

Jogo da Roleta

(*Roulette Game*)

Autores:

Manuel Henriques Nº47202

Tiago Pardal Nº46206

Miguel Almeida Nº47249

Projeto de

Laboratório de Informática e Computadores 2020 / 2021 inverno

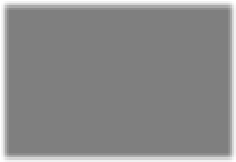
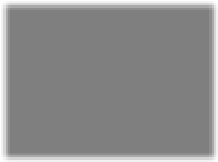
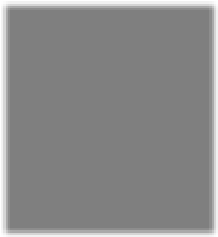
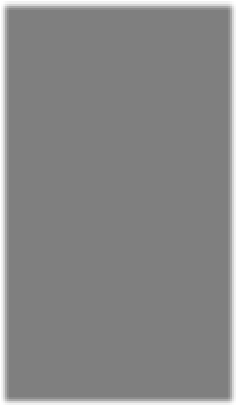
2020/2021 inverno

Jogo da Roleta (*Roulette Game*)

1. **INTRODUÇÃO 1**
2. **ARQUITETURA DO SISTEMA 2**
3. **INTERLIGAÇÕES ENTRE O HW E SW 3**
4. **CÓDIGO JAVA DA CLASSE *HAL* 4**
5. **CÓDIGO JAVA DA CLASSE *KBD* 5**
6. **CÓDIGO JAVA DA CLASSE *LCD* 6**
7. **CÓDIGO JAVA DA CLASSE *TUI* 8**
8. **CÓDIGO JAVA DA CLASSE *FILEACCESS* 9**
9. **CÓDIGO JAVA DA CLASSE *STATISTICS* 10**
10. **CÓDIGO JAVA DA CLASSE *ROULETTE DISPLAY* 11**
11. **CÓDIGO JAVA DA CLASSE *M* 12**
12. **CÓDIGO JAVA DA CLASSE *ROULETTEGAME - APP* 13**

**1 Introdução**

Neste projeto implementa-se o jogo da Roleta (*Roulette Game*), no qual a roleta compreende números entre 0 e 9, um jogador realiza apostas premindo as teclas de um teclado correspondentes aos números em que pretende apostar. Por cada aposta é debitado um crédito ao saldo acumulado do jogador, podendo o jogador apostar mais do que um crédito num mesmo número. Os créditos são obtidos pela introdução de moedas no moedeiro, este só aceita moedas de 1.00€, que corresponde a dois créditos. O sistema que implementa o jogo será constituído por: um PC (*Control*); um teclado de 12 teclas; um moedeiro (*Coin Acceptor*); um mostrador *Liquid Cristal Display* (*LCD*) de duas linhas com 16 caracteres; um mostrador da roleta (*Roulette Display*) e uma chave de manutenção designada por *M*, para colocação do sistema em modo de Manutenção. Na Figura 1 apresenta-se o diagrama de blocos do jogo da Roleta.



*Control*

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**\***

**0**

**#**

*LCD*

M

*Coin*

*Acceptor*

*Roulette*

*Display*

Figura 1 – Diagrama de blocos do jogo da Roleta (*Roulette Game*)

Sobre o sistema proposto podem realizar-se as seguintes ações em modo de Jogo:

* **Jogo –** O jogo inicia-se quando é premida a tecla ‘\*’ e existem créditos disponíveis. Utilizando as teclas numéricas (09) realizam-se as apostas, retirando-se um crédito ao saldo do jogador por cada aposta realizada. O jogador termina as apostas premindo a tecla ‘#’, o que dá início ao sorteio. Durante um tempo aleatório, o sistema simula o girar da Roleta no *Roulette Display*, permitindo ainda realizar apostas até 5 segundos antes desta parar. Ao parar a Roleta, o número sorteado é apresentado no *Roulette Display* e os créditos obtidos na jogada são apresentados no LCD. Os créditos obtidos são acumulados após 5 segundos ao saldo do jogador, também apresentado no LCD.

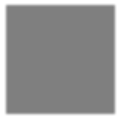
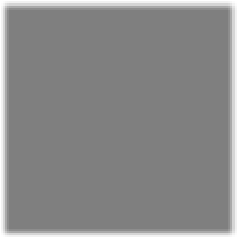
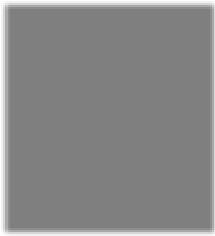
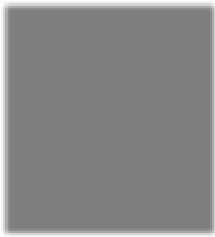
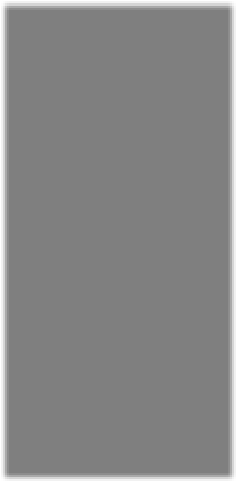
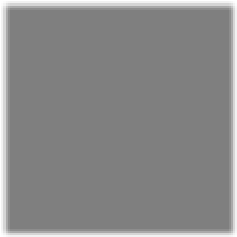
No modo Manutenção podem realizar-se as seguintes ações sobre o sistema:

* **Teste** – Premindo a tecla ‘\*’ inicia-se um jogo sem créditos e sem contabilizar os números sorteados.
* **Consultar os contadores de moedas e jogos** – Carregando na tecla ‘#’ permite-se a listagem dos contadores de moedas e jogos realizados.
* **Iniciar os contadores de moedas e jogos –** Premindo a tecla ‘#’ e em seguida a tecla ‘\*’, o sistema de gestão coloca os contadores de moedas e jogos a zero, iniciando um novo ciclo de contagem.
* **Consultar a lista de números sorteados** – Carregando na tecla ‘0’ permite-se a listagem dos números sorteados.
* **Iniciar a lista de números sorteados** **–** Premindo a tecla ‘0’ e em seguida a tecla ‘\*’, o sistema inicia um novo ciclo de estatística de números sorteados.

**Desligar –** Permite desligar o sistema, que encerra apenas após a confirmação do utilizador, ou seja, o programa termina e as estruturas de dados, contendo a informação dos contadores e da Lista de Números Sorteados, são armazenadas de forma persistente em dois ficheiros de texto, por linha e com os campos de dados separados por “;”. O primeiro ficheiro deverá conter o número de jogos realizados e o número de moedas guardadas no cofre do moedeiro. O segundo ficheiro deverá conter a Lista de Números Sorteados, que contém o número de saídas e os prémios atribuídos por cada número. Os dois ficheiros devem ser carregados para o sistema no seu processo de arranque.

