ISEL
DEETC
Departamento de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

Máquina de Venda de Bilhetes (*Ticket Machine*)

André Graça (47224)

Projeto
de
Laboratório de Informática e Computadores
2021 / 2022 verão

23 de Junho de 2022

1 Introdução

Neste projeto implementa-se um sistema de controlo de uma máquina de venda de bilhetes (*Ticket Machine*), que permite a aquisição de bilhetes de comboio. O percurso é definido pela estação de origem, local de compra do bilhete e pela seleção do destino digitando o identificador da estação ou através das teclas ↑ e ↓, sendo exibido no ecrã além do identificador da estação de destino o preço e o tipo de bilhete (ida ou ida/volta). A ordem de aquisição é dada através da pressão da tecla de confirmação, sendo impressa uma unidade do bilhete exibido no ecrã. A máquina não realiza trocos e só aceita moedas de: 0,05€; 0,10€; 0,20€; 0,50€; 1,00€; e 2,00€. Para além do modo de Dispensa, o sistema tem mais um modo de funcionamento designado por Manutenção, que é ativado por uma chave de manutenção. Este modo permite o teste da máquina de venda de bilhetes, além disso permite iniciar e consultar os contadores de bilhetes e moedas.

A máquina de venda de bilhetes é constituída pelo sistema de gestão (designado por *Control* na Figura 1) e pelos seguintes periféricos: um teclado de 12 teclas, um moedeiro (designado por *Coin Acceptor*), um ecrã *Liquid Cristal Display* (*LCD*) de duas linhas de 16 caracteres, um mecanismo de impressão de bilhetes (designado por *Ticket Dispenser*) e uma chave de manutenção (designada por *M*) que define se a máquina de venda de bilhetes está em modo de Manutenção, conforme o diagrama de blocos apresentado na Figura 1.

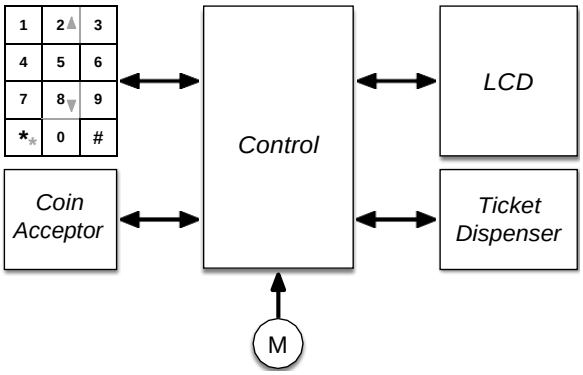


Figura 1 – Máquina de venda de bilhetes (*Ticket Machine*)

Sobre o sistema podem-se realizar as seguintes ações em modo Venda:

- **Consulta e venda** – A consulta de um bilhete é realizada digitando o identificador da estação de destino ou listando-a através das teclas ↑ e ↓. O processo de compra do bilhete inicia-se premindo a tecla '#'. Durante a inserção do respetivo valor monetário, é possível alterar o tipo de bilhete (ida ou ida/volta) premindo a tecla '0', com a consequente alteração do preço da viagem afixado no *LCD*, duplicando o valor no caso de ida/volta. Durante a compra ficam afixados no *LCD* as informações referentes ao bilhete pretendido até que o mecanismo de impressão de bilhetes confirme que a impressão já foi realizada e a recolha do bilhete efetuada. O modo de seleção ↑ e ↓ alterna com a seleção numérica por pressão da tecla '*'. A compra pode ser cancelada premindo a tecla '#', devolvendo as moedas inseridas.

Sobre o sistema podem-se realizar as seguintes ações em modo Manutenção:

- **Teste** – Esta opção do menu permite realizar um procedimento de consulta e venda de um bilhete, sem introdução de moedas e sem esta operação ser contabilizada como uma aquisição.
- **Consulta** – Para visualizar os contadores de moedas e bilhetes seleciona-se a operação de consulta no menu, e permite-se a listagem dos contadores de moedas e bilhetes, através das teclas ↑ e ↓.
- **Iniciar** – Esta opção do menu inicia os contadores de moedas e bilhetes a zero, iniciando um novo ciclo de contagem.
- **Desligar** – O sistema desliga-se ao selecionar-se esta opção no menu, ou seja, o software de gestão termina armazenando as estruturas de dados de forma persistente em ficheiros de texto. A informação do número de moedas no cofre do moedeiro e dos bilhetes vendidos deve ser armazenada em ficheiros separados. A informação em cada ficheiros deve estar organizada por linha, em que os campos de dados são separados por ";", com o respetivo formato: "*COIN;NUMBER*" (moedas) e "*PRICE;NUMBER;STATION_NAME*" (bilhetes vendidos). Estes ficheiros são lidos e carregados no início do programa e reescritos no final do programa.

