CAN ECU - Automotive Network Course

Felipe Bezerra Martins[1], Lucas Henrique Cavalcanti Santos[2], Roberto Costa Fernandes[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)

{fbm2, lhcs, rcf6}@cin.ufpe.br

1. Introdução

Este trabalho teve o objetivo de construir um ECU (Eletronic Control Unit) CAN (Controller Area Network) usando a linguagem de programação C/C++ em uma plataforma de prototipagem eletrônica. A ECU desenvolvida pela equipe deve ser capaz de se comunicar com outros ECUs por meio de um barramento CAN que será conectado através de um transceiver CAN que foi fornecido.

ECUs são unidades de controle com a função de controlar subsistemas em um veículo e se comunicam com outras ECUs através de mensagens utilizando o protocolo CAN. Esse protocolo foi desenvolvido pela Robert Bosch GmbH. e lançado em 1986 pela SAE (Society of Automotive Engineers).

O protocolo CAN é um protocolo orientado a mensagem, o que significa que todos os nós da rede recebem as mensagens do barramento e devem verificar se ele deve realizar alguma ação ou responder essa mensagem. Esse protocolo possui um arbitração não destrutiva, ou seja, o nó ao tentar enviar uma mensagem deve comparar a prioridade da sua mensagem com a que estiver sendo transmitida, pois a mensagem com menor prioridade perdem a arbitração, e devem ser transmitidas quando forem a de maior prioridade na rede. Essas e outras características do protocolo devem ser implementadas no desenvolvido neste projeto.

O ECU desenvolvido precisa ser capaz de codificar e decodificar mensagem CAN, enviar e receber mensagens CAN2.0A e CAN2.0B. CAN2.0A são mensagens standard, possuem identificadores de 11 bits de tamanho. CAN2.0B são mensagens extendidas, possuem identificadores de 29 bits de tamanho.

2. Estrutura

O projeto foi dividido em múltiplas entregas o que orientou o fluxo de trabalho da equipe onde inicialmente foram desenvolvidos máquinas de estados e diagramas de blocos para diferentes estruturas necessárias na composição do ECU, inicialmente pelo módulo de Bit Timing seguido do módulo de codificação e decodificação e depois de feito e validado foi implementado também em partes na mesma ordem.

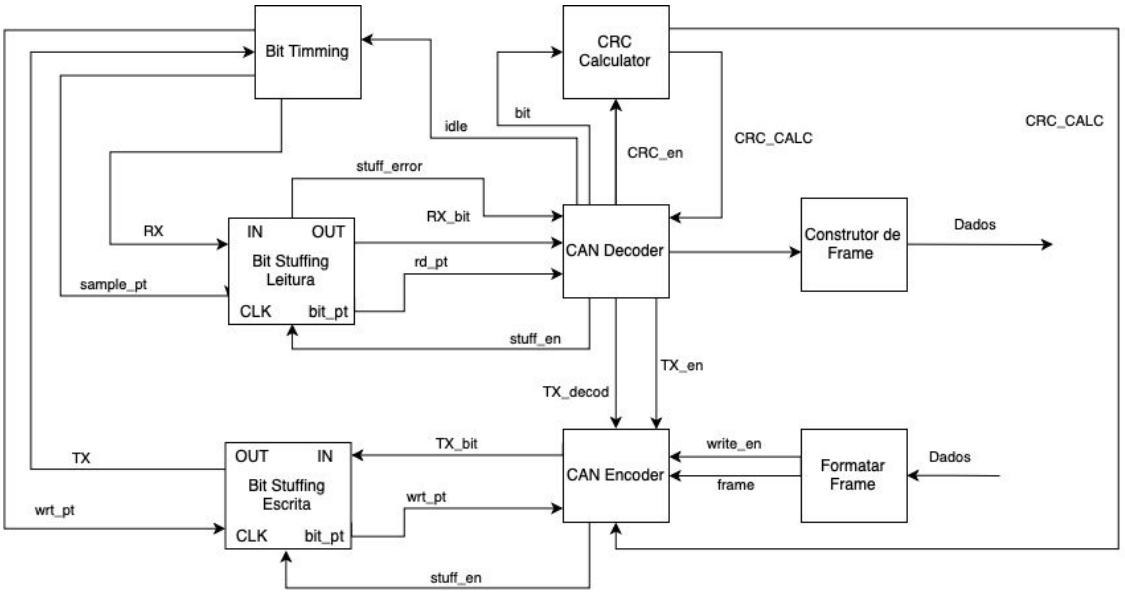


Figura 1 - Diagrama de Blocos ECU.

A figura 1 mostra uma visão geral do funcionamento dos diferentes blocos da ECU desenvolvida pela equipe. O Bit Timing define os pontos de escrita e leitura do barramento indicado pelo Writing Point e Sample Point e possui também a função de sincronização, os módulos de Bit Stuffing realiza a leitura ou escrita nos momentos indicados tratando o stuffing definido no protocolo e sinaliza caso um erro seja detectado.

O CAN decoder faz a montagem da mensagem a partir dos bits lidos e o CAN encoder possui funcionamento inverso definindo os bits que devem ser escritos a partir da mensagem que se deseja transmitir. O decoder identifica e sinaliza ao bloco de Bit Stuffing as áreas a se considerar stuffing, e as áreas de cálculo de CRC ao bloco que calcula o CRC. Quando nenhum frame está sendo escrito, o encoder entra em estado ocioso, e se responsabiliza pelas escritas de erros. Os dois módulos funcionam de modo simultâneo, o decoder, quando necessário realiza escrita indiretamente através do encoder e o barramento é lido pelo decoder mesmo quando uma mensagem está sendo transmitida.

Os outros módulos são o CRC Calculator que calcula o valor do CRC na medida em que os bits são lidos e este valor é usado tanto pelo decoder quanto pelo encoder. O Construtor de Frame vai traduzir a mensagem lida pelo decoder em dados e o Formatar Frame define a mensagem que vai ser transmitida pelo encoder a partir de dados que se deseja transmitir.