**2 Arquitetura do sistema**

O sistema é implementado numa solução híbrida de *hardware* e *software*, como apresentado no diagrama de blocos da Figura 2. A arquitetura proposta é constituída por quatro módulos principais: *i*) um leitor de teclado, designado por *Keyboard Reader*; *ii*) um módulo de interface com o *LCD* e com o mostrador da roleta, designado por *Serial Output Controller* (*SOC*); *iii*) um moedeiro, designado por *Coin Acceptor*; e *iv*) um módulo de controlo, designado por *Control*. Os módulos *i*) e *ii*) são implementados em *hardware,* o moedeiro será simulado utilizando um interruptor e um LED, enquanto o módulo de controlo é implementado em *software* escrito usando linguagem Java e executado num PC.



*Control*

*LCD*

*Display*

*Roulette*

M

*Keyboard*

*Reader*

*Serial*

*Output*

*Controller*

*Coin*

*Acceptor*

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**\***

**0**

**#**

Figura 2 – Arquitetura do sistema que implementa o jogo da Roleta (*Roulette Game*)

O módulo *Keyboard Reader* é responsável pela descodificação do teclado matricial de 12 teclas, determinando qual a tecla pressionada e disponibilizando o seu código, com quatro bits, ao módulo *Control*. Caso este não esteja disponível para o receber imediatamente, o código da tecla é armazenado internamente, até ao limite de dois códigos. O módulo *Control* processa os dados e envia a informação a apresentar no *LCD* através do módulo *SOC*. O *Roulette Display* é atuado pelo módulo *Control*, através do módulo *SOC*. Por razões de ordem física, e por forma a minimizar o número de fios de interligação, a comunicação entre o módulo *Control* e o módulo *SOC* é realizada através de um protocolo série.

A implementação do módulo *Control* foi realizada em *software*, usando a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 3.

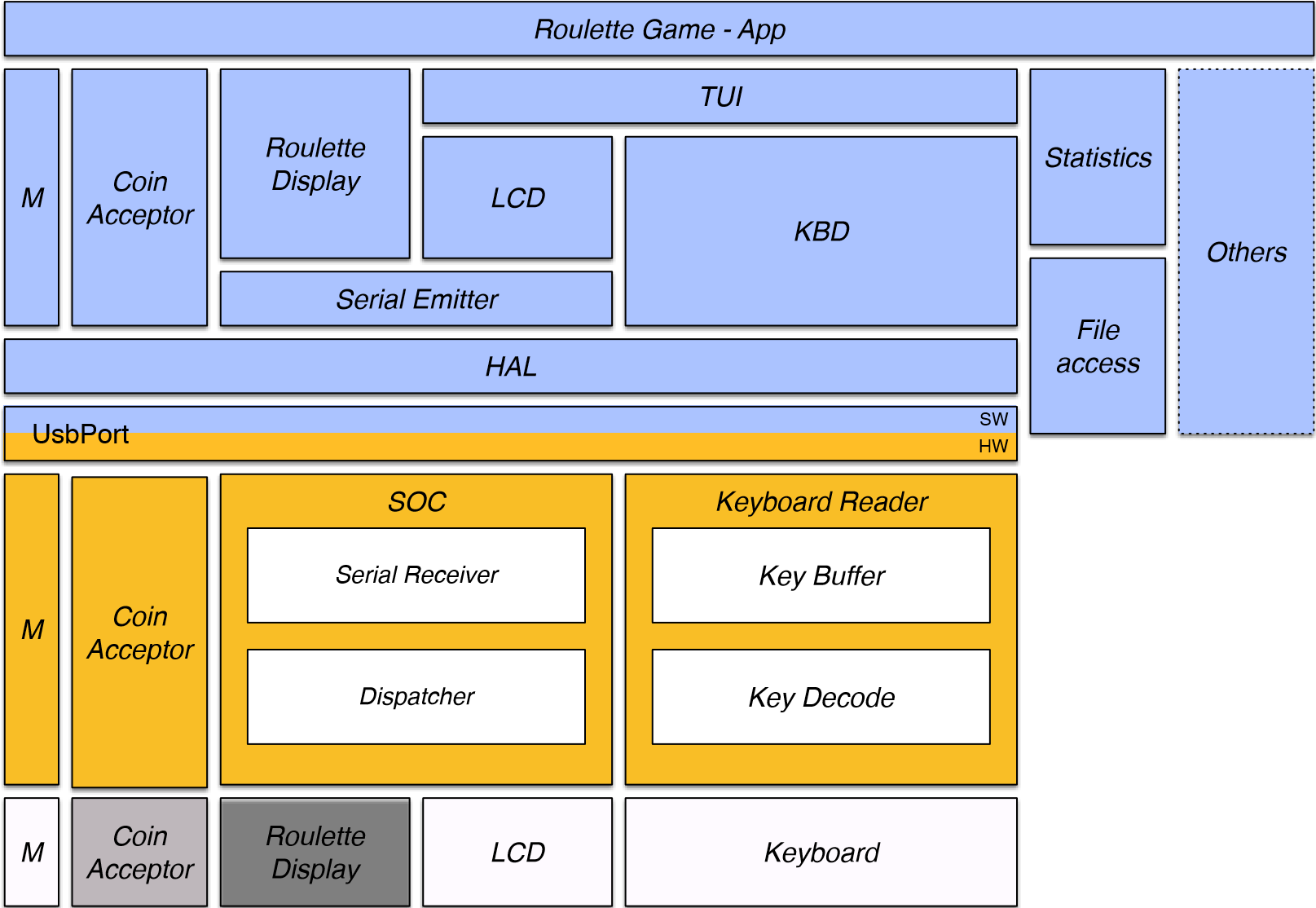


Figura 3 – Diagrama lógico do Jogo da Roleta (*Roulette Game*)

# A. Interligações entre o HW e SW

O módulo *Keyboard Reader* implementado é constituído por dois blocos principais: o descodificador de teclado (*Key Decode*); e o bloco de armazenamento e de entrega ao consumidor (designado por *Key Buffer*), conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em *software*, é a entidade consumidora.

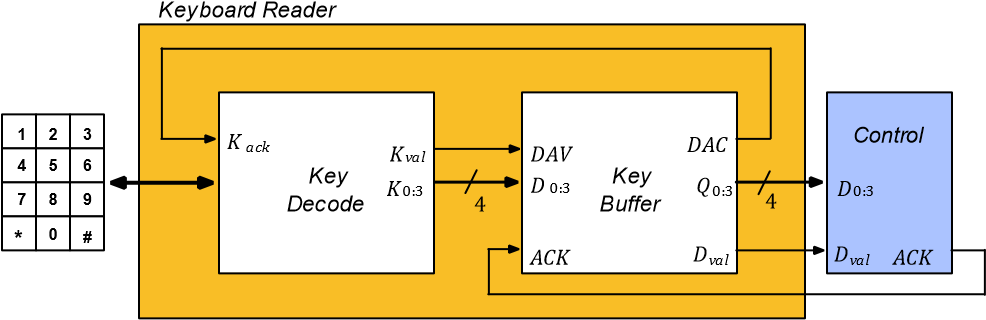


Figura 1 – Diagrama de blocos do módulo *Keyboard*

*Reader*

# 1 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x3 por *hardware*, sendo constituído por três sub-blocos: um teclado matricial de 4x3; o bloco *Key Scan,* responsável pelo varrimento do teclado; e o bloco *Key Control*, que realiza o controlo do varrimento e o controlo de fluxo, conforme o diagrama de blocos representado na Figura 2a. O controlo de fluxo de saída do bloco *Key Decode* (para o módulo *Key Buffer*), define que o sinal *Kval* é ativado quando é detetada a pressão de uma tecla, sendo também disponibilizado o código dessa tecla no barramento *K0:3*. Apenas é iniciado um novo ciclo de varrimento ao teclado quando o sinal *Kack* for ativado e a tecla premida for libertada.

