     Ordered_Queue<Inteiro>::Element e;
11 };
```

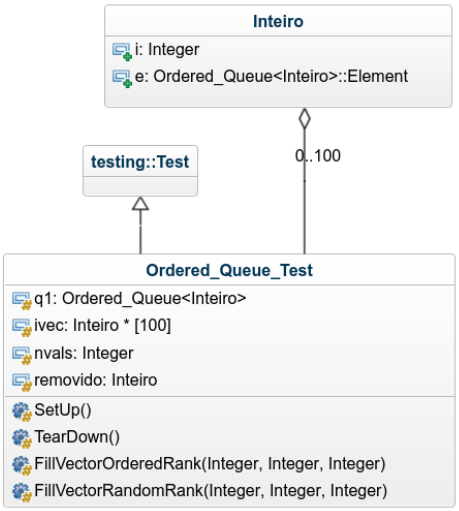
Com o objetivo de testar o funcionamento deste componente. Os seguintes testes foram realizados para esta classe:

- criação do tipo inteiro (*Inteiro*);
- inserção ordenada (*insert*);
- insersão de elemento nulo (*insert*);
- remoção em fila vazia (*empty*);
- remoção de elemento (*remove*);
- busca de elemento (*search*);
- retorno da cauda da fila (*tail*);
- retorno da cabeça da fila (*head*);
- tamanho da fila (*size*);

O Apêndice D apresenta o código completo que inclui criação do objeto auxiliar, classe acessória e o código dos testes citados até o fim deste capítulo.

Para realização dos testes uma classe acessória foi criada, chamada de *fixture* pela biblioteca googletest. Esta classe tem por objetivo instanciar objetos comuns aos testes, definir métodos de uso geral e liberar a memória utilizada nos testes. A Figura 14 apresenta o diagrama de classes para esta classe e seus atributos e métodos são apresentados na sequência.

Figura 14: Diagrama de classes para a *fixture* Ordered_Queue_Test.



Os objetos instanciados nesta classe são:

- **Ordered_Queue<Inteiro> q1:** Fila ordenada utilizada nos testes;
- **Inteiro * ivec [100]:** Array com capacidade para 100 inteiros;
- **Inteiro * removido:** Ponteiro auxiliar para a remoção de elementos;
- **int nvals:** Variável que armazena o número de de inteiros alocados na memória.

Esta classe implementa os seguintes métodos:

- **SetUp:** Atribui o valor 0 a variável *nvalue*, utilizada para controle de memória alocada aos objetos do tipo Inteiro;
- **FillVectorOrderedRank:** Insere *nvalues* valores no vetor de inteiros com rank conhecidos e valores aleatórios entre *min* e *max*;
- **FillVectorRandomRank:** Insere *nvalues* valores no vetor de inteiros com ranks aleatórios entre *min* e *max*. Os valores são iguais ao rank.
- **TearDown:** Libera a memória alocada para os *nvalues* inteiros durante cada teste.

Além disso foi necessário sobrescrever os arquivos *spin.h* (Algorimto 3) e *cpu.h* (Algorimto 4), substituindo os métodos *int_disable*, *int_enable*, *acquire*, *release*. Estes métodos forama apenas declarados para não causar problemas na compilação, visto que sua utilização não afeta os resultados dos testes.

Algoritmo 3: Classe *spin* simulada.

```
1 #ifndef spin_h
2 #define spin_h
3 class Spin{
4  public:
5    static void release(void);
6    static void acquire(void);
7 };
8 #endif
```

Algoritmo 4: Classe *cpu* simulada.

```
1 #ifndef cpu_h
2 #define cpu_h
3 class CPU{
4  public:
5    static void int_disable(void);
6    static void int_enable(void);
7 };
8 #endif
```

A implementação de cada teste é apresentado a seguir.

B. Implementação dos testes

1) Criação do tipo inteiro (Inteiro()): Este teste visa validar o objeto *Inteiro* criado para o teste da fila ordenada. Neste teste, é feita a inserção de objetos com diferentes valores e ranks e comparase se estes valores foram devidamente inseridos.

2) Inserção ordenada (insert()): Este teste faz a inserção de 100 valores com rank aleatório através do método *FillVectorRandomRank* e posteriormente sua remoção. No teste, verifica-se a ordenação da fila através da remoção do seu elemento da cabeça e comparação com o próximo elemento removido. Para a fila ser aprovada no teste, cada elemento removido deve ser maior ou igual ao elemento anterior.

C. Inserção de elemento nulo (insert(NULL))

Neste teste, é feita o valor nulo é passado como artumento do método *insert*. Espera-se que nada seja alterado na estrutura da fila sendo testada para um elemento nulo adicionado.

1) Remoção em fila vazia (empty): Nesta verificação, testa-se o comportamento da do método empty, que deve retornar *false* quando existem elementos na fila e *true* quando a fila se encontra vazia.

2) Remoção de elemento (remove): Este teste visa conferir o funcionamento do método *remove* quando o endereço de um objeto é passado como argumento. Para isto, é feita a inserção de 100 elementos de rank distintos. Posteriormente remove-se dois deles e inicia-se a remoção de elementos através do método *remove* sem argumentos. Para a aprovação deste teste, os mesmos elementos não podem devem aparecer nas remoções.

3) Busca de elemento (search): O método *search* é verificado neste teste. Para isto, cria-se 100 elementos com ranks entre 0 e 99 e apenas os 50 elementos de ranks pares são inseridos na fila ordenada. Faz-se a tentativa de remover todos os elementos, incluindo os não inseridos, e verifica-se o retorno do método. Para a aprovação, o método deve retornar os elementos quando para objetos de rank par passados como argumento e *false* para elementos de rank ímpar.

Tabela II: Testes realizados para a classe Ordered_Queue

Método da classe Ordered_Queue<Inteiro>	Possibilidades	Retorno esperado
insert(Inteiro &)	Elemento existente Elemento nulo	Inteiro inserido Nada ocorre
remove()	Fila com elementos Fila vazia	Inteiro Null
remove(Inteiro & x)	x contido na fila x não presente na fila	x Null
empty()	Fila com elementos Fila vazia	0 1
head()	Fila com elementos Fila vazia	Inteiro Null
tail()	Fila com elementos Fila vazia	Inteiro Null
head() e tail()	Fila com 1 elemento Fila com N>=2 elementos	head() == tail() head() != tail()
size()	Fila com N elementos Fila vazia	N 0

4) Retorno da cauda da fila (tail): Esta verificação consiste da inserção de elementos de rank conhecidos na fila e sua posterior remoção, verificando se o elemento esperado é o mesmo fornecido pelo método. Na remoção do último elemento, verifica-se se os elementos retornados por *head* e *tail* são os mesmos. Finalizando testa-se o método com a fila vazia, onde é esperado *null* como retorno.

5) Retorno da cabeça da fila (head: Neste teste aplica-se o mesmo procedimento utilizado no método *tail*, onde verifica-se os elementos retornados pelo método *head* e seu retorno para a fila vazia.

6) Tamanho da fila (size: Finalizando os testes, esta verificação garante o funcionamento do método *size* ao fazer a remoção de valores inseridos e comparar o retorno do método com o valor esperado.

D. Resultados dos testes

A Figura 15 apresenta o resultado dos testes para a classe Ordered_Queue. Observa-se um caso de *segmentation fault* para o teste de inserção de elemento nulo. Isto ocorre pois o método tenta acessar endereços de memórias inexistentes.

Figura 15: Resultado dos testes utilizando a biblioteca googletest.

```
[==========] Running 9 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from Inteiro
[ RUN      ] Inteiro.CriacaoDeInteiro
[       OK ] Inteiro.CriacaoDeInteiro (0 ms)
[----------] 1 test from Inteiro (0 ms total)

[----------] 8 tests from Ordered_Queue_Test
[ RUN      ] Ordered_Queue_Test.RetornoOrdenado
[       OK ] Ordered_Queue_Test.RetornoOrdenado (0 ms)
[ RUN      ] Ordered_Queue_Test.RemocaoElemento
[       OK ] Ordered_Queue_Test.RemocaoElemento (0 ms)
[ RUN      ] Ordered_Queue_Test.RetornoVazia
[       OK ] Ordered_Queue_Test.RetornoVazia (0 ms)
[ RUN      ] Ordered_Queue_Test.EncontraElemento
[       OK ] Ordered_Queue_Test.EncontraElemento (1 ms)
[ RUN      ] Ordered_Queue_Test.TailTest
[       OK ] Ordered_Queue_Test.TailTest (0 ms)
[ RUN      ] Ordered_Queue_Test.HeadTest
[       OK ] Ordered_Queue_Test.HeadTest (0 ms)
[ RUN      ] Ordered_Queue_Test.SizeTest
[       OK ] Ordered_Queue_Test.SizeTest (0 ms)
[ RUN      ] Ordered_Queue_Test.InsersaoElementoNulo
Segmentation fault (core dumped)
```

Para solucionar este problema uma simples correção no método *insert* foi efetuada para que o método seja encerrado antes de tentar efetuar qualquer ação caso o ponteiro passado seja nulo. O Algoritmo **??** apresenta esta correção.

Algoritmo 5: Correção do método *nsert* da classe Ordered_Queue.

```
1 //void insert(Element * e) { T::insert(
      e); }
2 void insert(Element * e) {
3   if(e==NULL) return;
4     T::insert(e);
5 }
```

A Figura 16 apresenta o resultado da execução dos testes após a correção do método *insert*. Observa-se que a fila ordenada cumpriu os requisitos do teste. A Tabela **??** apresenta de forma resumida os casos de teste e seus retornos esperados.

Figura 16: Resultado dos testes utilizando a biblioteca googletest.

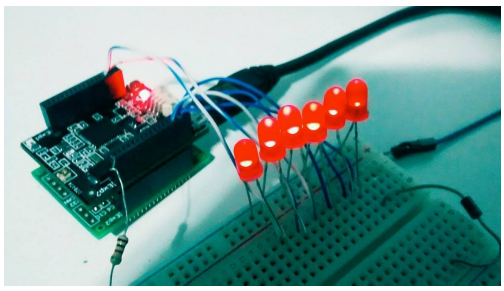
```
[==========] Running 9 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from Inteiro
[ RUN      ] Inteiro.CriacaoDeInteiro
[       OK ] Inteiro.CriacaoDeInteiro (0 ms)
[----------] 1 test from Inteiro (0 ms total)