2.1. Bit Timing

O Bit Timing trata condições com a finalidade de sincronizar a transmissão de bits, ele define o tamanho de cada bit transmitido sendo dividido em 4 segmentos SYNC_SEG, PROP_SEG, PHASE_SEG1 e PHASE_SEG2, o tamanho de cada segmento é definida em quantidade de Time Quantas e o tamanho de cada Time Quanta é um valor de tempo definido sendo este o tempo para cada iteração do Bit Timing.

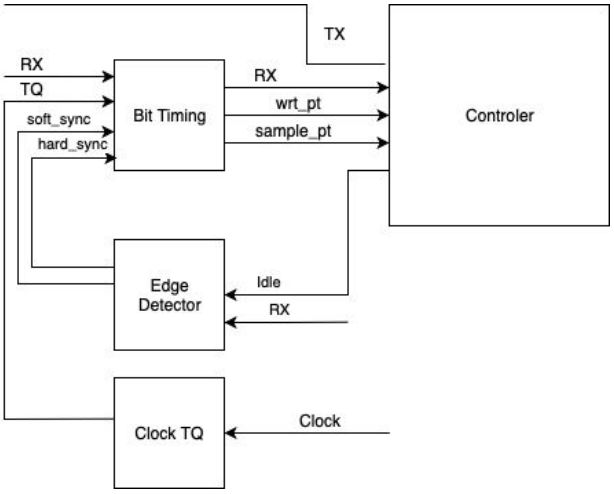


Figura 2 - Diagrama de Blocos Bit Timing.

Os segmentos PHASE_SEG1 e PHASE_SEG2 podem ter seu tamanho modificado para a finalidade de sincronização. Ocorrem dois tipos de sincronização, soft_sync e hard_sync:

- soft_sync: falling edge fora do estado de idle, identificando o ponto de um sample point.
- hard_sync: falling edge no estado de idle, identificando que um frame vai ser transmitido.

O funcionamento do Bit Timing é feito com uma máquina de estados, com apenas 3 estados, um para o SYNC e outros dois para o PHASE_SEG1 e PHASE_SEG2. Eles funcionam sequencialmente contando os Time Quantum especificados, a única maneira de mudar o contador deles é através de soft_sync, e no acontecimento de hard_sync a máquina sempre vai para PHASE_SEG1 para garantir a sincronização com o barramento.

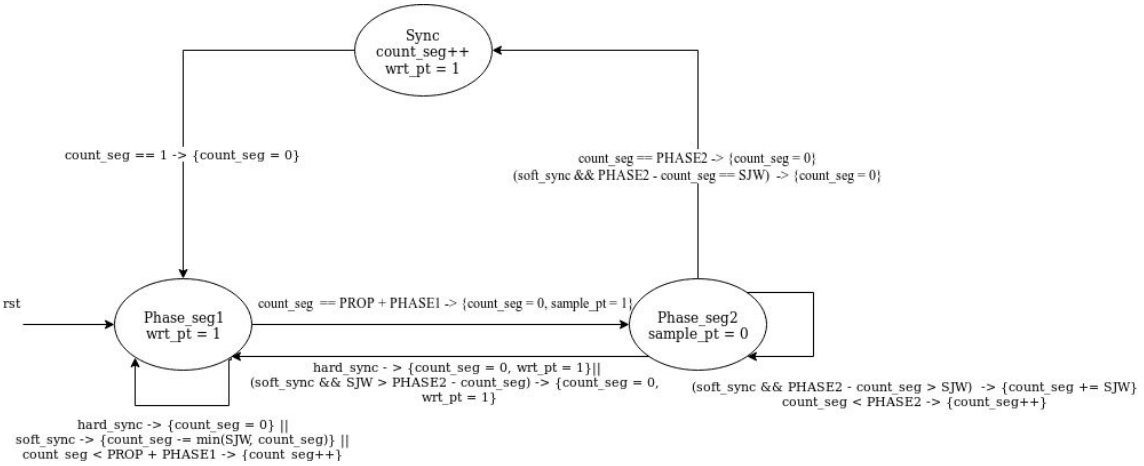


Figura 3 - Máquina de Estados Bit Timing

2.2. Edge Detector

O bloco Edge Detector, presente na figura 2, foi implementado para simbolizar o soft_sync ou hard_sync para o Bit Timing. A diferenciação das sincronizações foi feita através da verificação do estado de idle do Decoder. A detecção de borda foi realizada através de interrupção de descida presente nos microcontroladores.

2.3. Bit Stuffing

O Bit Stuffing é responsável por retirar para o decoder bit de stuff ou adiconar para o encoder. Essa atividade foi implementada sempre respeitando as áreas que devem ter ou não stuff. Através da flag de stuff_en do Decoder, o bloco de Bit Stuffing reconhece quando deve considerar stuff.

No caso de ocorrer um stuff no recebimento de um frame, o Bit Stuffing recebe o sample point, reconhece o bit como stuff, verifica se o mesmo está correto, e não repassa o sample point para o Decoder, mantendo-o pausado. No caso de erro no bit de stuff, o Bit Stuffing repassa o sample point juntamente com uma flag de stuff_error. Assim o Decoder reconhece a situação e executa as devidas providências. Quando a flag the stuff_en é desativada pelo Decoder, todo sample point que chega no Bit Stuffing é repassado para o Decoder.

No caso do Encoder o processo é similar, porém ao invés de retirar bit, é adicionado o bit de stuff aos bits que estão sendo enviados pelo Encoder, e quando este momento acontece o write point não é passado para o Encoder, com a finalidade de deixá-lo pausado enquanto o bit de stuff é enviado.

2.4. Decoder

O Decoder segue a máquina segundo figura 4. Nela cada campo do frame é decodificado enquanto os bits chegam, nenhuma tratamento de stuffing foi necessário pois já foi eliminado anteriormente. As únicas bifurcações foram feitas para acomodar a opção de frame estendido, e tratamento de erros.

Condições de erro, overload ou ACK requerem que bits sejam enviados para isto o Decoder faz a escrita através do Encoder mandado os bits que precisam ser transmitidos.