O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado na Figura 3, uma vez que consideramos que o uso de apenas um contador de dois bits tornaria o circuito mais simples sendo também necessários menos clocks para efetuar o varrimento do mesmo, sendo assim a solução mais rápida a efetuar a leitura de uma tecla.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. Conforme a solução apresentada, o Key Controla aguarda que uma tecla do Keyboard seja pressionada e, após verificar que DAC já não se encontra ativo, liberta o sinal Kval para assinalar que possui valores validos para leitura. Após isso, aguarda o sinal DAC a informar que os dados que está a disponibilizar foram recebidos e assim, após DAC se desligar novamente, recomeça o processo.

A descrição hardware do bloco *Key Decode* em CUPL segue em anexo no documento XXX

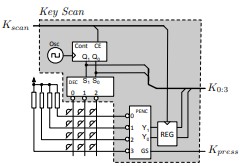


Figura 3 - Diagrama de blocosdo bloco *Key Scan*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo C. Vale notar que durante o decorrer do desenvolvimento deste modulo, optamos por deixar o clock do Counter invertido relativamente ao do KeyControl como modo de evitar possíveis conflitos. Relativamente ás resistências usadas na implementação do componente hardware do keyboard, usamos as que foram fornecidas e que correspondei ás listadas a documentação da peça.

# 2 Key Buffer

O módulo *Key* *Buffer* implementa uma estrutura de armazenamento de dados, com capacidade de uma palavra de quatro bits. A escrita de dados no *Key Buffer* inicia-se com a ativação do sinal *DAV* (*Data Available*) pelo sistema produtor, neste caso pelo *Key Decode*, indicando que tem dados para serem armazenados. Logo que tenha disponibilidade para armazenar informação, o *Key Buffer* escreve os dados *D0:3* em memória. Concluída a escrita em memória, ativa o sinal *DAC* (*Data Accepted*) para informar o sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal *DAV* ativo até que *DAC* seja ativado. O *Key Buffer* só desativa *DAC* depois de *DAV* ter sido desativado.

A implementação do *key Buffer* deverá ser baseada numa máquina de controlo (*Key Buffer Control*) e num registo externo (*Output Register*), o diagrama de blocos

Figura 5 – Diagrama de blocos do *Key Buffer*

O bloco *Key Buffer Control* do *Key Buffer* é também responsável pela interação com o sistema consumidor, neste caso o módulo *Control*. O *Control* quando pretende ler dados do *Key Buffer*, aguarda que o sinal *Dval* fique ativo, recolhe os dados e ativa o sinal *ACK* indicando que estes já foram consumidos.

O *Key Buffer Control*, logo que o sinal *ACK* fique ativo, deve invalidar os dados baixando o sinal *Dval*, só deverá voltar a armazenar uma nova palavra depois do *Control* ter desativado o sinal *ACK*.

O bloco *Key Buffer Control* foi implementado de acordo com o diagrama de blocos representado na Figura 6. Conforme as instruções do enunciado, o modulo Key Buffer Control, após receber o sinal que existe data para ser lida, grava essa mesma data no Register e de seguida encaminhaa para o Control.

A descrição hardware do bloco *Key Buffer Control* em CUPL encontra-se no Anexo C.

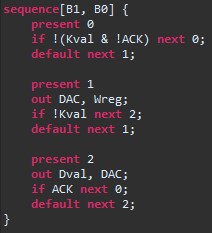


Figura 6 - Máquina de estados do bloco *Key Buffer*

*Control*

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo C. **[Justificar as opções tomadas, como por exemplo, as frequências de relógio, etc.]**

**3 Interface com o *Control***

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 7.

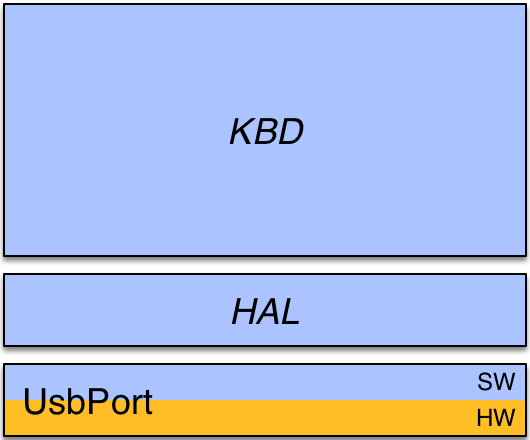


Figura 7 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos D e E, respetivamente.

**3.1 Classe *HAL***

Começando pelo método init(), este é responsável por iniciar a classe colocando à saída do USBPort o valor 0x00, já o método isBit(int mask) testa o bit selecionado pela mask no porto de entrada do USBPort através de um teste lógico entre mask e o valor à entrada, retornando true caso o mesmo bit esteja ligado ou a 1 lógico. O método readBits(int mask) trata de ler os bits do porto de entrada assinalados pelos respetivos bits da mask, sendo isto feito através da operação lógica entre o porto de entrada e a mask. O método setBits(int mask) afeta o porto de saída com o resultado da operação lógica entre a mask ou o campo lastValue guardado no mesmo campo lastValue, enquanto o método clrBits(int mask) afeta o porto de saída com o resultado da operação lógica entre a negação de mask e o campo lastValue guardando o resultado no mesmo campo lastValue. O último dos métodos que foi realizado nesta classe foi o método writeBits(int mask, int value) porque implicava o uso dos métodos setBits e clrBits, então, com estes métodos já feitos, o método writeBits implicava limpar os bits marcados por mask e escrever value por cima, sendo assim, fica resolvido através da chamada ao método clrBits(mask) e da chamada de setBits(value & mask)

**3.2 Classe *KBD***

Nesta classe KBD ficou ao nosso encargo realizar a composição dos três métodos encarregados à mesma, sendo estes explicados respetivamente em baixo no relatório, o método init(), o método getKey() e o método waitKey(long timeout). O método init() inicia a classe chamando o método init() da classe HAL anteriormente explicado e inicia o processo de procura de teclas pressionadas pelo utilizador chamando o método waitKey() afetando o campo Key da classe com o resultado. getKey() retorna o char correspondente ao valor lido do porto de entrada nos bits representados por mask chamando o readBits(0x0F) da classe HAL para tal efeito ou NONE caso não esteja nenhum char associado ao valor lido. O método waitKey(long timeout) é o método que trata de fazer a espera para ler a tecla pressionada e, para isso, espera também que o bit de mask 8 esteja ligado, simbolizando este a espera pelo sinal de Dval, só então, após o teste, ele retorna o método getKey().

