Fundamentos de Comunicação Digital
Codificação em Banda Base

Gustavo Corrêa Rodrigues[1], **Lívia Emanuelle Carrera Soares da Silva**[1]

[1]Faculdade de Computação – Instituto de Ciências Exatas e Naturais
Universidade Federal do Pará (UFPA)
Av. Augusto Correa 01, 66075-090 – Belém – PA – Brasil

{gustavo.rodrigues, livia.silva}@icen.ufpa.br

Abstract. *The goal of this paper is to describe and analyze different encoding schemes, furthermore, simulators of these schemes were implemented to help in the demonstration and analysis of the given subjects, the encoding schemes brought up on this paper are: NZRI, HDB4, B8ZS, Differential Manchester.*

Resumo. *O objetivo deste artigo é descrever e analisar esquemas de codificações diferentes, além disso, simuladores destes esquemas foram implementados para auxiliar na demonstração e analise destes, os esquemas abordados neste trabalho são: NZRI, HDB3, B8ZS E Manchester Diferencial.*

1. Introdução

Devido à evolução dos meios de comunicação, diversas técnicas de codificação e transmissão da informação digital foram criadas. Nesse cenário, a Banda Larga (BroadBand) transmite informação por meio da modulação de onda portadora com aparelhos de Modulação e Demodulação (MODEM). Do mesmo modo, no contexto de Banda Base (BaseBand), a informação é codificada diretamente sobre o par de fios, sendo que para isso ocorra os sinais são codificados como diferenças discretas de voltagem, combinando a um ou mais bits de informação, além disso, eles geram simplicidade e eficiência, por estas razões são amplamente utilizados em comunicação de dados em distâncias limitadas, como em rede local.

Ademais, verificam-se aspectos que tornam uma codificação ”exemplar”, os quais são:

1. A constante alteração de níveis lógicos (0 para 1 e 1 para 0);
2. Ocupar pouca largura de banda;
3. Ausência de componentes de Corrente Contínua (DC);
4. Capacidade de detecção de erro;
5. Imunidade a ruídos e interferências de sinais.

Esse artigo apresenta e implementa quatro dessas codificações, em ordem: i) Non-return to Zero Inverted (NRZ-I); ii) High Density Bipolar with 3 zero maximum tolerance prior to zero substitution (HDB3); iii) Bipolar with 8 zero substitution (B8ZS) e iv) Manchester Diferencial sendo estruturado como: a sessão 2 define a metodologia seguida durante o trabalho; a sessão 3 descreve e mostra a implementação das quatro codificações e a sessão 4 apresenta as considerações finais sobre o trabalho.

2. Metodologia de desenvolvimento

Dentre a avaliação dos requisitos levantados pela tarefa, na elaboração da metodologia de desenvolvimento deste trabalho tem-se em dois requisitos fundamentais:

- Codificar 4 códigos diferentes: NRZI, HDB3, B8ZS e Manchester Diferencial.
- O programa deve exibir o resultado em forma gráfica, a sequência original e os códigos resultantes.

Em relação à implementação dos códigos, foi de comum acordo entre os envolvidos a utilização do paradigma de Programação Orientada a Objetos (POO), pois assim seria possível estabelecer uma abstração computacional, centralizando e padronizando os recursos necessários em um único objeto. Para a exibição dos resultados de forma gráfica, foi um aspecto decisivo a escolha da Linguagem de programação Python 3 [1], visto que ela é multiparadigma (abrangendo nosso requisito inicial) e possui uma série de construções e bibliotecas amplamente utilizadas para visualização de dados, onde podemos destacar a biblioteca de plotagem Matplotlib [2], usada em nossos códigos. Os códigos-fonte podem ser encontrados em um repositório na plataforma de hospedagem de código GitHub [3].

A figura 1 mostra O diagrama de classes UML contendo a estrutura básica do código a ser desenvolvido.