[----------] 8 tests from Ordered_Queue_Test
[ RUN      ] Ordered_Queue_Test.RetornoOrdenado
[       OK ] Ordered_Queue_Test.RetornoOrdenado (0 ms)
[ RUN      ] Ordered_Queue_Test.RemocaoElemento
[       OK ] Ordered_Queue_Test.RemocaoElemento (0 ms)
[ RUN      ] Ordered_Queue_Test.RetornoVazia
[       OK ] Ordered_Queue_Test.RetornoVazia (0 ms)
[ RUN      ] Ordered_Queue_Test.EncontraElemento
[       OK ] Ordered_Queue_Test.EncontraElemento (1 ms)
[ RUN      ] Ordered_Queue_Test.TailTest
[       OK ] Ordered_Queue_Test.TailTest (0 ms)
[ RUN      ] Ordered_Queue_Test.HeadTest
[       OK ] Ordered_Queue_Test.HeadTest (0 ms)
[ RUN      ] Ordered_Queue_Test.SizeTest
[       OK ] Ordered_Queue_Test.SizeTest (0 ms)
[ RUN      ] Ordered_Queue_Test.InsersaoElementoNulo
[       OK ] Ordered_Queue_Test.InsersaoElementoNulo (0 ms)
[----------] 8 tests from Ordered_Queue_Test (1 ms total)

[----------] Global test environment tear-down
[==========] 9 tests from 2 test cases ran. (1 ms total)
[  PASSED  ] 9 tests.
```

VII. Desenvolvimento e Teste de Firmware

A. *Pinos de GPIO*

Um circuíto simples com LEDs e um resistor limitador de corrente foi montado para verificar as conexões com os pinos de saída utilizados na implementação. Baseado no algoritmo de exemplo *led_blink*, criou-se um algoritmo para piscar os LEDSs e verificar o funcionamento do sistema. A Figura 17 apresenta a montagem.

Figura 17: Teste dos pinos GPIO .

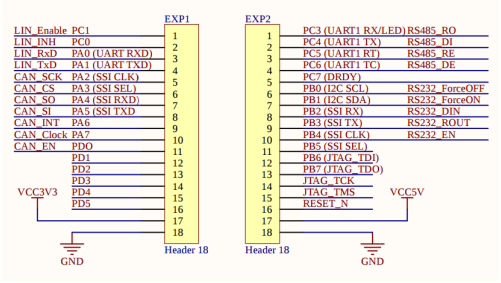


O código utilizado neste teste pode ser observado no Anexo E.

B. *Primeiros Passos do Driver*

Com base nos diagramas elétricos do sistema, observa-se que os pinos de recepção e transmissão de dados do CI de comunicação estão ligados aos pinos TX e RX da uart de número 1 do EPOS Mote (RS485_DO e RS485_DI). Desta forma é possível o uso da uart para o envio e recebimento de dados através do CI sendo necessário apenas o controle dos estados dos pinos de entrada que controlam os estados do CI (RS485_RE e RS485_DE).

Figura 18: Conexão dos pinos.



Baseado na disposição dos pinos decidiu-se criar a classe SerialRS485 através da herança da classe UART e implementar os métodos extras necessários para os controles dos pinos RS485_RE e RS485_DE. Uma primeira versão foi criada para testar o funcionamento básico do sistema e um as saídas do CI de comunicação foram conectadas a entrada de um conversor RS485-serial conectado a um Orange Pi Plus 2e com um script para realizar a leitura e envios de dados para a serial. A Figura 19 apresenta tal conexão.

Figura 19: Montagem para o teste básico de funcionamento do sistema.

A primeira implementação da classe de controle do CI para o teste de funcionamento da comunicação pode ser vista no Algoritmo 6.

Algoritmo 6: Código de teste para o funcionamento da comunicação

```
1
2  class SerialRS485 : public UART {
3
4    public:
5
6      //PC5 - RS485 RE - Receiving Enable
7      GPIO * nRE;
8            //PC6 - RS485 DE - Driver
                 Enable
9      GPIO * DE;
10
11
12     SerialRS485(unsigned int baud_rate,
           unsigned int data_bits,
         unsigned int parity, unsigned
         int stop_bits)
13     : UART(1, baud_rate, data_bits,
         parity, stop_bits)
14     {
```

```
15          nRE = new GPIO('C',5, GPIO::OUT);
16          DE = new  GPIO('C',6, GPIO::OUT);
17      }
18
19      void writeWord(char i){
20          sendingState();
21          put(i);
22      }
23
24      int readWord(){
25          receivingState();
26          return get();
27      }
28
29    private:
30
31      void sendingState(){
32          DE->set(1);
33      }
34
35      void shutdownState(){
36          DE->set(0);
37          nRE->set(1);
38      }
39
40      void receivingState(){
41          DE->set(0);
42          nRE->set(0);
43      }
44  };
45
46  int main()
47  {
48    SerialRS485 r(9600, 8, UART_Common::
          NONE, 1);
49    char msg []={"Mensagem de teste\0"};
50      int tam = strlen(msg);
51      r.sendMessage(msg, tam);
52      return 0;
53  }
```

Após a comprovação do funcionamento da comunicação iniciou-se a implementação/melhoria da classe baseada em testes. Observando esta primeira implementação, pode-se fazer as seguinte pergunta:
"O objeto da classe efetuará as chamadas necessária às classes instanciadas internamente durante sua execução?"

Através desta questão, observa-se a necessidade do teste de software para a classe criada. Com o objetivo de se realizar o teste de unidade para esta classe faz-se necessário a criação de *mocks* para as classes por ela utilizada, permitindo desta forma o seu completo isolamento e teste de unidade. Desta forma, as classes instanciadas dentro da classe SerialRS485 devem ser simuladas, sendo elas: GPIO e UART.

C. Desenvolvimento do Driver Baseado em Teste

De acordo com o livro de receitas da biblioteca googlemock (CREATING...,), a escolha entre as classes para ambientes de testes ou de produção deve ser feita em tempo de compilação. Para a realização de testes em classes que utilizam outras classes na sua contrução interna a solução apresentada consistem em "templatizar"esta classe, permitindo a escolha da classe interna como sendo a classe de simulação ou de produção.

Desta forma, para a implementação dos *mocks* das classes GPIO e UART, a classe SerialRS485 deve ser uma classe metaprogramada que permita a escolha dos *mocks* no tempo de compilação. O Algoritmo 7 apresenta tal modificação.

Algoritmo 7: "Templatização"do da classe SerialRS485

```
1  template <class GPIOClass, class
       UARTClass>
2  class SerialRS485 {
3    public:
4      //PC5 - RS485 RE - Receiving Enable
5      GPIO * nRE;
6      //PC6 - RS485 DE - Driver Enable
7      GPIO * DE;
8      UART * uart;
9      GPIOClass * gpioCreator;
10     UARTClass * uartCreator;
11
12     SerialRS485(unsigned int baud_rate,
           unsigned int data_bits,
           unsigned int parity, unsigned
           int stop_bits)
13     {
14       gpioCreator = new GPIOClass();
15       uartCreator = new UARTClass();
16
17       nRE    = gpioCreator->newGPIO('C'
             ,5, GPIOClass::OUT);
18       DE     = gpioCreator->newGPIO('C'
             ,6, GPIOClass::OUT);
19       uart   = uartCreator->newUART(1,
             baud_rate, data_bits, parity
             , stop_bits);
20       shutdownState();
21     }
22 }
```

Observa-se a adição do método newGPIO à classe GPIOClass permitindo a melhor configuração da classe de *mock*. Isto permite que objetos conhecidos sejam retornados a cada chamada do método, permitindo maior controle sobre o teste. Por exemplo, o Algoritmo retorna diferentes ponteiros para cada chamada e faz o teste para o número de vezes que o método newGPIO é chamado.

Algoritmo 8: Retorno de ponteiros de objetos conhecidos.

```
1  EXPECT_CALL(*this, newGPIO(_,_,_))
2          .Times(2)
3          .WillOnce(ReturnPointee(&nRE))
4          .WillOnce(ReturnPointee(&DE));
5  }
```

Os detalhes da implementação das classes de *mock* para as classes GPIO e UART são apresentadas abaixo.

VIII. Mock da classe GPIO

Estas classes foram criadas especicamente para os testes de unidade da classe SerialRS485. Primeiramente, criou-se uma classe de *mock* para a classe GPIO, emulando o construtor e os métodos *get* e

set, utilizados na configuração de estados do CI de comunicação.

Algoritmo 9: Mock da classe GPIO

```
1  class GPIO {
2   public:
3
4    int counter;
5
6    GPIO() {}
7    GPIO(char port, int pin, int inout)
         {}
8
9    MOCK_CONST_METHOD1(set,       void(bool
         bit));
10   MOCK_CONST_METHOD0(get,       int());
11 };
12 }
```

O *mock* da classe auxiliar para a criação de GPIOs é apresentada no Algoritmo 10. Dois objetos da classe GPIO são instanciados dentro da classe SerialRS485 e portanto a classe de *mock* instancia na sua construção dois objetos GPIO, permitindo a verificação da chamada dos métodos de cada um dos objetos separadamente. Além disso a classe recebe dois agumentos de template inteiros na sua criação que são utilizados para definir os métodos EXPECT_CALL dos pinos GPIO criados. O Algoritmo 10 apresenta a implementação desta classe, onde a linha contendo "..."representa a definição das expectativas de chamadas dos métodos *set* e *get* dos GPIOs nRE e DE que serão apresentados posteriormente.

Algoritmo 10: Mock da classe GPIOCreator

```
1  template <int nWrites=0, int nReads=0>
2  class MockGPIOCreator {
3    public:
4      enum { OUT = 0, IN = 1 };
5      GPIO * nRE;
6      GPIO * DE;
7
8      MockGPIOCreator() {
9        nRE = new GPIO();
10       DE = new GPIO();
11
12         ...
13
14       /*
15        * Espera apenas duas chamadas e
              retorna
16        * os enderecos de nRE e DE,
              nesta ordem.
17        */
18       EXPECT_CALL(*this, newGPIO(_,_,_)
             )
19         .Times(2)
20         .WillOnce(ReturnPointee(&nRE))
21         .WillOnce(ReturnPointee(&DE));
22       }
23
24     MOCK_CONST_METHOD3(newGPIO,    GPIO
         * (char A, int B, int C));
25 };
26 }
```

Baseado no número de escritas e leituras executadas pelo objeto da classe SerialRS485, pode-se saber o número exato de chamadas para os métodos *get* e *set* dos objetos GPIO instanciados, sendo eles:

A. *Pino nRE*

set(1): O método *set* com parâmetro 1 deve ser chamado apenas duas vezes durante a aplcação: no construtor e destrutor da classe SerialRS485, na qual o CI deve ser desligado.

Algoritmo 11: EXPECT_CALL para o método set(1), pino nRE, da classe UART

```
1        /*
2         * construtor -> shutdown state
3         * destructor -> shutdown state
4         */
5        EXPECT_CALL(*nRE, set(1))
6            .Times(1+1);
```

set(0): O método *set* com parâmetro 0 é chamado a cada leitura, para a *nReads* leituras executadas.

Algoritmo 12: EXPECT_CALL para o método set(0), pino nRE, da classe UART

```
1        /*
2     * readWord -> nReads vezes
3     */
4     EXPECT_CALL(*nRE, set(0))
5       .Times(nReads);
```

B. *Pino DE*

set(0): O método *set* com parâmetro 0 é chamado para o pino DE nas transições para os estados de desligado e de recebimento de dados. Este método é chamado uma vez no construtor, uma vez no destrutor e uma vez para cada palavra lida pelo driver.

Algoritmo 13: EXPECT_CALL para o método set(0), pino DE, da classe UART