# 1. Descrição CUPL do *KeyBoard Reader*

**Name** template ;

**PartNo** 00 ;

**Date** 10/10/10 ;

**Revision** 01 ;

**Designer** Engineer ;

**Company** None ;

**Assembly** None ;

**Location** ;

**Device** v750c ;

/\* Start Here \*/

**PIN** 1 = CLK;

**PIN** 2 = ACK;

**PIN** [3..6] = [I0..3]; /\*Inputs do PENC \*/

**PIN** [14..16] = [DEC0..2] ;

**PIN** 18 = A0;

**PIN** 19 = Kscan;

**PIN** [20..23] = [Q0..Q3];

**PIN** 17 = Dval;

**PINNODE** 28 = K2;

**PINNODE** 27 = K3;

**PINNODE** 26 = B1;

**PINNODE** 33 = K1;

**PINNODE** 30 = A1;

**PINNODE** 31 = B0;

**PINNODE** 32 = K0;

/\* \*\*\*\*\* COUNTER \*\*\*\*\* \*/

[K0..1].CK =!CLK & Kscan;

[K0..1].sp = 'b'0;

K0.t = 'b'1;

K1.t = 'b'1 & K0;

/\* \*\*\*\*\* DECODER \*\*\*\*\* \*/

DEC0 = K1 # K0;

DEC1 = K1 # !K0;

DEC2 = !K1 # K0;

/\* \*\*\*\*\*\* PENC & REGISTER \*\*\*\*\*\* \*/

[K2..3].CK = !Kscan;

[K2..3].sp = 'b'0;

K2.d = (!I1 & I0 # !I3 & I2 & I0);

K3.d = ((!I3 # !I2) & I1 & I0);

Kpress = !I0 # !I1 # !I2 # !I3;

/\* \*\*\*\*\*\* Key Control \*\*\*\*\*\* \*/

[A0..A1].AR ='b'0;

[A0..A1].sp = 'b'0;

[A0..A1].ck = CLK;

**sequence**[A0, A1]{

**present** 0

**OUT** Kscan;

**if** Kpress **next** 1;

**default** **next** 0;

**present** 1

**OUT** Kval;

**if** DAC **next** 2;

**default** **next** 1;

**present** 2

**if** !Kpress & !DAC **next** 0;

**default** **next** 2;

}

/\* Key Buffer Control \*/

[B0..B1].CK = !CLK;

[B0..B1].sp = 'b'0;

**sequence**[B1, B0] {

**present** 0

**if** Kval & !ACK **next** 1;

**default** **next** 0;

**present** 1

**out** DAC, Wreg;

**if** !Kval **next** 2;

**default** **next** 1;

**present** 2

**out** Dval;

**if** ACK **next** 0;

**default** **next** 2;

}

/\* Output Register \*/

[Q0..3].CK = Wreg;

[Q0..3].sp = 'b'0;

[Q0..3].d = [K0..3];

**2. Esquema elétrico do módulo *Keyboard Reader***

**B. *HAL***

import isel.leic.UsbPort;  
  
*// Virtualiza o acesso ao sistema UsbPort*public class HAL {  
  
 private static int *lastValue*;  
  
 public static void main(String[] args) {  
 *init*();  
 }  
  
 *// Inicia a classe* public static void init() { *out*(*lastValue* = 0); }  
  
 *// Retorna true se o bit tiver o valor lógico ‘1’* public static boolean isBit(int mask){  
 return *readBits*(mask) !=0;  
 }  
  
 *// Retorna os valores dos bits representados por mask presentes no UsbPort* public static int readBits(int mask) {  
 return (~UsbPort.*in*() & mask);  
 }  
  
 *// Escreve nos bits representados por mask o valor de value* public static void writeBits(int mask, int value) {  
 *clrBits*(mask);  
 *setBits*(value & mask);  
 }  
  
 *// Coloca os bits representados por mask no valor lógico ‘1’* public static void setBits(int mask) {  
 *out*(*lastValue* |= mask);  
 }  
  
 *// Coloca os bits representados por mask no valor lógico ‘0’* public static void clrBits(int mask){  
 *out*(*lastValue* &= ~mask);  
 }  
  
 private static void out(int val){  
 UsbPort.*out*(~val);  
 }  
}

**C. *KBD***

import isel.leic.utils.Time;  
  
public class KBD { *// Ler teclas. Métodos retornam ‘0’..’9’,’#’,’\*’ ou NONE.* private final static int *KVAL\_MASK* = 0x10; *// 0001 0000* private final static int *ACK\_MASK* = 0x80; *// 1000 0000* private final static int *KBD\_MASK* = 0x0F; *// 0000 1111* private final static char[] *keyboard* = {'1','4','7','\*','2','5','8','0','3','6','9','#'};  
   
 public static void main(String[] args) {  
 HAL.*init*();  
 *init*();  
 while(true) System.*out*.println(*waitKey*(5000000));  
 }

*// Inicia a classe* public static void init() { }  
  
 *// Retorna de imediato a tecla premida ou NONE se não há tecla premida.* public static char getKey() {  
 char key = 0;  
 if(HAL.*isBit*(*KVAL\_MASK*)) {  
 key = *keyboard*[*KBD\_MASK* & HAL.*readBits*(*KBD\_MASK*)];  
 HAL.*setBits*(*ACK\_MASK*);  
 while (HAL.*isBit*(*KVAL\_MASK*)) ;  
 HAL.*clrBits*(*ACK\_MASK*);  
 }return key;  
 }  
  
 *// Retorna quando a tecla é premida ou NONE após decorrido ‘timeout’ milisegundos.* public static char waitKey(long timeout) {  
 timeout += System.*currentTimeMillis*();  
 char key;  
 do {  
 key = *getKey*();  
 if (key != 0) return key;  
 } while (System.*currentTimeMillis*() < timeout);  
 return 0;  
 }  
}

**D. *SerialEmitter***   
  
public class SerialEmitter { *// Envia tramas para o módulo Serial Receiver.* public enum Destination {*RDisplay*,*LCD*};  
 private static int *SDX*;  
 private static final int *SOCSEL\_MASK* = 0x08;  
 private static final int *SDX\_MASK* = 0x02;  
 private static final int *CLOCK\_MASK* = 0x04;  
  
 *// Inicia a classe* public static void init(){  
 HAL.*writeBits*(0x0F, 0);  
 }  
  
