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*)

Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz (a47076@alunos.isel.pt)

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.

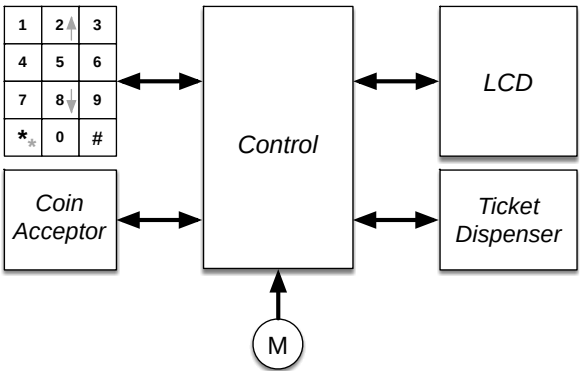


Figura 1 – Máquina de venda de bilhetes (*Ticket Machine*)

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.

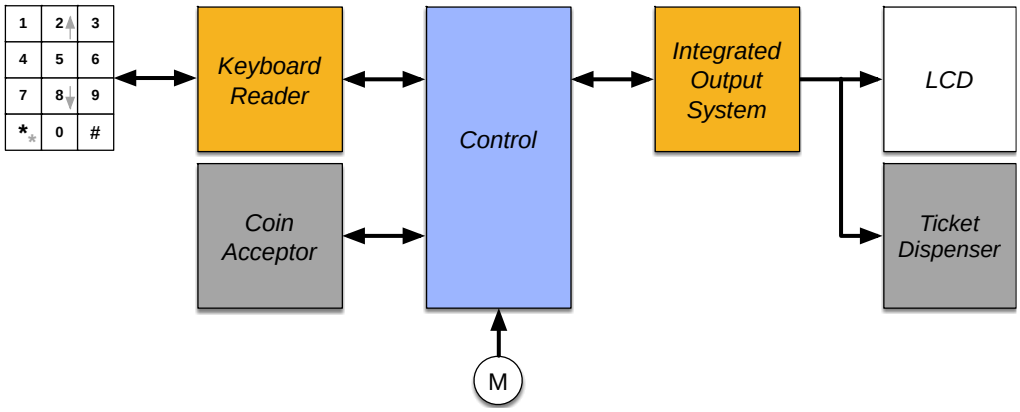


Figura 2 – Arquitetura do sistema que implementa a Máquina de Venda de Bilhetes (*Ticket Machine*)

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*.

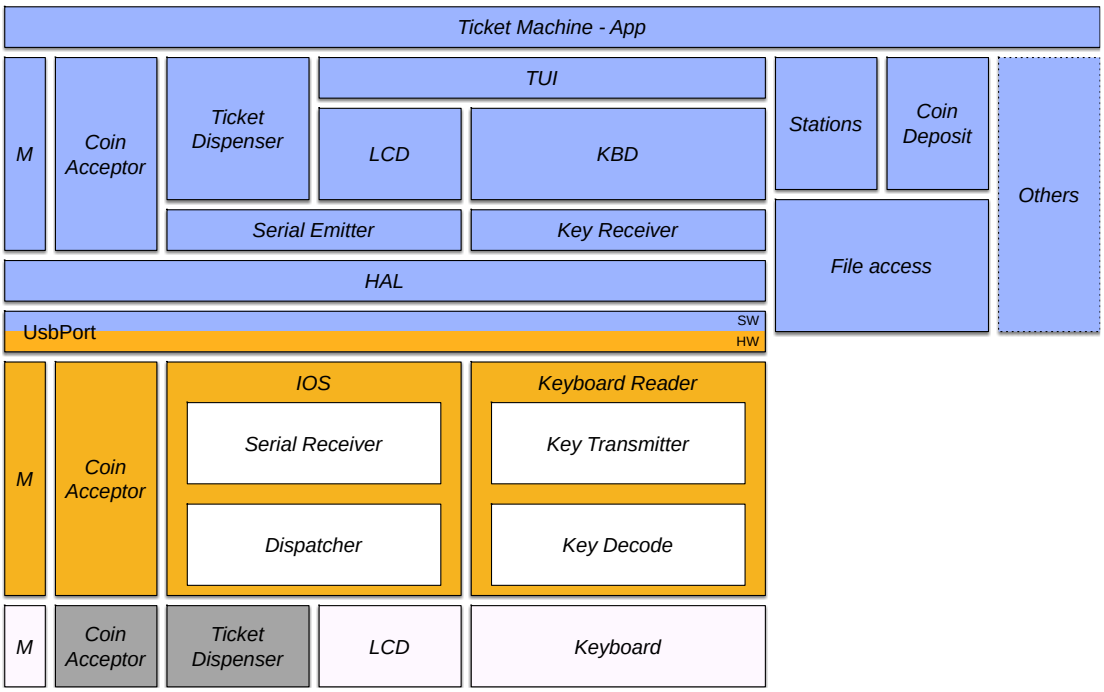
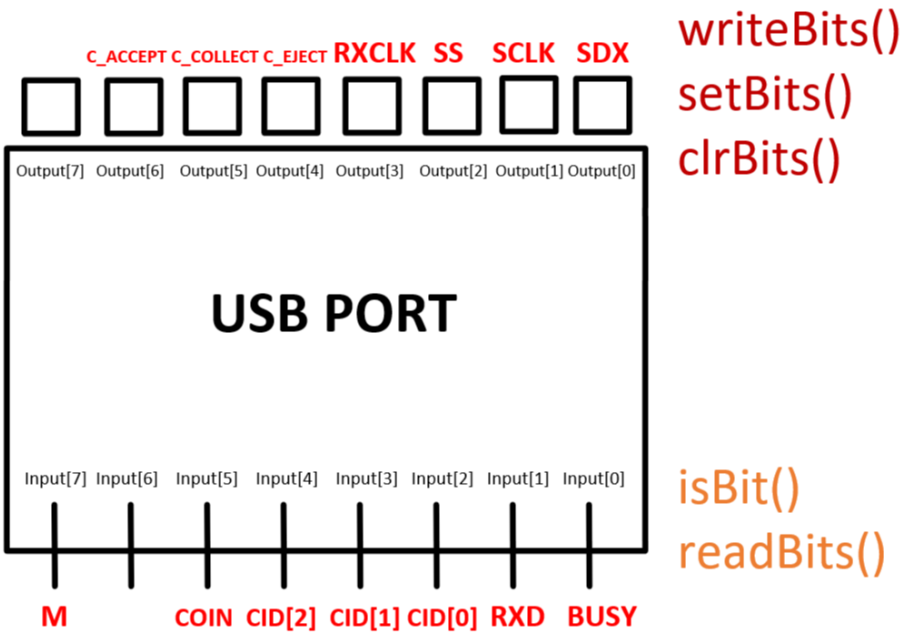


Figura 3 – Diagrama lógico do sistema de controlo da Máquina de Venda de Bilhetes (*Ticket Machine*)

A. Interligações entre o HW e SW



B. Código *Kotlin* da classe *HAL*

```kotlin
package ticketMachine

import isel.leic.UsbPort

// Virtualizes the access for the UsbPort system
object HAL {
    // Creates a mutable variable to store the value of the last output written
    private var lastOutput = 0
    // Initializes this class, which means to have a predefied output when UsbPort.write is executed
    fun init() {
        // Writes lastOutput value on the output of the board
        UsbPort.write(lastOutput)
    }
    // Returns true when a bit we want to evaluate is set to logical '1'
    fun isBit(mask: Int): Boolean {
        // Reads the current input on the board
        val currentInput = UsbPort.read()
        // Performs an AND logic operation, bit by bit, with the current input and the received mask
        // If it returns the mask (true), means that the bit we want to evaluate is indeed set to logical '1'
        // otherwise returns false, indicating that bit is set to logical '0'
        return currentInput.and(mask) == mask
    }
    // Returns the values of the bits represented by the mask in the UsbPort
    fun readBits(mask: Int): Int {
        // Reads the current input on the board
        val currentInput = UsbPort.read()
        // Returns the result of an AND logic operation, bit by bit, between the currentInput and the mask
        return currentInput.and(mask)
    }
    // Writes in the bits represented by the mask the given value
    fun writeBits(mask: Int, value: Int) {
        // Inverts mask bits
        val invertedMask: Int = mask.inv()
        // This secondMask variable will have the bits we want to keep and the bits we want to
        // change set to logical '0'
        val secondMask: Int = lastOutput.and(invertedMask)
        // For the output, we want to merge the bits represented by the value with the bits in secondMask
        lastOutput = secondMask.or(value)
        UsbPort.write(lastOutput)
    }
    // Sets bits represented by the mask as logical '1'
    fun setBits(mask: Int) {
        // Writes on the output of the board the result of an OR logic operation, bit by bit,
        // between the lastOutput and the mask
        lastOutput = lastOutput.or(mask)
        UsbPort.write(lastOutput)
```

```
    }
    // Sets bits represented by the mask as logical '0'
    fun clrBits(mask: Int) {
        // Inverts mask bits
        val invertedMask: Int = mask.inv()
        // Writes on the output of the board the result of an AND logic
operation, bit by bit,
        // between the lastOutput and the invertedMask
        lastOutput = lastOutput.and(invertedMask)
        UsbPort.write(lastOutput)
    }
}
```

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

```
package ticketMachine

// Initialize a constant to represent the absent of a valid key code
const val INVALID_KEYCODE = -1

// Receives the frame given by the KeyBoard Reader
object KeyReceiver {
    // Constant values for the masks used in the inputport of UsbPort
    private const val RXD_MASK = 0b00000010 // inputPort(1) of UsbPort
    // Constant values for the masks used in the outputport of UsbPort
    private const val RXCLK_MASK = 0b00001000 // outputPort(3) of UsbPort
    // Constant values
    private const val MAX_CLOCKS = 7
    // Initialize a clock counter to ensure Key Receiver is syncronised with Key
Transmitter after
    // TXD was set to logical '0'
    private var clockCounter = 0
    // Initializes this class
    fun init() {
        // Set RXCLK value with logical '0'
        HAL.clrBits(RXCLK_MASK)
    }
    // Receives a frame and returns the code of a key if it exists
    fun rcv(): Int {
        // Reset clock Counter
        clockCounter = 0
        // Create a mutable variable that will represent the code of a key
        // By default, is set to an invalid key code
        var keyCode: Int = INVALID_KEYCODE
        // Check if TXD value is set to logical '0', this way ensuring that RXCLK
activation only
        // occurs when Key Transmitter is ready to send a key code
        if (!HAL.isBit(RXD_MASK)) {
            // Enables 1 clock cycle for RXCLK
            enableRXCLKCycle()
            // Check if TXD value is set to logical '1' (START BIT)
            if (HAL.isBit(RXD_MASK)) {
                // Enables 1 clock cycle for RXCLK
                enableRXCLKCycle()
                // Start keyCode set to zero
                keyCode = 0
                // Check TXD value in order to build the key code Key Transmitter is
sending
                // Every key code is 4 bits long and Key Transmitter sends LSB->MSB
                for (i in 0..3) {
                    // Check if TXD value is set to logical '1'
                    if (HAL.isBit(RXD_MASK)) {
                        // Calculate the new bit position in the keyCode frame
                        val newBit: Int = 1.shl(bitCount = i)
                        // Add the new bit (K[i]) to the keyCode frame
                        keyCode = keyCode.or(newBit)
                    }
                    // Enables 1 clock cycle for RXCLK
                    enableRXCLKCycle()
```

```
                }
                // Check if TXD value is set to logical '0' (STOP BIT)
                if (!HAL.isBit(RXD_MASK)) {
                    // // Enables 1 clock cycle for RXCLK
                    enableRXCLKCycle()
                }
            }
            // While to ensure Key Transmitter has completed key code sending
protocol
            while (clockCounter < MAX_CLOCKS) {
                // Enables 1 clock cycle for RXCLK
                enableRXCLKCycle()
            }
        }
        return keyCode
    }
    // Enables 1 clock cycle for RXCLK and increments clock counter by 1
    private fun enableRXCLKCycle() {
        // Set RXCLK value with logical '1'
        HAL.setBits(RXCLK_MASK)
        // Set RXCLK value with logical '0'
        HAL.clrBits(RXCLK_MASK)
        // Increments clock counter by 1
        clockCounter++
    }
}
```

