ISEL

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Módulo 2

Comunicação Digital

Realizado por:
Turma 43D

Grupo 12
Bruno Ferreira nº50499

Docente:
José David Antão

Índice:

Introdução:

Este trabalho tem como principal foco codificação de canal, em que este pede então simulação,deteção e correção (onde possível) de erros que estão presentes em sistemas de comunicação digital. São aplicadas várias técnicas, desde código de repetição, código de hamming , CRC e até mesmo uso de arduino com uso do algoritmo de IP-CheckSum.

Exercício 1:

Este exercício irá conter três tipos de códigos diferentes para vermos as suas diferenças, em que estes são, sem controlo de erros , código de repetição (3,1) e código de hamming (7,4). Em que estes serão aplicados de acordo com a troca de 1 bit com várias probabilidades diferentes de acontecer.

Sem Controlo:

Para esta parte, a maioria do código do módulo 1 exercício 6 foi reutilizado, a única diferença é que para este foi feito uma conversão de símbolos para binário , para ser mais fácil simular um bit de erro ao longo da informação manipulada.

Como o nome do exercício indica, para esta fase não é pedido nenhuma forma de controlo de erros, desta forma apenas é realizada a troca de bits de acordo com as probabilidades e a verificação de símbolos diferentes presentes nos ficheiros de entrada comparado com os respectivos de saída.

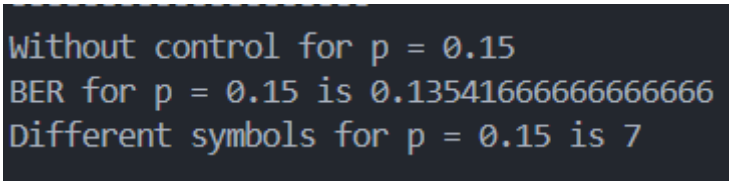

Figura 1: Exemplo do resultado para p=0.15 e um ficheiro com 12 símbolos(chars).

Código de repetição(3,1):

Aqui já nos é pedido um código para correção, sendo este o código de repetição (3,1) , em que este código através de um bit forma uma mensagem de 3 bits, e de forma padrão quando esta mensagem é 000 e 111 significa que o bit original é 0 e 1 respetivamente. Para os outros casos , ou seja por exemplo 001 aqui seria feito um voto por maioria em que este como **#0 > #1**, este iria supor que o bit original seria 0, podendo ser ou não.

Para aplicação deste algoritmo por cada bit recebido da mensagem este foi repetido três vezes, e cada bit repetido passava pela percentagem de troca de bit, desta forma simulando este código de repetição, resultando numa menor percentagem de erros do que o código sem controlo.

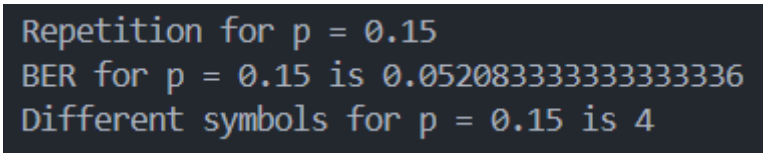

Figura 2: Resultado de código de repetição(3,1)

Código de Hamming (7,4):

Por fim temos o código de hamming (7,4) em que este nos indica que irá ter no total sete bits, em que destes sete, quatro serão a mensagem e três serão de paridade.

Desta forma a partir da informação recebida foi filtrada de sete em sete bits, em que os primeiros quatro foram selecionados para calcular os bits de paridade, sendo estes cálculos os mesmos fornecidos no PDF da disciplina fornecida pelo docente, ou seja:

$$p0 = m1 \oplus m2 \oplus m3$$
$$p1 = m0 \oplus m1 \oplus m3$$
$$p2 = m0 \oplus m2 \oplus m3$$

Por fim, a partir de todos os setes bits para verificar se existe erro ou não, basta verificar se a síndrome é diferente de zero, se for igual a zero significa que a informação está correta, se não contém pelo menos um bit errado, podendo ter mais que um. Para calcular a síndrome basta realizar o seguinte:

$$s0 = p0 \oplus d4$$
$$s1 = p1 \oplus d5$$
$$s2 = p2 \oplus d6$$

Caso fosse detectado um erro, a tabela das síndromes fornecida pelo docente era analisada e verifica-se a posição em que o erro está contido, após verificar a posição basta voltar a dar flip no bit de acordo com esta, caso exista dois bits errados na mesma subsequência de informação o código de hamming não consegue corrigir. A tabela de síndromes e posições foi colocada no código através de um dicionário.