 *// Envia uma trama para o Serial Receiver identificando o destino em addr e os bits de dados em‘data’.* public static void send(Destination addr, int data){  
 *init*();  
 int p = 0;  
 int value;  
 int LnD = (addr.ordinal() == Destination.*RDisplay*.ordinal()) ? 0x00 : 0x01;  
 *SDX* = (data << 1) | LnD;  
  
 HAL.*setBits*(*SOCSEL\_MASK*);  
 for (int i = 0; i <= 5; ++i){  
 value = *SDX* & 0x01;  
 if (value == 0x01){  
 HAL.*setBits*(*SDX\_MASK*);  
 ++p;  
 }  
 *SCLK*();  
 HAL.*clrBits*(*SDX\_MASK*);  
 *SDX* = *SDX* >> 1;  
 }  
  
 if ( p % 2 != 0) HAL.*setBits*(*SDX\_MASK*);  
 *SCLK*();  
 HAL.*clrBits*(*SDX\_MASK*);  
 HAL.*clrBits*(*SOCSEL\_MASK*);  
 }  
 private static void SCLK(){  
 HAL.*setBits*(*CLOCK\_MASK*);  
 HAL.*clrBits*(*CLOCK\_MASK*);  
 }  
}

**E. *LCD***

import isel.leic.utils.Time;  
  
public class LCD { *// Escreve no LCD usando a interface a 4 bits.* public static final int *LINES* = 2, *COLS* = 16; *// Dimensão do display.* private static final int *NIBBLE\_SIZE* = 4, *BYTE\_SIZE* = 8;  
 public static final int *NIBBLE\_MASK* = 0x0f, *NIBBLE\_MASK\_SIZE* = 0x10;  
  
 private static final int *FUNCTION\_SET\_TO8BIT* = 0x03, *FUNCTION\_SET\_TO4BIT* = 0x02;  
  
 private static final int *FUNCTION\_SET\_2LINES* = 0x28;  
 private static final int *DISPLAY\_OFF* = 0x08;  
 private static final int *CLEAR\_DISPLAY* = 0x01;  
 private static final int *ENTRY\_MODE\_SET\_DIR\_RIGHT* = 0x06;  
 private static final int *SET\_CGRAM\_ADRESS* = 0x40;  
  
 private static final int *FIRST\_INIT\_TIME* = 15, *SECOND\_INIT\_TIME* = 5, *THIRD\_INIT\_TIME* = 1, *WRITEBYTE\_SLEEP\_TIME* = 10;  
  
 private static final int *LINE0* = 0x00, *LINE1* = 0x40;  
 private static final int *CURSOR\_ON* = 0x0f, *DISPLAY\_ON* = 0x0f;  
 private static final int *CURSOR\_OFF* = 0x0c;  
 private static final int *TIME\_TO\_WRITE\_EACH\_CHAR\_ANIMATION* = 25;  
  
 private static final boolean *SERIAL\_INTERFACE* = true; *// Define se a interface com o LCD é série ou paralela  
  
 // Envia a sequência de iniciação para comunicação a 4 bits.* public static void main(String[] args) {  
 HAL.*init*();  
 *init*();  
 *write*(" Roulette Game ");  
 }

public static void init() {  
 Time.*sleep*(*FIRST\_INIT\_TIME*);  
 *writeNibble*(false,*FUNCTION\_SET\_TO8BIT*);  
 Time.*sleep*(*SECOND\_INIT\_TIME*);  
 *writeNibble*(false,*FUNCTION\_SET\_TO8BIT*);  
 Time.*sleep*(*THIRD\_INIT\_TIME*);  
 *writeNibble*(false,*FUNCTION\_SET\_TO8BIT*);  
 *writeNibble*(false,*FUNCTION\_SET\_TO4BIT*);  
 *writeCMD*(*FUNCTION\_SET\_2LINES*);  
 *writeCMD*(*DISPLAY\_OFF*);  
 *writeCMD*(*CLEAR\_DISPLAY*);  
 *writeCMD*(*ENTRY\_MODE\_SET\_DIR\_RIGHT*);  
 *writeCMD*(*DISPLAY\_ON*);  
 }  
  
 *// Escreve um nibble de comando/dados no LCD em paralelo* private static void writeNibbleParallel(boolean rs, int data) {  
 if(rs){  
 HAL.*writeBits*(*NIBBLE\_MASK\_SIZE*,0x10);  
 }else{  
 HAL.*writeBits*(*NIBBLE\_MASK\_SIZE*,0);  
 }  
 HAL.*writeBits*(0x20,0x20);  
 HAL.*writeBits*(*NIBBLE\_MASK*,data);  
 HAL.*writeBits*(0x20,0);  
 }  
  
 *// Escreve um nibble de comando/dados no LCD em série* private static void writeNibbleSerial(boolean rs, int data) {  
 data &= *NIBBLE\_MASK*;  
 data <<= 1;  
 if (rs) data |= 0x1;  
 SerialEmitter.*send*(SerialEmitter.Destination.*LCD*, data);  
 }  
  
 *// Escreve um nibble de comando/dados no LCD* private static void writeNibble(boolean rs, int data) {  
 if(!*SERIAL\_INTERFACE*)  
 *writeNibbleParallel*(rs,data);  
 else *writeNibbleSerial*(rs,data);  
 }  
  
 *// Escreve um byte de comando/dados no LCD* private static void writeByte(boolean rs, int data) {  
 int highData = ((data >>> *NIBBLE\_SIZE*) & *NIBBLE\_MASK*); *//Parte Alta do data* int lowData = (data & *NIBBLE\_MASK*); *//Parte Baixa do data  
 writeNibble*(rs,highData);  
 *writeNibble*(rs,lowData);  
 Time.*sleep*(*WRITEBYTE\_SLEEP\_TIME*);  
 }  
  
 *// Escreve um comando no LCD* private static void writeCMD(int data) {  
 *writeByte*(false,data);  
 }  
  
 *// Escreve um dado no LCD* private static void writeDATA(int data) {  
 *writeByte*(true,data);  
 }  
  
 *// Escreve um caráter na posição corrente.* public static void write(char c) {  
 *writeDATA*(c);  
 Time.*sleep*(*TIME\_TO\_WRITE\_EACH\_CHAR\_ANIMATION*);  
 }  
  
 *// Escreve uma string na posição corrente.* public static void write(String txt) {  
 for (int i = 0; i < txt.length(); i++) {  
 *write*(txt.charAt(i));  
 }  
 }  
  
 *// Envia comando para posicionar cursor (‘lin’:0..LINES-1 , ‘col’:0..COLS-1)* public static void cursor(int lin, int col) {  
 *writeCMD*(0x80 + (lin==1?*LINE1*:*LINE0*) + col);  
 }  
  