D. Código *Kotlin* da classe *KBD*

```
package ticketMachine

import isel.leic.utils.Time

// Read keys. Methods return '0'..'9','#','*' or NONE
object KBD {
    // This enum class will help declare which state KBD is currently at
    enum class State {
        NUMERIC, // Disables Arrow Keys Selection
        SELECTION // Enables Arrow Keys Selection
    }
    // Initialize a constant to represent the absence of a valid key
    const val NONE = 0
    // Initialize a mutable variable to represent the current state of KBD
    var currentState: State = State.NUMERIC
    // Create an array to store the all avalaible keys on KBD.
    // The index of a key correspond to its codification given by the Key Decode
hardware module
    private val KEY_CODE: List<Char> =
listOf('1','4','7','*','2','5','8','0','3','6','9','#')
    // Initializes this class
    fun init() {
        // Sets default KBD state to NUMERIC
        currentState = State.NUMERIC
    }
    // Implements the serial interaction with Key Transmitter
    private fun getKeySerial(): Char {
        // Get a Key code from Key Transmitter hardware module
        val keyCode = KeyReceiver.rcv()
        return if (keyCode != INVALID_KEYCODE || keyCode in KEY_CODE.indices) {
            // Returns the Key which the keyCode corresponds to in the KEY_CODE
array
            KEY_CODE[keyCode]
        } else {
            NONE.toChar()
        }
    }
    // Returns the pressed Key or NONE if no key is currently being pressed
    fun getKey(): Char {
       return getKeySerial()
    }
    // Returns when a key is pressed or NONE after a timeout, in milliseconds, as
occurred
    fun waitKey(timeout: Long): Char {
        // Create a variable to store the current time plus the timeout time
        val stopTime = Time.getTimeInMillis() + timeout
        // Get a Key from Key Receiver
        var key = getKey()
        // Active flag for when the key is found
        var found: Boolean = false
        while (Time.getTimeInMillis() < stopTime && !found) {
            // If a Key was pressed within the timeout time:
            if (key != NONE.toChar()) {
                // A valid key was found
```

```
                found = true
            } else {
                // Keep searching for the key
                key = getKey()
            }
        }
        return key
    }
}
```

E. Código *Kotlin* da classe *SerialEmitter*

```
package ticketMachine

// Send frames for the different modules of the Serial Receiver
object SerialEmitter {
    // Constant values for the masks used in the inputport of UsbPort
    private const val BUSY_MASK = 0b00000001 // inputPort(0) of UsbPort
    // Constant values for the masks used in the outputport of UsbPort
    private const val SDX_MASK  = 0b00000001 // outputPort(0) of UsbPort
    private const val SCLK_MASK = 0b00000010 // outputPort(1) of UsbPort
    private const val SS_MASK   = 0b00000100 // outputPort(2) of UsbPort
    // This enum class will help declare for which module we want to send the
frames
    enum class Destination { LCD, TICKET_DISPENSER }
    // Initializes a default mask to always target the first bit (bit 0)
    private const val LSB_MASK = 0b00000001
    // Initialize a flag to indicate when printing on console is required
    var EN_PRINT: Boolean = false
    // Initializes this class
    fun init() {
        // Set SDX value with logical '0'
        HAL.clrBits(SDX_MASK)
        // Set SCLK value with logical '0'
        HAL.clrBits(SCLK_MASK)
        // Set SS value with logical '1'
        HAL.setBits(SS_MASK)
        // Disables this function to print on console
        EN_PRINT = false
    }
    // Sends a frame to SerialReceiver identifying the destination with "addr"
and the
    // bits containing the data with "data", which has a total of 9 bits
    fun send(addr: Destination, data: Int) {
        // Waits for Busy signal to be disabled
        while (isBusy());
        // Set SS value with logical '0'
        HAL.clrBits(SS_MASK)
        // Build frame with 10 bits while making Destination bit LSB
        val din = (data.shl(1)).or(addr.ordinal)
        // Find parity bit
        val parityBit = findParityBit(din)
        // Full frame with MSB as the parity Bit
        var frame: Int = (parityBit.shl(10)).or(din)
        // With the full frame built, start a for loop to send each bit in
every ascending transition of SCLK
        for (i in 10 downTo 0) {
            // Evaluate if a frame has at least one bit set to logical '1'
            if (frame >= 1) {
                // Use checkLSB extension function to evaluate if the LSB of
the frame
                // is set to either '1' (true) or '0' (false)
                if (frame.checkLSB()) {
                    // Set SDX value with logical '1'
                    HAL.setBits(SDX_MASK)
                    if (EN_PRINT) println("SDX: 1")
```

```
                } else {
                    // Set SDX value with logical '0'
                    HAL.clrBits(SDX_MASK)
                    if (EN_PRINT) println("SDX: 0")
                }
                // Shift frame 1 bit to the right, this way discarding the LSB
                frame = frame.shr(1)
            } else {
                // Set SDX value with logical '0'
                HAL.clrBits(SDX_MASK)
                if (EN_PRINT) println("SDX: 0")
            }
            // Enables 1 clock cycle for SCLK
            enableSCLKCycle()
        }
        // Set SS value with logical '1'
        HAL.setBits(SS_MASK)
    }
    // Enables 1 clock cycle for SCLK
    private fun enableSCLKCycle() {
        // Set SCLK value with logical '1'
        HAL.setBits(SCLK_MASK)
        // Set SCLK value with logical '0'
        HAL.clrBits(SCLK_MASK)
    }
    // Returns true if busy, hardware output, is set to logical '1'
    fun isBusy(): Boolean = HAL.isBit(BUSY_MASK)
    // All the bits set to logical '1' in din have to be checked in order to
assert the
    // parity bit (MSB of the frame) with either 0 or 1, this way preserving
the agreed even parity.
    // This algorithm, which is generic, will find the parity bit while
consecutively compressing the initial number
    // with unassigned right shifts which will become smaller and smaller
until the end of din is reached
    // At that point, the parity bit can be found at the LSB
    // Time complexity: O(log(n)), whereas n is the number of bits of din
    private fun findParityBit(din: Int): Int {
        var d: Int = din
        // d = d xor (d.ushr(32))
        d = d xor (d.ushr(16))
        d = d xor (d.ushr(8))
        d = d xor (d.ushr(4))
        d = d xor (d.ushr(2))
        d = d xor (d.ushr(1))
        return (d and 1)
    }
    // Extension function to check if the LSB bit is either 1 (true) or 0
(false)
    fun Int.checkLSB(): Boolean {
        return this.and(LSB_MASK) == 1
    }
}
```