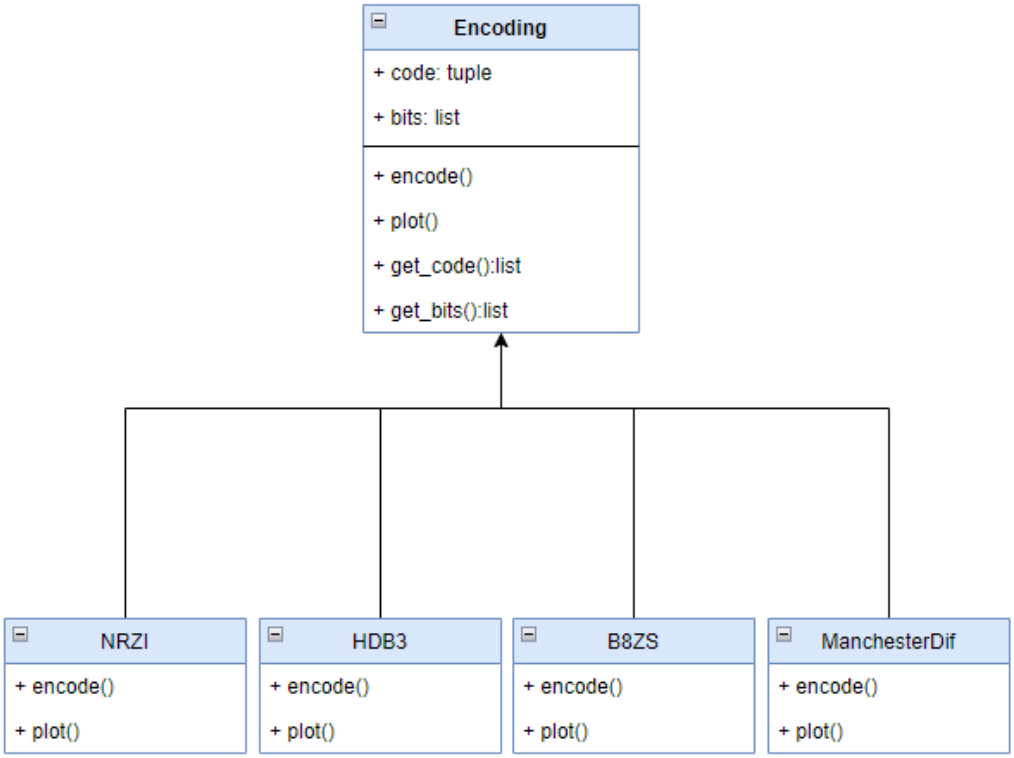


Figura 1. Diagrama de classes UML implementadas

[1] https://docs.python.org/3/
[2] https://matplotlib.org/stable/index.html
[3] https://github.com/Lixipluv/Codigos-Banda-Base

É definida uma classe Encoding, que contém todos os atributos e métodos necessários para a tarefa. Ela é uma classe genérica que servirá de protótipo para as implementações das demais classes específicas, utilizando o conceito de Hierarquia. Os dois atributos da classe fazem o armazenamento de uma lista de inteiros — contendo os bits originais da transmissão — e de uma tupla, que será atribuída ao retorno do método de codificação. Também há um método de codificação abstrato, implementado por cada subclasse, que retorna uma 2-upla (3-upla no caso do HDB3) contendo uma lista de inteiros com os bits codificados e uma lista de inteiros que auxilia na construção do gráfico final. Um método de plotagem irá criar a representação gráfica dos resultados.

Os códigos abordados serão descritos com detalhes na seção 3, onde são apresentadas as subclasses representativas das codificações dos sinais digitais. Portanto, cada subclasse específica implementa os métodos abstratos de codificação e plotagem.

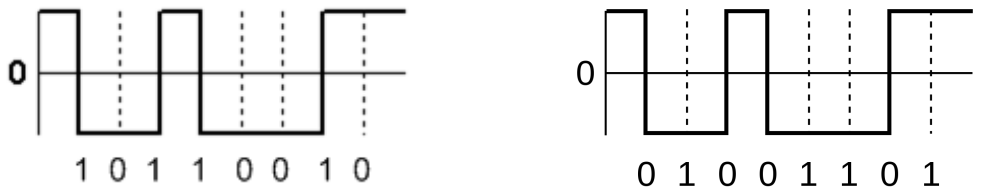
3. Códigos

3.1. NZRI

3.1.1. Descrição

No uso de computadores de mesa, notebooks e smartphones, o que todas essas tecnologias têm em comum é a necessidade de um método barato e confiável de transferência de dados, qualidades presentes na técnica NRZI (Nonreturn to Zero - Inverted). Essa técnica de codificação garante a integridade dos dados transferidos sem a necessidade de enviar um sinal de clock junto aos dados [Pai 2012].