Nota: A inserção de informação através do teclado tem o seguinte critério: *i*) se não for premida nenhuma tecla num intervalo de cinco segundos o comando em curso é abortado; *ii*) quando o dado a introduzir é composto por mais que um dígito, são considerados apenas os últimos dígitos, a inserção realiza-se do dígito de maior peso para o de menor peso.

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 três módulos principais: *i*) um leitor de teclado, designado por *Keyboard Reader*; *ii*) um módulo de interface com o *LCD* e com o mecanismo de dispensa de bilhetes, designado por *Integrated Output System* (*IOS*); e *iii*) um módulo de controlo, designado por *Control*. Os módulos *i*) e *ii*) deverão ser implementados em *hardware*, enquanto o módulo de controlo é implementado em *software* usando linguagem *Kotlin* executado num PC.

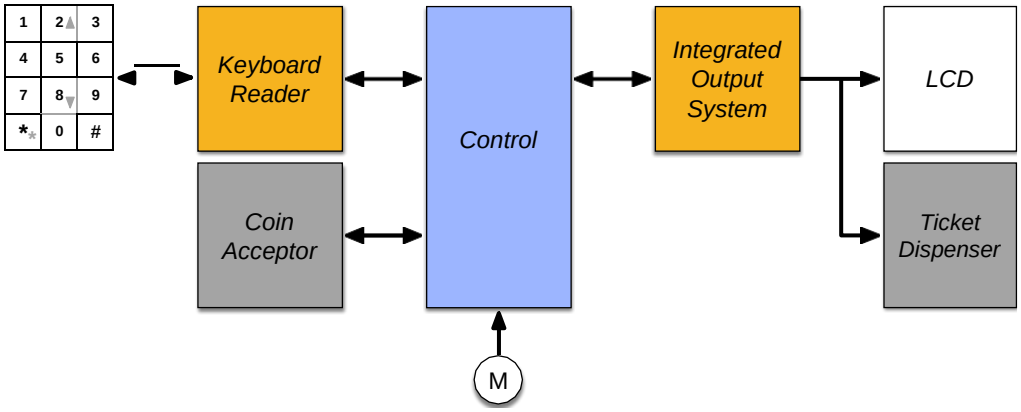


Figura 2 – Arquitetura do sistema que implementa a Máquina de Venda de Bilhetes (*Ticket Machine*)

O módulo *Keyboard Reader* é responsável pela descodificação do teclado matricial de 12 teclas, determinando qual a tecla pressionada e disponibilizando o seu código ao módulo *Control*. Caso este não esteja disponível para o receber imediatamente, o código da tecla é armazenado até ao limite de dois códigos. Por razões de ordem física, e por forma a minimizar o número de sinais de interligação, a comunicação entre o módulo *Control* e o módulo *Keyboard Reader* é realizada recorrendo a um protocolo série síncrono. O módulo *Control* processa os dados e envia a informação a apresentar no *LCD* através do módulo *IOS*. O mecanismo de dispensa de bilhetes, designado por *Ticket Dispenser*, é atuado pelo módulo *Control*, através do módulo *IOS*. A comunicação entre o módulo *Control* e o módulo *IOS* é também realizada recorrendo a um protocolo série síncrono, pelo mesmo motivo da comunicação entre o módulo *Control* e o módulo *Keyboard Reader*.