F. Código *Kotlin* da classe *LCD*

```
package ticketMachine

import isel.leic.utils.Time

// Writes to the LCD module using the 8 bit interface
object LCD {
    // Constant values
    private const val RS_VALUE = 1
    private const val DDRAM_DEFAULT_ADDR = 0x80
    private const val FIRST_ADDRESS_LOWER_LINE = 0x40
    private const val FIRST_COLUMN = 0
    // LCD initialization command values
    private const val INIT_SET = 0x30
    private const val INIT_SET_2 = 0x38
    private const val DISPLAY_OFF = 0x08
    private const val DISPLAY_CLEAR = 0x01
    private const val ENTRY_MODE_SET = 0x06 // 0x07 (To Enable S (LSB) Bit for
Display Shift)
    // LCD command values
    private const val CURSOR_OFF = 0x0C
    private const val CURSOR_ON = 0x0F
    private const val DISPLAY_ON = 0x0C
    // CGRAM starting addresses (64 bytes (8 avalaible cells each with 8 bytes) of
internal memory which means 8
    // starting addresses for a max of 8 custom characters that can be stored at the
same time in this memory)
    data class CGRAM_CELL(val addr: Int, val cell_n: Int)
    private val CGRAM_CELLS: List<CGRAM_CELL> = listOf(
        CGRAM_CELL(0x40, 0),
        CGRAM_CELL(0x48, 1),
        CGRAM_CELL(0x50, 2),
        CGRAM_CELL(0x58, 3),
        CGRAM_CELL(0x60, 4),
        CGRAM_CELL(0x68, 5),
        CGRAM_CELL(0x70, 6),
        CGRAM_CELL(0x78, 7)
    )
    // CGRAM character codes (Set #1)
    const val EURO_SIGN: Int = 0
    const val DOWNWARDS_ARROW: Int = 1
    const val UPWARDS_ARROW: Int = 2
    const val TRAIN_ICON: Int = 3
    const val RAIL_ICON: Int = 4
    const val TICKET_ICON: Int = 5
    const val COIN_ICON: Int = 6
    const val DEPARTURE_ICON: Int = 7
    // CGRAM character codes (Set #2)
    const val LOCK_ICON: Int = 0
    const val LOCKOPEN_ICON: Int = 1
    const val CONNECTOR_ICON: Int = 2
    const val OUTLET_ICON: Int = 3
    const val LEFT_PROGRESSBAR_ICON: Int = 4
    const val MIDDLE_FULL_PROGRESSBAR_ICON: Int = 5
```

```
    const val MIDDLE_EMPTY_PROGRESSBAR_ICON: Int = 6
    const val RIGHT_PROGRESSBAR_ICON: Int = 7
    // CGROM character codes
    const val RIGHT_ARROW = 0x7E
    const val LEFT_ARROW  = 0x7F
    const val LEFT_PARANTHESIS = 0x5B
    const val RIGHT_PARANTHESIS = 0x5D
    // Time values (in ms)
    private const val DELAY_5MS: Long = 5
    private const val DELAY_1MS: Long = 1
    // Custom Characters Patterns for CGRAM set #1:
    // Euro Sign (€)
    private val euroSignPatternList: List<Int> =
        listOf(0b00110, 0b01001, 0b11100, 0b01000, 0b11100, 0b01001, 0b00110,
0b00000)
    // Downwards Arrow
    private val downwardsArrowPatternList: List<Int> =
        listOf(0b00100, 0b00100, 0b00100, 0b00100, 0b10101, 0b01110, 0b00100,
0b00000)
    // Upwards Arrow
    private val upwardsArrowPatternList: List<Int> =
        listOf(0b00100, 0b01110, 0b10101, 0b00100, 0b00100, 0b00100, 0b00100,
0b00000)
    // Train Icon
    private val trainIconPatternList: List<Int> =
        listOf(0b11111, 0b10001, 0b10001, 0b11111, 0b10101, 0b11111, 0b01010,
0b11111)
    // Rail Icon
    private val railIconPatternList: List<Int> =
        listOf(0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b11111,
0b10101)
    // Ticket Icon
    private val ticketIconPatternList: List<Int> =
        listOf(0b10111, 0b11001, 0b11101, 0b10001, 0b10111, 0b11001, 0b10011,
0b11101)
    // Coin Icon
    private val coinIconPatternList: List<Int> =
        listOf(0b01110, 0b10001, 0b10011, 0b10011, 0b10011, 0b10011, 0b10101,
0b01110)
    // Departure Icon
    private val departureIconPatternList: List<Int> =
        listOf(0b11111, 0b11000, 0b00111, 0b00000, 0b11111, 0b11000, 0b00111,
0b00000)
    // Custom Characters Patterns for CGRAM set #2:
    // Lock Icon
    private val lockIconPatternList: List<Int> =
        listOf(0b01110, 0b10001, 0b10001, 0b11111, 0b11011, 0b11011, 0b11111,
0b00000)
    // Lock Open Icon
    private val lockOpenIconPatternList: List<Int> =
        listOf(0b01110, 0b10000, 0b10001, 0b11111, 0b11011, 0b11011, 0b11111,
0b00000)
    // Connector Open Icon
    private val connectorIconPatternList: List<Int> =
        listOf(0b01010, 0b01010, 0b11111, 0b10001, 0b11011, 0b01110, 0b00100,
0b00100)
```

```
    // Connector Open Icon
    private val outletIconPatternList: List<Int> =
        listOf(0b00100, 0b00100, 0b00100, 0b00100, 0b01110, 0b11111, 0b10101,
0b10101)
    // Left Progress Bar Icon
    private val leftProgressBarIconPatternList: List<Int> =
        listOf(0b01111, 0b11000, 0b10011, 0b10111, 0b10111, 0b10011, 0b11000,
0b01111)
    // Middle Progress Full Bar Icon
    private val middleProgressBarFullIconPatternList: List<Int> =
        listOf(0b11111, 0b00000, 0b11011, 0b11011, 0b11011, 0b11011, 0b00000,
0b11111)
    // Middle Progress Empty Bar Icon
    private val middleProgressBarEmptyIconPatternList: List<Int> =
        listOf(0b11111, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000,
0b11111)
    // Right Progress Bar Icon
    private val rightProgressBarFullIconPatternList: List<Int> =
        listOf(0b11110, 0b00011, 0b11001, 0b11101, 0b11101, 0b11001, 0b00011,
0b11110)
    // This enum class will help declare for which line we want to send data to
    enum class Line {
        UPPER,
        LOWER
    }
    // Writes a byte of command/data to the LCD module
    private fun writeByte(rs: Boolean, data: Int) {
        /** RS stands for register select:
            0 - IR (Instruction Register)
            1 - DR (Data Register)
        */
        if (rs) {
            // Sends a frame to the Serial Emitter with RS set to one
            SerialEmitter.send(addr = SerialEmitter.Destination.LCD, data =
(data.shl(1)).or(RS_VALUE))
        } else {
            // Sends a frame to the Serial Emitter with RS set to zero
            SerialEmitter.send(addr = SerialEmitter.Destination.LCD, data =
data.shl(1))
        }
    }
    // Writes a command to the LCD module
    private fun writeCMD(data: Int) {
        writeByte(false, data)
    }
    // Writes data to the LCD module
    private fun writeDATA(data: Int) {
        writeByte(true, data)
    }
    // Sends initialization sequence to the LCD module
    fun init() {
        /**
          *  INITIALIZE PROCESS
          *   7   6   5   4   3   2   1   0 (LSB)
          *  DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
          *   0   0   1   1   *   *   *   *     FUNCTION SET
```

```
            *  -----WAIT MORE THAN 4.1 MS-----
            *   0   0   1   1   *   *   *   *     FUNCTION SET
            *  -----WAIT MORE THAN 100 uS-----
            *   0   0   1   1   *   *   *   *     FUNCTION SET
            *  -------------------------------
            *   0   0   1   1   N   F   *   *     FUNCTION SET
            *   0   0   0   0   1   0   0   0     DISPLAY OFF
            *   0   0   0   0   0   0   0   1     DISPLAY CLEAR
            *   0   0   0   0   0   1  I/D  S     ENTRY MODE SET

            N = 1 (2 lines) / 0 (1 line)
            F = 1 (5x10 dots) / 0 (5x8 dots)
            I/D = 1 (Increments the DDRAM address by 1 when a character code is
written into or read from DDRAM
                 = 0 (Decrements, which means write in reverse order)
            S = 1 (Accompanies display shift)
          **/
          writeCMD(INIT_SET)
          //---WAIT MORE THAN 4.1 MS---
          Time.sleep(DELAY_5MS)
          writeCMD(INIT_SET)
          //---WAIT MORE THAN 100 uS---
          Time.sleep(DELAY_1MS)
          writeCMD(INIT_SET)
          //---------------------------
          writeCMD(INIT_SET_2)
          writeCMD(DISPLAY_OFF)
          writeCMD(DISPLAY_CLEAR)
          writeCMD(ENTRY_MODE_SET)
          //-----------Cursor----------
          writeCMD(CURSOR_ON)
          // Loads custom characters to CGRAM (set #2 by default)
          writeCGRAM(set = 2)
          // Sets DDRAM to address 0 since AC (Address Counter) was changed in the
previous command
          // to a CGRAM address
          cursor(0,0)
      }
      // Writes a character on the current cursor position
      fun write(c: Char) {
          writeDATA(c.code)
      }
      // Writes a string in the current cursor position
      fun write(text: String) {
          /**
            * Function explanation:
            * Reads the string char by char and send them individually to write(char)
          **/
          text.forEach { write(it) }
      }
      // Writes a CGROM character in the current cursor position
      fun writeCGROMchar(addr: Int) {
          writeDATA(addr)
      }
      // Writes a CGRAM character in the current cursor position
      fun writeCGRAMchar(addr: Int) {
```

```
        writeDATA(addr)
    }
    // Sends a command to place the cursor at a specific position ('line':0..LINES-1
, 'column':0..COLS-1)
    fun cursor(line: Int, column: Int) {
        /**
          * Function explanation:
          * To access DDRAM, DB7 must be set to logical '1', so we need to add this
          * value to data before using writeCMD function
          * If line == 1 (lower line) that means the user wants to put the cursor in
the lower line, so we need to
          * add 0x40 to the address cursor, since the first address of the lower
line is 0x40.
          **/
        var data = column.or(DDRAM_DEFAULT_ADDR)
        if (line == Line.LOWER.ordinal) data = data.or(FIRST_ADDRESS_LOWER_LINE)
        writeCMD(data)
    }
    // Enables or disables cursor and cursor blinking on LCD
    fun enableCursor(status: Boolean) {
        if (status) {
            writeCMD(CURSOR_ON)
        } else {
            writeCMD(CURSOR_OFF)
        }
    }
    // Enables or disables LCD display
    private fun enableDisplay(status: Boolean) {
        if (status) {
            writeCMD(DISPLAY_ON)
        } else {
            writeCMD(DISPLAY_OFF)
        }
    }
    // Displays a new CGRAM custom character set
    fun displayNewCGRAMset(set: Int) {
        enableDisplay(status = false)
        writeCGRAM(set)
        enableDisplay(status = true)
    }
    // Sends a command to clear the LCD screen and places the cursor on the position
(0,0)
    fun clear() {
        writeCMD(DISPLAY_CLEAR)
    }
    // Function to set a cursor to the start of a specified line
    fun moveCursorToLineStart(line: Line) {
        when (line) {
            Line.UPPER -> cursor(Line.UPPER.ordinal, FIRST_COLUMN)
            Line.LOWER -> cursor(Line.LOWER.ordinal, FIRST_COLUMN)
        }
    }
    // Function to delete text on LCD within a specified line and column range.
    fun deleteText(line: Line, col1: Int, col2: Int) {
        var str: String = ""
        // Create a string with length col2 - col1
```

```
        for (i in col1..col2) {
            str += " "
        }
        // Move cursor to specified location
        cursor(line.ordinal, col1)
        // Write on LCD
        write(str)
    }
    // Function to simulate LCD lower line shift
    fun simulateLowerLineShift() {
        Time.sleep(DELAY_1S)
        deleteText(Line.LOWER, 0, 15)
    }
    // Writes on CGRAM all created custom characters
    fun writeCGRAM(set: Int) {
        // Writes first set of custom characters
        if (set == 1) {
            loadEuroSign()
            loadDownwardsArrow()
            loadUpwardsArrow()
            loadTrainIcon()
            loadRailIcon()
            loadTicketIcon()
            loadCoinIcon()
            loadDepartureIcon()
        } else {
            loadLockIcon()
            loadLockOpenIcon()
            loadConnectorIcon()
            loadOutletIcon()
            loadLeftProgressBarIcon()
            loadMiddleFullProgressBarIcon()
            loadMiddleEmptyProgressBarIcon()
            loadRightProgressBarIcon()
        }
    }
    /*****************************************************************
     * First Set of custom CGRAM characters
     *****************************************************************/
    private fun loadEuroSign() {
        // Specify CGRAM address (Selected CGRAM cell #0)
        writeCMD(CGRAM_CELLS[0].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in euroSignPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadDownwardsArrow() {
        // Specify CGRAM address (Selected CGRAM cell #2)
        writeCMD(CGRAM_CELLS[1].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in downwardsArrowPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadUpwardsArrow() {
```

```
        // Specify CGRAM address (Selected CGRAM cell #3)
        writeCMD(CGRAM_CELLS[2].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in upwardsArrowPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadTrainIcon() {
        // Specify CGRAM address (Selected CGRAM cell #4)
        writeCMD(CGRAM_CELLS[3].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in trainIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadRailIcon() {
        // Specify CGRAM address (Selected CGRAM cell #4)
        writeCMD(CGRAM_CELLS[4].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in railIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadTicketIcon() {
        // Specify CGRAM address (Selected CGRAM cell #5)
        writeCMD(CGRAM_CELLS[5].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in ticketIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadCoinIcon() {
        // Specify CGRAM address (Selected CGRAM cell #6)
        writeCMD(CGRAM_CELLS[6].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in coinIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadDepartureIcon() {
        // Specify CGRAM address (Selected CGRAM cell #7)
        writeCMD(CGRAM_CELLS[7].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in departureIconPatternList) {
            writeDATA(pattern)
        }
    }
    /*****************************************************************
    * Second Set of custom CGRAM characters
    *****************************************************************/
    private fun loadLockIcon() {
        // Specify CGRAM address (Selected CGRAM cell #0)
        writeCMD(CGRAM_CELLS[0].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in lockIconPatternList) {
            writeDATA(pattern)
```

```
        }
    }
    private fun loadLockOpenIcon() {
        // Specify CGRAM address (Selected CGRAM cell #1)
        writeCMD(CGRAM_CELLS[1].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in lockOpenIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadConnectorIcon() {
        // Specify CGRAM address (Selected CGRAM cell #2)
        writeCMD(CGRAM_CELLS[2].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in connectorIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadOutletIcon() {
        // Specify CGRAM address (Selected CGRAM cell #3)
        writeCMD(CGRAM_CELLS[3].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in outletIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadLeftProgressBarIcon() {
        // Specify CGRAM address (Selected CGRAM cell #4)
        writeCMD(CGRAM_CELLS[4].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in leftProgressBarIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadMiddleFullProgressBarIcon() {
        // Specify CGRAM address (Selected CGRAM cell #5)
        writeCMD(CGRAM_CELLS[5].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in middleProgressBarFullIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadMiddleEmptyProgressBarIcon() {
        // Specify CGRAM address (Selected CGRAM cell #6)
        writeCMD(CGRAM_CELLS[6].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in middleProgressBarEmptyIconPatternList) {
            writeDATA(pattern)
        }
    }
    private fun loadRightProgressBarIcon() {
        // Specify CGRAM address (Selected CGRAM cell #7)
        writeCMD(CGRAM_CELLS[7].addr)
        // Write to CGRAM all patterns for this custom character
        for (pattern in rightProgressBarFullIconPatternList) {
            writeDATA(pattern)
```

```
                }
            }
        }
```