```
1        /*
2     * constructor  -> shutdown state
3     * receiving    -> nReads vezes
4     * destructor   -> shutdown state
5     */
6     EXPECT_CALL(*DE, set(0))
7       .Times(nReads+1+1);
```

set(1):O método *set* com parâmetro 1 para o pino DE é chamado a cada envio de palavra pela UART. Portanto o número de vezes esperado é dado por *nWrites*.

Algoritmo 14: EXPECT_CALL para o método set(1), pino DE, da classe UART

```
1        /*
2     * writeWord -> nWrites vezes
3     */
4     EXPECT_CALL(*DE, set(1))
5       .Times(nWrites);
```

O arquivo completo pode ser visto no Anexo G.

IX. Mock da classe UART

Estas classes também foram criadas especicamente para os testes de unidade da classe SerialRS485. Primeiramente, criou-se uma classe de *mock* para a classe UART, emulando o construtor e os métodos *get* e *put*, utilizados na construção da classe de comunicação.

Algoritmo 15: Mock da classe UART

```
1  class UART {
2   public:
3    UART() {}
4    UART(unsigned int unit, unsigned int
         baud_rate, unsigned int
         data_bits, unsigned int parity,
         unsigned int stop_bits){}
5
6    MOCK_CONST_METHOD1(put,      void(int
         word));
7    MOCK_CONST_METHOD0(get,      int());
8
9  };
```

O Algoritmo 16 apresenta o *mock* criado para a classe de criação de objetos da classe UARTCreator que é utilizada na criação de um objeto UART utilizado na comunicação. Observa-se a alocação de um objeto do tipo UART nesta classe para o posterior controle da chamada de cada método utilzado nas trocas de mensagens. Os três pontinhos da linha 16 representam os métodos EXPECT_CALL para cada um dos métodos da clases UART utilizado e são apresentados nos Algoritmos 17, 18 e 19.

Algoritmo 16: Mock da classe UARTCreator

```
1  /*
2   * Passa como parametro template o
        valor 'n'
3   * que refere-se ao numero de vezes que
        o metodo
4   * put sera chamado.
5   */
6  template <int nWrites=0, int nReads=0>
7  class MockUARTCreator {
8   public:
9    enum {OUT = 0, IN = 1};
10   UART * uart1;
11   MockUARTCreator(){
12
13     uart1 = new UART();
14     ...
15   }
16
17   MOCK_CONST_METHOD5(newUART, UART * (
         unsigned int unit, unsigned int
         baud_rate, unsigned int
         data_bits, unsigned int parity,
         unsigned int stop_bits));
18 };
```

O Algoritmo 17 apresenta a expectativa para o método *newUART*. Observa-se que o método deve ser chamado apenas uma vez, na criação do objeto UART dentro da classe SerialRS485, retornando o endereço do objeto uart1 previamente criado.

Algoritmo 17: EXPECT_CALL para o método newUART

```
1    EXPECT_CALL(*this, newUART(_,_,_,_,
         _))
2      .Times(1)
3      .WillOnce(ReturnPointee(&uart1));
```

O Algoritmo 18 define a expectativa para o método *put* da classe UART na qual espera-se que o método seja chamado *nWrites* vezes.

Algoritmo 18: EXPECT_CALL para o método put

```
1    EXPECT_CALL(*uart1, put(_))
2      .Times(nWrites);
```

O último teste, apresentado no Algoritmo 19, foi criado para o método *get* e implementado a partir da criação de uma expectativa para cada leitura efetuadas pelo teste implementado que é passada através do argumento *nReads* do template. Através da implementação utilizando o laço *for*, pode-se definir valores de retorno conhecidos que permitam a verificação do correto número de leituras da classe SerialRS485.

A ordem de retorno de valores desta expectativa é dada como: 0, 1, 2, 3, ..., *nReads*. Isto occore pois, de acordo com o laço for implementado, a última expectativa atribui como retorno o valor 0 através do comando *.WillOnce(Return(–i))*. O comando *.RetiresOnSaturation()* faz com que cada expectativa seja chamada apenas uma vez, dado que uma expectativa é definida para cada leitura a ser feita.

Algoritmo 19: EXPECT_CALL para o método get.

```
1    /*
2     * Define uma expectiva por leitura
          a ser efetuada
3     * desta forma permite que se
          definam 0 leituras
4     * para testes que usam apenas
          escrita.
5     * A ultima expectativa criada eh a
          primeira atendida,
6     * desta forma, a orderm de retorno
          sera: 0, 1, 2, ... nReads-1.
7     */
8    for (int i = nReads; i > 0 ; ) {
9      EXPECT_CALL(*uart1, get())
10       .WillOnce(Return(--i))
11       .RetiresOnSaturation();
12   }
```

O arquivo completo pode ser visto no Anexo H.

X. Testes da classe de comunicação

Devido à criação das classes de *mock* com argumentos template, o número de chamadas dos métodos *get* e *set* da classe GPIO e dos métodos *get* e *put* da classe UART já são automaticamente testados. Cada teste consiste basicamente da definição do número de leituras e escritas através das constantes *nWrites* e *nReads* e da criação da instanciação de um objeto da classe SerialRS485 passando como argumentos template as classes de *mock* que simulam as classes GPIO e UART que

são as classes *MockGPIOCreator* e *MockUARTCreator*, respectivamente. O Algoritmo 20 apresenta o inicío comum para os testes da classe SerialRS485.

Algoritmo 20: Criação de objeto SerialRS485 para teste utilizando classes de *mock*

```
1 TEST(SerialRS485Test,
      testWriteWordOverfolow){
2   const int nWrites=10;
3   const int nReads=0;
4   SerialRS485 < MockGPIOCreator<nWrites
        , nReads>, MockUARTCreator<
        nWrites, nReads> > r(9600, 8,
        UART_Common::NONE, 1);
5
6     ...
7 }
```

Os testes implementados são descritos a seguir e o Apêndice J apresenta os códigos para estes testes. Devido à implementação das classes de *mock* definirem expectativas para a verificação do acesso aos métodos das classes GPIO e UART, não se faz necessária a implementação desta etapa nos testes aqui apresentados. Em outras palavras, a simples criação de um objeto *mock* destas classes com os corretos valores passados como argumento de template já definem os testes de acesso ao hardware. Consequentemente, faz-se necessário apenas a implementação dos testes referentes ao comportamento dos métodos da classe SerialRS485.

1) Escrita de palavras (writeWord):: Este teste realiza a escrita de 100 caracteres esperando que os testes de acesso aos pinos GPIO e métodos da UART sejam corretamente chamados, conforme definido no construtor do *mock* de cada uma destas classes. O Algoritmo 21 apresenta tal teste.

Algoritmo 21: Teste do método *writeWord* da classe SerialRS485

```
1 TEST(SerialRS485Test, testWriteWord){
2
3   const int nWrites=100;
4   const int nReads=0;
5   SerialRS485 < MockGPIOCreator<nWrites
        , nReads>, MockUARTCreator<
        nWrites, nReads> > r(9600, 8,
        UART_Common::NONE, 1);
6
7     for(int i=0;i<nWrites;i++)
8     r.writeWord(i);
9 }
```

2) Leitura de palavras (readWord):: Da mesma forma que o teste anterior, este teste deixa o controle de acesso ao hardware para as expectativas criadas nos contrutores das classes de *mock*. Adicionalmente cria-se expectativas para os retornos do método *readWord* dados que seus valores de retorno são bem conhecidos, conforme implementado no construtor da classe *MockUARTCreator*.

Algoritmo 22: Teste do método *readWord* da classe SerialRS485

```
1 /*
2  * Testa a leitura de uma sequencia de
       caracteres
3  * bem definido. Verificando o valor
       recebido a cada
4  * chamada do metodo readWord()
5  */
6
7 TEST(SerialRS485Test, testReadWord){
8   const int nWrites=0;
9   const int nReads=100;
10  SerialRS485 < MockGPIOCreator<nWrites
        , nReads>, MockUARTCreator<
        nWrites, nReads> > r(9600, 8,
        UART_Common::NONE, 1);
11
12  char leitura;
13    for(char i=0;i<nReads;i++){
14    leitura = r.readWord();
15    EXPECT_EQ(i, leitura);
16  }
17  cout<<endl;
18
19  /*
20   * Esta implementacao se faz possivel
         pois a classe MockUARTCreator
21   * retorna valores de 0 a nReads (
         passado como argumetno template
         ) sequenciamente.
22   */
23 }
```

O resultado dos testes é apresentado na Figura 20 onde pode-se observar que a classe foi aprovada.

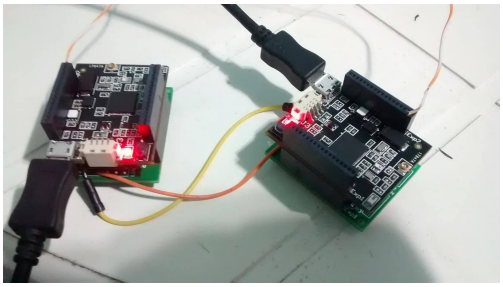
Figura 20: Resultado dos testes da classe SerialRS485.

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from SerialRS485Test
[ RUN      ] SerialRS485Test.testWriteWord
[       OK ] SerialRS485Test.testWriteWord (0 ms)
[ RUN      ] SerialRS485Test.testReadWord

[       OK ] SerialRS485Test.testReadWord (2 ms)
[----------] 2 tests from SerialRS485Test (2 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (2 ms total)
[  PASSED  ] 2 tests.
```

XI. Teste de comunicação entre EPOSs

Um teste de troca de mensagems foi realizado entre dois dispositivos EPOS MOTE III (Figura 21) com algoritmos utilizando diferentes versões de drivers para o CI ISL83485.

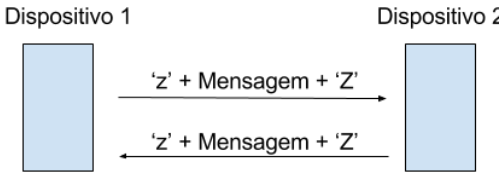
Figura 21: Ligação entre diferentes dispositivos.



Para esta troca de mensagens, criou-se um esquema simples para a sincronização entre as aplicações de troca de mensagem. Chamado de protocolo

'zZ' (zezinho zezão), este esquema consiste da adição da letra Z minúscula no início das mensagems e maiúscula ao seu final. A Figura 22 apresenta uma simples troca de mensagens para este esquema que possui como desvantagem a impossibilidade do envio dos caracteres 'z' e 'Z', mas é eficaz na demonstração do funcionamento.

Figura 22: Troca de mensagems pelo protocolo 'zZ'.



As Figuras 23 e 24 apresentam as saídas de ambos os algoritmos na troca de mensagens entre os dispositivos.

Figura 23: Dispositivo 1 enviando mensagem e recebendo resposta.