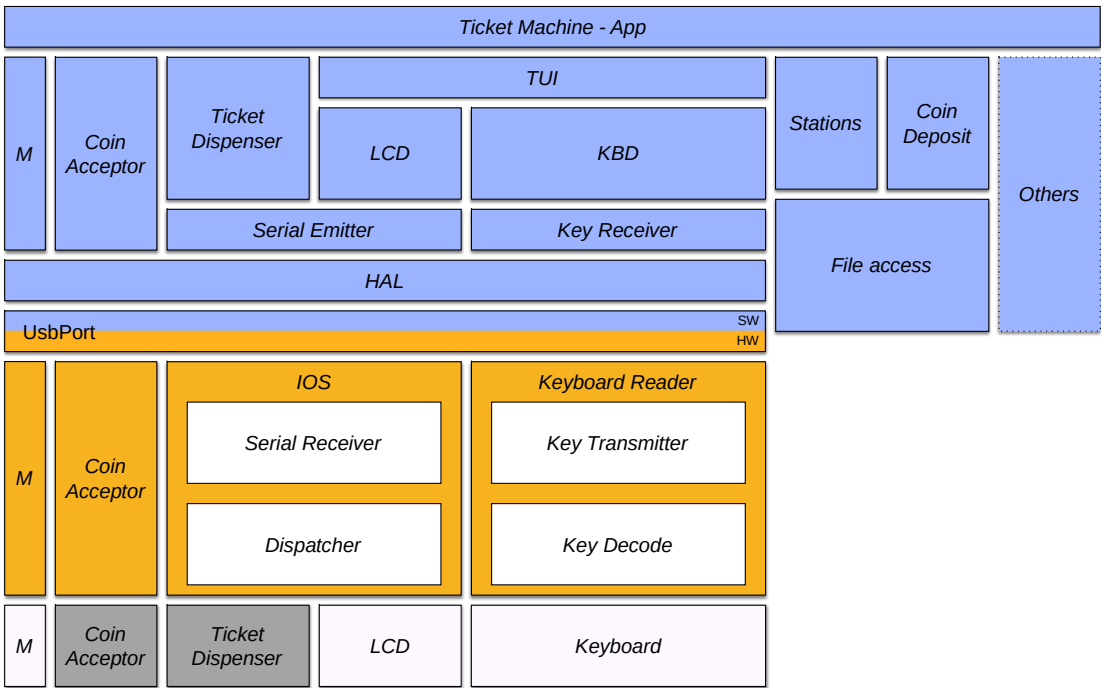
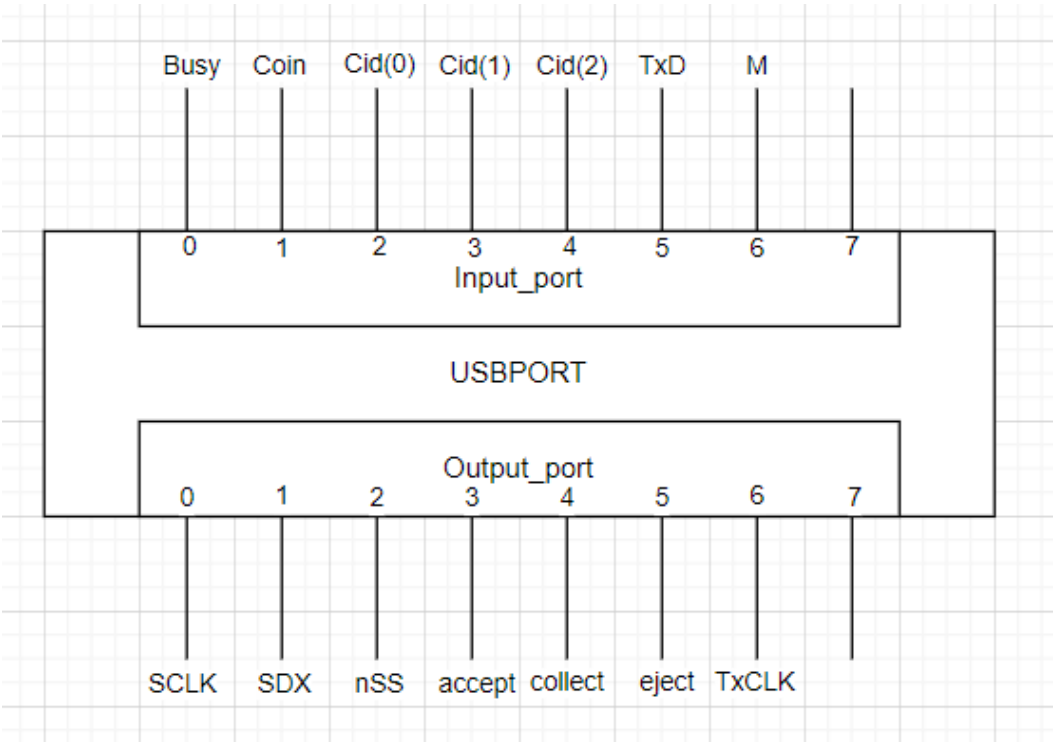


