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

Licenciatura em Engenharia Informática e de Computadores
Ano letivo 2021/2022

Máquina de venda de bilhetes

DOCUMENTAÇÃO MÓDULO KEYBOARD READER

Docentes responsáveis:
Pedro Miguens — Manuel Carvalho
Diego Passos — Nuno Sebastião
Trabalho realizado por:
Francisco Engenheiro — Nº49428
João Perleques — Nº49498
Mariana Muñoz — Nº47076
Turmas 22D/23D

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

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

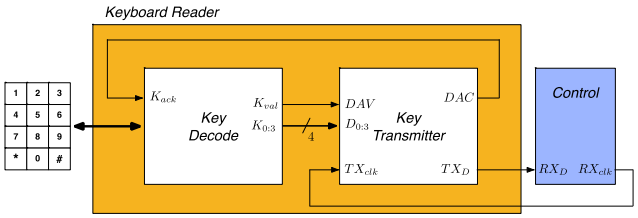
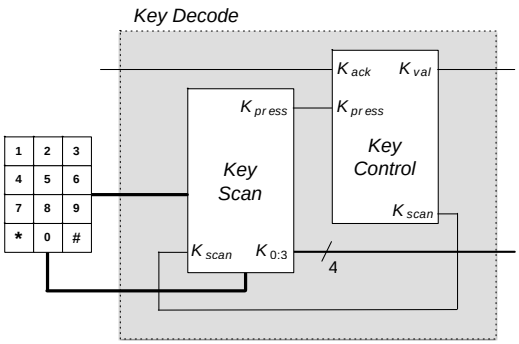


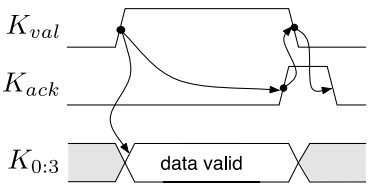
Figura 1 – Diagrama de blocos do módulo Keyboard Reader

1 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x3 por *hardware*, sendo constituído por três sub-blocos: *i)* um teclado matricial de 4x3; *ii)* o bloco *Key Scan*, responsável pelo varrimento do teclado; e *iii)* o bloco *Key Control*, que realiza o controlo do varrimento e o controlo de fluxo, conforme o diagrama de blocos representado na *Figura 2*a. O controlo de fluxo de saída do bloco *Key Decode* (para o módulo *Key Buffer*), define que o sinal K_{val} é ativado quando é detetada a pressão de uma tecla, sendo também disponibilizado o código dessa tecla no barramento $K_{0:3}$. Apenas é iniciado um novo ciclo de varrimento ao teclado quando o sinal K_{ack} for ativado e a tecla premida for libertada. O diagrama temporal do controlo de fluxo está representado na *Figura 2*b.



a) Diagrama de blocos



b) Diagrama temporal

Figura 2 – Bloco Key Decode

O bloco Key Scan foi implementado de acordo com o diagrama de blocos representado na Figura 3.

Foi escolhida a versão III para implementar a arquitetura do bloco *Key Scan*. Comparativamente às outras duas versões esta versão necessita de menos ciclos de relógio para efetuar o varrimento do teclado, sendo, portanto, mais eficiente. Como é utilizado um *priority encoder*, não existe necessidade de percorrer as suas quatro entradas como teríamos de fazer nas outras versões em que é utilizado um multiplexer, reduzindo assim o processo de varrimento às saídas do *decoder*, visto que é obtido logo o valor correto da parte baixa do barramento de dados *K*. No pior caso esta versão precisará de 3 ciclos de relógio (tendo em conta que, para esta versão, foi implementado um contador *mod* 3 de forma a não percorrer a última coluna de teclas do *key pad*, que corresponde a um conjunto de teclas não utilizadas no âmbito deste projeto) para encontrar uma tecla pressionada, já no caso da versão II (com este contador modificado) seriam precisos 7 (4 linhas + 3 colunas) ciclos de relógio e na versão I seriam necessários 12 (4 linhas x 3 colunas) ciclos de relógio.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4.