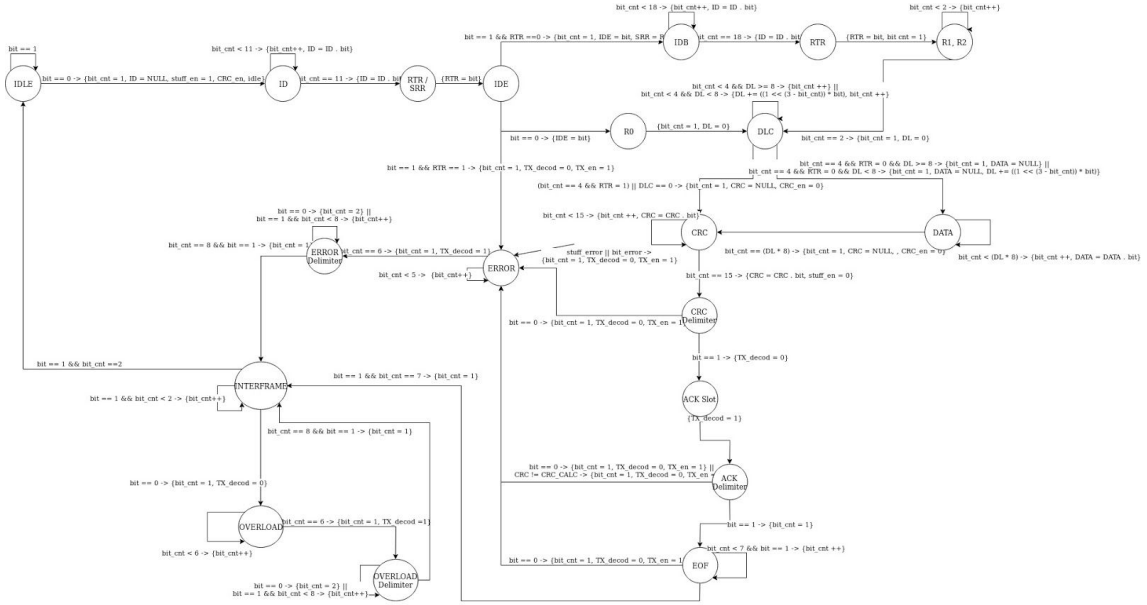


Figura 4 - Máquina de Estados Decoder

Pelo seu funcionamento simultâneo com o encoder, o decoder tem a função de verificar se o bit escrito corresponde ao bit lido, levantando um bit_error caso ocorra uma divergência. Salvo exceções de arbitração onde um bit diferente indica uma perda de arbitração e não de sinaliza erro, e também o ACK onde já é esperado ler um valor diferente do escrito e também não configura como erro.

2.5. Encoder

O Encoder foi montado conforme figura 5. Existe uma semelhança com o Decoder, pois o frame recebido possui mesma estrutura dos enviados, porém existe uma simplificação do Encoder, ao não tratar erros nele, e apenas no Decoder, pois tudo que o Encoder envia, o Decoder também recebe.

O Encoder realiza a escrita de bits mesmo quando não está transmitindo uma mensagem, recebendo diretamente do Decoder a informação de qual bit transmitir para tratar condições de ERROR, OVERLOAD e reconhecimento de recebimento de mensagem no ACK.

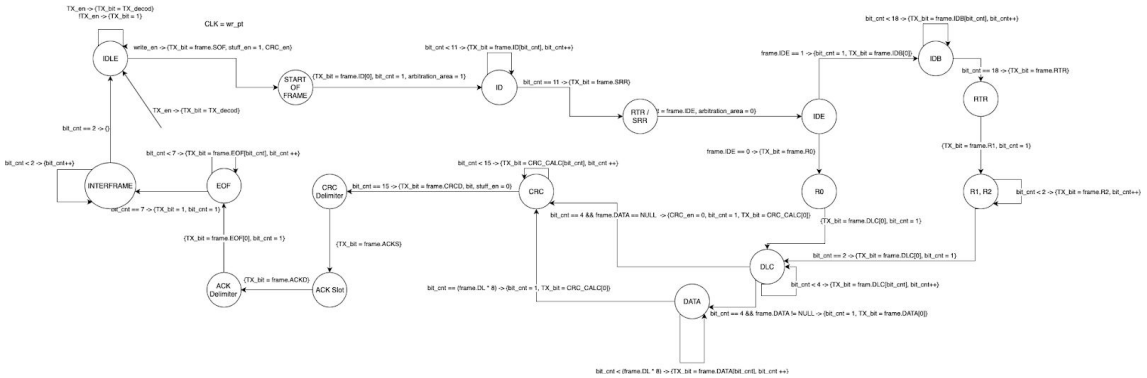


Figura 5 - Máquina de Estados Encoder

3. Implementação

A implementação desses módulos foi feita na placa de desenvolvimento NUCLEO-F767ZI da STMicroelectronics. Essa placa foi escolhida para se encaixar nos requisitos do projeto e pela alta frequência de funcionamento fornecida pelo seu processador ARM. Nessa placa foi escolhido utilizar o sistema operacional Mbed, por sua facilidade de uso e seu desempenho.

Nas próximas seções serão expostas a forma de implementação de cada submódulo descrito na seção 2.

3.1. Bit Timing

O módulo de Bit Timing é executado a cada Time Quanta, para os testes realizados pela equipe o Time Quanta foi definido em 10 ms. Com isso a função de execução desse bloco foi anexado a uma interrupção de tempo a cada Time Quanta, como pode ser visto na figura 6. A figura 7 mostra a implementação da função de Bit Timing, que consiste de um seletor de estados, dentre os três estados definidos na máquina de estados.

```
tq_clock.attach(bitTimingSM, TIME_QUANTA_S);
```

Figura 6 - Módulo de Bit Timing anexado a uma interrupção de tempo

```
void bitTimingSM(){
  static int state = PHASE1_ST;
  static int count = 0;
  switch(state){
    case SYNC_ST:⋯
    case PHASE1_ST:⋯
    case PHASE2_ST:⋯
  }
```

Figura 7 - Seletor de estados do Bit TIming

3.2. Edge Detector

Como descrito na seção 2.2 o módulo de Edge Detector tem a finalidade de detectar um Falling Edge no RX. Por causa do uso do Mbed OS foi possível anexar uma interrupção em uma transição negativa do RX, como pode ser visto na figura 8. Na figura 9 é possível observar a função do Edge Detector, que foi facilitada pelo uso do artifício da interrupção.

```
RX.fall(&edgeDetector);
```