Figura 3 – Diagrama lógico do sistema de controlo da Máquina de Venda de Bilhetes (*Ticket Machine*)

A. Interligações entre o HW e SW



B. Código *Kotlin* da classe *HAL*

```kotlin
object HAL {
    var lastOutput = 0
    fun init() {
        UsbPort.write(0)
    }

    fun isBit(mask: Int): Boolean {
    val value = UsbPort.read()
    val final = value and mask
    if (final != 0) return true else return false
    }

    fun readBits(mask: Int): Int {
        val value = UsbPort.read()
        val final = value and mask
        return final
    }

    fun writeBits(mask: Int, value: Int) {
        var finalOutput = 0
        finalOutput = (mask.inv() and lastOutput) or (value and mask)
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }

    fun setBits(mask: Int) {
        var finalOutput = 0
        finalOutput = lastOutput or mask
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }

    fun clrBits(mask: Int) {
        var finalOutput = 0
        finalOutput = mask.inv() and lastOutput
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }
}
```

C. Código *Kotlin* da classe *Key Receiver*

```
object KeyReceiver { // Recebe trama do Keyboard Reader.
    private val TxD_MASK = 1 shl 5      //entrada usbport
    private val TxCLK_MASK = 1 shl 6    //saida usbport

    // Inicia a classe
    fun init() {
        HAL.init()
    }
    // Recebe uma trama e retorna o código de uma tecla caso exista
    fun rcv(): Int {
        var result = -1
        var frame = 0
        if (!HAL.isBit(TxD_MASK)) {
            for (i in 0 .. 5) {
                HAL.setBits(TxCLK_MASK)
                Thread.sleep(1)
                frame = frame or HAL.readBits(TxD_MASK).shl(i)
                HAL.clrBits(TxCLK_MASK)
                Thread.sleep(1)
            }
            frame = frame.shr(5)
        if ((frame and 1) == 0) return -1
            if ((frame and (1 shl 5)) != 0) return -1
            result = frame.shr(1)
        }
        return result
    }
}
```

D. Código *Kotlin* da classe *KBD*

```
object KBD { // Ler teclas. Métodos retornam '0'..'9','#','*' ou NONE.
    const val NONE = -1;
    // Inicia a classe
    fun init() {
    KeyReceiver.init()
    }

    // Implementa a interação série com o Key Transmitter, utilizado na 2ªfase do
projeto
    private fun getKeySerial(): Char {
        val key = KeyReceiver.rcv()
        return when(key) {
            0 -> '1'
            1 -> '4'
            2 -> '7'
            3 -> '*'
            4 -> '2'
            5 -> '5'
            6 -> '8'
            7 -> '0'
            8 -> '3'
            9 -> '6'
            10 -> '9'
            11 -> '#'
            else -> NONE.toChar()
        }
    }
    // Retorna de imediato a tecla premida ou NONE se não há tecla premida.
    fun getKey(): Char {
      return getKeySerial()
    }
    // Retorna quando a tecla for premida ou NONE após decorrido 'timeout'
milisegundos.
    fun waitKey(timeout: Long): Char {
    val timeinit = Time.getTimeInMillis()
        while (Time.getTimeInMillis() - timeinit < timeout) {
            val key = getKeySerial()
            if (key != NONE.toChar()){
                return key
            }
        }
        return NONE.toChar()
    }

}
```