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

```
package ticketMachine

// Controls the state of the mechanism for ticket printing
object TicketDispenser {
    // Initializes this class
    fun init() {
        // Since the hardware of the TicketDispenser doesn't need prior
activation to enable its use,
        // this function is redundant, but it was kept in order to be coherent
with the current
        // steps used in this application for OOP
    }
    // Sends a command to print and dispense a ticket
    fun print(destinyId: Int, originId: Int, roundTrip: Boolean) {
        // Starts a mutable variable that will hold the entire frame while
it's being built
        // Step1: Shift Origin station code 4 bits to the left, in order to
allocate space for the next instruction
        var data = originId.shl(4)
        // Step2: Perform a OR logic operation between the current frame and
the Destiny station code,
        // to be able to add it to the frame
        data = data.or(destinyId)
        // Step3: Shift current frame 1 bit to the left, in order to allocate
space for the next instruction
        data = data.shl(1)
        // Step4: Perform an OR logic operation between the current frame and
the roundTrip bit,
        // to be able to add it to the frame
        // The boolean roundTrip identifies if a ticket is: a two-way ticket
(true) or a one-way ticket (false)
        if (!roundTrip) {
            data = data.or(1)
        }
        // Send data to the Serial Emitter
        SerialEmitter.send(addr = SerialEmitter.Destination.TICKET_DISPENSER,
data = data)
    }
}
```

H. Código *Kotlin* da classe *TUI*

```
package ticketMachine

import isel.leic.utils.Time

// Implements interaction between KBD and LCD objects
object TUI {
    // Constant values
    private const val MAX_COLUMNS: Int = 16
    // This enum class will help declare for which Position the user wants to align
the text on LCD
    enum class Position {
        LEFT,
        MIDDLE,
        RIGHT,
    }
    fun init() {
        // This function is redundant, but it was kept in order to be coherent with
the current
        // steps used in this application for OOP
    }
    // Function to align a string on the LCD module with a specified position and
line
    fun alignStringPos(pos: Position, str: String, line: LCD.Line) {
        // Initialize a variable to store the length of the given string
        val length = str.length
        // Initialize a variable to store the new cursor column position
        val newcursor_xpos: Int
        // Evaluate given position
        when (pos) {
            Position.LEFT -> {
                // Move cursor to the chosen line start
                LCD.moveCursorToLineStart(line)
                // Write on the LCD the given String
                LCD.write(str)
            }
            Position.MIDDLE -> {
                // Calculate the new cursor position
                newcursor_xpos = ((MAX_COLUMNS - (length)) / 2)
                // Move cursor to the new position
                LCD.cursor(line.ordinal, newcursor_xpos)
                // Write on the LCD the given String
                LCD.write(str)
            }
            Position.RIGHT -> {
                // Calculate the new cursor position
                newcursor_xpos = MAX_COLUMNS - length
                // Move cursor to the new position
                LCD.cursor(line.ordinal, newcursor_xpos)
                // Write on the LCD the given String
                LCD.write(str)
            }
        }
    }
    /*************************************************************
```

```
     * Display functions
     ************************************************************/
    // Function to display a starting screen message and a progress bar upon
initialization on LCD
    fun displayStartingScreen() {
        // First display message
        alignStringPos(Position.MIDDLE, "Ticket Machine", LCD.Line.UPPER)
        Time.sleep(DELAY_1S)
        alignStringPos(Position.MIDDLE, "Welcome", LCD.Line.LOWER)
        Time.sleep(DELAY_1S)
        // Clears LCD previous content
        LCD.clear()
        // Second display message
        alignStringPos(Position.MIDDLE, "Initializing", LCD.Line.UPPER)
        // Draws empty progress bar
        LCD.moveCursorToLineStart(LCD.Line.LOWER)
        LCD.writeCGRAMchar(LCD.LEFT_PROGRESSBAR_ICON)
        for (col in 1..10) {
            LCD.writeCGRAMchar(LCD.MIDDLE_EMPTY_PROGRESSBAR_ICON)
        }
        LCD.writeCGRAMchar(LCD.RIGHT_PROGRESSBAR_ICON)
        Time.sleep(DELAY_500MS)
        // Draws progress bar being filled in
        LCD.cursor(1, 1)
        for (col in 1..10) {
            Time.sleep(DELAY_250MS)
            alignStringPos(Position.RIGHT, "${col*10}%", LCD.Line.LOWER)
            LCD.cursor(LCD.Line.LOWER.ordinal, col)
            LCD.writeCGRAMchar(LCD.MIDDLE_FULL_PROGRESSBAR_ICON)
            if (col == 2) {
                Time.sleep(DELAY_1S)
            }
            if (col == 4) {
                Time.sleep(DELAY_500MS)
            }
            if (col == 7) {
                Time.sleep(DELAY_1S)
            }
            if (col == 8) {
                Time.sleep(DELAY_500MS)
            }
        }
        LCD.simulateLowerLineShift()
        alignStringPos(Position.MIDDLE, "Complete", LCD.Line.LOWER)
        Time.sleep(DELAY_1S)
    }
    // Function to display the home screen for the App on LCD
    fun displayHomeScreen() {
        // Clears LCD previous content
        LCD.clear()
        // Set App HomeScreen layout
        alignStringPos(Position.MIDDLE, "Ticket Machine", LCD.Line.UPPER)
        LCD.moveCursorToLineStart(LCD.Line.LOWER)
        LCD.writeCGRAMchar(LCD.RAIL_ICON)
        LCD.writeCGRAMchar(LCD.TRAIN_ICON)
        LCD.writeCGRAMchar(LCD.RAIL_ICON)
```

```
        alignStringPos(Position.MIDDLE, "Press #", LCD.Line.LOWER)
        LCD.cursor(1, 13)
        LCD.writeCGRAMchar(LCD.RAIL_ICON)
        LCD.writeCGRAMchar(LCD.TRAIN_ICON)
        LCD.writeCGRAMchar(LCD.RAIL_ICON)
    }
    // Function to display a printing ticket message on LCD
    fun displayPrintingTicketScreen() {
        // Clears LCD previous content
        LCD.clear()
        // Sets printing ticket message
        alignStringPos(Position.MIDDLE, "Printing", LCD.Line.UPPER)
        LCD.cursor(1, 4)
        LCD.write("Ticket ")
        LCD.writeCGRAMchar(LCD.TICKET_ICON)
    }
    // Function to display a reset message on LCD
    fun displayResetScreen(reset: Boolean) {
        // Evaluates given reset state
        if (!reset) {
            // Clears LCD previous content
            LCD.clear()
            // Sets reset counters message
            alignStringPos(Position.MIDDLE, "Resetting", LCD.Line.UPPER)
            LCD.cursor(1, 2)
            LCD.write("Counters")
            Time.sleep(DELAY_1S)
            for (i in 0..2) {
                LCD.write(".")
                Time.sleep(DELAY_1S)
            }
        } else {
            // Displays complete message
            LCD.simulateLowerLineShift()
            alignStringPos(Position.MIDDLE, "Complete", LCD.Line.LOWER)
            Time.sleep(DELAY_1S)
        }
    }
    // Function to display shutdown messages on LCD
    fun displayShutdownScreen(dataSent: Boolean) {
        // Evaluates given dataSent state
        if (!dataSent) {
            // Clears LCD previous content
            LCD.clear()
            // Sets shutdown screen before data was sent
            LCD.writeCGRAMchar(LCD.OUTLET_ICON)
            LCD.moveCursorToLineStart(LCD.Line.LOWER)
            LCD.writeCGRAMchar(LCD.CONNECTOR_ICON)
            alignStringPos(Position.MIDDLE, "Sending Data", LCD.Line.UPPER)
            LCD.cursor(0, 15)
            LCD.writeCGRAMchar(LCD.OUTLET_ICON)
            LCD.cursor(1, 15)
            LCD.writeCGRAMchar(LCD.CONNECTOR_ICON)
            alignStringPos(Position.MIDDLE, "to Server", LCD.Line.LOWER)
            Time.sleep(DELAY_2S)
        } else {
```

```
            // Sets shutdown screen after data was sent
            LCD.deleteText(LCD.Line.LOWER, 3, 11)
            alignStringPos(Position.MIDDLE, "Complete", LCD.Line.LOWER)
            Time.sleep(DELAY_1S)
            LCD.deleteText(LCD.Line.LOWER, 0, 0)
            LCD.deleteText(LCD.Line.LOWER, 15, 15)
            Time.sleep(DELAY_500MS)
            // Clears LCD previous content
            LCD.clear()
        }
    }
    // Function to display M starting screen on LCD
    fun displayMStartingScreen() {
        // Clears LCD previous content
        LCD.clear()
        // Sets M starting screen
        alignStringPos(Position.MIDDLE, "Maintenance", LCD.Line.UPPER)
        alignStringPos(Position.MIDDLE, "Mode", LCD.Line.LOWER)
        LCD.cursor(0, 15)
        LCD.writeCGRAMchar(LCD.LOCKOPEN_ICON)
        Time.sleep(DELAY_1S)
        drawMModeLock()
        Time.sleep(DELAY_1S)
    }
    // Function to display M complete screen on LCD
    fun displayMCompleteScreen() {
        // Clears LCD previous content
        LCD.clear()
        // Sets M complete screen
        alignStringPos(Position.MIDDLE, "Maintenance", LCD.Line.UPPER)
        alignStringPos(Position.MIDDLE, "Complete", LCD.Line.LOWER)
        drawMModeLock()
        Time.sleep(DELAY_1S)
        LCD.cursor(0, 15)
        LCD.writeCGRAMchar(LCD.LOCKOPEN_ICON)
        Time.sleep(DELAY_1S)
    }
    // Function to display Show State on LCD
    fun displayShowStateScreen() {
        alignStringPos(Position.LEFT, "Modes:", LCD.Line.UPPER)
        drawMModeLock()
    }
    /*****************************************************************
     * Draw on LCD functions
     *****************************************************************/
    // Function to draw an indication of the current selected KBD state on LCD
    fun drawKBDMode() {
        // Evaluates current KBD state
        when (KBD.currentState) {
            KBD.State.NUMERIC -> {
                // Erases KBD SELECTION state indication from LCD
                LCD.cursor(1, 2)
                LCD.write(":")
                LCD.deleteText(LCD.Line.LOWER, 3, 4)
            }
            KBD.State.SELECTION -> {
```

```
                LCD.cursor(1, 2)
                // Draws two arrows on the bottom left side to indicate KBD
SELECTION mode is active
                LCD.writeCGRAMchar(LCD.UPWARDS_ARROW)
                LCD.writeCGRAMchar(LCD.DOWNWARDS_ARROW)
            }
        }
    }
    // Function to draw the current station ID on LCD
    fun drawStationID(stationID: Int) {
        LCD.moveCursorToLineStart(LCD.Line.LOWER)
        if (stationID in 0..9) {
            // Add a 0 to represent 2 digits for Station's IDs with only 1 digit
            // Example: "5" -> "05"
            LCD.write("0$stationID")
        } else {
            LCD.write("$stationID")
        }
    }
    // Function to draw the current coin ID on LCD
    fun drawCoinID(coinID: Int) {
        LCD.moveCursorToLineStart(LCD.Line.LOWER)
        LCD.write("0$coinID")
    }
    // Function to indicate current roundTrip selection on LCD
    fun drawRoundTripOnLCD(roundTrip: Boolean) {
        LCD.moveCursorToLineStart(LCD.Line.LOWER)
        // Evaluates roundTrip state
        if (!roundTrip) {
            LCD.writeCGRAMchar(LCD.UPWARDS_ARROW)
            LCD.write(" ")
        } else {
            LCD.writeCGRAMchar(LCD.UPWARDS_ARROW)
            LCD.writeCGRAMchar(LCD.DOWNWARDS_ARROW)
        }
    }
    // Function to convert given value in cents to price format in euros (€)
    // Example: value = 250 -> "2,50"
    private fun convertCentsToEuros(value: Int) = String.format("%.2f",
value.toDouble()/100)
    // Function to draw the station correspondent ticket price on LCD
    fun drawTicketPrice(value: Int) {
        // Convert given value to a price format in euros (€)
        val price = convertCentsToEuros(value)
        // Draw price on LCD
        LCD.write(price)
        // Writes custom char (€) next to the price
        LCD.writeCGRAMchar(LCD.EURO_SIGN)
    }
    // Function to draw an indication that M Mode is activated on LCD
    fun drawMModeLock() {
        // Writes a lock icon custom character on the top right side of LCD to
indicate M Mode is active
        LCD.cursor(LCD.Line.UPPER.ordinal, 15)
        LCD.writeCGRAMchar(LCD.LOCK_ICON)
    }
```

```
}
```