Figura 8 - Anexo do módulo de Edge Detector a uma transição negativa

```
void edgeDetector(){
  if (idle)
  {
    soft_sync = 0;
    hard_sync = 1;
    idle = 0;
  }
  else
  {
    hard_sync = 0;
    soft_sync = 1;
  }
  debug(pc.printf("Edge: Soft: %d, Hard: %d\n", soft_sync, hard_sync));
}
```

Figura 9 - Função de Edge Detector

3.3. Bit Stuffing

O módulo de Bit Stuffing, assim como o de Edge Detector, foi possível ser simplificado após o início da implementação. Isso foi possível, pois foi utilizada uma variável para salvar o valor do bit que se está verificando o stuff. Com isso a função do Bit Stuffing conta com apenas dois estados, como pode ser visto na figura 10. A figura 11 mostra um pouco mais detalhadamente o estado de contagem do Bit Stuffing.

```
void bitstuffREAD()
{
  static int count = 0;
  static int state = 0;
  static int last_rx;
  last_rx = RX_bit;
  RX_bit = RX.read(); // TRANSCIEVER
  switch(state)
  {
    case(START):…
    case(COUNT):…
  }
}
```

Figura 10 - Função do Bit Stuffing

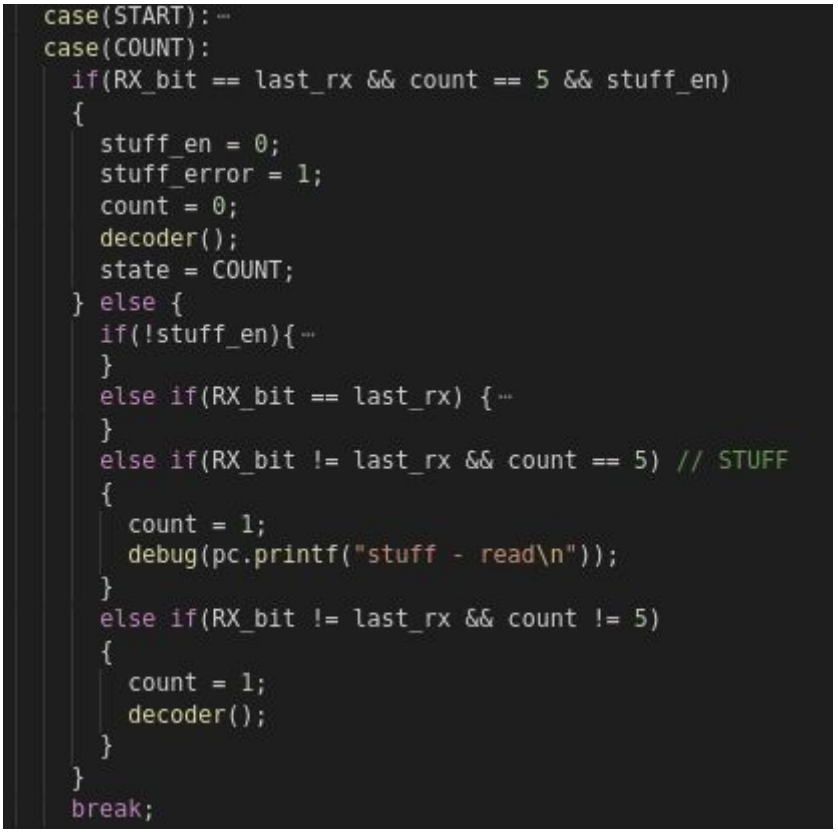


Figura 11 - Estado de contagem do Bit Stuffing

3.4. Encoder

O Encoder foi implementado como um função sendo chamado pelo *Bit Stuffing* de escrita, a cada iteração ele carrega na variável global *TX_bit* o valor do bit que vai ser escrito no barramento no próximo *writing point*.

```
void encoder(){
  static int state = 0;
  static int bit_cnt = 0;
  if(TX_en){
    state = IDLE;
  }
```

Figura 12 - Função Encoder

```
  switch(state){
    case IDLE: …
    case SOF: …
    case ID: …
    case SRR: …
    case IDE: …
    case R0: …
    case IDB: …
    case RTR: …
    case R1: …
    case R2: …
    case DLC: …
    case DATA: …
    case CRC_V: …
    case CRC_D: …
    case ACK_S: …
    case ACK_D: …
    case EOFRAME: …
    case INTERFRAME: …
  }
```

Figura 13 - Estados Encoder

A figura 12 mostra a declaração da função e suas condições iniciais. Uma condição trata o recebimento de um sinal *TX_en* enviado pelo Decoder na ocorrência de erro ou overload, mudando o estado do Encoder para *Idle* onde ele enviará os bits recebidos do Decoder. A figura 13 mostra a implementação da máquina de estados do Encoder.

```
case R0:
  if(RX_bit != TX_bit) //BIT_ERROR
  {
    state = IDLE;
    TX_bit = 1;
    bit_error = 1;
    break;
  }
  TX_bit = (frame_send.DLC >> 3)&1;
  bit_cnt = 1;
  state = DLC;
  break;
```

Figura 14 - Estado R0

A cada iteração é comparado o bit escrito com o bit lido no barramento, esta condição é verificada em cada estado em caso de divergência podendo significar a perda de arbitração ou um bit error dependendo do estado atual, com exceção do estado de ACK onde já é esperado ler um valor diferente do escrito. A figura 14 mostra a implementação do estado R0, onde se pode observar que o *TX_bit* é referente a próxima iteração pois será escrito no próximo *writing point.*

3.5. Decoder

Na implementação do Decoder primeira coisa a ser feita é identificar se foi levantado algum erro detectado fora do Decoder. Esses erros externos podem ser erro de Bit Stuff detectado pelo módulo de Bit Stuffing ou Bit Error detectado pelo Encoder. Além desses erros o Decoder é responsável por identificar outros erros, como por exemplo, erro de CRC, erro de ACK. A figura 15 mostra o seletor de estados do Decoder.