E. Código *Kotlin* da classe *SerialEmitter*

```
object SerialEmitter {

    enum class Destination {
        LCD, TICKET_DISPENSER
    }

    private val nSS_MASK = 1 shl 2

    private val SDX_MASK = 1 shl 1

    private val SCLK_MASK = 1

    private val BUSY_MASK = 1

    private var FRAME_LENGTH = 10

    fun init () {    //inicia classe e mete respetivos bits a estado inicial
        HAL.init()
        HAL.setBits(nSS_MASK)
    }

    fun send(addr: Destination, data: Int) {
    var send_MASK = 0

        while (isBusy()) {
        }

        var mMask = 0

        if (addr == Destination.LCD) {
            mMask = 0
        }   else if (addr == Destination.TICKET_DISPENSER) {
            mMask = 1
        }

        send_MASK = mMask or data.shl(1)  //juntar data e destino
        val parityMask = if (!parityCheck(send_MASK)) 1 shl FRAME_LENGTH else 0
       // send_MASK = send_MASK shl 1
        //se sendMask for impar o meu parity bit ira ser 1 para os 11 bits ficarem
par
        send_MASK = send_MASK or parityMask

        HAL.clrBits(nSS_MASK)    //inicia o serial receiver para receber bits

        for (i in 0 .. FRAME_LENGTH) {     //mandar SDX respetivo para placa
            if (send_MASK and (1 shl i) != 0){
              // print(1)
                HAL.setBits(SDX_MASK)
            } else {
              // print(0)
                HAL.clrBits(SDX_MASK)
            }
            HAL.clrBits(SCLK_MASK)
            Thread.sleep(1)
            HAL.setBits(SCLK_MASK)
```

```
            Thread.sleep(1)
        }
        HAL.setBits(nSS_MASK)    //acabo transmissao para serial receiver
    }

    private fun parityCheck(data: Int): Boolean {
        var numOnes = 0
        var tempMASK = 1
        for (i in 0 until FRAME_LENGTH) {
            if (data and tempMASK != 0) numOnes ++
            tempMASK = tempMASK shl 1
        }
        return numOnes % 2 == 0
    }

    /** Retorna true se o canal série estiver ocupado  */
    fun isBusy(): Boolean {
        return HAL.isBit(BUSY_MASK)
    }
}
```

F. Código *Kotlin* da classe *LCD*

```
object LCD { // Escreve no LCD usando a interface a 8 bits.
    private const val LINES = 2
    private const val COLS = 16; // Dimensão do display.
    private val twoline8bits = 56
    private val sleepCMD = 48

        // Escreve um byte de comando/dados no LCD em série
    private fun writeByteSerial(rs: Boolean, data: Int) {
    val rs = if(rs) 1 else 0
        val finaldata = rs or data.shl(1)
        SerialEmitter.send(SerialEmitter.Destination.LCD,finaldata)
    }
    // Escreve um byte de comando/dados no LCD
    private fun writeByte(rs: Boolean, data: Int) {
    writeByteSerial(rs, data)
    }
    // Escreve um comando no LCD
    private fun writeCMD(data: Int) {
    writeByte(false, data)
    }
    // Escreve um dado no LCD
    private fun writeDATA(data: Int) {
    writeByte(true, data)
    }
    // Envia a sequência de iniciação para comunicação a 8 bits.
    fun init() {
        Thread.sleep(50)
        writeCMD(sleepCMD)
        Thread.sleep(5)
        writeCMD(sleepCMD)
        Thread.sleep(1)
        writeCMD(sleepCMD)
        writeCMD(twoline8bits)
        writeCMD(8)
        writeCMD(1)
        writeCMD(6)
        writeCMD(15)
    }
    // Escreve um caráter na posição corrente.
    fun write(c: Char) {
    writeDATA(c.code)
    }
    // Escreve uma string na posição corrente.
    fun write(text: String) {
        for (i in text.indices) {
            write(text[i])
        }
    }
    // Envia comando para posicionar cursor ('line':0..LINES-1 , 'column':0..COLS-1)
    fun cursor(line: Int, column: Int) {
    var jump = 128 or line.shl(6) or column
        writeCMD(jump)
    }
    // Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
    fun clear() {
    writeCMD(1)
```

```
        }
}
```

G. Código *Kotlin* da classe *Ticket Dispenser*

```
object TicketDispenser {
    // Inicia a classe, estabelecendo os valores iniciais.
    fun init() {
    SerialEmitter.init()
    }
    // Envia comando para imprimir e dispensar um bilhete
    fun print(destinyId: Int, originId: Int, roundTrip: Boolean) {
       val RT:Int
        if (roundTrip) RT = 1 else RT = 0
        var data = RT or destinyId.shl(1) or originId.shl(5) //meter bits na posição
final para enviar
        SerialEmitter.send(SerialEmitter.Destination.TICKET_DISPENSER, data)
    }
}
```