Nesse método de codificação, somente valores de carga positiva e negativa são considerados. Começando do um positivo (1), sempre que houver uma transição para o um negativo (-1), está sendo representada a codificação do bit ”1”, assim como quando há a transição do estado negativo para o estado positivo. Por outro lado, a ausência de transição de estados indica o bit 0 [Schools 1961]. Todavia, há versões onde transições indicam o bit 0 e a ausência de transição indica o bit 1 [Pai 2012] e [Guri et al. 2016].



(a) NRZI padrão **(b) NRZI Alternativo**

Por mais que primariamente não haja a necessidade do uso de clock na codificação NRZI, uma longa série de 0 (ou 1, na versão alternativa) traz dois problemas: a dificuldade de sincronia com o receptor e a presença de componentes de Corrente Contínua, as quais não são desejáveis visando uma possível retransmissão. Para evitar isso, alguns protocolos inserem um bit de transição a cada intervalo de 4 a 6 bits neutros [Pai 2012]. Assim, uma sequência como 101100000 se tornaria 1011000010. Essa manobra impede os dois problemas citados anteriormente.

3.1.2. Implementação

O código fonte completo está disponível no anexo II.

Após herdar as características da superclasse Encoding conforme descrito em 2, é definido o método encode do NRZI, o qual retorna uma tupla contendo uma lista com os bits codificados seguindo o padrão NRZI e uma lista auxiliar que fará as coordenadas com o eixo x de modo a permitir a correta plotagem dos dados no gráfico final.

Nesse método, vale-se destacar que, conforme se itera sobre a lista de inteiros que representa os bits informados pelo usuário, é realizada a verificação de bit. Caso o bit lido seja "1", a variável state troca de valor de forma ternária, representando a troca de estados. Todavia, quando o bit lido for "0"não é necessária a troca de estados.

Por fim, o método plot configura os objetos `figure` e `axs` de forma a incluir no eixo x os bits ao invés de posições $(0, 1, 2, ..., n)$ e no eixo y apenas os pontos 1 e -1, representando os únicos estados possíveis. O eixo x é povoado com o segundo elemento da tupla retornada por encode enquanto o eixo y é povoado com o primeiro elemento da mesma tupla. Com ambos, x e y preenchidos e com o mesmo tamanho, é possível plotar o gráfico correspondente à mensagem binária fornecida pelo usuário.

3.1.3. Avaliação

De forma geral, o NRZI é um método de simples implementação e entendimento, usado até hoje nas tecnologias de disco ótico (CDs) [Pai 2012] e USBs [Guri et al. 2016], ele garante com certa confiança e por baixo custo a integridade dos dados [Pai 2012]. Outrossim, a codificação é parcialmente sujeita a dessincronização e a componentes de Corrente Contínua (DC), que podem acontecer devido a uma longa sequência de bits neutros (0, por padrão), porém, alguns mecanismos de segurança, como adicionar bits de transição após uma certa sequência de bits neutros diminuem a possibilidade desses erros [Pai 2012].

Para protocolos que permitem a configuração de codificação, é aconselhável o uso do NRZI devido à sua relativa imunidade a erros de configuração de inversão de sinal. Entretanto, ele ainda é passível de erros durante sua codificação, por isso, recomenda-se utilizar Taxa de Verificação de Quadro (FCS), um código de detecção de erros, quando a codificação Nonreturn to Zero Inverted estiver em uso [Simpson 1994].

3.2. HDB3

3.2.1. Descrição

A codificação HDB-3 (High Density Bipolar With 3 zero maximum tolerance prior to zero substitution) é uma codificação que tem como uma das principais propostas eliminar a componente DC do sinal. Ela faz isso de duas formas, a primeira, alternando a polaridade de bits de sinal a todo momento. Bits de informação consecutivos nunca tem a mesma polaridade, mas sim voltagens invertidas, já a segunda é sinalizando os sinais que apresentam 4 bits "0"seguidos e os substituindo por bits de violação e/ou sinalização. A escolha é feita para garantir que violações consecutivas sejam de polaridade diferente; isto é, separados por um número ímpar de marcas + ou - normais. Uma cadeia de bits

0000 é substituída por S00V ou 000V, dependendo da polaridade da quantidade de bits de informação presentes desde a última violação. Se a paridade de bits desde a violação anterior é par, será do tipo S00V. Se não, será do tipo 000V.