A descrição hardware do bloco *Key Decode* em VHDL encontra-se no Anexo A.

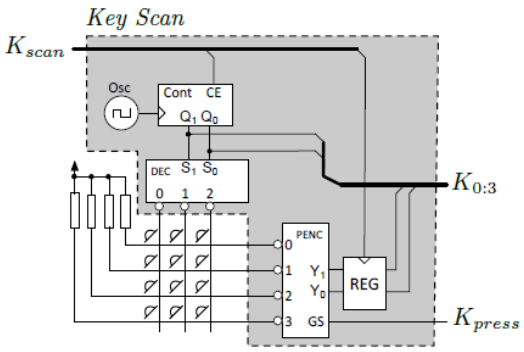


Figura 3 - Diagrama de blocos do bloco Key Scan

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

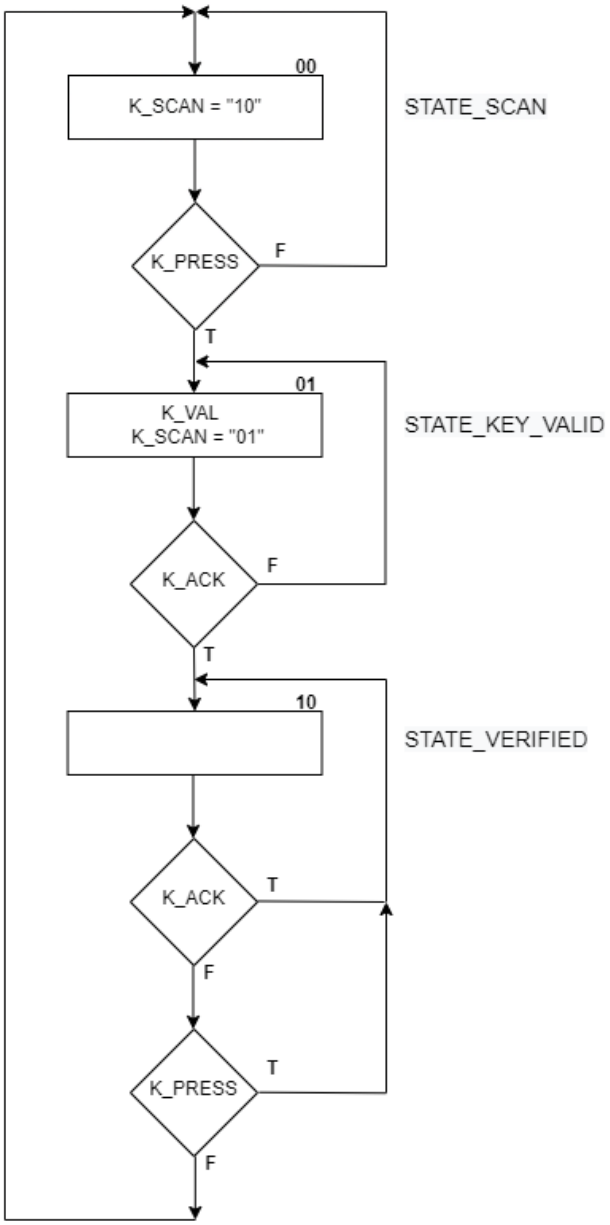


Figura 4 – Máquina de estados do bloco Key Control

Como descrito anteriormente, foi escolhida a versão III para a arquitetura do bloco *Key Scan*, o que implicou tornar o *Key Scan* num vetor de 2 bits, sendo que o *MSB* controla o *enable* do contador e o *LSB* o *clock* do registo que coloca 2 bits na parte baixa do barramento de dados *K*, correspondentes à codificação de uma tecla.