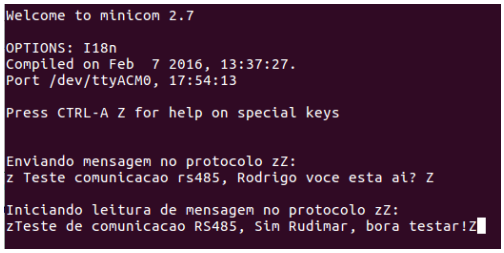
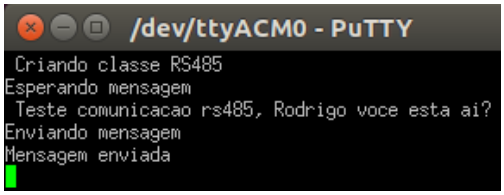


Figura 24: Dispositivo 2 recebendo mensagem e enviando resposta.



O algoritmo utilizado nesta comunicação pode ser visto no Apêndice K.

XII. Conclusões

A utilização de classes de teste no desenvolvimento de softwares permite a organização do fluxo de trabalho e a verificação de não conformidades no código.

A utilização da biblioteca *googletest* permitiu a verificação e correção do componente Ordered_Queue das bibliotecas disponíveis no sistema EPOS. Já a utilização da biblioteca *googlemock* permitiu o desenvolvimento da classe SerialRS485 na plataforma Linux sem a necessidade da utilização dos hardwares para o qual o *driver* foi implementado. A utilização dos *mocks* também permitiu a identificação de não conformidades no desenvolvimento da classe implementada. Por exemplo, após a definição das classes de *mock*, a primeira execução do teste apresentou tal aviso:
"*rs485test.cc:41: ERROR: this mock object (used in test SerialTest.newGPIO) should be deleted but never is. Its address is @0x1f6f360.*"
indicando a falta do comando *delete* para os objetos de GPIO instanciados no construtor da classe Serial485.
O mesmo ocorrou para o objeto UART criado a partir do *mock*:
"*mockUART.h:65: ERROR: this mock object (used in test SerialTest.testWriteWord) should be deleted but never is. Its address is @0x1159a50.*"

Estas mensagens geradas pelos testes permitem que o desenvolvedor faça correções no seu código e desta forma evitando o vazamento de memória.

Por fim, a classe mediadora do hardware foi testada em comunicação com outro dispositivo e mostrou-se funcional. Os códigos das classes GPIOCreator, UARTCreator e SerialRS485 são apresentados no Apêndice I.

Referências

CREATING Mock Classes. <https://github.com/google/googlemock/blob/master/googlemock/docs/v1_6/CookBook.md>. (Accessed on 06/10/2017).
EMBEDDED Parallel Operating System | EPOSMote III. LISHA, 2017. (Accessed on 03/25/2017). Disponível em: <http://epos.lisha.ufsc.br/EPOSMote+III>.
RS232 to RS485 Converter and Adapter Wiki. Magneto Tech, 2011. (Accessed on 04/16/2017). Disponível em: <http://www.rs232-to-rs485.com/>.
RS485, RS232, RS422, RS423, Quick Reference Guide. Re Smith, 2017. (Accessed on 04/21/2017). Disponível em: <http://www.rs485.com/rs485spec.html>.
SERIAL Communication - learn.sparkfun.com. Sparkfun, 2017. (Accessed on 04/17/2017). Disponível em: <https://learn.sparkfun.com/tutorials/serial-communication/rules-of-serial>.

APÊNDICE A
CODIGO PROMELA PARA VERIFICAÇÃO DO MODELO INICIAL

```
mtype = {m, serialStop, serialStart, serialSend, serialReceive, baudrate, nbits,
    stopbits};




chan sline = [0] of {mtype};
chan cmd  = [0] of {mtype};

active proctype serialDriver(){

  goto SERIAL_OFF;

SERIAL_OFF:
  do
  :: cmd?serialStart -> goto SERIAL_IDLE
  :: cmd?serialStop  -> goto SERIAL_OFF
  od;

SERIAL_IDLE:
      do
  ::  if
    :: cmd?serialSend    -> goto SERIAL_SENDING
    :: cmd?serialReceive -> goto SERIAL_RECEIVING
    :: cmd?serialStop    -> goto SERIAL_OFF
    :: skip
    fi;
  od;

SERIAL_RECEIVING:


  if
  ::sline?m -> goto SERIAL_IDLE
  ::timeout -> goto SERIAL_IDLE
  fi;

SERIAL_SENDING:
  if
  ::sline!m -> goto SERIAL_IDLE
  fi;
}

active proctype slave(){
  do
  :: if
     :: sline?m -> sline!m
     :: sline?m
     fi;
  od;
}

active proctype application(){
  cmd!serialStart
  do
  ::cmd!serialSend
    cmd!serialReceive
  od;
  cmd!serialStop
}
```

APÊNDICE B
CODIGO PROMELA PARA VERIFICAÇÃO DO MODELO COMPLETO

```
mtype = {m, serialStop, serialStart, serialSend, serialReceive, baudrate, nbits,
    stopbits, odd, even, none, startBit, stopBit, high, low, outputMode, inputMode,
    error, success};

#define MAX_DEVS 32

typedef config {
  int _baudrate;
```

```
7   short _nbits;
8   short _stopbits;
9   mtype _parity;
10 };
11
12
13 chan sline [MAX_DEVS] = [0] of {mtype};
14 chan cline = [0] of {mtype};
15 chan cmd    = [0] of {mtype};
16
17 bool nRE, DE;
18 int ndevs = 1;
19 int novoid = 0;
20 config c;
21
22
23
24 inline sendParityBit()
25 {
26   if
27   ::c._parity == odd ->
28     if
29     ::n1s%2==1 ->
30       cline!low
31       nsent=nsent+1
32     ::else ->
33       cline!high
34       nsent=nsent+1
35       n1s=n1s+1
36     fi
37   ::c._parity == even ->
38     if
39     ::n1s%2==1 ->
40       cline!high
41       n1s=n1s+1
42     ::else ->
43       cline!low
44       nsent=nsent+1
45     fi
46   ::c._parity == none ->
47     skip
48   fi
49 }
50
51 inline checkParityBit()
52 {
53   if
54   ::c._parity == odd ->
55     if
56     ::n1s%2==1 -> cmd!success //success
57     ::else -> cmd!error       //error
58     fi
59   ::c._parity == even ->
60     if
61     ::n1s%2==1 -> cmd!error//error
62     ::else -> cmd!success
63     fi
64   ::c._parity == none ->
65     cmd!success
66   fi
67 }
68
69 inline setInputMode(){
70   atomic{
71     DE  = 0
72     nRE = 0
73   }
74 }
75
76 inline setOutputMode(){
77   atomic{
78     DE = 1
79   }
80 }
81
82 inline shutDown(){
83   atomic{
```

```
84      nRE = 1
85      DE  = 0
86    }
87  }
88
89  inline receiveWord(){
90    nrecv=0;
91    n1s=0;
92    do
93    ::(c._parity!=none && nrecv != c._nbits + 1) ->
94      if
95      :: cline?high ->
96      nrecv=nrecv+1
97      n1s=n1s+1
98      :: cline?low ->
99        nrecv=nrecv+1;
100           :: timeout ->
101       shutDown();
102       cmd!error
103       break
104     fi;
105   ::(c._parity==none && nrecv != c._nbits) ->
106     if
107     :: cline?high ->
108       nrecv=nrecv+1
109       n1s=n1s+1
110     :: cline?low ->
111       nrecv=nrecv+1;
112           :: timeout ->
113       shutDown();
114       cmd!error
115       break
116     fi;
117   :: else ->
118
119     if
120     ::c._stopbits==2 ->
121       if
122       ::cline?high -> skip //stopBit1
123       ::timeout ->
124         shutDown();
125         cmd!error
126         break
127       fi
128
129       if
130       ::cline?high -> skip //stopBit2
131       ::timeout ->
132         shutDown()
133         cmd!error
134         break
135       fi
136     ::else ->
137       if
138       ::cline?high -> skip //stopBit
139       ::timeout ->
140         cmd!error
141         shutDown();
142         break
143       fi
144     fi
145
146     shutDown();
147     checkParityBit(); //sends success
148     break
149   od;
150 }
151
152
153
154
155 active proctype serialDriver(){
156
157   shutDown();
158
159   int nrecv = 0
160   int n1s = 0
```

```
161   int nsent = 0
162
163
164 //baudrate configured
165   if
166   :: c._baudrate  = 57600;
167   :: c._baudrate  = 38400;
168   :: c._baudrate  = 19200;
169   :: c._baudrate  = 9600;
170   :: c._baudrate  = 4800;
171   fi
172 //nbis defined
173   if
174   :: c._nbits   = 8;
175   :: c._nbits   = 7;
176   :: c._nbits   = 6;
177   :: c._nbits   = 5;
178   fi
179 //stop bits defined
180   if
181   :: c._stopbits  = 2;
182   :: c._stopbits  = 1;
183   fi
184 //parity defined
185   if
186   :: c._parity  = odd;
187   :: c._parity  = even;
188   //:: c._parity  = none;
189   :: c._parity  = none;
190   fi
191
192 SERIAL_OFF:
193   shutDown()
194   do
195   :: cmd?serialStart -> goto SERIAL_IDLE
196   :: cmd?serialStop  -> goto SERIAL_OFF
197   od;
198
199 SERIAL_IDLE:
200   shutDown()
201       do
202 //   ::   if
203     :: cmd?serialSend     ->
204
205     SERIAL_SENDING:
206
207       //Sending data
208       setOutputMode();
209       if
210       ::cline!low -> skip //startBit
211       ::timeout ->
212         cmd!error;
213         goto SERIAL_IDLE
214       fi
215       nsent=0;
216       n1s=0;
217       //enqto n enviou a msg
218       do
219         :: nsent == c._nbits -> break
220         :: else ->
221             if
222             :: cline!high ->
223           nsent=nsent+1
224           n1s=n1s+1
225             :: cline!low ->
226           nsent=nsent+1
227           fi;
228       od;
229
230       if
231       ::c._stopbits==2 ->
232         cline!high //stopBit1
233         cline!high //stopBit2
234       ::else ->
235         cline!high //stopBit
236       fi
237
```

```
238
239         sendParityBit()
240         cmd!success
241         goto SERIAL_IDLE
242
243       :: cmd?serialReceive ->
244
245       SERIAL_RECEIVING:
246         setInputMode();
247         if
248         ::cline?low; //start bit
249         ::timeout->
250           cmd!error
251           goto SERIAL_IDLE
252
253
254         fi
255         receiveWord();
256       :: cmd?serialStop     ->
257         goto SERIAL_OFF
258
259   //      fi;
260   od;
261 }
262
263 proctype receivingSlave(){
264
265   int nrecv = 0
266   int n1s = 0
267   int id;
268   atomic{
269     ndevs  = ndevs + 1
270     id = ndevs
271     printf("\nExistem %d dispositivos\n", ndevs);
272   }
273
274   do
275   :: sline[id]?low -> skip
276   :: sline[id]?high -> skip
277   od
278
279 }
280
281 inline sendParityBitSlave()
282 {
283   if
284   ::c._parity == odd ->
285     if
286     ::n1s%2==1 ->
287       if
288       ::sline[1]!low->
289         nsent=nsent+1
290         break
291       ::timeout -> break
292       fi
293
294     ::else ->
295       if
296       ::sline[1]!high->
297         nsent=nsent+1
298         n1s=n1s+1
299         break
300       ::timeout -> break
301       fi
302
303     fi
304   ::c._parity == even ->
305     if
306     ::n1s%2==1 ->
307       if
308       ::sline[1]!high->
309         nsent=nsent+1
310         n1s=n1s+1
311         break
312       ::timeout -> break
313       fi
314     ::else ->
```

```
315         if
316         ::sline[1]!high->
317           nsent=nsent+1
318           break
319         ::timeout -> break
320         fi
321       fi
322     ::c._parity == none ->
323       skip
324     fi
325   }
326
327   proctype sendingSlave(){
328
329     //Slave sending data
330
331     int nsent=0;
332     int n1s=0;
333     if
334     ::sline[1]!low -> //start bit
335       do
336       :: nsent == c._nbits ->
337
338         sendParityBitSlave()
339
340         if
341         ::c._stopbits==2 ->
342           sline[1]!high //stopBit1
343           sline[1]!high //stopBit2
344           break
345         ::c._stopbits==1 ->
346           sline[1]!high //stopBit
347           break
348         ::timeout->break
349         fi
350
351         //sline[1]!high //stop bit
352
353
354       :: nsent != c._nbits ->
355         if
356         :: sline[1]!high ->
357           nsent=nsent+1
358           n1s=n1s+1
359         :: sline[1]!low ->
360           nsent=nsent+1;
361         ::timeout ->  break
362         fi
363       od
364     ::timeout -> skip;
365     fi
366
367   }
368
369
370
371   active proctype ISL83483(){
372
373     bool DI, VCC, B, A, GND, RO;
374     GND = 0;
375     VCC = 1;
376     nRE=1
377     DE=0
378
379     //chipDown:
380
381     //fica aqui ate chip estar ligado
382     //(nRE == 1 && DE == 0)==0
383
384
385     int i;
386
387     do
388     ::(DE==1)-> //Transmiting
389       //ISL83483: Chip in transmiting mode
390       if
391       ::cline?high-> //high, stopbit
```

```
392         DI = 1 //A=1 B=0, high
393         for(i : 2 .. ndevs){
394           sline[i]!high
395         }
396       ::cline?low -> //low, startbit
397         DI = 0 //A=0 B=1, low
398         for(i : 2 .. ndevs){
399           sline[i]!low
400         }
401       ::timeout -> skip
402       fi
403     ::(DE==0 && nRE==0)-> //Receiving
404       //ISL83483: Chip in receiving mode
405       if
406       ::sline[1]?high ->
407         if
408         ::cline!high
409         ::timeout->skip
410         fi
411         RO = 1
412
413       ::sline[1]?low->
414         if
415         ::cline!low
416         ::timeout->skip
417         fi
418         RO = 0
419
420       ::(DE==0 && nRE==0)==0-> //if this is false
421         //printf("no more in receiving mode")
422         skip
423       fi
424     //::(nRE==1 && DE==0)->
425     ::(nRE==1 && DE==0)->
426     //chip esta desligado, qualquer acao nos pinos nao servira para nada
427       if
428       ::(nRE==1 && DE==0)==0->skip
429       ::cline?high  ->skip
430       ::cline?low ->skip
431       ::sline[1]?high ->skip
432       ::sline[1]?low  ->skip
433       //::timeout ->skip
434       fi
435   od
436 }
437
438 active proctype application(){
439
440   run receivingSlave()
441   run receivingSlave()
442
443   cmd!serialStart
444
445   do
446   ::cmd!serialReceive ->
447     run sendingSlave()
448     if
449     ::cmd?error -> skip
450     ::cmd?success -> skip
451     //::timeout -> break
452     fi
453   ::cmd!serialSend ->
454     if
455     ::cmd?error -> skip
456     ::cmd?success -> skip
457     //::timeout -> break
458     fi
459   ::timeout->skip
460   od
461   //cmd!serialStop
462 }
```