Paridade de +/− bits desde V anterior	Padrão	Pulso anterior	Código
Par	B00V	+	−00−
		−	+00+
Ímpar	000V	+	000+
		−	000-

Figura 3. Regras para a substituição de 4 zeros consecutivos.

Um exemplo de código HDB-3 pode ser observado na imagem abaixo:

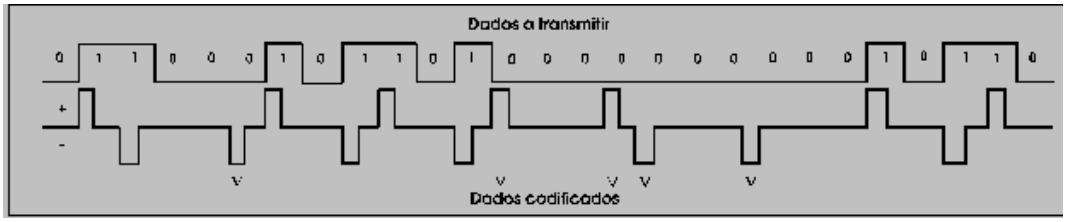


Figura 4. Exemplo de sinal comum (acima) e HDB-3 (abaixo) para o código 011000101101000000000010110.

3.2.2. Implementação

O código-fonte completo está disponível no anexo III.

A implementação em código do HDB3 foi pensada de uma forma fácil de ser compreendida pelo leitor, algumas variáveis a mais foram adicionadas com o intuito de mostrar ao usuário o máximo de informações possíveis do sinal transmitido, como números de 1, 0, violações e sinalizações.

3.2.3. Avaliação

Tal codificação é usada em todos os níveis do sistema E-carrier europeu, um formato de transmissão de dados. Também é utilizado no Japão. É uma codificação poderosa, pois evita a criação de componentes DC e também facilita na sincronização do clock entre o receptor e o transmissor.

3.3. B8ZS

3.3.1. Descrição

A codificação ”bipolar with 8-zero substitution”(Bipolar com substituição de 8 zeros) ou ”B8ZS”foi criada com o objetivo de manter a sincronia caso ocorra uma sequencia grande

de zeros binários na informação transmitida, é considerado uma versão aprimorada do AMI (Alternate Mark Inversion), portanto, as técnicas de codificação funcionam de forma extremamente similar.

AMI (Alternate Mark Inversion) é uma técnica de codificação que usa pulsos bipolares para representar o 1 lógico. O próximo 1 é representado por um pulso de polaridade invertida. Portanto, a sequência de 1s lógicos é representada por uma sequência de pulsos de polaridade alternada. A codificação alternada previne a formação de um nível de corrente DC formar no cabo. [Fairhust 2020]. A principal adição do B8ZS no AMI é a substituição de uma sequência de oito zeros (00000000) pela sequência ”000VB0VB”, onde V representa uma violação, ou seja, pulso na polaridade utilizada anteriormente e B representa uma transição bipolar correta.