 *// Envia comando para limpar o ecrã e posicionar o cursor em (0,0)* public static void clear() {  
 *writeCMD*(*CLEAR\_DISPLAY*); *// Clear Display e Coloca o cursor na posicao 0,0* }  
  
 public static void saveCustomChar(int charNum){  
 *writeCMD*( *SET\_CGRAM\_ADRESS*+(charNum\**BYTE\_SIZE*));  
 for(int i = 0; i < *BYTE\_SIZE*; i++) {  
*writeByte*(true,RouletteGameApp.*specialChar*[i+(charNum\**BYTE\_SIZE*)]);  
 }  
 }  
  
 public static void customChar(int charNum){  
 *writeByte*(true,charNum);  
 }  
  
 public static void customChar(int charNum,int line,int col){  
 *cursor*(line,col);  
 *writeByte*(true,charNum);  
 }  
  
 public static void displayCursor(boolean cursor){  
 if (cursor) *writeCMD*(*CURSOR\_ON*);  
 else *writeCMD*(*CURSOR\_OFF*);  
 }  
}

**F. *TUI***

public class TUI {  
  
 private static final int *OFFSET* = -1;  
  
 public static void init(){  
 *//grava carateres especiais* LCD.*saveCustomChar*(0);  
 LCD.*saveCustomChar*(1);  
 LCD.*saveCustomChar*(2);  
 *displayCursor*(false);  
 }  
  
 public static void write(String text){  
 LCD.*write*(text);  
 *displayCursor*(false);  
 }  
  
 public static void write(String text, int line, int col){  
 LCD.*cursor*(line,col);  
 LCD.*write*(text);  
 *displayCursor*(false);  
 }  
  
 public static void writeOnCenter(String txt, int line){  
 *setCursor*(line,0);  
 int i = 0;  
 for(;LCD.*COLS* >= txt.length()+i\*2;++i);  
 if(LCD.*COLS* == txt.length()+i\*2) TUI.*write*(txt,line,i);  
 else TUI.*write*(txt,line,i+*OFFSET*); *//caso valor seja impar offset txt +1 para a direita* }  
  
 public static void clearScreen(){  
 LCD.*clear*();  
 }  
  
 public static int digitDim(int digit){  
 int spaces;  
 if(digit < 10) spaces = 1;  
 else if(digit < 100) spaces = 2;  
 else spaces = 3;  
 return spaces;  
 }  
  
 public static void setCursor(int line, int col){  
 LCD.*cursor*(line, col);  
 }  
  
 public static void displayCursor(boolean cursor){  
 LCD.*displayCursor*(cursor);  
 }  
}

**G. *FileAccess***

import java.io.\*;  
import java.util.ArrayList;  
import java.util.InputMismatchException;  
import java.util.Scanner;  
  
public class FileAccess {  
 public static ArrayList<String> load(String fileName, int initialCapacity) {  
 if (initialCapacity <= 2) initialCapacity = 2;  
 ArrayList<String> SL = new ArrayList<>(initialCapacity);  
 Scanner in = null;  
 try {  
 in = new Scanner(new FileInputStream(fileName));  
 while (in.hasNextLine()) {  
 SL.add(in.nextLine());  
 }  
 } catch (FileNotFoundException | InputMismatchException e) {  
 System.*out*.println("Error loading file \"" + fileName + "\":\n" + e.getMessage());  
 } finally {  
 if (in != null) in.close(); *// Close the file* }  
 return SL;  
 }  
  
 public static void save(String fileName, ArrayList<String> SL) {  
 BufferedWriter out = null;  
 try {  
 out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(fileName)));  
 if (SL != null)  
 for (String s : SL) {  
 out.write(s);  
 out.newLine();  
 }  
 } catch (IOException e) {  
 System.*out*.println("Error saving file \"" + fileName + "\":\n" + e.getMessage());  
 }  
 try {  
 if (out != null) {  
 out.flush();  
 out.close(); *// Close the file* }  
 } catch (IOException e) {  
 System.*out*.println("Error saving file \"" + fileName + "\":\n" + e.getMessage());  
 }  
 }  
}

**H. *Stastiscs***

import java.util.ArrayList;  
  
class Statistics {  
  
 private final static String *STATISTICSFILENAME* = "Statistics.txt";  
 private final static String *ROULETTE\_STATSFILENAME* = "Roulette\_Stats.txt";  
 static int *games*;  
 static int *coins*;  
 private static String *betsWon* = "";  
 private static String *betsWonValue* = "";  
  
 public static void init(){  
 *load*();  
 }  
  
 public static void addGame(){  
 *games*++;  
 *save*();  
 }  
  
 public static void addCoins(int c){  
 *coins*+=c;  
 *save*();  
 }  
 public static int getGames(){return *games*;}  
 public static int getCoins(){return *coins*;}  
  
 public static void clear(){  
 *coins*=0;  
 *games*=0;  
 *//TODO* }  
  
 *//Carrega estatisticas a partir de um ficheiro* private static void load(){  
 *clear*();  
  
 ArrayList<String> SL=FileAccess.*load*(*STATISTICSFILENAME*,2);  
 if (SL.size()>=2) {  
 *games* = Integer.*parseInt*(SL.get(0) );  
 *coins* = Integer.*parseInt*(SL.get(1) );  
 }  
 ArrayList<String> RSL = FileAccess.*load*(*ROULETTE\_STATSFILENAME*,10);  
 for(int i = 0; i < 10; i++){  
 *betsWon* = "" + RSL.get(i).charAt(2);  
 RouletteGameApp.*betsWon*[i] = Integer.*parseInt*(*betsWon*);  
 *betsWonValue* = "" + RSL.get(i).charAt(4);  
 RouletteGameApp.*betsWonValue*[i] = Integer.*parseInt*(*betsWonValue*);  
 }  
 }  
  
 *// Grava as estatisticas* public static void save(){  
 ArrayList<String> SL=new ArrayList<>(2);  
 SL.add(""+*games*);  
 SL.add(""+*coins*);  
 FileAccess.*save*(*STATISTICSFILENAME*,SL);  
  
 ArrayList<String> RSL = new ArrayList<>(10);  
 for(int i = 0; i < 10; i++)  
 RSL.add(""+ i +";" + RouletteGameApp.*betsWon*[i] + ";" + RouletteGameApp.*betsWonValue*[i]);  
 FileAccess.*save*(*ROULETTE\_STATSFILENAME*,RSL);  
 }  
}

**I. *Roulette Display***

import java.lang.Math;  
import isel.leic.utils.\*;  
  