APÊNDICE C
DIAGRAMAS DE ESTADOS

Figura 25: Diagrama de estados do funcionamento do driver serial completo gerado pelo software Spin

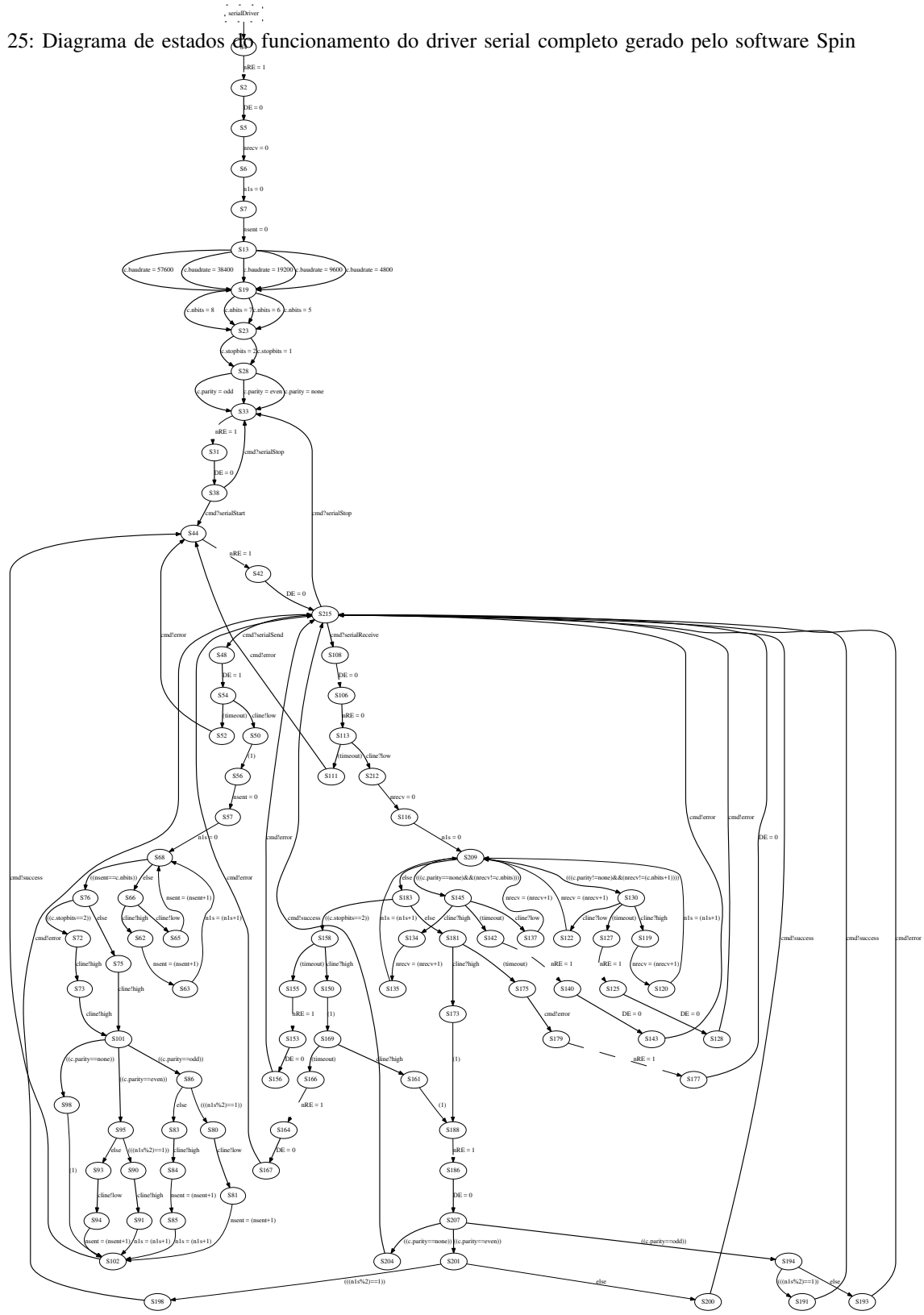


Figura 26: Diagrama de estados do funcionamento do CI de comunicação gerado pelo software Spin

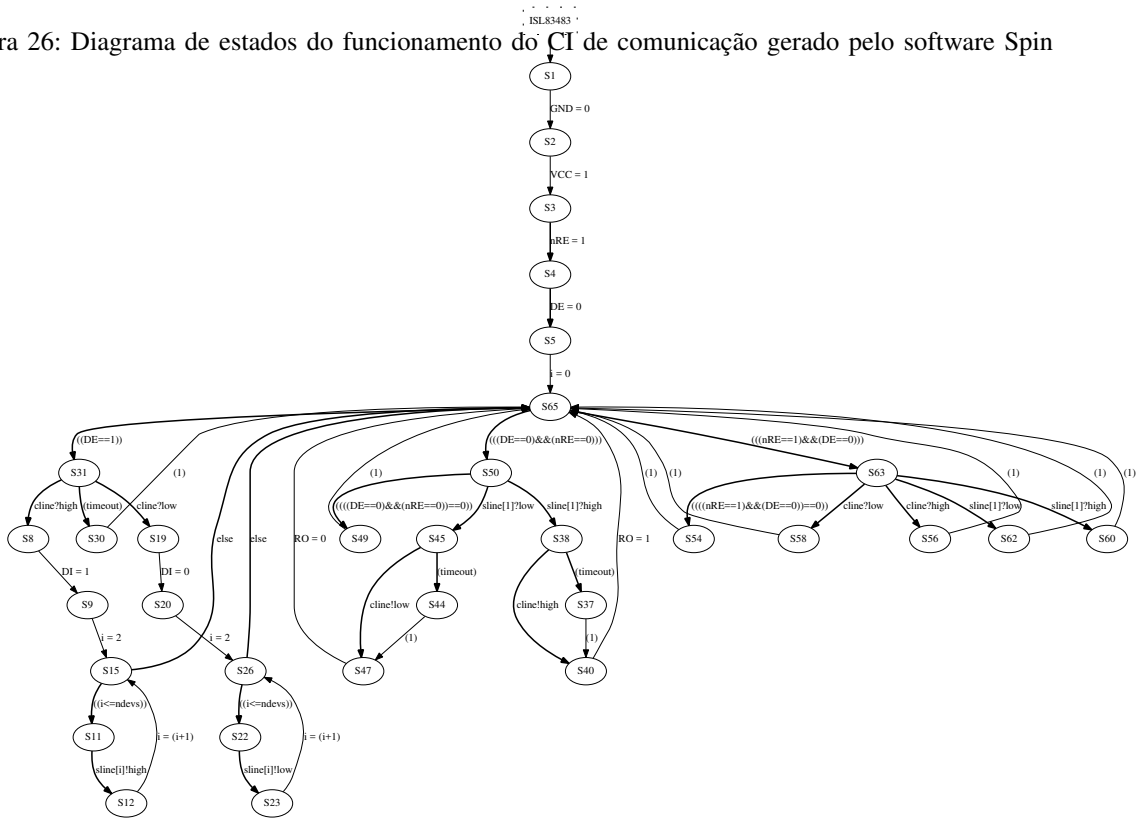
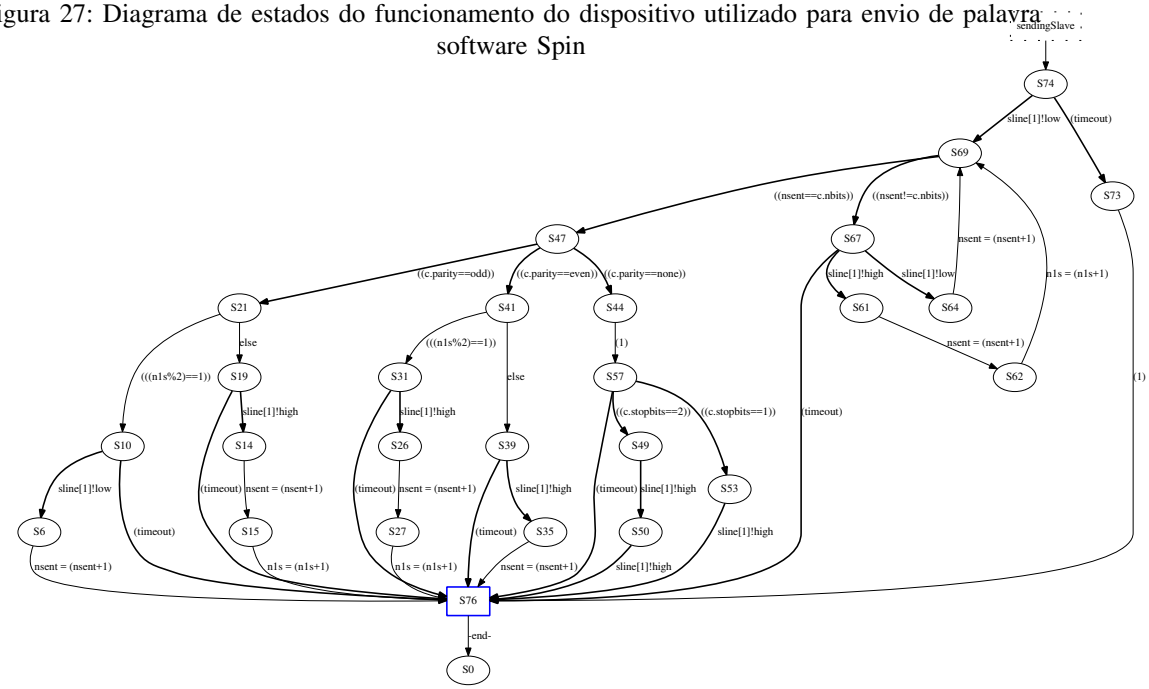