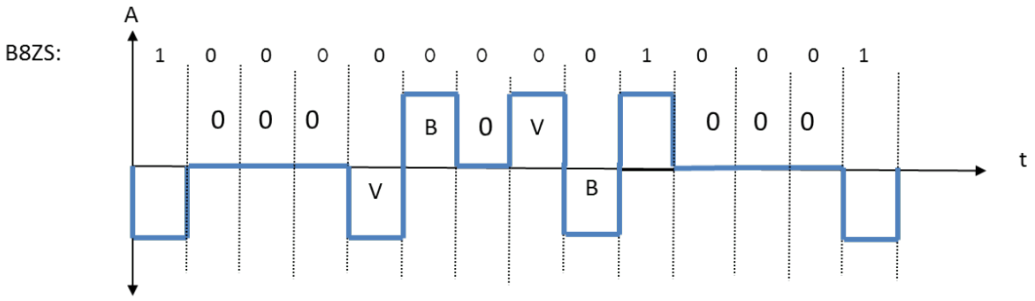


Figura 5. Exemplo de codificação B8ZS para a sequência 10000000010001

3.3.2. Avaliação

Essa codificação é notoriamente utilizada no sinal Norte Americano T1, o principal padrão de telefonia usada nos países norte americanos desenvolvido pela Bell Labs. O B8ZS oferece sincronia a transmissão, apesar de existirem técnicas superiores, é uma boa codificação para sistemas onde a sincronia é necessária, porém não é prioridade.

3.3.3. Implementação

O código-fonte completo está disponível no anexo IV.

Nesta implementação, é utilizada uma variável que guarda o zero encontrado mais cedo e checa essa variável de forma constante para encontrar sequencias de oito zeros, depois, por sabermos o índice do zero mais cedo encontrado, são realizadas substituições simples no vetor que representa os dados codificados.

3.4. Manchester Diferencial

3.4.1. Descrição

Na codificação Manchester Diferencial pelo menos uma transição de fase é garantida por bit transmitido, garantido a sincronia da transmissão sem a necessidade de uma transmissão separada para o clock. O manchester diferencial é uma forma aprimorada do

manchester comum, que utilizava a direção da transição no meio do período para codificar os dados.

Já o Manchester Diferencial utiliza a presença ou ausência de uma transição no início do período do bit para codificar os dados da transmissão, enquanto a transição no meio do período é reservada apenas para transmissão do clock, utilizando o mesmo canal dos dados.

Por não utilizar a direção de uma transição e sim a presença ou ausência para carregar os dados, o manchester diferencial acaba aprimorando o manchester no quesito direcionalidade, pois, caso a polaridade de todo o sinal seja invertida, os dados da mensagem não são perdidos.

3.4.2. Avaliação

Usado em redes lan de token ring e definido pelo padrão IEEE 802.5, a codificação manchester diferencial funciona de forma elegante e garante a sincronia da transmissão, pois transmite o clock e os dados em um único sinal, economizando recursos, porém permitindo a transmissão ocorrer sem compromissos.

3.4.3. Implementação

O código-fonte completo está disponível no anexo V.

No caso do Manchester Diferencial, é utilizado uma variável que guarda o estado da codificação com o intuito de garantir que as transições ocorram na direção correta, essa variável é necessária para garantir a direcionalidade da codificação, se seu valor for invertido, a polaridade do sinal codificado também é invertido, mas os dados são os mesmos.

3.5. Resultados

3.5.1. NRZI

O exemplo da figura 6 ilustra a codificação NRZI para a sequência de bits ”10110010011”, escolhida especificamente para conter todos as combinações de níveis para este esquema.

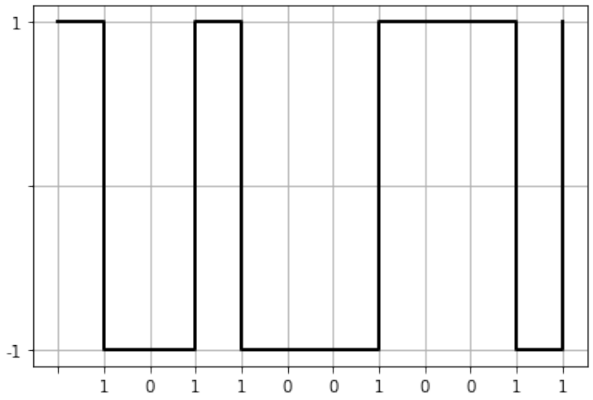


Figura 6. Exemplo de codificação NRZI

3.5.2. HDB3

O exemplo da figura 7 ilustra a codificação HDB3 para a sequência de bits ”1011000000”, escolhida especificamente para conter todos as combinações de níveis para este esquema.