Este bloco foi idealizado para representar 3 estados: um estado zero (***STATE_SCAN***) que ativa o enable do contador de forma a poder ser feito um novo ciclo de varrimento do teclado; um primeiro estado (***STATE_KEY_VALID)*** que ativa a saída *K_VAL* consoante a deteção de uma tecla premida (*K_PRESS*), sendo esta responsável por indicar a existência de uma tecla válida ao bloco *Key Transmitter* e também ativa o *clock* do registo que coloca 2 bits na parte baixa do barramento de dados *K*, correspondentes à codificação dessa tecla; um segundo estado (***STATE_VERIFIED)*** que fica à espera que a tecla deixe de ser premida pela desativação do sinal *K_PRESS* e que a mesma tenha sido aceite pelo bloco *Key Transmitter* pela desativação do sinal *K_ACK*, só desta forma é possível que ocorra um novo ciclo de varrimento do teclado.

2 Key Transmitter

O módulo *Key Transmitter* implementa uma estrutura de armazenamento de dados, com capacidade de uma palavra de quatro bits. A escrita de dados no *Key Transmitter* inicia-se com a ativação do sinal *DAV* (*Data Available*) pelo sistema produtor, neste caso pelo *Key Decode*, indicando que tem dados para serem armazenados. Logo que tenha disponibilidade para armazenar informação, o *Key Transmitter* escreve os dados $D_{0:3}$ em memória interna. Concluída a escrita em memória, ativa o sinal *DAC* (*Data Accepted*) para informar o sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal *DAV* ativo até que *DAC* seja ativado. O *Key Transmitter* só desativa *DAC* depois de *DAV* ter sido desativado.

A implementação do *key Transmitter* é baseada numa máquina de controlo (*Key Transmitter Control*), um registo de 4 bits, um contador de 4 bits e um multiplexer 8x3, conforme o diagrama de blocos apresentado na *Figura 5*.

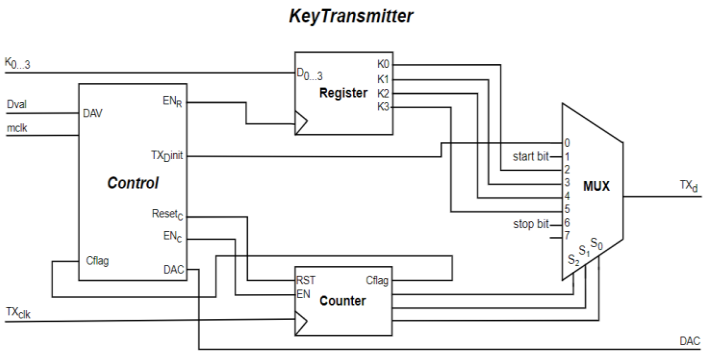


Figura 5 – Diagrama de blocos do Key Transmitter

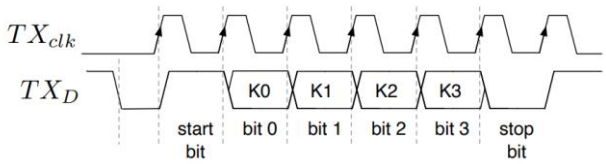


Figura 6 – Protocolo de comunicação com o módulo Key Transmitter

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

O bloco *Key Transmitter Control* do *Key Transmitter* é também responsável pela interação com o sistema consumidor, neste caso o módulo *Control*.

O bloco Key Transmitter Control foi implementado de acordo com a máquina de estados representada na Figura .

*O bloco Key Transmitter Control foi idealizado para representar 3 estados: um estado zero (**STATE_IDLE**) que indica que o Key Transmitter está à espera da sinalização de uma tecla válida por parte do Key Decode, neste estado o TXD_INIT é mantido com o valor lógico '1' e é feito reset ao contador (RESET_C); um primeiro estado (**STATE_ACK**) que indica que a tecla foi aceite ativando a saída DAC, libertando assim o Key Decoder para fazer um novo varrimento do teclado (ficando só dependente que a tecla deixe de ser premida), além disso, é dado enable ao registo (ENR) para poder guardar a tecla recebida e o TXD_INIT mantêm-se novamente com o valor lógico '1'; um segundo estado (**STATE_SEND**) que sinaliza o início do protocolo de comunicação com a entidade Control em software, neste estado é feito enable ao contador (ENC) e é colocado TXD a 0, pela alteração do valor lógico para '0' da saída TXD_INIT.*