Acoplado ao módulo do Decoder está a função de cálculo do CRC. Essa função é chamada a cada bit lido, porém o cálculo só é executado se a flag CRC_en estiver ativada. A implementação desenvolvida pela equipe para o cálculo do CRC pode ser observada na figura 16.

```
void decoder(){
  static int state = 0;
  static int bit_cnt = 0;
  bool bit = RX_bit;
  if(stuff_error || bit_error){
    stuff_en = 0;
    bit_cnt = 0;
    TX_decod = 0;
    TX_en = 1;
    state = ERROR_FLAG;
    debug(pc.printf("Error Detected: %s\n", (stuff_error)?"STUFF_ERROR": "BIT_ERROR"));
    stuff_error = 0;
    bit_error = 0;
  }
  switch(state){
    case(IDLE):…
    case(ID):…
    case(SRR):…
    case(IDE):…
    case(R0):…
    case(IDB):…
    case(RTR):…
    case(R1):…
    case(R2):…
    case(DLC):…
    case(DATA):…
    case(CRC_V):…
    case(CRC_D):…
    case(ACK_S):…
    case(ACK_D):…
    case(EOFRAME):…
    case(INTERFRAME):…
    case(OVERLOAD):…
    case(OVERLOAD_D):…
    case(ERROR_FLAG):…
    case(ERROR_D):…
  }
  calculateCRC(bit);
}
```

Figura 15 - Implementação do Decoder

```
void calculateCRC(bool bit)
{
  if (CRC_en) {
    CRC_CALC <<= 1;
    if ((CRC_CALC >= (1 << 15)) ^ bit) { // um smente no bit mais significativo
      CRC_CALC ^= 0x4599;
    }
    CRC_CALC &= 0x7fff; // zero no bit mais significativo e um no resto
  }
}
```

Figura 16 - Implementação do cálculo do CRC

4. Testes e Resultados

4.1 Casos de Teste

Foram utilizados 12 frames diferentes variando seus parâmetros. Destes frames testados 9 alimentam o Encoder, que automaticamente utiliza o Decoder para identificar o frame que está sendo passado para o barramento. Os outros 3 frames foram utilizados para testar erros, e como o Encoder não envia frames errados, apenas o Decoder foi considerado nesse momento.

Os 12 frames possuíam como parâmetro:

- Frame 1:
  - ID = 0x0672;
  - RTR = 0;
  - IDE = 0;
  - DLC = 8;
  - DATA = 0xAAAAAAAAAAAAAAAA;
  - CRC_V = 81;
- Frame 2:
  - ID = 0x0672;
  - RTR = 0;
  - IDE = 0;
  - DLC = 7;
  - DATA = 0xAAAAAAAAAAAAAA;
  - CRC_V = 22941;
- Frame 3:
  - ID = 0x0672;
  - RTR = 0;
  - IDE = 0;
  - DLC = 3;
  - DATA = 0xAAAAAA;
  - CRC_V = 9665;

Os três primeiros frames variam apenas o número de dado enviados, é possível reconhecer através do DLC e do DATA, e consequentemente o CRC também é alterado.

- Frame 4:
  - ID = 0x0672;
  - RTR = 0;
  - IDE = 0;
  - DLC = 0;
  - DATA = 0;
  - CRC_V = 13013;
- Frame 5:
  - ID = 0x0672;
  - RTR = 1;
  - IDE = 0;
  - DLC = 0;

    - DATA = 0;
    - CRC_V = 16656;
- Frame 6:
    - ID = 0x0672;
    - RTR = 1;
    - IDE = 0;
    - DLC = 1;
    - DATA = 0;
    - CRC_V = 1161;

Os frames 4, 5 e 6, são 3 frames remotos e são ativos de 3 formas diferentes. O frame 4 é sinalizado através de DLC = 0 e sem DATA, o frame 5 é remoto pois o RTR = 1, e o frame 6 testa se mesmo com DLC = 1, o frame continua remoto pois RTR = 1.

- Frame 7:
    - ID = 0x0449;
    - SRR = 1;
    - RTR = 0;
    - IDE = 1;
    - IDB = 0x3007A;
    - DLC = 8;
    - DATA = 0xAAAAAAAAAAAAAAAA;
    - CRC_V = 31733;
- Frame 8:
    - ID = 0x0449;
    - SRR = 1;
    - RTR = 1;
    - IDE = 1;
    - IDB = 0x3007A;
    - DLC = 8;
    - DATA = 0;
    - CRC_V = 10742;
- Frame 9:
    - ID = 3;
    - SRR = 1;
    - RTR = 0;
    - IDE = 1;
    - IDB = 0;
    - DLC = 15;
    - DATA = 0xFFFFFFFFFFFFFFF;
    - CRC_V = 20214;

Os frames 7, 8 e 9 são frames com ID estendido. O frame 7 é apenas um frame de dados com ID estendido simples, já o frame 8 é um frame remoto por conta do RTR = 1, mesmo com DLC = 8. Já o frame 9, é um frame de dados com DLC maior que 8, justamente para testar se o Encoder e Decoder limitam o DLC a 8.

- Frame 10:

01100111001000010001010101010101010101010101010101010101010101010101
01010101010100000100001010001110000001111111111

Sequência de bits que significa:

  - ID = 672;
  - SRR = 0;
  - RTR = 0;
  - IDE = 0;
  - DLC = 8;
  - DATA = 0xAAAAAAAAAAAAAAAA;
  - CRC_V = 81;
  - Com **Ack Error**

- Frame 11:

01100111001000010001010101010101010101010101010101010101010101010101
01010101010100000000000001111111111

Sequência de bits que significa:

  - ID = 672;
  - SRR = 0;
  - RTR = 0;
  - IDE = 0;
  - DLC = 8;
  - DATA = 0xAAAAAAAAAAAAAAAA;
  - Com **Stuff Error**

- Frame 12:

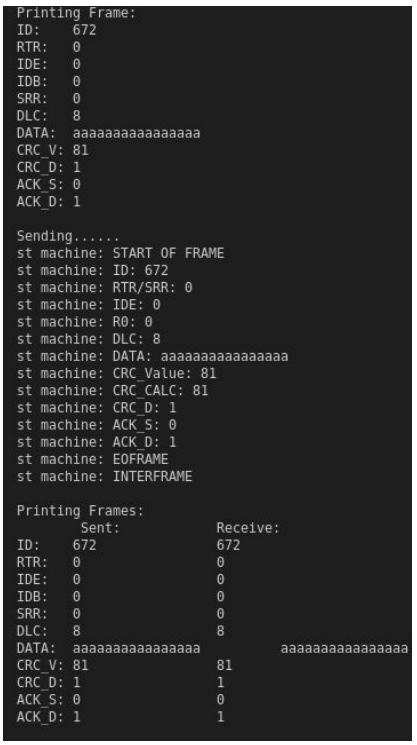
01100111001000010001010101010101010101010101010101010101010101010101
01010101010100000100001110001101000001111111111

Sequência de bits que significa:

  - ID = 672;
  - SRR = 0;
  - RTR = 0;
  - IDE = 0;
  - DLC = 8;
  - DATA = 0xAAAAAAAAAAAAAAAA;
  - Com **CRC Error**

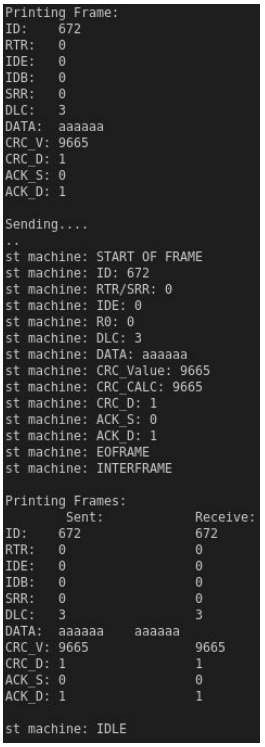
4.1 Resultados

- **Frames 1, 2 e 3 respectivamente.**



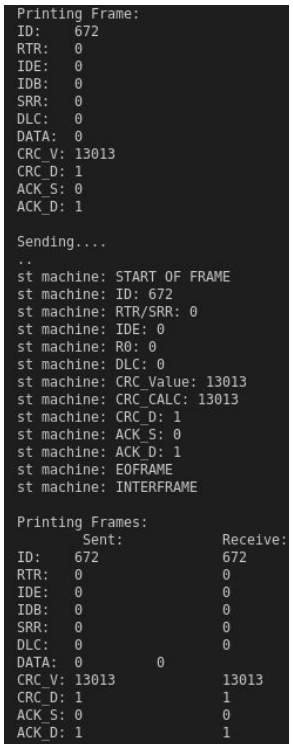
```
Printing Frame:
ID:    672
RTR:   0
IDE:   0
IDB:   0
SRR:   0
DLC:   8
DATA:  aaaaaaaaaaaaaaa
CRC_V: 81
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending......
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 0
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 8
st machine: DATA: aaaaaaaaaaaaaaa
st machine: CRC_Value: 81
st machine: CRC_CALC: 81
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    672              672
RTR:   0                0
IDE:   0                0
IDB:   0                0
SRR:   0                0
DLC:   8                8
DATA:  aaaaaaaaaaaaaaa          aaaaaaaaaaaaaaa
CRC_V: 81               81
CRC_D: 1                1
ACK_S: 0                0
ACK_D: 1                1
```

```
Printing Frame:
ID:    672
RTR:   0
IDE:   0
IDB:   0
SRR:   0
DLC:   7
DATA:  aaaaaaaaaaaaaa
CRC_V: 22941
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending......
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 0
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 7
st machine: DATA: aaaaaaaaaaaaaa
st machine: CRC_Value: 22941
st machine: CRC_CALC: 22941
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    672              672
RTR:   0                0
IDE:   0                0
IDB:   0                0
SRR:   0                0
DLC:   7                7
DATA:  aaaaaaaaaaaaaa   aaaaaaaaaaaaaa
CRC_V: 22941            22941
CRC_D: 1                1
ACK_S: 0                0
ACK_D: 1                1
```

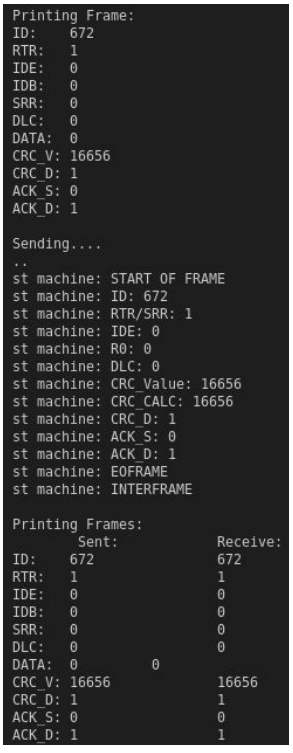


```
Printing Frame:
ID:    672
RTR:   0
IDE:   0
IDB:   0
SRR:   0
DLC:   3
DATA:  aaaaaa
CRC_V: 9665
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 0
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 3
st machine: DATA: aaaaaa
st machine: CRC_Value: 9665
st machine: CRC_CALC: 9665
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
       Sent:           Receive:
ID:    672             672
RTR:   0               0
IDE:   0               0
IDB:   0               0
SRR:   0               0
DLC:   3               3
DATA:  aaaaaa    aaaaaa
CRC_V: 9665            9665
CRC_D: 1               1
ACK_S: 0               0
ACK_D: 1               1

st machine: IDLE
```

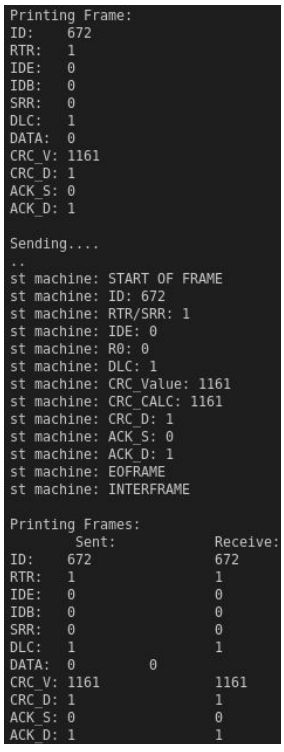
- **Frames 4, 5 e 6 respectivamente.**



```
Printing Frame:
ID:    672
RTR:   0
IDE:   0
IDB:   0
SRR:   0
DLC:   0
DATA:  0
CRC_V: 13013
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 0
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 0
st machine: CRC_Value: 13013
st machine: CRC_CALC: 13013
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    672              672
RTR:   0                0
IDE:   0                0
IDB:   0                0
SRR:   0                0
DLC:   0                0
DATA:  0        0
CRC_V: 13013            13013
CRC_D: 1                1
ACK_S: 0                0
ACK_D: 1                1
```