*// Controla o Roulette Display.*public class RouletteDisplay {  
  
 private static final int *MAX\_ANIMATION\_TIME\_ROTATINGSEGMENT* = 15;  
 private static final int *MIN\_ANIMATION\_TIME\_ROTATINGSEGMENT* = 5;  
 private static final int *MAX\_ANIMATION\_TIME\_ROTATINGNUMBERS* = 15;  
 private static final int *MIN\_ANIMATION\_TIME\_ROTATINGNUMBERS* = 5;  
 private static final int *LAST\_NUMBER\_SHOW\_TIME* = 5;  
 private static final int *LASTBUTONE\_NUMBER\_SHOW\_TIME* = 2;  
 private static final int *WR\_BIT* = 0x40;  
 private static final int *ANIM\_BIT* = 0x0a;  
 private static final int *DISPLAY\_OFF* = 0x1c;  
 private static final int *WAIT\_TIME* = 300;  
 private static final int *WAIT\_TIME\_NUMBER* = 200;  
 private static final int *WAIT\_TIME\_HALF\_SECOND* = 500;  
 private static final int *WAIT\_TIME\_ONE\_AND\_HALF\_SECOND* = 1500;  
 private static final int *WAIT\_TIME\_TWO\_AND\_HALF\_SECOND* = 2500;  
  
 public static void main(String[] args) {  
 HAL.*init*();  
 *init*();  
 *showNumber(5);*  
 }

*// Inicia a classe, estabelecendo os valores iniciais.* public static void init() {  
 *clearDisplay*();  
 }

*// Envia comando para apresentar o número sorteado* public static void showNumber(int number) {  
 HAL.*clrBits*(0xff);  
 HAL.*setBits*(number);  
 HAL.*setBits*(*WR\_BIT*);  
 }

public static void animationRotatingSegment(){  
 int animationDuration = (int) (Math.*random*() \* (*MAX\_ANIMATION\_TIME\_ROTATINGSEGMENT* - *MIN\_ANIMATION\_TIME\_ROTATINGSEGMENT* + 1) \* 1000);  
 int stopAnimationTime = (int) (Time.*getTimeInMillis*() + animationDuration);  
 int i;  
 while (true) {  
 for (i = 0; i < 6 & stopAnimationTime > (int)Time.*getTimeInMillis*(); i++) {  
 *showNumber*(*ANIM\_BIT* + i);  
 RouletteGameApp.*bet*(*WAIT\_TIME*);  
 }if(stopAnimationTime < (int)Time.*getTimeInMillis*()) break;  
 }  
 }  
 public static void animationRotatingNumbers(int rouletteNumber) {  
 int animationDuration = 1000;  
 int animationTimeForFirstNumbers;  
 if(rouletteNumber>=6) animationTimeForFirstNumbers = animationDuration/(rouletteNumber-3);  
 else {  
 animationTimeForFirstNumbers = animationDuration / (rouletteNumber + 10 - 3);  
 *animationCompleteRotation*(rouletteNumber, animationTimeForFirstNumbers);  
 }  
 for(int i = 0;i <= rouletteNumber;++i) {  
 if ((rouletteNumber-3) > 0) *showNumberAnim*(i,animationTimeForFirstNumbers);  
 else if ((rouletteNumber-2) > 0) *showNumberAnim*(i,*WAIT\_TIME\_HALF\_SECOND*);  
 else if((rouletteNumber-1) > 0) *showNumberAnim*(i,*WAIT\_TIME\_ONE\_AND\_HALF\_SECOND*);  
 else if(i <= rouletteNumber) *showNumberAnim*(i,*WAIT\_TIME\_TWO\_AND\_HALF\_SECOND*);  
 }  
 }  
 private static void animationCompleteRotation(int rouletteNumber,int animationTimeForFirstNumbers){  
 for(int i = 0;i <= 9;++i) {  
 if ((i+3) <= (rouletteNumber+10)) *showNumberAnim*(i, animationTimeForFirstNumbers);  
 else if ((i+2) <= (rouletteNumber+10)) *showNumberAnim*(i, *WAIT\_TIME\_HALF\_SECOND*);  
 else if ((i+1) <= (rouletteNumber+10)) *showNumberAnim*(i, *WAIT\_TIME\_ONE\_AND\_HALF\_SECOND*);  
 }  
 }  
 public static void showNumberAnim(int number,int time){  
 *showNumber*(number);  
 Time.*sleep*(time);  
 }  
 public static void blinkNumber(int number){  
 for(int i=0; i < 10;i++){  
 Time.*sleep*(*WAIT\_TIME\_HALF\_SECOND*);  
 *showNumber*(*DISPLAY\_OFF*);  
 Time.*sleep*(*WAIT\_TIME\_NUMBER*);  
 *showNumber*(number);  
 }  
 }  
  
 public static void clearDisplay(){  
 *showNumber*(*DISPLAY\_OFF*);  
 }  
}

**J. *M***

public class M {  
  
 public static void main(String[] args) {  
  
 }  
  
 public static char maintenanceMenu() {  
 RouletteGameApp.*checkIfMaintenanceButtonOff*();  
  
 char pressed = '?';  
 boolean b = false;  
 int c = 1;  
 TUI.*clearScreen*();  
 TUI.*write*(" On Maintenance ");  
  
 while (pressed != '0' & pressed != '#' & pressed != '\*' & pressed != '8'){  
 RouletteGameApp.*checkIfMaintenanceButtonOff*();  
 int i = b ? 1 : 0;  
 TUI.*write*(RouletteGameApp.*KEYOPTIONS*[i], 1, 0);  
 pressed = KBD.*waitKey*(RouletteGameApp.*WAIT\_TIME\_5SEC*);  
 b = !b;  
 }return pressed;  
 }  
}

**K. *RouletteGame - App***

import isel.leic.utils.Time;  
  
import java.lang.Math;  
  
public class RouletteGameApp {  
  
 private static final int *MAX\_BET* = 9;  
 private static final int *COIN\_VALUE* = 2;  
 private static final int *MIN\_ROL\_NUM* = 0;  
 private static final int *MAX\_ROL\_NUM* = 9;  
 private static final int *MAINTENANCE\_COINS* = 100;  
  
 private static final int *MAINTENANCE\_BUTTON* = 0x80;  
 static int[] *betsWon* = {0,0,0,0,0,0,0,0,0,0};  
 static int[] *betsWonValue* = {0,0,0,0,0,0,0,0,0,0};  
 private static final int[] *currentBets* = {0,0,0,0,0,0,0,0,0,0};  
 public static final String[] *KEYOPTIONS* = {"0-Stats #-Count ", "\*-Play 8-ShutD "};  
  
 private static int *totalCoins* = 10;  
 private static int *coinsAvailable* = 0;  
 private static int *rouletteNumber*;  
 public static final int *WAIT\_TIME\_5SEC* = 5000; *//5seg* public static void main(String[] args){  
 *init*();  
 *gameRotation*(false);  
 }  
  
 public static void init(){  
 HAL.*init*();  
 KBD.*init*();  
 LCD.*init*();  
 RouletteDisplay.*init*();  
 TUI.*init*();  
 Statistics.*init*(); *// Load's previous Statistics from Statistics.txt file* }  
  