I. Código *Kotlin* da classe *FileAccess*

```
package ticketMachine

import java.io.BufferedReader
import java.io.FileReader
import java.io.PrintWriter

// Function to read from a specified file and return data as an Array of Strings
fun readFile(fileName: String): Array<String> {
    // Creates a reader
    val reader = BufferedReader(FileReader(fileName))
    // Creates an array to store all data lines
    var dataArray: Array<String> = emptyArray()
    // Creates a mutable variable to store the current line that bufferedReader is
reading
    var currentLine = reader.readLine()
    while (currentLine != null) {
        // Adds data line to the array
        dataArray += currentLine
        // Moves to next data line
        currentLine = reader.readLine()
    }
    // Closes reader
    reader.close()
    return dataArray
}

// Function to write given data as an Array of Strings to a specified file
fun writeFile(fileName: String, dataArray: Array<String>) {
    // Creates a writer
    val writer = PrintWriter(fileName)
    for (data in dataArray) {
        // Writes the current data line
        writer.println(data)
    }
    // Closes writer
    writer.close()
}
```

J. Código *Kotlin* da classe *Stations*

```
package ticketMachine

import isel.leic.utils.Time

/**
  * Stations file format:
  * PRICE;NUMBER;STATION_NAME
  * EXAMPLE:
  * 100;2;Porto
  * Means: 2 tickets to Porto (each 1€) were sold
**/

// Implements interaction with Stations current information
object Stations {
    // Constant values
    private const val STATIONS_FILENAME: String = "Stations.txt"
    // Data class to caracterize a station
    data class Station (var price: Int, var currentTicketCount: Int, var name:
String, var ID: Int)
    // Initialize an array which will store all station's information
    var stationsInfo: Array<Station> = emptyArray()
    // Creates a variable to indicate the first station ID
    var firstStationID: Int = 0
    // Creates a variable to indicate the last station ID
    var lastStationID: Int = 0
    // Initializes this class
    fun init() {
        // Loads Station's file information
        loadStationsInfo()
        // Sets firstStationID value
        firstStationID= stationsInfo.first().ID
        // Sets lastStationID value
        lastStationID = stationsInfo.last().ID
    }
    // Loads Station's file information
    private fun loadStationsInfo() {
        // Retrieves data from Station's file
        val dataArray = readFile(STATIONS_FILENAME)
        // Initializes a mutable variable to set current Station ID
        var setStationID: Int = 0
        for (line in dataArray.indices) {
            // Splits the current line in three halves: (PRICE;NUMBER;STATION_NAME -
> [0]PRICE [1]NUMBER [2]STATION_NAME)
            val lineList = dataArray[line].split(";")
            // Sets Station ticket price value
            val loadedTicketPrice: Int = lineList[0].toInt()
            // Sets Station ticket count value
            val loadedTicketCount: Int = lineList[1].toInt()
            // Sets Station name
            val loadStationName: String = lineList[2]
            // Sets Station ID value
            val identification = setStationID++
            // Initializes a variable to caracterize the current station
```

```
            val station = Station(loadedTicketPrice, loadedTicketCount,
loadStationName, identification)
            // Assigns previously created station to stationsInfo array
            stationsInfo += station
        }
    }
    // Function to increment the current ticket counter for a given station ID
    fun addTicket(stationID: Int) {
        for (index in stationsInfo.indices) {
            // Search for the corresponding station
            if (stationID == stationsInfo[index].ID) {
                // Increment its current ticket counter by 1
                stationsInfo[index].currentTicketCount++
            }
        }
    }
    // Function to reset all station's ticket counters
    fun resetTicketCounters() {
        for (index in stationsInfo.indices) {
            // Resets current ticket counter
            stationsInfo[index].currentTicketCount = 0
        }
    }
    // Writes data to output file
    fun writeFile() {
        // Create an array to write all data lines to
        var outputArray: Array<String> = emptyArray()
        for (station in stationsInfo) {
            // Calculates the ticket amount to store in the file for the current
station
            val total: Int = station.currentTicketCount
            // Sets a new data line
            val data: String = "${station.price};$total;${station.name}"
            // Adds current data line to outputArray
            outputArray += data
        }
        // Prints to file
        writeFile(fileName = STATIONS_FILENAME, dataArray = outputArray)
    }
}

fun main() {
    // Reads stored information from Stations File
    Stations.init()
    // Resets all station's ticket counters
    Stations.resetTicketCounters()
    // Writes in Stations File
    Stations.writeFile()
    Time.sleep(DELAY_5S)
    // Adds a ticket which was sold for the station with ID: 3
    Stations.addTicket(3)
    // Adds a ticket which was sold for the station with ID: 7
    Stations.addTicket(7)
    Stations.writeFile()
    // Writes in Stations File
    Time.sleep(DELAY_5S)
```

```
    // Resets all station's ticket counters
    Stations.resetTicketCounters()
    Stations.writeFile()
}
```

K. Código *Kotlin* da classe *Coin Deposit*

```
package ticketMachine

import isel.leic.utils.Time

/**
  * Coins file format:
  * COIN;NUMBER
  * EXAMPLE:
  * 2;3
  * Means: 3 coins of 2€ are deposited in the Coin Deposit vault
**/

// Implements interaction with Coin Acceptor and the Coin Deposit vault
object CoinDeposit {
    // Constant values
    private const val CD_FILENAME: String = "CoinDeposit.txt"
    // Data class to caracterize a coin type
    data class Coin (val type: Int, var currentCount: Int, var ID: Int)
    // Initialize an array which will store all coin's information currently in the
Coin Deposit vault
    var storedCoins: Array<Coin> = emptyArray()
    // Initialize an array which will store all coin's information during a ticket
purchase
    var insertedCoins: Array<Coin> = emptyArray()
    // Creates a variable to indicate the first coin ID
    var firstCoinID: Int = 0
    // Creates a variable to indicate the last coin ID
    var lastCoinID: Int = 0
    // Initializes this class
    fun init() {
        // Loads Coin's file information
        loadStoredCoins()
        // Sets firstCoinID value
        firstCoinID = storedCoins.first().ID
        // Sets lastCoinID value
        lastCoinID = storedCoins.last().ID
        // Reset insertedCoins counters
        ejectInsertedCoins()
    }
    // Loads Coin's file information
    private fun loadStoredCoins() {
        // Retrieves data from Coin's file
        val dataArray = readFile(CD_FILENAME)
        // Initializes a mutable variable to set current Station ID
        var setCoinID: Int = 0
        for (line in dataArray.indices) {
            // Splits the current line in two halves: (COIN;NUMBER -> [0]COIN
[1]NUMBER)
            val lineList = dataArray[line].split(";")
            // Sets Coin type
            val loadedCoinType = lineList[0].toInt()
            // Sets Coin count
            val loadedCoinCount = lineList[1].toInt()
            // Sets Station ID value
```

```
            val identification = setCoinID++
            // Initializes a variable to caracterize the current coin
            val coin1 = Coin(loadedCoinType, loadedCoinCount, identification)
            // Assigns previously created coin to storedCoins array
            storedCoins += coin1
            // Another instance of the same coin was created so that it can be
referenced by the insertedCoins
            // array without altering previous created objects for the storedCoins
array
            val coin2 = Coin(loadedCoinType, loadedCoinCount, identification)
            // Initially insertedCoins will mimic stored coins
            insertedCoins += coin2
        }
    }
    // Function to increment the current counter of the given type of coin
    fun add(type: Int) {
        for (index in insertedCoins.indices) {
            if (type == insertedCoins[index].type) {
                // Increments current counter for this coin type
                insertedCoins[index].currentCount++
                // Type was found, there's no need to keep searching
                break
            }
        }
    }
    // Function to collect all inserted stored coins into the Coin Deposit vault
    fun collectStoredCoins() {
        for (index in storedCoins.indices) {
            // The collected coins for this type will be the given by the current
stored coins
            // plus the current inserted coins
            storedCoins[index].currentCount += insertedCoins[index].currentCount
        }
    }
    // Function to eject all inserted coins counters
    fun ejectInsertedCoins() {
        for (coin in insertedCoins) {
            // Resets current counter
            coin.currentCount = 0
        }
    }
    // Function to reset all stored coin counters
    fun resetCoinCounters() {
        for (coin in storedCoins) {
            // Resets current counter
            coin.currentCount = 0
        }
    }
    // Writes data to output file
    fun writeFile() {
        // Create an array to write all data lines to
        var outputArray: Array<String> = emptyArray()
        for (coin in storedCoins) {
            // Calculates the coin amount to store in the file for the current coin
type
            val total: Int = coin.currentCount
```

```
            // Sets a new data line
            val data: String = "${coin.type};$total"
            // Adds current data line to outputArray
            outputArray += data
        }
        // Prints to file
        writeFile(fileName = CD_FILENAME, dataArray = outputArray)
    }
}

fun main () {
    CoinDeposit.init()
    CoinDeposit.resetCoinCounters()
    // Should not change the file
    CoinDeposit.add(0)
    // Adds a 0.05€ coin
    CoinDeposit.add(5)
    // Adds a 1€ coin
    CoinDeposit.add(100)
    // Adds a 0.20€ coin
    CoinDeposit.add(20)
    // Adds a 1€ coin
    CoinDeposit.add(100)
    // Writes File
    CoinDeposit.writeFile()
    Time.sleep(DELAY_5S)
    // Should not change the file
    CoinDeposit.add(0)
    // Adds a 0.05€ coin
    CoinDeposit.add(5)
    // Adds a 2€ coin
    CoinDeposit.add(200)
    // Adds a 0.20€ coin
    CoinDeposit.add(20)
    // Adds a 0.20€ coin
    CoinDeposit.add(20)
    // Should not change the file
    CoinDeposit.add(3)
    CoinDeposit.writeFile()
    Time.sleep(DELAY_5S)
    CoinDeposit.resetCoinCounters()
    Time.sleep(DELAY_5S)
    CoinDeposit.writeFile()
}
```