Figura 27: Diagrama de estados do funcionamento do dispositivo utilizado para envio de palavra software Spin



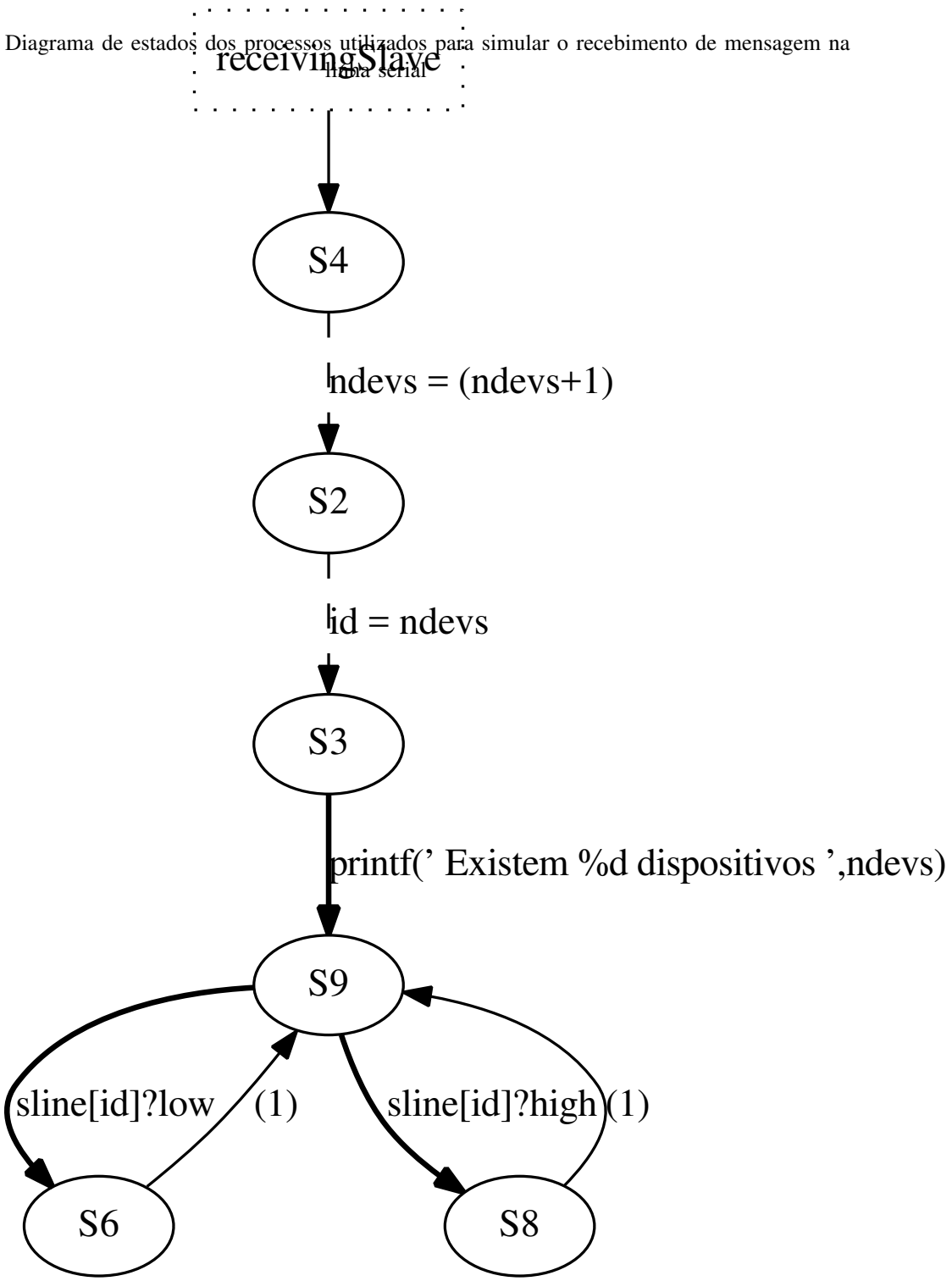


Figura 28: Diagrama de estados dos processos utilizados para simular o recebimento de mensagem na linha serial