 public static int[] *specialChar* = {0,0b00011111,0b00010001,0b00010101,0b00010001,0b00011111,0,0, *// 0*0,0b00011111,0b00010101,0b00010001,0b00010101,0b00011111,0,0, *// 1* 0,0b00011111,0b00010011,0b00010101,0b00011001,0b00011111,0,0}; *// 2* public static void gameRotation(boolean maintenance){  
 while(true) {  
 *coinsAvailable* = (maintenance)? *MAINTENANCE\_COINS* : *totalCoins*;  
 if(!maintenance){*firstMenu*();  
 while (*coinsAvailable* == 0) ;  
 *waitforPlay*();  
 }  
 *betsMenu*();  
 char currentKey;  
 while (true) {  
 currentKey = *readKey*();  
 *placeBet*(currentKey - '0');  
 *updateTotalCoins*();  
 if (currentKey == '#') {  
 *rouletteRoll*();  
 RouletteDisplay.*animationRotatingNumbers*(*rouletteNumber*);  
 *calculateWinsAndLosses*();  
 if(!maintenance) *totalCoins* = *coinsAvailable*;  
 break;  
 }  
 }  
 *clearPlacedBets*();  
 RouletteDisplay.*clearDisplay*();  
 if(maintenance) *maintenanceOptions*(M.*maintenanceMenu*());  
 }  
 }  
  
 public static void firstMenu(){  
 TUI.*clearScreen*();  
 TUI.*write*(" Roulette Game ",0,0);  
 TUI.*setCursor*(1,0);  
 for(int i=0;i<3;i++){  
 TUI.*write*(" " + ((char)(i+'1')) + " ");  
 LCD.*customChar*(i);  
 }  
 TUI.*setCursor*(1,15-TUI.*digitDim*(*coinsAvailable*));  
 TUI.*write*("$" + *coinsAvailable*);  
 }  
  
 public static void betsMenu(){  
 TUI.*clearScreen*();  
 TUI.*setCursor*(1,0);  
 TUI.*write*("0123456789 ");  
 TUI.*setCursor*(1,15-TUI.*digitDim*(*coinsAvailable*));  
 TUI.*write*("$" + *coinsAvailable*);  
 }  
  
 public static void bet(int time){  
 char currentKey = KBD.*waitKey*(time);  
 *placeBet*(currentKey - '0');  
 *updateTotalCoins*();  
 }  
  
 public static void rouletteRoll(){  
 *rouletteNumber*= (int)(Math.*random*()\*(*MAX\_ROL\_NUM* - *MIN\_ROL\_NUM* +1));  
 RouletteDisplay.*animationRotatingSegment*();  
 }  
  
 public static void updateTotalCoins(){  
 TUI.*setCursor*(1,14-TUI.*digitDim*(*coinsAvailable*));  
 TUI.*write*(" $" + *coinsAvailable*);  
 }  
  
 public static int addCoin() {  
 return *totalCoins* += *COIN\_VALUE*;  
 }  
  
 public static void coinPlacedOnBets(){  
 *coinsAvailable* -= 1;  
 }  
  
 public static void placeBet(int bet){  
 LCD.*cursor*(0,bet);  
 if(bet>=0 && *currentBets*[bet]<*MAX\_BET* && *coinsAvailable*>0){  
 *coinPlacedOnBets*();  
 TUI.*write*(String.*valueOf*(++*currentBets*[bet]));  
 }  
 }  
  
 public static void clearPlacedBets(){ for(int n=0;n<=9;n++) *currentBets*[n] = 0; }  
  
 public static void calculateWinsAndLosses(){  
 int won = 0, lost = 0, coinsWonLoss;  
 String winOrLoss;  
 for(int n=0;n<=9;n++){  
 if(n == *rouletteNumber*) won = *currentBets*[n];  
 else if(*currentBets*[n]>0) lost += *currentBets*[n];  
 }  
 if(won!=0) won\*=2;  
 coinsWonLoss = won-lost;  
 *coinsAvailable* += won;  
 winOrLoss = (coinsWonLoss > 0)?"W":"L";  
 coinsWonLoss = Math.*abs*(coinsWonLoss);  
 TUI.*write*(winOrLoss + "$" + coinsWonLoss,0,14-TUI.*digitDim*(coinsWonLoss));  
 RouletteDisplay.*blinkNumber*(*rouletteNumber*);  
 }  
  
 public static char readKey(){  
 char key = 0;  
 while (key == 0) key = KBD.*getKey*();  
 return key;  
 }  
  
 public static void waitforKey(char keyExpected){  
 char key = 0;  
 while (key != keyExpected) key = KBD.*getKey*();  
 }  
  
 public static void waitforPlay(){  
 char key = 0;  
 while (key != '\*'){  
 key = KBD.*getKey*();  
 *checkIfMaintenanceButtonOn*();  
 }  
 }  
 public static void checkIfMaintenanceButtonOn(){  
 if(HAL.*readBits*(*MAINTENANCE\_BUTTON*) == *MAINTENANCE\_BUTTON*) { *maintenanceOptions*(M.*maintenanceMenu*());}  
 }  
  
 public static void checkIfMaintenanceButtonOff(){  
 if(HAL.*readBits*(*MAINTENANCE\_BUTTON*) != *MAINTENANCE\_BUTTON*) *gameRotation*(false);  
 }  
 public static void maintenanceOptions(char pressed){  
 if(pressed == '0') {  
 TUI.*clearScreen*();  
 int line = 1;  
 TUI.*write*(""+(line-1)+": -> "+*betsWon*[line-1]+" $:"+*betsWonValue*[line-1],0,0);  
 TUI.*write*(""+line+": -> "+*betsWon*[line]+" $:"+*betsWonValue*[line],1,0);  
 char key = KBD.*waitKey*(*WAIT\_TIME\_5SEC*);  
 do {  
 if(key == '2' && line > 1) --line;  
 if(key == '8' && line < 9) ++line;  
 if(key != '8' && key != '2') break;  
 TUI.*write*(""+(line-1)+": -> "+*betsWon*[line-1]+" $:"+*betsWonValue*[line-1],0,0);  
 TUI.*write*(""+line+": -> "+*betsWon*[line]+" $:"+*betsWonValue*[line],1,0);  
 key = KBD.*waitKey*(*WAIT\_TIME\_5SEC*);  
 }while(key != 0);  
 }else if(pressed == '#'){  
 TUI.*clearScreen*();  
 TUI.*write*("Games: " + Statistics.*getGames*(),0,0);  
 TUI.*write*("Coins: " + Statistics.*getCoins*(),1,0);  
 char key = KBD.*waitKey*(*WAIT\_TIME\_5SEC*);  
 }else if(pressed == '\*') *gameRotation*(true);  
 else if(pressed == '8') *shutdownMenu*();  
 *checkIfMaintenanceButtonOn*();  
 }  
  
 private static void shutdownMenu(){  
 TUI.*write*(" Shutdown ",0,0);  
 TUI.*write*("5-Yes other-No ",1,0);  
 char key = KBD.*waitKey*(*WAIT\_TIME\_5SEC*);  
 if (key == '5') {  
 Statistics.*save*(); *//Saves scores and stats to Statistics.txt file* System.*exit*(0);  
 }M.*maintenanceMenu*();

}  
}