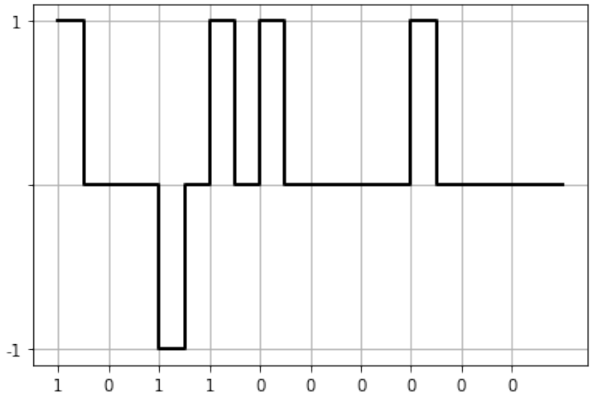


Figura 7. Exemplo de codificação HDB3

3.5.3. B8ZS

O exemplo da figura 8 ilustra a codificação HDB3 para a sequência de bits ”100000000101”, escolhida especificamente para conter todos as combinações de níveis para este esquema.

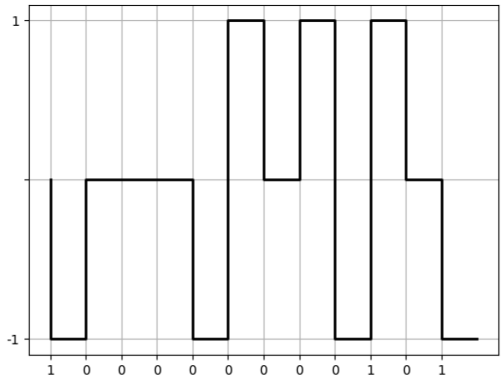


Figura 8. Exemplo de codificação B8ZS

3.5.4. Manchester Diferencial

O exemplo da figura 9 ilustra a codificação Manchester Diferencial para a sequência de bits ”0010010110”, escolhida especificamente para conter todos as combinações de níveis para este esquema.

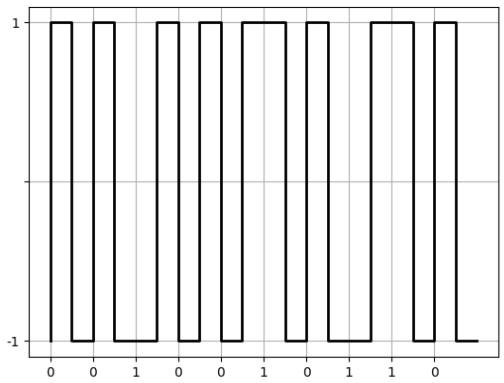


Figura 9. Exemplo de codificação Manchester Diferencial

4. Considerações Finais

Conforme apresentado nesse trabalho, há diversos tipos de codificação que tentam, à sua forma, resolver os problemas a qual elas são propostas. Com suas próprias características, mecanismos de segurança e formas de evitar componentes de corrente contínua, as codificações NRZI, HDB3...

Com a linguagem de programação Python3, o uso das bibliotecas Matplotlib e Numpy, e a aplicação de técnicas de Programação Orientada a Objetos, foi construido um simulador para cada uma dessas codificações, que pode, de forma didática, apresentar os conceitos fundamentais de cada esquema de codificação, através da criação de gráficos estáticos em tempo real, bastando alterar a cadeia de caracteres de entrada.

Referências

Fairhust, G. (2020). Alternate mark inversion (ami).

Guri, M., Monitz, M., and Elovici, Y. (2016). Usbee: Air-gap covert-channel via electromagnetic emission from USB. *CoRR*, abs/1608.08397.

Pai, Y.-T. (2012). Sub-trees modification of huffman coding for stuffing bits reduction and efficient nrzi data transmission. *IEEE Transactions on Broadcasting*, 58:221–227.

Schools, R. S. (1961). Recording and reproduction of nrzi signals. *Journal of Applied Physics*, 32(3):42–43.

Simpson, W. (1994). PPP in HDLC-like Framing. Technical Report 1662, RFC Editor.