APÊNDICE D
ROTINA DE TESTES DA FILA ORDENADA

```
1  // EPOS Queue Utility Test Program
2
3  #include <iostream>
4  #include "queue.h"
5
6  #include "gtest/gtest.h"
7  #include "gmock/gmock.h"
8
9  #define DEBUG 0
10
11 using namespace std;
12
13 struct Inteiro {
14     //Construtor
15     //e._object = this; e.rank = _r;
16     Inteiro(int valor, int rank) :  e(this, rank), i(valor)
17   {}
18
19     int i; //valor do elemento
20     Ordered_Queue<Inteiro>::Element e;
21 };
22
23 TEST(Inteiro, CriacaoDeInteiro) {
24
25   if(DEBUG) cout<<"\n\n------- TESTE DE CRIACAO DE INTEIRO --------"<<endl;
26
27   //Ordered_Queue<Inteiro> q1;
28   Inteiro i1(1,1), i2(-2,2), i3(9999,3);
29
30   EXPECT_EQ(1, i1.i)      << "Elemento nao possui o mesmo  da sua criacao";
31   EXPECT_EQ(1, i1.e.rank())   << "Elemento nao possui o mesmo rank da sua criacao";
32
33   EXPECT_EQ(-2, i2.i)     << "Elemento nao possui o mesmo  da sua criacao";
34   EXPECT_EQ(2, i2.e.rank())   << "Elemento nao possui o mesmo rank da sua criacao";
35
36   EXPECT_EQ(9999, i3.i)     << "Elemento nao possui o mesmo  da sua criacao";
37   EXPECT_EQ(3, i3.e.rank())   << "Elemento nao possui o mesmo rank da sua criacao";
38 }
39
40 /*
41  * Fixture utilizada para todos os testes, cria automaticamente uma fila
42  * ordenada, um vetor de dados, e tambem libera a memoria no final do teste
43  */
44 class Ordered_Queue_Test : public ::testing::Test
45 {
46   protected:
47      Ordered_Queue<Inteiro> q1;
48      Inteiro * ivec[100];
49      Inteiro * removido;
50      int nvals;
51
52      virtual void SetUp()
53      {
54        nvals=0;
55        //q1 = new Ordered_Queue;
56      }
57
58      virtual void TearDown()
59      {
60        if(DEBUG) cout<<"Limpando "<<nvals<<" inteiros"<<endl;
61        for(int i=1;i<nvals;i++)
62         delete ivec[i];
63      }
64
65      /*
66       * Insere nvalues valores com rank sequenciais
67       * Os valores sao aleatorios e estao entre min e max
68       */
69      void FillVectorOrderedRank(int nvalues=100, int min =0, int max = 100){
70
71       nvals=nvalues; //Para propositos de liberacao de memoria
72
73        //Inserindo valores
74       if(DEBUG) cout<<"Inserindo: ";
```

```
75        for(int i=0;i<nvalues;i++){
76          //gerando valores aleatorios
77          int value = rand()%(max-min + 1) + min;
78          if(DEBUG) cout<<endl<<"("<<value<<","<<i<<") ";
79          ivec[i]= new Inteiro(value, i);
80
81          //inserindo na fila
82          q1.insert(&(ivec[i]->e));
83        }
84       }
85
86      /*
87        * Insere nvalues valores com rank aleatorios
88        * e valores sao aleatorios, entre min e max
89        */
90       void FillVectorRandomRank(int nvalues=100, int min =0, int max = 100){
91
92         nvals=nvalues; //Para propositos de liberacao de memoria
93
94         //Inserindo valores
95        if(DEBUG) cout<<"Inserindo: ";
96        for(int i=0;i<nvalues;i++){
97          //gerando valores aleatorios
98          int value = rand()%(max-min + 1) + min;
99          if(DEBUG) cout<<endl<<"("<<value<<","<<value<<") ";
100         ivec[i]= new Inteiro(value, value);
101
102         //inserindo na fila
103         q1.insert(&(ivec[i]->e));
104       }
105      }
106 };
107
108 /*
109  * Neste teste, faz-se a insersao de valores com rank aleatorios
110  * e espera-se que cada numero seja maior ou igual o anterior.
111  */
112 TEST_F(Ordered_Queue_Test, RetornoOrdenado) {
113   if(DEBUG) cout<<"\n\n------- TESTE DE ORDENACAO ---------"<<endl;
114
115   //Inserindo valores
116   FillVectorRandomRank();
117
118   if(DEBUG) cout <<"\nRemovendo: ";
119   int head=0, head_rank=0, last_head=0, last_head_rank=0;
120
121   removido = q1.remove()->object();
122   last_head=removido->i;
123   last_head_rank=removido->e.rank();
124
125   for(int i=1;i<nvals;i++){
126
127     if(DEBUG) cout<<endl<<"("<<last_head<<","<<last_head_rank<<") ";
128
129     //Removendo o primeiro elemento
130     removido = q1.remove()->object();
131     head = removido->i;
132     head_rank = removido->e.rank();
133
134     //Verificando ordem
135     EXPECT_GE(head_rank,  last_head_rank);
136
137     //atualizando valores
138     last_head=head;
139     last_head_rank=head_rank;
140   }
141 }
142
143 /*
144  * Neste teste, faz-se a insersao de valores com ranks conhecidos.
145  * Testa-se se apos a remocao do elemento atraves da funcao search ele foi
146  * de fato removido da fila.
147  */
148 TEST_F(Ordered_Queue_Test, RemocaoElemento) {
149
150   if(DEBUG) cout<<"\n\n------- TESTE DE REMOCAO DE ELEMENTO ---------"<<endl;
151
```

```
152   //Inserindo valroes de rank conhecido
153   FillVectorOrderedRank();
154
155   //Removendo dois valores
156   q1.remove(ivec[10]);
157   q1.remove(ivec[15]);
158
159   //Testando com elemento que ja nao esta na fila
160   EXPECT_FALSE(q1.remove(ivec[15]));
161
162   //Removendo valores da fila, que nao podem ser
163   //iguais aos removidos anteriormente
164   while(!q1.empty()){
165     removido = q1.remove()->object();
166     EXPECT_NE(ivec[10], removido);
167     EXPECT_NE(ivec[15], removido);
168   }
169 }
170
171 /*
172  * Neste teste, faz-se a insersao de valores com diferentes ranks.
173  * Verifica-se se a fila retorna nulo ao se tentar remover algum  quando
174  * a mesma esta vazia.
175  * Tambem vefica-se o metodo empty()
176  */
177 TEST_F(Ordered_Queue_Test, RetornoVazia) {
178
179   if(DEBUG) cout<<"\n\n------- TESTE DE RETORNO DE FILA VAZIA ---------"<<endl;
180
181   //verificando o metodo empty
182   EXPECT_TRUE(q1.empty());
183
184   //tentando remover antes de inserir
185   EXPECT_FALSE(q1.remove());
186
187   Inteiro i1(1,7), i2(4,10), i3(10,5);
188   q1.insert(&i1.e);
189   q1.insert(&i2.e);
190   q1.insert(&i3.e);
191
192   //verificando o metodo empty
193   EXPECT_FALSE(q1.empty());
194
195   q1.remove();
196   q1.remove();
197   q1.remove();
198
199   //tentando remover apos a fila ter sido esvaziada
200   EXPECT_FALSE(q1.remove());
201
202   //verificando o metodo empty
203   EXPECT_TRUE(q1.empty());
204 }
205
206
207
208 /*
209  * Neste teste faz-se a insersao de valores com ranks conhecidos, valores impares.
210  * Testa-se se o metodo search para verificar se todos os valores inseridos sao
211  * encontrados
212  */
213 TEST_F(Ordered_Queue_Test, EncontraElemento) {
214
215   if(DEBUG) cout<<"\n\n------- TESTE DO METODO SEARCH ---------"<<endl;
216
217   int nvalues = 100;
218   int max = 100;
219   int min = 0;
220
221   //Inserindo valores impares
222   if(DEBUG) cout<<"Inserindo ranks impares: ";
223   for(int i=0;i<nvalues;i++){
224     int value = rand()%(max-min + 1) + min;
225     ivec[i]= new Inteiro(value, i);
226     //Apenas insere se for impar
227     if(i%1==0){
228       if(DEBUG) cout<<endl<<"("<<value<<","<<i<<") ";
```

```
229         q1.insert(&(ivec[i]->e));
230       }
231     }
232
233     //Testando o metodo search
234     for(int i=0;i<nvalues;i++){
235       if(i%1==0) //Se or impar, deve estar
236         EXPECT_EQ(ivec[i], q1.search(ivec[i])->object());
237       else //Se for par, nao esta na fila
238         EXPECT_FALSE(q1.search(ivec[i]));
239     }
240 }
241
242 /*
243  * Neste teste faz-se o teste do metodo tail, removendo-se sempre o ultimo
244  * elemento e verificando se o  de tail eh o mesmo que o esperado
245  */
246 TEST_F(Ordered_Queue_Test, TailTest) {
247
248   if(DEBUG) cout<<"\n\n------- TESTE DO METODO TAIL ---------"<<endl;
249
250   //Inserindo valroes de rank conhecido
251   FillVectorOrderedRank();
252
253   //Testando o metodo search
254   for(int i=nvals-1;i>0;i--){
255     EXPECT_EQ(ivec[i], q1.tail()->object());
256     q1.remove(ivec[i]);
257   }
258
259   EXPECT_EQ(q1.head(), q1.tail());
260   q1.remove(ivec[0]);
261
262   EXPECT_FALSE(q1.tail());
263 }
264
265 /*
266  * Neste teste faz-se o teste do metodo head, removendo-se sempre o primeiro
267  * elemento e verificando se o  de tail eh o mesmo que o esperado
268  */
269 TEST_F(Ordered_Queue_Test, HeadTest) {
270
271   if(DEBUG) cout<<"\n\n------- TESTE DO METODO TAIL ---------"<<endl;
272
273   //Inserindo valroes de rank conhecido
274   FillVectorOrderedRank();
275
276   //Testando o metodo search
277   for(int i=0;i<nvals;i++){
278     EXPECT_EQ(ivec[i], q1.head()->object());
279     q1.remove(ivec[i]);
280   }
281
282   EXPECT_FALSE(q1.head());
283 }
284
285 /*
286  * Neste teste faz-se o teste do metodo size. Insere-se um numero conhecido de
        valores
287  * e verifica-se o tamanho apos a remocao de cada elemento.
288  */
289 TEST_F(Ordered_Queue_Test, SizeTest) {
290
291   if(DEBUG) cout<<"\n\n------- TESTE DO METODO SIZE ---------"<<endl;
292
293   //Inserindo valroes de rank conhecido
294   FillVectorOrderedRank();
295
296   //Testando o metodo search
297   for(int i=nvals;i>0;i--){
298     EXPECT_EQ(i, q1.size());
299     q1.remove();
300   }
301
302   EXPECT_FALSE(q1.size());
303 }
304
```

```
305 TEST_F(Ordered_Queue_Test, InsersaoElementoNulo) {
306   if(DEBUG) cout<<"\n\n------- TESTE DE INSERSAO DE ELEMENTO NULO ---------"<<endl;
307   q1.insert(NULL);
308 
309   EXPECT_EQ(0, q1.size());
310   EXPECT_EQ(1, q1.empty());
311   EXPECT_FALSE(q1.tail());
312   EXPECT_FALSE(q1.head());
313 
314   Inteiro i1(1,7);
315   q1.insert(&i1.e);
316 
317   q1.insert(NULL);
318 
319   EXPECT_EQ(1, q1.size());
320   EXPECT_EQ(q1.head(), q1.tail());
321 }
322 
323 
324 int main( int argc, char *argv[] ) {
325     ::testing::InitGoogleMock( &argc, argv );
326     return RUN_ALL_TESTS( );
327 }
```

APÊNDICE E

TESTE DA CLASSE GPIO, CONSISTINDO DE UMA VERSÃO DO ALGORITMO 'PISCA LED' PARA OS GPIOS UTILIZADOS NESTE TRABALHO.

```
1  #include <gpio.h>
2  #include <alarm.h>
3  #include <utility/ostream.h>
4
5  using namespace EPOS;
6
7  OStream cout;
8
9  void delay(int max){
10   for(volatile int t=0;t<max;t++);
11 }
12
13 int main()
14 {
15     GPIO uart1rx('C',3, GPIO::OUT);
16     GPIO uart1tx('C',4, GPIO::OUT);
17     GPIO RE('C',5, GPIO::OUT);
18     GPIO DE('C',6, GPIO::OUT);
19
20     GPIO uart0tx('A',1, GPIO::OUT);
21     GPIO uart0rx('A',0, GPIO::OUT);
22
23     int max = 0x2ffff;
24
25     for(bool b=false;;b=(b+1)%2)
26     {
27         cout << "teste\n";
28         uart1rx.set(b);    delay(max);
29         uart1rx.set(!b);
30
31         uart1tx.set(b);    delay(max);
32         uart1tx.set(!b);
33
34         RE.set(b);   delay(max);
35         RE.set(!b);
36
37         DE.set(b);   delay(max);
38         DE.set(!b);
39
40         uart0rx.set(b);    delay(max);
41         uart0rx.set(!b);
42
43         uart0tx.set(b);    delay(max);
44         uart0tx.set(!b);
45
46         max = max-10000;
47
48         if(max<=30000)
49       max = 0x2ffff;
50
51     }
52     return 0;
53 }
```

APÊNDICE F
TESTE DE COMUNICAÇÃO ENTRE UARTS

```
#include <iostream>
int main()
{
#include <gpio.h>
#include <alarm.h>
#include <utility/ostream.h>
#include <uart.h>

using namespace EPOS;
OStream cout;


void sleep(int n){
  for(int i=0;i<n;i++){
    for(volatile int t=0;t<0xfffff;t++);
  }
}


int uartSend(int uart)
{
    UART r(uart, 9600, 8, UART_Common::NONE, 1);
    int i=0;
    while(1){
    sleep(1);
    cout<<"Escrevendo na uart "<<uart<<": "<<i<<endl;
    r.put(i++);
  }
    return 1;
}

int uartReceive(int uart)
{
    UART s(uart, 9600, 8, UART_Common::NONE, 1);
    while(1){
    int val = s.get();
    cout << "Lendo na uart "<<uart<<": " << val << "." <<endl;
  }
    return 1;
}


Thread * uarts[2];

int main()
{
  cout<<"Criando uarts"<<endl;
  uarts[0] = new Thread(&uartSend, 1);
  uarts[1] = new Thread(&uartReceive, 0);

  cout<<"Join nas uarts"<<endl;
  for(int i = 0; i < 2; i++)
        int ret = uarts[i]->join();

    return 0;
}
}
```