L. Código *Kotlin* da classe *CoinAcceptor*

```
package ticketMachine

import isel.leic.utils.Time

// Constant values
const val DELAY2: Long = 2000

// Implements the iterface with the Coin Acceptor hardware module
object CoinAcceptor {
    // Constant values for the masks used in the inputport of UsbPort
    // CID stands for Coin ID
    private const val CID_MASK  = 0b00011100 // inputPort(4, 3, 2) of UsbPort
    private const val COIN_MASK = 0b00100000 // inputPort(5) of UsbPort
    // Constant values for the masks used in the outputport of UsbPort
    private const val COIN_EJECT_MASK   = 0b00010000 // outputPort(4) of UsbPort
    private const val COIN_COLLECT_MASK = 0b00100000 // outputPort(5) of UsbPort
    private const val COIN_ACCEPT_MASK  = 0b01000000 // outputPort(6) of UsbPort
    // Constant values
    private const val FIRST_CID = 0 // Index 0
    private const val LAST_CID = 6 // Actual last index is 5 but until keyword is
exclusive, so it was
                                   // incremented to 6
    // Lists
    val coinCode_List: List<Int> = listOf(5, 10, 20, 50, 100, 200, 0, 0) // Coins in
cents
    // Initializes this class
    fun init() {
        // Set COIN_ACCEPT value with logical '0'
        HAL.clrBits(COIN_ACCEPT_MASK)
        // Set COIN_EJECT value with logical '0'
        HAL.clrBits(COIN_EJECT_MASK)
        // Set COIN_COLLECT value with logical '0'
        HAL.clrBits(COIN_COLLECT_MASK)
    }
    // Returns true if a new coin was inserted
    fun hasCoin(): Boolean {
        return HAL.isBit(COIN_MASK)
    }
    // Returns the true value of the inserted coin
    fun getCoinValue(): Int {
        // Check if a coin was inserted
        if (hasCoin()) {
            // Create a mutable variable that will represent the true value of the
inserted coin (value frame)
            val value = HAL.readBits(CID_MASK).shr(2)
            // Check if coin value is in the accepted coin range
            if (value in FIRST_CID until LAST_CID) {
                return coinCode_List[value]
            }
        }
        // No coin was inserted
        return 0
    }
    // Informs the CoinAcceptor that the coin was accounted for
```

```
    fun acceptCoin() {
        if (hasCoin()) {
            // Set COIN_ACCEPT value with logical '1'
            HAL.setBits(COIN_ACCEPT_MASK)
            // Checks if coin switch was returned to its original position
            while (hasCoin());
            // Set COIN_ACCEPT value with logical '0'
            HAL.clrBits(COIN_ACCEPT_MASK)
        }
    }
    // Returns all coins currently stored in the CoinAcceptor
    fun ejectCoins() {
        // Set COIN_EJECT value with logical '1'
        HAL.setBits(COIN_EJECT_MASK)
        // Ejects inserted coins
        CoinDeposit.ejectInsertedCoins()
        // Active only for 2 seconds
        Time.sleep(DELAY2)
        // Set COIN_EJECT value with logical '0'
        HAL.clrBits(COIN_EJECT_MASK)
    }
    // Collects all coins currently stored in the CoinAcceptor
    fun collectCoins() {
        // Set COIN_COLLECT value with logical '1'
        HAL.setBits(COIN_COLLECT_MASK)
        // Collects stored coins
        CoinDeposit.collectStoredCoins()
        // Active only for 2 seconds
        Time.sleep(DELAY2)
        // Set COIN_COLLECT value with logical '0'
        HAL.clrBits(COIN_COLLECT_MASK)
    }
}
```

M. Código *Kotlin* da classe *M*

```
package ticketMachine

// Implements M (Maintenance Mode) Interface
object M {
    // Constant values for the masks used on the inputport of UsbPort
    private const val M_MASK = 0b10000000 // inputPort(7) of UsbPort
    fun init() {
        // This function is redundant, but it was kept in order to be coherent with
the current
        // steps used in this application for OOP
    }
    // Check if M Mode was actived (true) or disabled (false)
    fun status(): Boolean = HAL.isBit(M_MASK)
}
```