Portanto, quando o bloco Key Transmitter Control se encontra no STATE_SEND, é iniciado o envio da codificação da tecla e por isso é colocado na saída do Multiplexer 8x3, que corresponde ao valor atual do TXD, as entradas que os seletores do Multiplexer vão selecionando. Como estes seletores estão ligados ao output do contador que por sua vez é está dependente do sinal RXCLK proveniente do Control, é enviado desta forma os bits individualmente, onde a codificação da tecla está inserida, para o Control, sendo que o ritmo desse envio é controlado por este.

Quando o contador chega a 7, o TXD terminou de enviar o último bit (STOP BIT) e por isso é feito reset ao contador usando o 7 com uma flag (C_FLAG) para indicar esse efeito. Consequentemente o Key Transmitter Control muda de estado, voltando novamente ao estado inicial (STATE_IDLE).

A descrição hardware do bloco *Key Transmitter Control* em VHDL encontra-se no Anexo B.

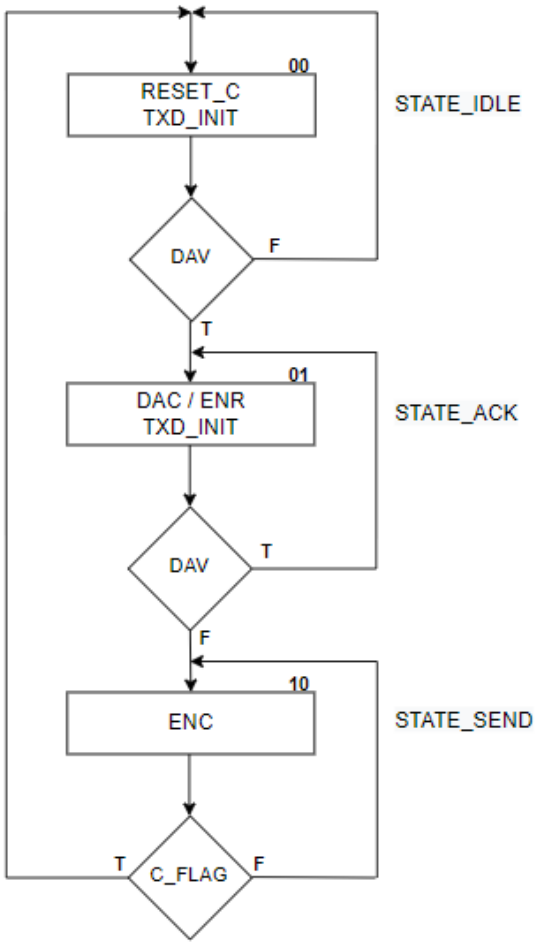


Figura 7 - Máquina de estados do bloco Key Transmitter Control

Com base nas descrições do bloco *Key Decode* e do bloco *Key Transmitter* implementou-se o módulo *KeyBoard Reader*.

De notar que, para o *Key Decode*, foi utilizado o módulo *CLK DIV* para baixar a frequência a que o varrimento do teclado estava a ser feito, de *50Mhz* (20ns) para *200Hz*(0.01s), visto que uma frequência tão alta não era necessária, pois 0,01 segundos de varrimento era mais que suficiente para o pretendido, para além de que, por vezes, uma única tecla era contabilizada diversas vezes (fenómeno *key bounce*), e não era o que se pretendia.

De notar que o módulo *CLK DIV*, além do propósito referido, foi colocado nos módulos Key Scan e Key Control em fases diferentes de forma a evitar que ocorra uma deteção errada de uma tecla, visto que ao estarem na mesma fase, o contador contava mais um ciclo quando a máquina

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

de estados mudava de estado, contribuindo para a deteção de uma tecla incorreta.