APÊNDICE G
MOCK DA CLASSE GPIO

```
#ifndef MOCKGPIO_H
#define MOCKGPIO_H

using ::testing::Return;
using ::testing::ReturnPointee;
using ::testing::_;
using namespace std;

class GPIO_Common
{
protected:
    GPIO_Common() {}

public:
    enum Level {
        HIGH,
        LOW
    };

    enum Edge {
        RISING,
        FALLING,
        BOTH
    };

    enum Direction {
        IN,
        OUT,
        INOUT
    };

    enum Pull {
        UP,
        DOWN,
        FLOATING
    };
};


/*
 * Mock da classe GPIO para simular os
 * metodos set, get e construtor.
 */
class GPIO {
 public:

  int counter;

  GPIO() {}
  GPIO(char port, int pin, int inout) {}

  MOCK_CONST_METHOD1(set,      void(bool bit));
  MOCK_CONST_METHOD0(get,      int());

};

/*
 * Passa como parametro template os valores 'nWrites' e 'nReads'
 * e com base nestes parametros cria as expectativas para cada
 * chamada dos metodos da GPIO
 */
template <int nWrites=0, int nReads=0>
class MockGPIOCreator {
  public:

    GPIO * nRE;
    GPIO * DE;

    MockGPIOCreator() {

      /*
       * Espera apenas duas chamadas e retorna
       * os enderecos de nRE e DE, nesta ordem.
       */
```

```
75
76          nRE = new GPIO();
77          DE = new GPIO();
78
79          EXPECT_CALL(*this, newGPIO(_,_,_))
80             .Times(2)
81             .WillOnce(ReturnPointee(&nRE))
82             .WillOnce(ReturnPointee(&DE));
83
84             /*
85              * construtor -> shutdown state
86              * destructor -> shutdown state
87              */
88          EXPECT_CALL(*nRE, set(1))
89             .Times(1+1);
90
91          /*
92           * readWord    -> nReads vezes
93           */
94          EXPECT_CALL(*nRE, set(0))
95             .Times(nReads);
96
97          /*
98           * constructor  -> shutdown state
99           * receiving   -> nReads vezes
100          * destructor -> shutdown state
101          */
102         EXPECT_CALL(*DE, set(0))
103            .Times(nReads+1+1);
104
105         /*
106          * writeWord   -> nWrites vezes
107          */
108         EXPECT_CALL(*DE, set(1))
109            .Times(nWrites);
110
111     }
112
113     MOCK_CONST_METHOD3(newGPIO,     GPIO * (char A, int B, int C));
114 };
115
116 #endif
```

APÊNDICE H
MOCK DA CLASSE UART

```
#ifndef MOCKUART_H
#define MOCKUART_H

using ::testing::Return;
using ::testing::ReturnPointee;
using ::testing::_;
using namespace std;

class UART_Common
{
protected:
    UART_Common() {}

public:
    // Parity
    enum {
        NONE = 0,
        ODD = 1,
        EVEN = 2
    };
};

class UART {
 public:

  int counter;

  UART() {}
  UART(unsigned int unit, unsigned int baud_rate, unsigned int data_bits, unsigned
      int parity, unsigned int stop_bits){}

  MOCK_CONST_METHOD1(put,      void(int word));
  MOCK_CONST_METHOD0(get,      int());

};

/*
 * Passa como parametro template o valor 'n'
 * que refere-se ao numero de vezes que o metodo
 * put sera chamado.
 */
template <int nWrites=0, int nReads=0>
class MockUARTCreator {

 public:

  UART * uart1;

  MockUARTCreator(){

    uart1 = new UART();

    EXPECT_CALL(*this, newUART(_,_,_,_,_))
        .Times(1)
        .WillOnce(ReturnPointee(&uart1));

    EXPECT_CALL(*uart1, put(_))
       .Times(nWrites);

    /*
     * Define uma expectiva por leitura a ser efetuada
     * desta forma permite que se definam 0 leituras
     * para testes que usam apenas escrita.
     * A ultima expectativa criada eh a primeira atendida,
     * desta forma, a orderm de retorno sera: 0, 1, 2, ... nReads-1.
     */
    for (int i = nReads; i > 0 ; ) {
      EXPECT_CALL(*uart1, get())
        .WillOnce(Return(--i))
        .RetiresOnSaturation();
    }

  }

```

```
74    MOCK_CONST_METHOD5(newUART, UART * (unsigned int unit, unsigned int baud_rate,
          unsigned int data_bits, unsigned int parity, unsigned int stop_bits));
75
76
77
78  };
79
80
81  #endif
```

APÊNDICE I
IMPLEMENTAÇÃO DAS CLASSES GPIOCREATOR, UARTCREATOR E SERIALRS485

```
#ifndef RS485_H
#define RS485_H


#ifdef RS_485_TEST
  #include "gtest/gtest.h"
  #include "gmock/gmock.h"
  #include "mockGPIO.h"
  #include "mockUART.h"
#else
  #include <gpio.h>
  #include <alarm.h>
  #include <utility/ostream.h>
  #include <uart.h>
  using namespace EPOS;
  OStream cout;
#endif


/* Classe utilizada para a criacao de
 * objetos GPIO, passada como parametro template
 * para a classe SerialRS485
 */
class GPIOCreator {
  public:
    GPIO * newGPIO(char port, int pin, GPIO_Common::Direction inout)
    {
      return new GPIO(port, pin, inout);
    }
};


/* Classe utilizada para a criacao de
 * objetos UART, passada como parametro template
 * para a classe SerialRS485
 */
class UARTCreator {
  public:
   UART * newUART(unsigned int unit, unsigned int baud_rate, unsigned int data_bits,
        unsigned int parity, unsigned int stop_bits){
     return new UART(unit, baud_rate, data_bits, parity, stop_bits);
   }
};

/*
 * Esta classe utiliza a classe UART pois
 * DI ja esta conectado em TX da UART
 * e RO ja esta conectado no RX da UART
 *
 * Como a classe UART1 ja implementa baudrate, paridade, numero de bits
 * basta extender a classe
 */

template <class GPIOClass, class UARTClass>
class SerialRS485 {

  public:

    //PC5 - RS485 RE - Receiving Enable
    GPIO * nRE;
    //PC6 - RS485 DE - Driver Enable
    GPIO * DE;
    UART * uart;

    GPIOClass * gpioCreator;
    UARTClass * uartCreator;


    SerialRS485(unsigned int baud_rate, unsigned int data_bits, unsigned int parity,
        unsigned int stop_bits)
    {
      gpioCreator = new GPIOClass();
      uartCreator = new UARTClass();

```

```
73        nRE    = gpioCreator->newGPIO('C',5, GPIO_Common::OUT);
74        DE     = gpioCreator->newGPIO('C',6, GPIO_Common::OUT);
75        uart   = uartCreator->newUART(1, baud_rate, data_bits, parity, stop_bits);
76        shutdownState();
77      }
78
79      ~SerialRS485(){
80        shutdownState();
81        delete nRE;
82        delete DE;
83        delete uart;
84        delete gpioCreator;
85        delete uartCreator;
86      }
87
88      //DI ja esta conectado em TX da UART
89
90      //Se char der overflow??
91      void writeWord(char i){
92        sendingState();
93        uart->put(i);
94      }
95
96      //RO ja esta conectado no RX da UART
97      char readWord(){
98        receivingState();
99        char i = uart->get();
100       return i;
101     }
102
103
104   private:
105
106     void sendingState(){
107       //cout<<"Sending: DE = 1"<<endl;
108       DE->set(1);
109     }
110
111     void shutdownState(){
112       //cout<<"Shutdown: DE = 0, nRE=1"<<endl;
113       DE->set(0);
114       nRE->set(1);
115     }
116
117     void receivingState(){
118       //cout<<"Receiving: DE = 0, nRE=0"<<endl;
119       DE->set(0);
120       nRE->set(0);
121     }
122 };
123
124 #endif
```

APÊNDICE J
TESTES DA CLASSE SERIALRS485

```
1  //#include <gpio.h>
2  //#include <alarm.h>
3
4  //#include <utility/ostream.h>
5  #include <iostream>
6  #define RS_485_TEST
7  #include "rs485.h"
8
9
10   TEST(SerialRS485Test, testWriteWord){
11
12     const int nWrites=100;
13     const int nReads=0;
14     SerialRS485 < MockGPIOCreator<nWrites, nReads>, MockUARTCreator<nWrites, nReads>
           > r(9600, 8, UART_Common::NONE, 1);
15
16     for(int i=0;i<nWrites;i++)
17       r.writeWord(i);
18   }
19
20 /*
21  * Testa a leitura de uma sequencia de caracteres
22  * bem definido. Verificando o valor recebido a cada
23  * chamada do metodo readWord()
24  */
25
26 TEST(SerialRS485Test, testReadWord){
27   const int nWrites=0;
28   const int nReads=100;
29   SerialRS485 < MockGPIOCreator<nWrites, nReads>, MockUARTCreator<nWrites, nReads> >
         r(9600, 8, UART_Common::NONE, 1);
30
31   char leitura;
32     for(char i=0;i<nReads;i++){
33     leitura = r.readWord();
34     EXPECT_EQ(i, leitura);
35   }
36   cout<<endl;
37
38   /*
39    * Esta implementacao se faz possivel pois a classe MockUARTCreator
40    * retorna valores de 0 a nReads (passado como argumetno template) sequenciamente.
41    */
42
43 }
44
45 /*
46 TEST(SerialRS485Test, testFailExample){
47
48   const int nWrites=2;
49   const int nReads=1;
50   SerialRS485 < MockGPIOCreator<nWrites, nReads>, MockUARTCreator<nWrites, nReads> >
         r(9600, 8, UART_Common::NONE, 1);
51
52     r.writeWord('1');
53     r.writeWord('2');
54 }
55 */
56
57 int main( int argc, char *argv[] ) {
58     ::testing::InitGoogleMock( &argc, argv );
59     return RUN_ALL_TESTS( );
60 }
```

APÊNDICE K
CÓDIGO UTILIZADO PARA COMUNICAÇÃO COM OUTRO DISPOSITIVO UTILIZANDO A CLASSE SERIALRS485

```
#include "rs485.h"

void sleep(int n){
  for(int i=0;i<n;i++){
    for(volatile int t=0;t<0xfffff;t++);
  }
}

int main( int argc, char *argv[] ) {
  sleep(6);
    SerialRS485<GPIOCreator, UARTCreator> r(9600, 8, UART_Common::NONE, 1);
    char arr[3] = " \0";
    const char * msg = "z Teste comunicacao rs485, Rodrigo voce esta ai? Z";
    int send = 1;
    int i =0;
    char m;
    while(1){
    //ENVIO DE MENSAGEM
    if (send){
      cout<<"\nEnviando mensagem no protocolo zZ:"<<endl;
      while(msg[i]!='Z'){
        r.writeWord(msg[i]);
        m = msg[i];
        cout<<m;
        i++;
      }
      r.writeWord(msg[i]);
      m = msg[i];
      cout<<m;
      cout<<"\n";
      sleep(1);
      i=0;
      send = 0;
    }
    //RECEBIMENTO DE MENSAGEM
    else{
      m=r.readWord();
      while(m!='z'){
        cout<<m;
        m=r.readWord();
      }
      cout<<"\nIniciando leitura de mensagem no protocolo zZ:"<<endl;
      while(m!='Z'){
        cout<<m;
        m=r.readWord();
      }
      cout<<m;
    }
  }
}
```