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

```
package ticketMachine

import isel.leic.utils.Time
import kotlin.system.exitProcess

// Time constant values (in ms)
const val DELAY_250MS: Long = 250
const val DELAY_500MS: Long = 500
const val DELAY_1S: Long = 1000
const val DELAY_2S: Long = 2000
const val DELAY_5S: Long = 5000

// Implements Ticket Machine App Interface
object App {
    // Constant values
    private const val EMPTY_CHAR: Char = ' '
    private const val ZERO_CODE: Int = '0'.code
    private const val FIVE_CODE: Int = '5'.code
    const val VENDING_SET: Int = 1 // Represents a specific set of CGRAM custom
characters
    const val M_SET: Int = 2 // Represents a specific set of CGRAM custom characters
    // This enum class will help declare which mode Ticket Machine App is currently
at
    enum class Mode {
        VENDING, // Vending Mode
        M, // Maintenance mode
    }
    // This enum class will help declare which state Ticket Machine App is currently
at
    enum class State {

/*************************************************************************
         *  Mode VENDING Avalaible States

*************************************************************************/
        HOMESCREEN, // Represents the default state for this mode
                    // Displays a default message before a purchase process is
initiated by the user
        PURCHASE, // Prompts the user to choose a station to buy a ticket to.
Stations can be searched
                  // by its corresponding ID or with KBD.SELECTION mode toggled
        PAYMENT, // Allows the user to buy the selected ticket and choose between a
one-way or a two-way ticket,
                 // and the price reflects the change in this last selection
        PRINT, // Prints selected ticket and displays important information for the
user

/*************************************************************************
         *  Mode M Avalaible States

*************************************************************************/
        SHOW, // Represents the default state for this mode
              // Shows all avalaible Maintenance Modes to select from on LCD
```

```
        TICKET_TEST, // Allows the consultation and sale of a ticket without
inserting any of its corresponding
                     // currencies and without this transaction being counted as an
acquisition
        TICKET_CNT, // Allows the visualization of every ticket printed for a
specified station and the
                    // listing of all stations can be done with KBD.SELECTION mode
toggled
        COINS_CNT, // Allows the visualization of every coin counter and the listing
of all coin
                   // counters can be done with KBD.SELECTION mode toggled
        RESET, // Sets the coins and ticket counters to zero, starting a new
counting cycle
        SHUTDOWN // System shutdown command. Writes all information regarding coin
amount stored
                 // and tickets sold in their corresponding files
    }
    // Initialize a mutable variable to represent the current selected Mode of the
Ticket Machine App
    var currentMode: Mode = Mode.VENDING
    // Initialize a mutable variable to represent the current state of the Ticket
Machine App
    var currentState: State = State.HOMESCREEN
    // Initialize a mutable variable to represent the current Station ID
    var currentStationID: Int = 0
    // Initialize a mutable variable to represent the current Coin ID
    var currentCoinID: Int = 0
    // Initialize a boolean variable to represent roundTrip selection:
    // RoundTrip identifies if a ticket is: a two-way ticket (true) or a one-way
ticket (false)
    var roundTrip: Boolean = false
    // Initialize a mutable variable to represent the last key pressed by the user
    var lastKeyPressed: Char = EMPTY_CHAR
    // Initialize a mutable variable to represent the current ticket left to pay
price
    var ticketleftToPayPrice: Int = 0
    // Initialize a mutable variable to represent the current total true value of
the coins inserted by the user
    var coinAmount: Int = 0
    // Initializes this class
    fun init() {
        // Initializes HAL object
        HAL.init()
        // Initializes SerialEmitter object
        SerialEmitter.init()
        // Initializes KeyReceiver object
        KeyReceiver.init()
        // Initializes KeyBoardReader object
        KBD.init()
        // Initializes LCD module
        LCD.init()
        // Initializes TicketDispenser object
        TicketDispenser.init()
        // Initializes TUI object
        TUI.init()
        // Initializes CoinAcceptor object
```

```
        CoinAcceptor.init()
        // Initializes CoinDeposit object
        CoinDeposit.init()
        // Initializes Stations object
        Stations.init()
        // Initializes M object
        M.init()
        // Sets default Ticket Machine App mode to VENDING
        currentMode = Mode.VENDING
        // Sets default Ticket Machine App state to HOMESCREEN
        currentState = State.HOMESCREEN
        // Disables LCD cursor
        LCD.enableCursor(false)
        // Displays starting screen
        TUI.displayStartingScreen()
    }
    /************************************************************
     * Display functions
     ************************************************************/
    // Function to display the current station information on LCD, depending on the
current App mode
    // and a boolean to indicate if the price is shown or not
    fun displayCurrentStationInfo(mode: Mode = Mode.VENDING, price: Boolean = true)
{
        // Clears LCD previous content
        LCD.clear()
        // Initializes a variable to represent the current station
        val station = Stations.stationsInfo[currentStationID]
        TUI.alignStringPos(TUI.Position.MIDDLE, station.name, LCD.Line.UPPER)
        // Draws station ID in the bottom left corner
        TUI.drawStationID(stationID = currentStationID)
        if (mode == Mode.M && !price) {
            // Draws a ticket icon in the upper right corner
            LCD.cursor(0, 15)
            LCD.writeCGRAMchar(LCD.TICKET_ICON)
            // Initializes a variable to represent the current ticket count for this
station
            val ticketCount: Int = station.currentTicketCount
            TUI.alignStringPos(TUI.Position.RIGHT, "$ticketCount", LCD.Line.LOWER)
        } else {
            // Draws ticket price in the bottom right corner of the screen
            LCD.cursor(1, 11)
            TUI.drawTicketPrice(station.price)
        }
        TUI.drawKBDMode()
    }
    // Function to display the current coin selected information on LCD
    fun displayCurrentCoinInfo() {
        // Clears LCD previous content
        LCD.clear()
        LCD.cursor(0, 5)
        // Initializes a variable to represent the current selected coin type
        val coinType = CoinDeposit.storedCoins[currentCoinID].type
        // Draws selected coin in the middle of the upper line
        TUI.drawTicketPrice(coinType)
        // Draws a coin in the upper right corner
```

```
            LCD.cursor(0, 15)
            LCD.writeCGRAMchar(LCD.COIN_ICON)
            // Draws coin ID in the bottom left corner
            TUI.drawCoinID(coinID = currentCoinID)
            // Initializes a variable to represent the current selected coin counter
            val coinCount: Int = CoinDeposit.storedCoins[currentCoinID].currentCount
            TUI.alignStringPos(TUI.Position.RIGHT, "$coinCount", LCD.Line.LOWER)
            TUI.drawKBDMode()
        }
        // Function to display an abort purchase message
        fun displayAbortedPurchaseScreen() {
            // Clears LCD previous content
            LCD.clear()
            // Displays message
            TUI.alignStringPos(TUI.Position.MIDDLE, "Vending", LCD.Line.UPPER)
            TUI.alignStringPos(TUI.Position.MIDDLE, "Was aborted", LCD.Line.LOWER)
            // Returns inserted coins to the user
            CoinAcceptor.ejectCoins()
        }
        // Function to display Payment information on LCD
        fun displayPaymentScreen(mode: Mode) {
            // Clears LCD previous content
            LCD.clear()
            // Initializes a variable to represent the current selected station
            val station = Stations.stationsInfo[currentStationID]
            TUI.alignStringPos(TUI.Position.MIDDLE, station.name, LCD.Line.UPPER)
            // Draws current roundTrip indication in the bottom left corner
            TUI.drawRoundTripOnLCD(roundTrip = roundTrip)
            if (mode == Mode.VENDING) {
                LCD.cursor(1, 5)
                // Initializes a variable to represent the price to be paid by the user
                val price = if (roundTrip) {
                    // Double price variable value
                    station.price * 2
                } else {
                    station.price
                }
                // Update ticketleftToPayPrice
                ticketleftToPayPrice = price - coinAmount
                // Draw left to pay ticket price in the middle of the lower line
                TUI.drawTicketPrice(value = ticketleftToPayPrice)
            } else {
                TUI.alignStringPos(TUI.Position.MIDDLE, "*-To Print", LCD.Line.LOWER)
            }
        }
        // Function to display Ticket Printing information on LCD
        fun displayPrintScreen(mode: Mode, ticketCollected: Boolean) {
            // Evaluates if the ticket was collected
            if (!ticketCollected) {
                // Clears LCD previous content
                LCD.clear()
                // Initializes a variable to represent the current selected station
                val station = Stations.stationsInfo[currentStationID]
                // Display message
                TUI.alignStringPos(TUI.Position.MIDDLE, "Dst: ${station.name}",
LCD.Line.UPPER)
```

```
            TUI.alignStringPos(TUI.Position.MIDDLE, "Collect Ticket",
LCD.Line.LOWER)
        } else {
            // Clears LCD previous content
            LCD.clear()
            // First display message
            TUI.alignStringPos(TUI.Position.MIDDLE, "Printing", LCD.Line.UPPER)
            TUI.alignStringPos(TUI.Position.MIDDLE, "Receipt", LCD.Line.LOWER)
            Time.sleep(DELAY_2S)
            // Second display message
            TUI.alignStringPos(TUI.Position.MIDDLE, "Consult Train", LCD.Line.UPPER)
            TUI.alignStringPos(TUI.Position.LEFT, " Departures ", LCD.Line.LOWER)
            LCD.writeCGRAMchar(LCD.DEPARTURE_ICON)
            LCD.writeCGRAMchar(LCD.DEPARTURE_ICON)
            LCD.writeCGRAMchar(LCD.DEPARTURE_ICON)
            Time.sleep(DELAY_2S)
            // Update state depending on current App mode
            currentState = if (mode == Mode.M) {
                State.SHOW
            } else {
                State.HOMESCREEN
            }
        }
    }
    // Function to display a query request screen depending on the given state
    fun displayQueryRequestScreen(state: State) {
        // Clears LCD previous content
        LCD.clear()
        // Draws a lock custom character on the top right side of LCD to indicate M
Mode is active
        TUI.drawMModeLock()
        // Writes display message based on the given state
        if (state == State.RESET) {
            TUI.alignStringPos(TUI.Position.LEFT, "Reset Counters", LCD.Line.UPPER)
            TUI.alignStringPos(TUI.Position.LEFT, "5-Yes Other-No", LCD.Line.LOWER)
        } else if (state == State.SHUTDOWN) {
            TUI.alignStringPos(TUI.Position.MIDDLE, "Shutdown", LCD.Line.UPPER)
            TUI.alignStringPos(TUI.Position.MIDDLE, "5-Yes Other-No",
LCD.Line.LOWER)
        }
    }
    /*****************************************************************
     * Auxiliary functions
     *****************************************************************/
    // Function to reset App variables before starting a new state
    fun resetVariables() {
        // Sets KBD mode to NUMERIC
        KBD.currentState = KBD.State.NUMERIC
        // Sets currentStationID counter to first station ID
        currentStationID = Stations.firstStationID
        // Sets currentCoinID counter to first coin ID
        currentCoinID = CoinDeposit.firstCoinID
        // Sets roundTrip default choice to false (One-way)
        roundTrip = false
        // Resets user input coin amount
        coinAmount = 0
```

```
        // Resets ticket price value
        ticketleftToPayPrice = 0
    }
    // Function to enable currentCoinID variable to be updated if the key received
actual value is a valid CID
    private fun updateCurrentCoinID(key: Char) {
        // Initialize a variable to place in a string the given key
        val value = key.toString()
        // Evaluates if this value is within the accepted coin IDs
        if (value.toInt() in CoinDeposit.firstCoinID..CoinDeposit.lastCoinID) {
            // Update currentCoinID with an actual value
            currentCoinID = value.toInt()
        }
    }
    // Function to evaluate a key according to the current App state
    fun evaluateKey(key: Char) {
        when (currentState) {
            State.PURCHASE -> {
                when (key) {
                    '2' -> {
                        // Evaluates current KBD state
                        when (KBD.currentState) {
                            KBD.State.NUMERIC -> {
                                // Evaluates last key pressed:
                                if (lastKeyPressed == EMPTY_CHAR) {
                                    // Initializes a variable to place in a string
the given key
                                    val value = key.toString()
                                    // Updates currentStationID with an actual value
                                    currentStationID = value.toInt()
                                } else {
                                    // Initializes a variable to store both keys as a
string
                                    val num: String = lastKeyPressed.toString() + key
                                    // Updates currentStationID with an actual value
                                    currentStationID = num.toInt()
                                    // Updates lastKeyPressed
                                    lastKeyPressed = EMPTY_CHAR
                                }
                            }
                            KBD.State.SELECTION -> {
                                if (currentStationID == Stations.lastStationID) {
                                    // If the currentStation counter reaches the last
station correspondent ID it
                                    // resets to the first station ID
                                    currentStationID = Stations.firstStationID
                                } else {
                                    // Increases currentStation counter
                                    currentStationID++
                                }
                            }
                        }
                    }
                    '8' -> {
                        // Evaluates current KBD state
                        when (KBD.currentState) {
```

```
                            KBD.State.NUMERIC -> {
                                // Initialize a variable to place in a string the
given key
                                val value = key.toString()
                                // Update currentStationID with an actual value
                                currentStationID = value.toInt()
                                // Update lastKeyPressed
                                lastKeyPressed = EMPTY_CHAR
                            }
                            KBD.State.SELECTION -> {
                                if (currentStationID == Stations.firstStationID) {
                                    // If the currentStationID counter reaches the
first station correspondent ID it
                                    // resets to the last station ID
                                    currentStationID = Stations.lastStationID
                                } else {
                                    // Decreases currentStation counter
                                    currentStationID--
                                }
                            }
                        }
                    }
                    '1' -> {
                        // If KBD NUMERIC is selected:
                        if (KBD.currentState == KBD.State.NUMERIC) {
                            // Evaluates last key pressed:
                            if (lastKeyPressed == EMPTY_CHAR) {
                                // Initializes a variable to place in a string the
given key
                                val value = key.toString()
                                // Updates currentStationID with an actual value
                                currentStationID = value.toInt()
                                // Updates lastKeyPressed
                                lastKeyPressed = key
                            } else {
                                // Initializes a variable to store both keys as a
string
                                val num: String = lastKeyPressed.toString() + key
                                // Updates currentStationID with an actual value
                                currentStationID = num.toInt()
                                // Updates lastKeyPressed
                                lastKeyPressed = EMPTY_CHAR
                            }
                        }
                    }
                    else -> {
                        // If KBD NUMERIC is selected
                        if (KBD.currentState == KBD.State.NUMERIC) {
                            // Evaluates last key pressed:
                            if (lastKeyPressed == EMPTY_CHAR) {
                                // Initializes a variable to place in a string the
given key
                                val value = key.toString()
                                // Updates currentStationID with an actual value
                                currentStationID = value.toInt()
                                // Updates lastKeyPressed
```

```
                                lastKeyPressed = EMPTY_CHAR
                            } else if (key.code in ZERO_CODE..FIVE_CODE) {
                                // Initializes a variable to store both keys as a
string
                                val num: String = lastKeyPressed.toString() + key
                                // Updates currentStationID with an actual value
                                currentStationID = num.toInt()
                                // Updates lastKeyPressed
                                lastKeyPressed = EMPTY_CHAR
                            }
                        }
                    }
                }
            }
            State.SHOW -> {
                // Updates currentState
                currentState = when (key) {
                    '1' -> State.TICKET_TEST
                    '2' -> State.TICKET_CNT
                    '3' -> State.COINS_CNT
                    '4' -> State.RESET
                    '5' -> State.SHUTDOWN
                    else -> State.SHOW
                }
            }
            State.COINS_CNT -> {
                when (key) {
                    '2' -> {
                        // Evaluates current KBD state
                        when (KBD.currentState) {
                            KBD.State.NUMERIC -> {
                                // Updates Current Coin ID if given key actual value
is a valid CID
                                updateCurrentCoinID(key = key)
                            }
                            KBD.State.SELECTION -> {
                                if (currentCoinID == CoinDeposit.lastCoinID) {
                                    // If the currentCoinID counter reaches the last
coin correspondent
                                    // ID it resets to the first coin ID
                                    currentCoinID = CoinDeposit.firstCoinID
                                } else {
                                    // Increases currentCoinID counter
                                    currentCoinID++
                                }
                            }
                        }
                    }
                    '8' -> {
                        when (KBD.currentState) {
                            KBD.State.NUMERIC -> {
                                // Updates Current Coin ID if given key actual value