H. Código *Kotlin* da classe *TUI*

```
object TUI {

    const val NONE = -1

    fun init() {
        KBD.init()
        LCD.init()
    }

    fun clearDisplay(){
        LCD.clear()
    }

    fun getKey(): Char {
        return KBD.getKey()
    }

    fun waitForKey() : Char{
        return KBD.waitKey(5000)
    }

    fun writeLCDStation(station : String , price : String , number : Int){//expor a
estações
        val firstPos = 8 - (station.length)/2
        LCD.cursor(0,firstPos)
        LCD.write(station)

        val priceString = ("${price[0]}.${price[1]}${price[2]}E")
        val pricePos = 16 - priceString.length
        LCD.cursor(1, pricePos)
        LCD.write(priceString)

        LCD.cursor(1,0)
        LCD.write(number.toString())
    }

    fun sellWriteLCDStation(station: String,price: String){//no pagamento
        val firstPos = 8 - (station.length)/2
        LCD.cursor(0,firstPos)
        LCD.write(station)

        var priceString = "000"
        if(price.length == 3 ){
            priceString = ("${price[0]}.${price[1]}${price[2]}E")
        }
        if(price.length <= 2){
            priceString = ("0.${price[0]}${price[1]}E")
        }

        LCD.cursor(1,firstPos)
        LCD.write(priceString)
    }

    fun writeMaintenanceMenu(){
        LCD.cursor(0,0)
        LCD.write("Maintenance Mode")
        LCD.cursor(1,0)
        LCD.write("Type:1,2,3,4or 5")
```

```
    }

    fun writeLCDMainMenu(){
        LCD.cursor(0,1)
        LCD.write("Ticket to Ride")
        LCD.cursor(1,0)
        LCD.write("Press # to Start")
    }

    fun writeYesOrNo(){ // shutdown has 8 letters
        val pos = 4 // posição da primeira letra de shutdown
        LCD.cursor(0,pos)
        LCD.write("Shutdown")

        LCD.cursor(1,0)
        LCD.write("5-Yes  other-No")
    }


    fun writeOnLCD() : Char {//fazer verificação da tecla obtida pela função
getKey() e na main manter a correr indefenidamente
        val key = KBD.waitKey(5000)// timeout = 5 seconds
        if (key != NONE.toChar()) {
            println(key)
            LCD.write(key)
            return key
        }
        return key
    }

    fun vendingAborted(){
        LCD.cursor(0,0)
        LCD.write("Vending Aborted")
    }

}
```

I. Código *Kotlin* da classe *FileAccess*

```
object FileAcess{

    fun createReader(fileName : String) : BufferedReader {
        return BufferedReader(FileReader(fileName))
    }

    fun createWriter(fileName: String): PrintWriter {
        return PrintWriter(fileName)
    }

}
```

J. Código *Kotlin* da classe *Stations*

```
object Station{
    data class Ticket(val Price : String , var quantity : Int , val city : String)

    fun getInfo(FileName : String) : Array<Ticket?>{
        val reader = FileAcess.createReader(FileName)
        var line = reader.readLine()
        var array = arrayOfNulls<Ticket>(16) // devemos deixar 16 ou deveriamos
colocar outro valor?
        var i = 0
        while(line != null){
            val list = line.split(";")
            val ticket = Ticket(list[0],list[1].toInt(),list[2])
            array[i] = ticket
            line = reader.readLine()
            i++
        }
        return array
    }

    fun writeInfo(FileName: String, array : Array<Ticket?>){
        val writer = FileAcess.createWriter(FileName)
        for(i in array.indices){

writer.println("${array[i]?.Price};${array[i]?.quantity};${array[i]?.city}")
        }
        writer.close()
    }
}
```