Anexo I - Código-fonte da classe Encoding

```python
from abc import abstractmethod

class Encoding:
    def __init__(self, bits:str):
        valid = True
        for i in bits:
            if (i != '0') and (i != '1'):
                valid = False

        if valid == True:
            self.bits = [int(bit) for bit in bits]
            self.code = ()
        else:
            raise ValueError ("Available entry: characters 1 or 0")

    def get_bits(self) -> list:
        return self.bits

    def get_code(self) -> list:
        return self.code[0]

    @abstractmethod
    def encode(self):
        pass

    @abstractmethod
    def plot(self):
        pass
```

Anexo II - Código-fonte da classe NRZI

```python
from encoding import Encoding

import matplotlib.pyplot as plt
import numpy as np

class NRZI(Encoding):
    def __init__(self, bits:str):
        super().__init__(bits)
        self.code = self.encode()


    def encode(self) -> tuple:
        code = [1, 1]
        timestamp = [0]

        counter = 0
        states = (1, -1)
        state = 0

        for i in self.bits: #10110010

            counter += 1

            if i == 1:
                state = 1 if (state == 0) else 0
                code.append(states[state])
                code.append(states[state])

                timestamp.append(counter)
                timestamp.append(counter)

            elif i == 0:
                code.append(states[state])
                timestamp.append(counter)

        code.pop()

```

```
        return code, timestamp


    def plot(self):
        fig, axs = plt.subplots(1)

        y_axis = self.code[0]
        x_axis = self.code[1]

        bit_code = [str(i) for i in self.bits]
        bit_code.insert(0, '')
        plt.xticks(np.arange(len(bit_code)), bit_code)
        plt.yticks([-1, 0, 1], ['-1', '', '1'])

        plt.plot(x_axis, y_axis, 'black', linewidth=2)
        plt.grid()
        plt.show()
```

Anexo III - Código-fonte da classe HDB3

```
from encoding import Encoding

import matplotlib.pyplot as plt
import numpy as np

class HDB3(Encoding):
    def __init__(self, bits:str):
        super().__init__(bits)
        self.code = self.encode()


    def encode(self) -> tuple:
        consecutive_zero_counter = 0
        counter = 0
        informations_number = 0
        VIOLATION_NUMBER = 3
        timestamp = []
        code = []
        converted_data = []
        converted_data_y = []
        information_polarity = True
        violation_polarity = True

        for i in range(len(self.bits)):
            posterior_sublist = self.bits[i+1:i+4]

            if self.bits[i] == 1:
                converted_data.append(1)
                consecutive_zero_counter = 0
                if information_polarity == True:
                    timestamp.append(counter)
                    code.append(1)
                    counter += 0.5
                    timestamp.append(counter)
                    code.append(1)
                    timestamp.append(counter)
                    code.append(0)
                    counter += 0.5
                    timestamp.append(counter)
                    code.append(0)
                    converted_data_y.append(1)
                elif information_polarity == False:
                    timestamp.append(counter)
                    code.append(-1)
                    counter += 0.5
                    timestamp.append(counter)
                    code.append(-1)
                    timestamp.append(counter)
                    code.append(0)
                    counter += 0.5
```

```
                    timestamp.append(counter)
                    code.append(0)
                    converted_data_y.append(-1)
                informations_number += 1
                information_polarity = not information_polarity

            elif self.bits[i] == 0:
                converted_data.append(0)
                if (consecutive_zero_counter == 0) and (sum(posterior_sublist) == 0) and
 (len(posterior_sublist) == 3) and (informations_number%2 == 1 and
informations_number != 1):
                    if (violation_polarity == True):
                        timestamp.append(counter)
                        code.append(1)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(1)
                        timestamp.append(counter)
                        code.append(0)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(0)
                        converted_data_y.append("+s")
                    elif (violation_polarity == False):
                        timestamp.append(counter)
                        code.append(-1)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(-1)
                        timestamp.append(counter)
                        code.append(0)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(0)
                        converted_data_y.append("-s")
                    consecutive_zero_counter += 1

                elif (consecutive_zero_counter == VIOLATION_NUMBER):
                    if (violation_polarity == True):
                        timestamp.append(counter)
                        code.append(1)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(1)
                        timestamp.append(counter)
                        code.append(0)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(0)
                        converted_data_y.append("+v")
                    elif (violation_polarity == False):
                        timestamp.append(counter)
                        code.append(-1)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(-1)
                        timestamp.append(counter)
                        code.append(0)
                        counter += 0.5
                        timestamp.append(counter)
                        code.append(0)
                        converted_data_y.append("-v")
                    violation_polarity = not violation_polarity
                    consecutive_zero_counter = 0

                elif (consecutive_zero_counter != VIOLATION_NUMBER):
                    timestamp.append(counter)
                    code.append(0)
                    counter += 1
                    timestamp.append(counter)
```

```
                    code.append(0)

                    consecutive_zero_counter += 1
                    converted_data_y.append(0)

        return code, timestamp, converted_data

    def plot(self):

        fig, axs = plt.subplots(1)

        x_axis = self.code[1]
        y_axis = self.code[0]
        converted_data = self.code[2]

        plt.xticks(np.arange(len(converted_data)), converted_data)
        bit_code = [str(i) for i in self.bits]
        bit_code.insert(0, '')
        plt.yticks([-1, 0, 1], ['-1', '', '1'])

        plt.plot(x_axis, y_axis, 'black', linewidth=2)
        plt.grid()
        plt.show()
```