Síndroma	Padrão de Erro	Observações
000	0000000	Ausência de erro
011	1000000	1.º bit em erro
110	0100000	2.º bit em erro
101	0010000	3.º bit em erro
111	0001000	4.º bit em erro
100	0000100	5.º bit em erro
010	0000010	6.º bit em erro
001	0000001	7.º bit em erro

Figura 3: Tabela de síndromes

```
Hamming for p = 0.15
BER for p = 0.15 is 0.08333333333333333
Different symbols for p = 0.15 is 5
```

Figura 4: Resultado utilizando código de hamming (7,4)

Conclusão exercício 1:

Para todos estes códigos referidos foram utilizados as mesmas probabilidades e o mesmo ficheiro de entrada, em que as probabilidades de erro foram: 5% , 15%, 35% e 80%. Já para o ficheiro de entrada foi utilizado o seguinte:

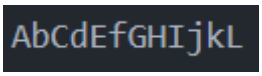

Figura 5: Ficheiro de teste de entrada

Em seguida podemos verificar exemplos de ficheiros de saída para cada código para as probabilidades 15% e 80%.

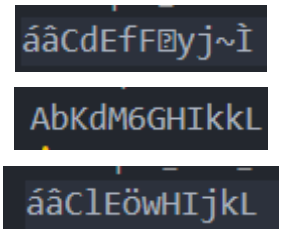

Figura 6: Exemplos de códigos de saída para sem controlo, repetição e hamming respectivamente com 15% de probabilidade de erro

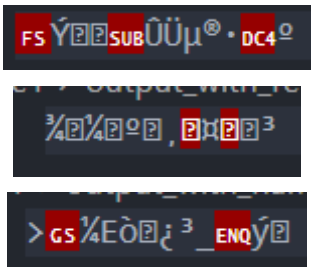

Figura 7: Exemplos de códigos de saída para sem controlo, repetição e hamming respectivamente com 80% de probabilidade de erro

Como podemos verificar, conseguimos ver quais são os códigos que foram mais eficientes na correção de erro, neste caso podemos concluir que o código de repetição foi o mais eficiente, isto acontece pois para o código de hamming este deve ter detectado muitos códigos com dois bits errados, mas apenas tem a capacidade de corrigir um destes, enquanto que o de repetição tem capacidade de corrigir caso exista dois erros, mas para percentagens maiores de erro podemos também verificar que o código de hamming conseguiu corrigir um símbolo enquanto que o de repetição não corrigiu nenhum.

Por fim podemos ver algumas tabelas que nos indica a quantidade de símbolos diferentes e o valor do BER confirmando assim o analisado:

		Sem controlo		
Probabilidade de erro	5%	15%	35%	80%
BER	0,03	0,1354	0,36458	0,7708
Simbolos diferentes	2	7	12	12

		Repetição(3,1)		
Probabilidade de erro	5%	15%	35%	80%
BER	0,0104	0,0520	0,291	0,8958
Simbolos diferentes	1	4	12	12

		Hamming(7,4)		
Probabilidade de erro	5%	15%	35%	80%
BER	0,0104	0,0833	0,5104	0,6875
Simbolos diferentes	1	5	12	12

Figura 8: Tabelas de resultados analisados

Exercício 2:

Este exercício consiste em simular erros em burst e aplicar deteção de erros com CRC numa sequência de bits.

Burst de erros:

Primeiramente para este exercício é pedido para simular erros entre comunicações através de burst de erros, este exercício é maioritariamente parecido com o que foi realizado anteriormente, no caso simulação de erros sem controlo do exercício um, a única diferença é que este em vez de ser a troca de um bit, será em burst, ou seja múltiplos bits seguidos serão trocados como podemos verificar no exemplo seguinte.

```
Simulation with burst error for p = 0.1 and Length of burst = 6
input:  [0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 0 0 0
 0 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 1 0 0 0 1 0 0 1 0 1 0 0
 1 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 0 1 0 1 0]
output:  [1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0
 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1
 0 1 1 1 1 1 0 0 1 0 0 1 0 1 1 0 0 0 1 0 1]
Number of different bits:  52
```

Figura 9: Exemplo do resultado de erro em burst

Como podemos verificar quando este encontra um erro , os próximos 6 bits foram trocados. Os tamanhos escolhidos para este exercício foram 2,4,6 e 8 bits.

CRC:

Para detecção de erros foi pedido para ser aplicado o CRC , importante lembrar que este código apenas detecta erros e não os corrige, também por recomendação do docente foi introduzido uma sequência de 64 bits e é utilizado CRC32. Para ocorrer a deteção de bits é realizado a seguinte equação:

$$resto\ [c(x)\ /\ g(x)]$$

Em que c(x) é o cálculo do crc que é realizado a partir da mensagem inicial e do polinômio gerador, e g(x) é o polinómio gerador neste caso CRC 32, caso este resto resulte em 0 significa que não existe síndrome e que não ocorreu erros , caso contrário existe, este código é extremamente eficiente com uma detecção de erro de cerca de 98%. Em seguida podemos ver uma deteção de erros e o polinómio gerador de CRC32:

CRC32: X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1

```
Simulation with CRC for p = 0.1 and Length of burst = 2
input:  [0 0 0 0 1 1 1 1 0 0 1 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1 0 0 1
 1 1 0 0 1 1 0 0 1 0 1 1 1 0 0 1 1 1 0 0 0 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0
 1 1 1 0 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0]
output:  [0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1 0 0 1
 0 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 0
 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0]
Number of different bits:  16
CRC Check:  False
```

Figura 10: Detecção de erros com CRC check , false indica que encontrou erro

Conclusão Exercício 2:

Existe uma situação em que o erro não é detectável através do CRC. Isso ocorre quando, mesmo com a presença de erro, o resto da divisão de polinômios resulta em zero. Em outras palavras, se o padrão de erro introduzido for um múltiplo exato do polinômio gerador, o CRC calculado não mudará, tornando o erro indetectável.

Exercício 3:

Este exercício tem um maior foco na utilização do arduino, em que primeiramente iremos apenas aplicar um algoritmo para ficarmos mais familiarizados com o arduino, após isto então iremos aplicar o algoritmo do IP-Checksum, neste exercício é nos pedido que seja uma ligação simplex em que o arduino é o emissor, e este irá passar números primos até N de acordo com o utilizador.

Sem IP-CheckSum:

Para o primeiro exercício temos de distinguir os códigos que serão utilizados, pois para além de linguagens diferentes (C para o arduino , python para o computador), ambos terão funcionalidades diferentes, onde o arduino primeiramente terá de se preparar para enviar informação, após estar pronto irá ler o número pedido pelo usuário e irá então processar um algoritmo simples de números primos, e cada número primo detectado será então enviado para o computador, em que este apenas tem de receber, processar e escrever na consola estes números.

Para garantir a funcionalidade correta , primeiramente foi executado apenas o código do arduino e verificado o resultado no serial monitor do arduino IDE, como podemos verificar a seguir:

```
Received N: 10
2
3
5
7
Done!
```

Figura 11: Execução do código apenas no arduino

Por fim é então feito o código em python com os ajustes necessários, dentre estes certos tempos de sleep foram aplicados para garantir que o arduino está pronto, no final temos este resultado:

```
Enter the value of N: 10
Waiting for prime numbers from the Arduino...
Received: Arduino is ready
Received: Received N: 10
Received: 2
Received: 3
Received: 5
Received: 7
Received: Done!
All prime numbers received.
```

Figura 12: Execução do código com arduino e computador.

Com IP-CheckSum:

Por fim , nesta fase final é apenas aplicado o algoritmo do IP-checksum , tanto no arduino , tanto no computador , isto porque ambos têm de calcular o IP-Checksum, desta forma o arduino também irá enviar o seu IP-Checksum para além do número primo correspondente, em seguida o computador irá calcular o checksum do número recebido, se ambos forem iguais não terá problemas, caso contrário existirá erros, como é um código simples muito dificilmente ocorrerá erros , então os erros serão forçados no computador, em que de cinco em cinco números primos recebidos este será incrementado para obrigar que o cálculo do checksum será diferente como podemos ver a seguir:

```
Enter the value of N: 30
Waiting for prime numbers from the Arduino...
Received: Arduino is ready
Received: Received N: 30
Received: Number: 2 checksum: 65533
Received: 2 with valid checksum 65533
Received: Number: 3 checksum: 65532
Received: 3 with valid checksum 65532
Received: Number: 5 checksum: 65530
Received: 5 with valid checksum 65530
Received: Number: 7 checksum: 65528
Received: 7 with valid checksum 65528
Received: Number: 11 checksum: 65524
Checksum mismatch for number 12: received 65524, calculated 65523
Received: Number: 13 checksum: 65522
Received: 13 with valid checksum 65522
Received: Number: 17 checksum: 65518
Received: 17 with valid checksum 65518
Received: Number: 19 checksum: 65516
Received: 19 with valid checksum 65516
Received: Number: 23 checksum: 65512
Received: 23 with valid checksum 65512
Received: Number: 29 checksum: 65506
Checksum mismatch for number 30: received 65506, calculated 65505
Received: Done!
All prime numbers received.
```

Figura 13: Execução do código com IP-CheckSum

Como referido anteriormente tanto no arduino como no computador foi aplicado o mesmo algoritmo de checksum , em que este basicamente separa o valor em blocos de 16 bits, após isso irá somar todos estes blocos e inverter este valor, resultando assim o valor do ip-checksum.

Conclusão:

Com este trabalho conseguimos perceber a diferença entre os vários tipos de algoritmos diferentes para detecção e correção de erros que são utilizados no dia-a-dia, também nos trouxe um certo conhecimento de como trabalhar com um componente exterior, neste caso o arduino. Em certos exercícios foram encontrados alguns problemas , especialmente com o arduino por ser algo diferente, mas nada muito preocupante.

Referências:

ISEL moodle: Comunicação digital - LEIC - 2324SV : https://2324moodle.isel.pt