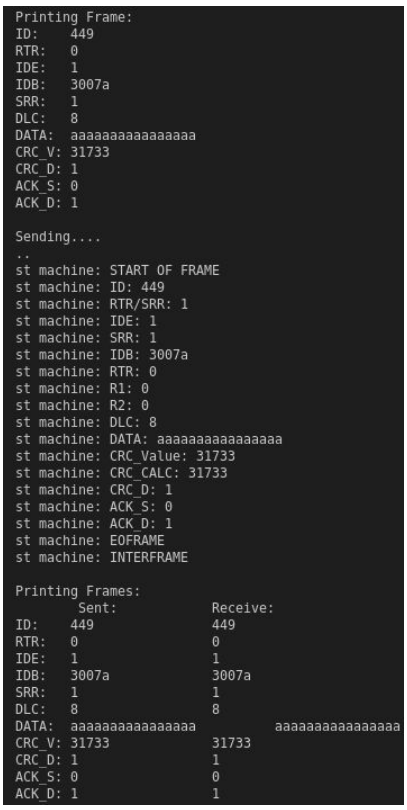


```
Printing Frame:
ID:    672
RTR:   1
IDE:   0
IDB:   0
SRR:   0
DLC:   0
DATA:  0
CRC_V: 16656
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 1
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 0
st machine: CRC_Value: 16656
st machine: CRC_CALC: 16656
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    672              672
RTR:   1                1
IDE:   0                0
IDB:   0                0
SRR:   0                0
DLC:   0                0
DATA:  0        0
CRC_V: 16656            16656
CRC_D: 1                1
ACK_S: 0                0
ACK_D: 1                1
```

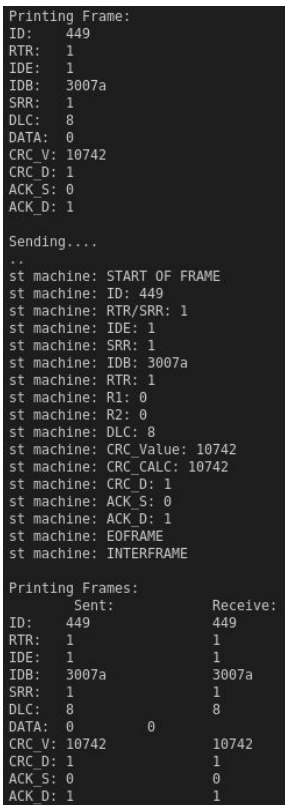


```
Printing Frame:
ID:    672
RTR:   1
IDE:   0
IDB:   0
SRR:   0
DLC:   1
DATA:  0
CRC_V: 1161
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 1
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 1
st machine: CRC_Value: 1161
st machine: CRC_CALC: 1161
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    672              672
RTR:   1                1
IDE:   0                0
IDB:   0                0
SRR:   0                0
DLC:   1                1
DATA:  0        0
CRC_V: 1161             1161
CRC_D: 1                1
ACK_S: 0                0
ACK_D: 1                1
```

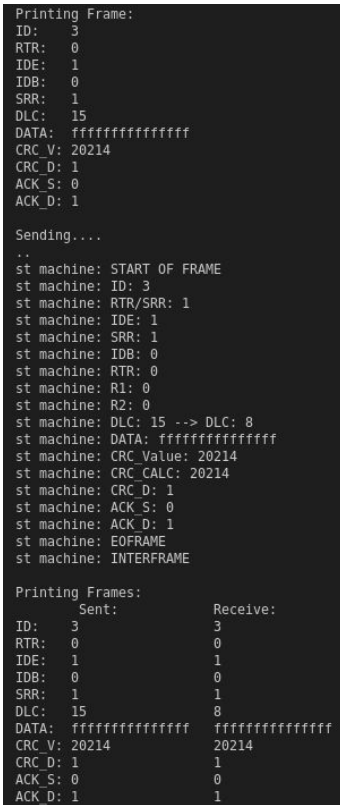
- **Frames 7, 8 e 9 respectivamente.**



```
Printing Frame:
ID:    449
RTR:   0
IDE:   1
IDB:   3007a
SRR:   1
DLC:   8
DATA:  aaaaaaaaaaaaaaaa
CRC_V: 31733
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 449
st machine: RTR/SRR: 1
st machine: IDE: 1
st machine: SRR: 1
st machine: IDB: 3007a
st machine: RTR: 0
st machine: R1: 0
st machine: R2: 0
st machine: DLC: 8
st machine: DATA: aaaaaaaaaaaaaaaa
st machine: CRC_Value: 31733
st machine: CRC_CALC: 31733
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    449             449
RTR:   0               0
IDE:   1               1
IDB:   3007a           3007a
SRR:   1               1
DLC:   8               8
DATA:  aaaaaaaaaaaaaaaa        aaaaaaaaaaaaaaaa
CRC_V: 31733           31733
CRC_D: 1               1
ACK_S: 0               0
ACK_D: 1               1
```