Anexo IV - Código-fonte da classe B8ZS

```
from encoding import Encoding

import matplotlib.pyplot as plt
import numpy as np

class B8ZS(Encoding):
  def __init__(self, bits:str):
    super().__init__(bits)
    self.code = self.encode()


  def encode(self) -> tuple:
    timestamp = [0]
    counter = 0
    earliestZero = -1
    state = 1
    code = [0]

    for index, bit in enumerate(self.bits):
      if bit == 1:
        earliestZero = -1
        state = 1 if state == -1 else -1
        code.append(state)
        timestamp.append(counter)
        counter += 1
        timestamp.append(counter)
        code.append(state)

      elif bit == 0:
        if earliestZero == -1:
          earliestZero = index

        if index - earliestZero == 7:
          code.append(0)
          timestamp.append(counter)
          counter += 1
          code.append(0)
          timestamp.append(counter)

          code[earliestZero*2 + 7] = state
          code[earliestZero*2 + 8] = state
          state = 1 if state == -1 else -1
          code[earliestZero*2 + 9] = state
          code[earliestZero*2 + 10] = state
```

```
          code[earliestZero*2 + 13] = state
          code[earliestZero*2 + 14] = state
          state = 1 if state == -1 else -1
          code[earliestZero*2 + 15] = state
          code[earliestZero*2 + 16] = state

          earliestZero = -1

        else:
          code.append(0)
          timestamp.append(counter)
          counter += 1
          code.append(0)
          timestamp.append(counter)

    return code, timestamp


  def plot(self):
        fig, axs = plt.subplots(1)

        y_axis = self.code[0]
        x_axis = self.code[1]

        bit_code = [str(i) for i in self.bits]
        plt.xticks(np.arange(len(bit_code)), bit_code)
        plt.yticks([-1, 0, 1], ['-1', '', '1'])

        plt.plot(x_axis, y_axis, 'black', linewidth=2)
        plt.grid()
        plt.show()
```

Anexo V - Código-fonte da classe Manchester Diferencial

```
from encoding import Encoding

import matplotlib.pyplot as plt
import numpy as np

class ManchesterDif(Encoding):
  def __init__(self, bits:str):
    super().__init__(bits)
    self.code = self.encode()


  def encode(self) -> tuple:
    timestamp = [0]
    counter = 0
    state = -1
    code = [-1]

    for bit in self.bits:
      if bit == 0:
        state = -1 if state == 1 else 1
        code.append(state)
        timestamp.append(counter)
        counter += .5
        code.append(state)
        timestamp.append(counter)

      elif bit == 1:
        code.append(state)
        timestamp.append(counter)
        counter += .5
        code.append(state)
        timestamp.append(counter)

      state = -1 if state == 1 else 1
      code.append(state)
```

```
36      timestamp.append(counter)
37      counter += .5
38      code.append(state)
39      timestamp.append(counter)
40
41    return code, timestamp
42
43
44  def plot(self):
45        fig, axs = plt.subplots(1)
46
47        y_axis = self.code[0]
48        x_axis = self.code[1]
49
50        bit_code = [str(i) for i in self.bits]
51        plt.xticks(np.arange(len(bit_code)), bit_code)
52        plt.yticks([-1, 0, 1], ['-1', '', '1'])
53
54        plt.plot(x_axis, y_axis, 'black', linewidth=2)
55        plt.grid()
56        plt.show()
```