K. Código *Kotlin* da classe *CoinDeposit*

```
object CoinDeposit{

    data class Coin(val CoinId : Int , var CoinQT : Int)

    fun getInfo(FileName : String) : Array<Coin?>{
        val reader = FileAcess.createReader(FileName)
        var line = reader.readLine()
        var array = arrayOfNulls<Coin>(6) // devemos deixar 16 ou deveriamos colocar
outro valor?
        var i = 0
        while(line != null){
            val list = line.split(";")
            val coin = Coin(list[0].toInt(),list[1].toInt())
            array[i] = coin
            line = reader.readLine()
            i++
        }
        return array
    }

    fun writeInfo(FileName: String, array : Array<Coin?> ){
        val array = array
        val writer = FileAcess.createWriter(FileName)
        for(i in array.indices){
            writer.println("${array[i]?.CoinId};${array[i]?.CoinQT}")
        }
        writer.close()
    }

}
```

L. Código *Kotlin* da classe *CoinAcceptor*

```
object CoinAcceptor { // Implementa a interface com o moedeiro.
        //Entradas do UsbPort
    private val Coin_MASK = 1 shl 1
    private val Cid0_MASK = 1 shl 2
    private val Cid1_MASK = 1 shl 3
    private val Cid2_MASK = 1 shl 4
        //saidas do UsbPort
    private val accept_MASK = 1 shl 3
    private val collect_MASK = 1 shl 4
    private val return_MASK = 1 shl 5
    // Inicia a classe
    fun init() {
    HAL.init()
    }

    // Retorna true se foi introduzida uma nova moeda.
    fun hasCoin(): Boolean {
   return HAL.isBit(Coin_MASK)
    }

    // Retorna o valor facial da moeda introduzida.
    fun getCoinValue(): Int {
        var finalMask = 0
        var amount = 0
    finalMask = HAL.readBits(Cid0_MASK) or HAL.readBits(Cid1_MASK) or
HAL.readBits(Cid2_MASK)
        finalMask = finalMask.shr(2)

        when(finalMask) {
            0 -> amount = 5
            1 -> amount = 10
            2 -> amount = 20
            3 -> amount = 50
            4 -> amount = 100
            5 -> amount = 200
        }
        return amount
    }

    fun acceptCoin(){// informa o moedeiro que a moeda foi contabilizada
        if(hasCoin()){
            HAL.setBits(accept_MASK)
            while(hasCoin()){//aguardar que hasCoin mude
            }
            HAL.clrBits(accept_MASK)
        }
    }

    // Devolve as moedas que estão no moedeiro.
    fun  ejectCoins()  {
    HAL.setBits(return_MASK)
    }
    // Recolhe as moedas que estão no moedeiro.
    fun collectCoins() {
    HAL.setBits(collect_MASK)
    }
}
```

M. Código *Kotlin* da classe *M*

```
object M {
    private const val M_mask = 1 shl 6

    fun init(){
        HAL.init()
    }
 fun verify() :Boolean{
     return HAL.isBit(M_mask)
 }

}
```