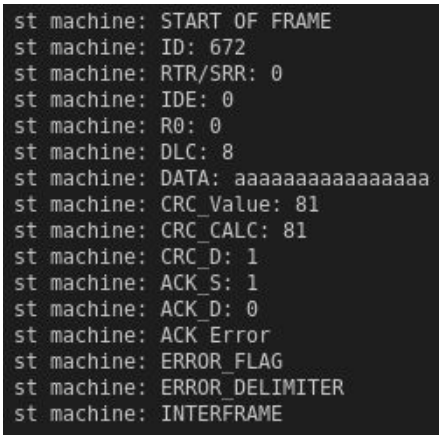
```
Printing Frame:
ID:    449
RTR:   1
IDE:   1
IDB:   3007a
SRR:   1
DLC:   8
DATA:  0
CRC_V: 10742
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 449
st machine: RTR/SRR: 1
st machine: IDE: 1
st machine: SRR: 1
st machine: IDB: 3007a
st machine: RTR: 1
st machine: R1: 0
st machine: R2: 0
st machine: DLC: 8
st machine: CRC_Value: 10742
st machine: CRC_CALC: 10742
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    449             449
RTR:   1               1
IDE:   1               1
IDB:   3007a           3007a
SRR:   1               1
DLC:   8               8
DATA:  0         0
CRC_V: 10742           10742
CRC_D: 1               1
ACK_S: 0               0
ACK_D: 1               1
```



```
Printing Frame:
ID:    3
RTR:   0
IDE:   1
IDB:   0
SRR:   1
DLC:   15
DATA:  fffffffffffff
CRC_V: 20214
CRC_D: 1
ACK_S: 0
ACK_D: 1

Sending....
..
st machine: START OF FRAME
st machine: ID: 3
st machine: RTR/SRR: 1
st machine: IDE: 1
st machine: SRR: 1
st machine: IDB: 0
st machine: RTR: 0
st machine: R1: 0
st machine: R2: 0
st machine: DLC: 15 --> DLC: 8
st machine: DATA: fffffffffffff
st machine: CRC_Value: 20214
st machine: CRC_CALC: 20214
st machine: CRC_D: 1
st machine: ACK_S: 0
st machine: ACK_D: 1
st machine: EOFRAME
st machine: INTERFRAME

Printing Frames:
        Sent:           Receive:
ID:    3               3
RTR:   0               0
IDE:   1               1
IDB:   0               0
SRR:   1               1
DLC:   15              8
DATA:  fffffffffffff   fffffffffffff
CRC_V: 20214           20214
CRC_D: 1               1
ACK_S: 0               0
ACK_D: 1               1
```

Nos casos anteriores é possível observar que todo caso de teste enviado, foi reconhecido corretamente, inclusive os casos com DLC > 8 e frame remoto com DLC != 0.

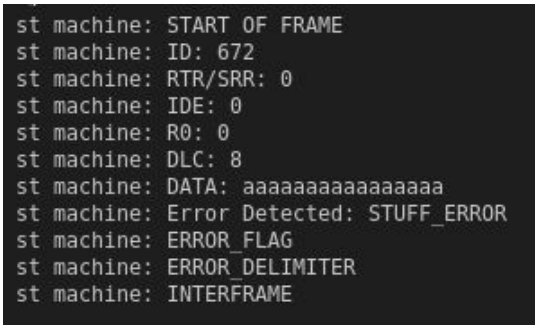
- **Frame 10.**



```
st machine: START OF FRAME
st machine: ID: 672
st machine: RTR/SRR: 0
st machine: IDE: 0
st machine: R0: 0
st machine: DLC: 8
st machine: DATA: aaaaaaaaaaaaaaaa
st machine: CRC_Value: 81
st machine: CRC_CALC: 81
st machine: CRC_D: 1
st machine: ACK_S: 1
st machine: ACK_D: 0
st machine: ACK Error
st machine: ERROR_FLAG
st machine: ERROR_DELIMITER
st machine: INTERFRAME
```

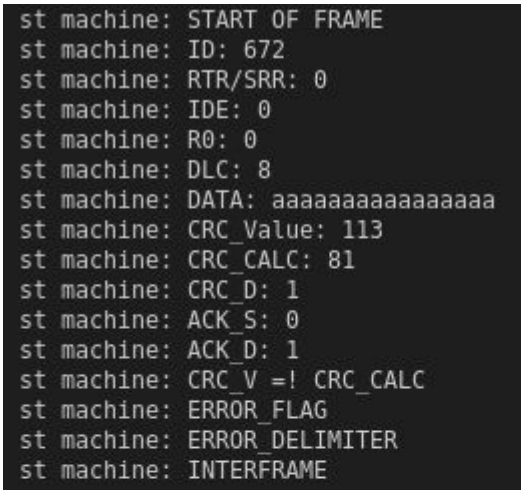
Pela imagem acima é possível ver um ACK erro logo após o ACK delimiter, pois o valor de ACK slot veio 1.

- **Frame 11**



  Pela imagem acima é possível ver que o frame 11 tem um stuff error logo após o DATA.

- **Frame 12**



  Pela imagem acima é possível ver a detecção de erro no CRC, e ele é levantado logo após o ACK delimiter.

5. Dificuldades encontradas

Durante o desenvolvimento do projeto encontramos as seguintes dificuldades:

- Transceiver entregue não funcionou;
- Implementação de uma ECU em microcontrolador não funciona;
- Dificuldade de validar o projeto, a não ser com o que foi desenvolvido pela equipe;
- Interface de debug lenta;
- Microcontrolador não suporta um rede CAN a 500 kbps;

6. Divisão de tarefas

A divisão de tarefas foi da seguinte forma:

	Felipe	Lucas	Roberto
Bit Timing	60 %	20 %	20 %
Bit Stuffing	20 %	20 %	60 %
Encoder	33 %	33 %	33 %
Decoder	33 %	33 %	33 %
Integração	30 %	35 %	35 %
Validação	20 %	60 %	20 %
Relatório	35 %	30 %	35 %

7. Conclusão

Com esse trabalho foi possível entender todas funcionalidades e as nuâncias do protocolo CAN. Além disso também foi possível entender como funciona a redundância e confiabilidade deste protocolo. Infelizmente, por causa do tempo não foi possível realizar teste da ECU desenvolvida com as outras equipes, nem em um barramento CAN de um carro.

References

Bosch. CAN Specification. http://can.marathon.ru/files/can2spec.pdf

STMicroelectronics. STM32 Nucleo-144 boards User Manual. https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/26/49/90/2e/33/0d/4a/da/DM00244518/files/DM00244518.pdf/jcr:content/translations/en.DM00244518.pdf

Mbed OS Arm. https://www.mbed.com/en/