is a valid CID
                                updateCurrentCoinID(key = key)
                            }
                            KBD.State.SELECTION -> {
```

```
                                if (currentCoinID == CoinDeposit.firstCoinID) {
                                    // If the currentCoinID counter reaches the
first coin correspondent
                                    // ID it resets to the last coin ID
                                    currentCoinID = CoinDeposit.lastCoinID
                                } else {
                                    // Decreases currentCoinID counter
                                    currentCoinID--
                                }
                            }
                        }
                    }
                    else -> {
                        // Updates Current Coin ID if given key actual value is a
valid CID
                        updateCurrentCoinID(key = key)
                    }
                }
            }
            State.RESET -> {
                when (key) {
                    '5' -> {
                        // Displays a message letting the user know the reset is yet
to be completed
                        TUI.displayResetScreen(reset = false)
                        // Resets every station ticket counter
                        Stations.resetTicketCounters()
                        // Resets every coin type counter
                        CoinDeposit.resetCoinCounters()
                        // Displays a message letting the user know the reset has
been completed
                        TUI.displayResetScreen(reset = true)
                        // Updates currentState
                        currentState = State.SHOW
                    }
                    else -> currentState = State.SHOW // Updates currentState
                }
            }
            State.SHUTDOWN -> {
                when (key) {
                    '5' -> {
                        // Displays a message letting the user know the shutdown
process is yet to be completed
                        TUI.displayShutdownScreen(dataSent = false)
                        // Writes run-time modifications to Stations file
                        Stations.writeFile()
                        // Writes run-time modifications to CoinDeposit file
                        CoinDeposit.writeFile()
                        // Displays a message letting the user know the shutdown
process has been completed
                        TUI.displayShutdownScreen(dataSent = true)
                        // Closes Application
                        exitProcess(0)
                    }
                    else -> currentState = State.SHOW
                }
```

```
                }
                // If other state calls this function, no changes are made
                else -> {}
            }
        }
    }

// Main function
fun main() {
    // Initializes Ticket Machine Application Software
    App.init()
    // Initialize a mutable variable to store user input key
    var key: Char
    // Initialize three mutable variables to store current values, in order to only
update LCD
    // if these values are different from the actual correspondent ones, meaning a
change was
    // made by the user
    var lastStationID: Int = 0
    var lastCoinAmount: Int = 0
    var lastCoinID: Int = 0
    // Initialize a boolean variable to indicate if price should or not appear on
the LCD
    var showPrice: Boolean = false
    // Initialize a mutable variable to function has an index counter
    var cycleIndex: Int = 0
    // List of all avalaible Modes to show on LCD
    val listM_Modes: List<String> = listOf(
        "1-Ticket Test",
        "2-Ticket Cnt.",
        "3-Coins Cnt.",
        "4-Reset",
        "5-Shutdown"
    )
    // App state machine loop
    while (true) {
        // Evaluates current App Mode
        when (App.currentMode) {
            App.Mode.VENDING -> {
                // Evaluates current App State
                when (App.currentState) {
                    App.State.HOMESCREEN -> {
                        // Sets a new CGRAM custom character set
                        LCD.displayNewCGRAMset(App.VENDING_SET)
                        TUI.displayHomeScreen()
                        while (true) {
                            // Evaluates M state
                            if (!M.status()) {
                                // Wait for '#' key to be pressed
                                key = KBD.getKey()
                                // Start ticket purchase process if key '#' was
pressed
                                if (key == '#') {
                                    // Reset variables
                                    App.resetVariables()
                                    // Reset lastStationID
```

```
                                lastStationID = 0
                                // Update current App state
                                App.currentState = App.State.PURCHASE
                                App.displayCurrentStationInfo()
                                break
                            }
                        } else {
                            // Maintenance mode was activated:
                            // Sets a new CGRAM custom character set
                            LCD.displayNewCGRAMset(App.M_SET)
                            // Update current App mode
                            App.currentMode = App.Mode.M
                            break
                        }
                    }
                }
                App.State.PURCHASE -> {
                    // This check ensures the LCD is only written when the user
changes to a different
                    // station ID than the one being currently displayed
                    if (lastStationID != App.currentStationID) {
                        App.displayCurrentStationInfo()
                    }
                    // Waits 5 seconds for a key to be pressed
                    key = KBD.waitKey(DELAY_5S)
                    when (key) {
                        '#' -> {
                            // A station was selected for a ticket purchase
                            App.displayPaymentScreen(mode = App.currentMode)
                            // Update current App state
                            App.currentState = App.State.PAYMENT
                        }
                        '*' -> {
                            // Switch KBD state if key '*' was pressed
                            when (KBD.currentState) {
                                KBD.State.NUMERIC -> KBD.currentState =
KBD.State.SELECTION
                                KBD.State.SELECTION -> KBD.currentState =
KBD.State.NUMERIC
                            }
                            TUI.drawKBDMode()
                        }
                        KBD.NONE.toChar() -> {
                            // Since no key was pressed, current App state is
updated
                            App.currentState = App.State.HOMESCREEN
                        }
                        else -> {
                            // Updates lastStationID with the currentStationID
value
                            lastStationID = App.currentStationID
                            // Evaluates a pressed key or the absence of one
                            App.evaluateKey(key)
                        }
                    }
                }
```

```
                    App.State.PAYMENT -> {
                        // This check ensures the LCD is only written when the user
inserts a valid coin
                        if (lastCoinAmount != App.coinAmount) {
                            App.displayPaymentScreen(mode = App.currentMode)
                        }
                        // Evaluates if ticket price was paid by the user
                        if (App.ticketleftToPayPrice <= 0) {
                            // Ticket price was paid in full
                            TUI.displayPrintingTicketScreen()
                            // Collects inserted coins and stores them in the Coin
Deposit vault
                            CoinAcceptor.collectCoins()
                            // Sets inserted coins counters to zero
                            CoinDeposit.ejectInsertedCoins()
                            // Updates current App state
                            App.currentState = App.State.PRINT
                            continue
                        }
                        // Evaluates if a key was pressed
                        key = KBD.getKey()
                        if (key == '0') {
                            // Updates roundTrip value
                            App.roundTrip = !App.roundTrip
                            App.displayPaymentScreen(mode = App.currentMode)
                        } else if (key == '#') {
                            App.displayAbortedPurchaseScreen()
                            // Updates state since ticket purchase was aborted by
the user
                            App.currentState = App.State.HOMESCREEN
                        }
                        // Updates lastCoinAmount
                        lastCoinAmount = App.coinAmount
                        // Initializes a variable to represent the actual value of
the
                        // user inserted coin
                        val coin = CoinAcceptor.getCoinValue()
                        if (coin != 0) {
                            // Accepts coin if coin switch is still active
(indicating a coin was inserted)
                            CoinAcceptor.acceptCoin()
                            // Adds inserted coin to Coin Deposit, but not the vault
                            CoinDeposit.add(coin)
                            // Updates coinAmount with the received coin
                            App.coinAmount += coin
                        }
                    }
                    App.State.PRINT -> {
                        App.displayPrintScreen(mode = App.currentMode,
ticketCollected = false)
                        // Waits for the client to remove the printed ticket
                        TicketDispenser.print(destinyId = App.currentStationID,
originId = 0, App.roundTrip)
                        App.displayPrintScreen(mode = App.currentMode,
ticketCollected = true)
                        Stations.addTicket(stationID = App.currentStationID)
```

```
                    }
                    else -> {
                        // Default VENDING Mode state
                        App.currentState = App.State.HOMESCREEN
                    }
                }
            }
            App.Mode.M -> {
                when (App.currentState) {
                    App.State.SHOW -> {
                        // Sets a new CGRAM custom character set
                        LCD.displayNewCGRAMset(App.M_SET)
                        // Clears LCD previous content
                        LCD.clear()
                        App.resetVariables()
                        // Resets lastStationID value
                        lastStationID = 0
                        TUI.displayShowStateScreen()
                        while (true) {
                            // Evaluates M state
                            if (M.status()) {
                                // Retrieves a pressed key
                                key = KBD.getKey()
                                if (key == KBD.NONE.toChar()) {
                                    TUI.alignStringPos(TUI.Position.LEFT,
listM_Modes[cycleIndex++], LCD.Line.LOWER)
                                    if (cycleIndex == listM_Modes.lastIndex + 1) {
                                        // Resets cycleIndex value
                                        cycleIndex = 0
                                    }
                                    LCD.simulateLowerLineShift()
                                } else {
                                    // Evaluates a pressed key or the absence of one
                                    App.evaluateKey(key)
                                }
                            } else {
                                TUI.displayMCompleteScreen()
                                // Updates current App state, since M mode was
deactivated
                                App.currentMode = App.Mode.VENDING
                                break
                            }
                            if (App.currentState != App.State.SHOW) {
                                if (App.currentState == App.State.COINS_CNT) {
                                    // Sets a new CGRAM custom character set
                                    LCD.displayNewCGRAMset(App.VENDING_SET)
                                    App.displayCurrentCoinInfo()
                                }
                                break
                            }
                        }
                    }
                    App.State.TICKET_TEST -> {
                        // Sets a new CGRAM custom character set
                        LCD.displayNewCGRAMset(App.VENDING_SET)
```

```
                        // Enables showPrice flag since in this state ticket price
is displayed
                        showPrice = true
                        App.displayCurrentStationInfo(mode = App.currentMode, price
= showPrice)
                        // Updates current App state
                        App.currentState = App.State.PURCHASE
                    }
                    App.State.TICKET_CNT -> {
                        // Sets a new CGRAM custom character set
                        LCD.displayNewCGRAMset(App.VENDING_SET)
                        // Disables showPrice flag since in this state ticket price
isn't displayed
                        showPrice = false
                        App.displayCurrentStationInfo(mode = App.currentMode, price
= showPrice)
                        // Updates current App state
                        App.currentState = App.State.PURCHASE
                    }
                    App.State.PURCHASE -> {
                        // This check ensures the LCD is only written when the user
changes to a different
                        // station ID than the one being currently displayed
                        if (lastStationID != App.currentStationID) {
                            App.displayCurrentStationInfo(mode = App.currentMode,
price = showPrice)
                        }
                        // Waits 5 seconds for a key to be pressed
                        key = KBD.waitKey(DELAY_5S)
                        when (key) {
                            '#' -> {
                                // Evaluates showPrice flag
                                if (showPrice) {
                                    App.displayPaymentScreen(App.currentMode)
                                    // Updates current App state
                                    App.currentState = App.State.PAYMENT
                                } else {
                                    // Updates current App state
                                    App.currentState = App.State.SHOW
                                }
                            }
                            '*' -> {
                                // Switch KBD state if key '*' was pressed
                                when (KBD.currentState) {
                                    KBD.State.NUMERIC -> KBD.currentState =
KBD.State.SELECTION
                                    KBD.State.SELECTION -> KBD.currentState =
KBD.State.NUMERIC
                                }
                                TUI.drawKBDMode()
                            }
                            KBD.NONE.toChar() -> {
                                // Since no key was pressed, current App state is
updated
                                App.currentState = App.State.SHOW
                            }
```

```
                        else -> {
                            // Updates lastStationID with the currentStationID
value
                            lastStationID = App.currentStationID
                            // Evaluates a pressed key or the absence of one
                            App.evaluateKey(key)
                        }
                    }
                }
                App.State.PAYMENT -> {
                    // Retrieves a pressed key
                    key = KBD.getKey()
                    when (key) {
                        '0' -> {
                            // Updates roundTrip
                            App.roundTrip = !App.roundTrip
                            App.displayPaymentScreen(App.currentMode)
                        }
                        '#' -> {
                            App.displayAbortedPurchaseScreen()
                            // Updates state since ticket purchase was aborted
by the user
                            App.currentState = App.State.SHOW
                        }
                        '*' -> {
                            // Updates state since ticket purchase was completed
by the user
                            App.currentState = App.State.PRINT
                        }
                    }
                }
                App.State.PRINT -> {
                    App.displayPrintScreen(mode = App.currentMode,
ticketCollected = false)
                    // Wait for ticket removal
                    TicketDispenser.print(destinyId = App.currentStationID,
originId = 0, App.roundTrip)
                    App.displayPrintScreen(mode = App.currentMode,
ticketCollected = true)
                }
                App.State.COINS_CNT -> {
                    // This check ensures the LCD is only written when the user
changes to a different
                    // coin ID than the one being currently displayed
                    if (lastCoinID != App.currentCoinID) {
                        App.displayCurrentCoinInfo()
                    }
                    // Waits 5 seconds for a key to be pressed
                    key = KBD.waitKey(DELAY_5S)
                    when (key) {
                        '#' -> {
                            // Updates current App state
                            App.currentState = App.State.SHOW
                        }
                        '*' -> {
                            // Switch KBD state if key '*' was pressed
```

```
                                when (KBD.currentState) {
                                    KBD.State.NUMERIC -> KBD.currentState =
KBD.State.SELECTION
                                    KBD.State.SELECTION -> KBD.currentState =
KBD.State.NUMERIC
                                }
                                TUI.drawKBDMode()
                            }
                            KBD.NONE.toChar() -> {
                                // Since no key was pressed, current App state is
updated
                                App.currentState = App.State.SHOW
                            }
                            else -> {
                                // Updates lastCoinID with the currentCoinID value
                                lastCoinID = App.currentCoinID
                                // Evaluates a pressed key or the absence of one
                                App.evaluateKey(key)
                            }
                        }
                    }
                    App.State.RESET -> {
                        App.displayQueryRequestScreen(state = App.currentState)
                        // Waits 5 seconds for a key to be pressed
                        key = KBD.waitKey(DELAY_5S)
                        // Evaluates a pressed key or the absence of one
                        App.evaluateKey(key)
                    }
                    App.State.SHUTDOWN -> {
                        // Sets a new CGRAM custom character set
                        LCD.displayNewCGRAMset(App.M_SET)
                        App.displayQueryRequestScreen(state = App.currentState)
                        // Waits 5 seconds for a key to be pressed
                        key = KBD.waitKey(DELAY_5S)
                        // Evaluates a pressed key or the absence of one
                        App.evaluateKey(key)
                    }
                    else -> {
                        TUI.displayMStartingScreen()
                        // Default M Mode state
                        App.currentState = App.State.SHOW
                    }
                }
            }
        }
    }
}
```