Interface com o *Control*

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

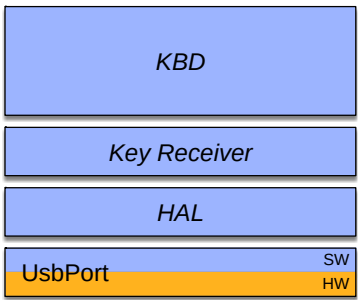


Figura 8 – Diagrama lógico do módulo Control de interface com o módulo Keyboard Reader

Os módulos de software *HAL*, *KeyReceiver* e *KBD* desenvolvidos são descritos nas secções 2.1., 2.3 e 2.4, e o código fonte desenvolvido nos Anexos D, E e F, respetivamente.

2.1 HAL

Comparativamente à versão apresentada na documentação do *Módulo Integrated Output System*, não foram feitas alterações.

2.2 LCD

Esta classe estabelece a ligação com o LCD, são enviadas tramas de dados vindas do software para o hardware. Numa primeira fase são enviadas 10 tramas de dados que fazem parte do protocolo de comunicação e têm o papel de inicializar o LCD. Essas mesmas tramas para além de ligar o LCD ainda definem por exemplo o número de linhas (uma ou duas) e o número de quadriculas por caracteres. Além das funções mencionadas no enunciado foram criadas funções auxiliares, como por exemplo, para mover o cursor para a primeira posição de uma linha, para eliminar texto ou até para carregar e escrever caracteres customizados, entre outras. Foi também criada uma *enum class* para distinguir as linhas do *LCD*.

2.3 KeyReceiver

Esta classe foi construída como objeto, para que não existam várias instâncias do *KeyReceiver* a receber tramas do *Key Transmitter*, mas apena uma.

Este objeto é responsável por receber os bits que *TXD* representa e, de acordo com o protocolo comunicação com o *Key Transmitter*, construir uma trama que corresponderá ao index que essa tecla representa num array, sendo que esse array contem todas as teclas mapeadas. De notar que tal array pode ser facilmente trocado por outro, mudando assim o mapeamento do teclado, permitindo ter, por exemplo, várias linguagens para teclado carregadas em memória e alternar entre as mesmas, tal como nos foi explicado em aula prática.

Durante a realização dos testes para esta objeto, foi sempre assumido que a interação entre o *KeyReceiver* e o *Key Transmitter* ocorre de forma sincronizada, no entanto, tal pode não acontecer e por isso foi criado um contador interno de forma a poder contabilizar cada ciclo dado ao *RXCLK*, garantindo assim que é dado o número de clocks suficientes a este, de forma a que o *Key Transmitter* volte ao estado inicial, permitindo que um novo ciclo de envio seja possível. Não existe, por isso, retransmissão, visto que o protocolo de comunicação é "*best effort*".

Se a garantia de uma sincronização, no final do protocolo de envio, entre o *Control* e o *Key Transmitter*, não fosse feito, este último ficaria preso no meio do protocolo de envio, corrompendo assim o sistema e o funcionamento normal do circuito.

2.4 *KBD*

Esta classe foi construída como objeto, para que não existam várias instâncias do *KBD* a receber informação sobre a codificação das teclas pelo *Key Receiver*, mas apena uma.

Foi criada uma *enum class* para representar quando o teclado está em modo de seleção, possibilitando o uso das

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

teclas, ou em modo numérico, ou seja, sem essa possibilidade.

Este objeto apresenta métodos para usar a informação obtida pelo *KeyReceiver* e, usando um array que foi referenciado anteriormente, é possível representar em software qual a tecla que foi premida no momento. Outro método, deste objeto, apresenta a possibilidade de esperar um tempo específico para que uma tecla seja premida.