N. Código *Kotlin* da classe *TicketMachine - App*

```
object TicketMachine{

    private const val NONE = -1
    var arrayStations = arrayOfNulls<Station.Ticket>(16)
    var arrayCoins = arrayOfNulls<CoinDeposit.Coin>(6)
    var indice = 0
    var key = '0'

    fun init(){
        arrayStations = Station.getInfo("stations.txt")
        arrayCoins = CoinDeposit.getInfo("CoinDeposit.txt")
        //Vai haver problema em ter o mesmo protocolo de inicialização que o M? sim
        //criar uma variável que deteta se a inicialização já foi feita , ou então,
colocar por ordem
        TUI.init()
        //displayMainMenu()
    }

    private fun teste(destinyId : Int, originId : Int, roundTrip : Boolean){//modo
de manutenção
        TicketDispenser.print(destinyId,originId,roundTrip)
    }

    private fun resetCount(){//modo de manutenção
        for(i in arrayStations.indices){
            arrayStations[i]?.quantity = 0
        }
        for(i in arrayCoins.indices){
            arrayCoins[i]?.CoinQT = 0
        }
    }

    private fun shutDown(){//modo de manutenção
        Station.writeInfo("stations.txt", arrayStations)
        CoinDeposit.writeInfo("CoinDeposit.txt", arrayCoins)
    }

    private fun maintenance(){
        key = TUI.waitForKey()
        while(true){
            when(key){
                '1'->{ // print ticket
                    teste(4,6,true) // valores de teste
                }
                '2'->{ // stations count
                    TODO("Not yet implemented")
                }
                '3'->{ // Coin Count
                    TODO("Not yet implemented")
                }
                '4'->{ // Reset Count
                    resetCount()
                }
                '5' ->{ // shutDown
                    while(true){
                        TUI.clearDisplay()
                        TUI.writeYesOrNo()
                        key = TUI.waitForKey()
```

```
                    if(key == '5') shutDown()// yes
                    else break
                }
            }
        }
    }
}

private fun stationsAndSell() {// mostras as estações e inicia a venda
    displayStation(indice)
    while (true) {
        key = TUI.waitForKey()
        if(key == '#') {
            sell(indice)
            key = indice.toChar()
            TUI.clearDisplay()
            ticketMachine()
        }
        while (true){
            if(key != '#' && key !='*' && key != NONE.toChar()){
                indice = key.toString().toInt()
                break
            }
            else break
        }
        if (key == '1') {
            while (true) {
                key = TUI.waitForKey()
                when (key) {
                    '0' -> indice = 10
                    '1' -> indice = 11
                    '2' -> indice = 12
                    '3' -> indice = 13
                    '4' -> indice = 14
                    '5' -> indice = 15
                }
                break
            }
        }
        var prev = true
        if (key == '*' && prev) { // arrow mode
            while (true) {
                key = TUI.waitForKey()
                if (key == '2') {
                    indice--
                    if(indice <= -1 ) indice = 15
                    TUI.clearDisplay()
                    displayStation(indice)
                    key = NONE.toChar()
                }
                if (key == '8') {
                    indice++
                    if(indice >= 16) indice = 0
                    TUI.clearDisplay()
                    displayStation(indice)
                    key = NONE.toChar()
                }
                if(key == '*'){
                    prev = false
                    break
```

```
                    }
                    if (key != '2' || key != '8') continue
                }
            }
            if (key != NONE.toChar() && key != '*' && key != '#'){
                TUI.clearDisplay()
                displayStation(indice)
                key = NONE.toChar()
                continue
            }else continue
        }
    }

    private fun sell( Index : Int ){ // modo de consulta e venda
        TUI.clearDisplay()
        val city = arrayStations[Index]!!.city
        var priceDoubleOrNot = arrayStations[Index]!!.Price
        val priceOriginal = arrayStations[Index]!!.Price
        TUI.sellWriteLCDStation(city , priceOriginal)
        var prev = true
        while(true){
            var key = TUI.waitForKey()
            if(key == '*') {
                priceDoubleOrNot = (priceDoubleOrNot.toInt() * 2).toString()
                if((priceOriginal.toInt() * 2) >= priceDoubleOrNot.toInt() ){
                    TUI.sellWriteLCDStation(city, priceDoubleOrNot)
                    payment(priceDoubleOrNot.toInt())
                    continue
                }else payment(priceDoubleOrNot.toInt())
            }else payment(priceDoubleOrNot.toInt())
            if(key == '#' && prev){
                prev = false
                LCD.clear()
                TUI.vendingAborted()
                while (true){
                    key = TUI.waitForKey()
                    if(key=='#') return
                    else break
                }
            } else continue
        }
    }

    private fun displayStation(Index : Int){//mostra as estações
        val city = arrayStations[Index]!!.city
        val price = arrayStations[Index]!!.Price
        TUI.writeLCDStation(city, price,Index)
    }

    private fun displayMainMenu(){//mostra o menu principal
        TUI.clearDisplay()
        TUI.writeLCDMainMenu()
    }

    private fun displayMaintenanceMenu(){
        TUI.clearDisplay()
        TUI.writeMaintenanceMenu()
    }

    private fun payment(value : Int) { // value == valor a pagar, mudar para não
```

```
estar sempre a escrever
        var currentValue = value
        while (true) {
            if(CoinAcceptor.hasCoin()){
                currentValue -= CoinAcceptor.getCoinValue()
                CoinAcceptor.acceptCoin()

TUI.sellWriteLCDStation(arrayStations[indice]!!.city,currentValue.toString())
            }
            continue
        }
    }

    fun ticketMachine(){
        displayMainMenu()
        while (true) {
            if (M.verify()) { // modo de manutenção
                displayMaintenanceMenu()
                maintenance()
            }
            var key = TUI.waitForKey()
            if (key == '#') { // modo de venda
                TUI.clearDisplay()
                stationsAndSell()
            }
        }
    }

}
```