3 Conclusões

O desenvolvimento do *Keyboard* Reader foi menos atribulado comparativamente ao desenvolvimento do módulo *IOS*, devido à experiência adquirida pela aprendizagem dos erros cometidos no desenvolvimento deste, tanto na programação em *VHDL* como em *Kotlin*.

De referir que, o bloco que verificamos mais dificuldade no desenvolvimento foi o *Key Scan* devido ao tempo dedicado à perceção de qual a melhor versão para implementar este bloco, e posteriormente, à sua implementação e validação, tanto no simulador *Altera* como na placa.

Mais uma vez, este relatório permitiu que ficasse claro a importância e a necessidade de descrever e documentar a informação pertinente, no momento de desenvolvimento de módulos em software e hardware, de forma a que, outros colegas possam perceber de forma sucinta e clara, e replicar com sucesso o que foi feito.

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

A. Descrição VHDL do bloco *Key Decode*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity KEY_DECODE is

    port( KACK, MCLK, RESET: in STD_LOGIC;
          LIN: in STD_LOGIC_VECTOR(3 downto 0);
          K, COL: out STD_LOGIC_VECTOR(3 downto 0);
          KVAL: out STD_LOGIC );

end KEY_DECODE;

architecture arq_KEY_DECODE of KEY_DECODE is

component KEY_SCAN

    port( MCLK, RESET: in STD_LOGIC;
          Kscan: in STD_LOGIC_VECTOR(1 downto 0);
          Lin: in STD_LOGIC_VECTOR(3 downto 0);
          K: out STD_LOGIC_VECTOR(3 downto 0);
          Kpress: out STD_LOGIC;
          Col: out STD_LOGIC_VECTOR(3 downto 0) );

end component;

component KEY_CONTROL

    port( MCLK, RESET, Kack, Kpress: in STD_LOGIC;
          Kval: out STD_LOGIC;
          Kscan: out STD_LOGIC_VECTOR(1 downto 0) );

end component;

component CLKDIV

    generic(div: natural := 250000);
    port( clk_in: in std_logic;
          clk_out: out std_logic );

end component;

signal KPRESS_LINK, M_LINK: STD_LOGIC;
signal KSCAN_LINK: STD_LOGIC_VECTOR (1 downto 0);

begin

    -- Using CLK_DIV to lower the avaliable frequency to 200Hz(0.01s)
    M0: CLKDIV port map (
        clk_in => MCLK,
        clk_out => M_LINK );

    U1: KEY_SCAN port map (
        Kscan => KSCAN_LINK,
        MCLK => M_LINK,
        RESET => RESET,
        K => K,
        Kpress => KPRESS_LINK,
        Col => COL,
```

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

```
        Lin => LIN );

    U2: KEY_CONTROL port map (
        MCLK => not M_LINK,
        RESET => RESET,
        Kack => KACK,
        Kpress => KPRESS_LINK,
        Kval => KVAL,
        Kscan => KSCAN_LINK );

end arq_KEY_DECODE;
```

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

B. Descrição VHDL do bloco *Key Transmitter*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity KEY_TRANSMITTER is

    port( MCLK, RESET, DAV, TXclk: in STD_LOGIC;
          D: in STD_LOGIC_VECTOR(3 downto 0);
          DAC, TXD: out STD_LOGIC );

end KEY_TRANSMITTER;

architecture arq_KEY_TRANSMITTER of KEY_TRANSMITTER is

component KEY_TRANSMITTER_CONTROL

    port( MCLK, RESET, DAV, Cflag: in STD_LOGIC;
          ENR, RESET_C, ENC, DAC, TXD_INIT: out STD_LOGIC );

end component;

component REGISTER4BITS

    port( D: in STD_LOGIC_VECTOR(3 downto 0);
          CLK, RESET, SET, EN: in STD_LOGIC;
          Q: out STD_LOGIC_VECTOR(3 downto 0) );

end component;

component COUNTER4BITS

    port( CLR, SCLK, EN: in STD_LOGIC;
          O: out STD_LOGIC_VECTOR(3 downto 0) );

end component;

component Mux8x3

    port( A: in STD_LOGIC_VECTOR(7 downto 0);
          S: in STD_LOGIC_VECTOR(2 downto 0);
          O: out STD_LOGIC );

end component;

signal C_FLAG, ENR_LINK, ENC_LINK, RESETC_LINK, TXD_INIT_LINK: STD_LOGIC;
signal S_LINK: STD_LOGIC_VECTOR(2 downto 0);
signal O_LINK, Q_LINK: STD_LOGIC_VECTOR(3 downto 0);
signal M_LINK: STD_LOGIC_VECTOR(7 downto 0);

begin

    -- Active flag when counter equals to 7 (111)
    C_FLAG <= (O_LINK(2) and O_LINK(1) and O_LINK(0)) and not TXclk;

    M0: KEY_TRANSMITTER_CONTROL port map (
        MCLK => MCLK,
        RESET => RESET,
        DAV => DAV,
        Cflag => C_FLAG,
```

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

```
        ENR => ENR_LINK,
        RESET_C => RESETC_LINK,
        ENC => ENC_LINK,
        DAC => DAC,
        TXD_INIT => TXD_INIT_LINK );

    M1: REGISTER4BITS port map (
        D => D,
        CLK => MCLK,
        RESET => RESET,
        SET => '0',
        EN => ENR_LINK,
        Q => Q_LINK);

    M2: COUNTER4BITS port map (
        CLR => RESETC_LINK,
        SCLK => TXclk,
        EN => ENC_LINK,
        O => O_LINK );

    -- Counter last 3 bits assignment:
    S_LINK(2 downto 0) <= O_LINK(2 downto 0);

    -- Multiplexer Assignments:
    M_LINK(0) <= TXD_INIT_LINK; -- Either '1' (Default TXD value) or '0' (INIT BIT)
    M_LINK(1) <= '1'; -- Start bit
    M_LINK(2) <= Q_LINK(0); -- K(0)
    M_LINK(3) <= Q_LINK(1); -- K(1)
    M_LINK(4) <= Q_LINK(2); -- K(2)
    M_LINK(5) <= Q_LINK(3); -- K(3)
    M_LINK(6) <= '0'; -- Stop bit
    M_LINK(7) <= '1'; -- Place holder value

    M3: MUX8x3 port map (
        A => M_LINK,
        S => S_LINK,
        O => TXD );



end arq_KEY_TRANSMITTER;
```

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

C. Descrição VHDL do módulo *KeyboardReader*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity KBD is

    port( MCLK, TXclk, RESET: in STD_LOGIC;
          LIN: in STD_LOGIC_VECTOR(3 downto 0);
          TXD: out STD_LOGIC;
          COL: out STD_LOGIC_VECTOR(3 downto 0) );

end KBD;

architecture arq_KBD of KBD is

component KEY_DECODE

    port( KACK, MCLK, RESET: in STD_LOGIC;
          LIN: in STD_LOGIC_VECTOR(3 downto 0);
          K, COL: out STD_LOGIC_VECTOR(3 downto 0);
          KVAL: out STD_LOGIC );

end component;

component KEY_TRANSMITTER

    port( MCLK, RESET, DAV, TXclk: in STD_LOGIC;
          D: in STD_LOGIC_VECTOR(3 downto 0);
          DAC, TXD: out STD_LOGIC );

end component;

signal DAC_LINK, KVAL_LINK: STD_LOGIC;
signal K_LINK: STD_LOGIC_VECTOR(3 downto 0);

begin

    U1: KEY_DECODE port map (
        KACK => DAC_LINK,
        MCLK => MCLK,
        RESET => RESET,
        LIN => LIN,
        K => K_LINK,
        COL => COL,
        KVAL => KVAL_LINK );

    U2: KEY_TRANSMITTER port map (
        MCLK => MCLK,
        RESET => RESET,
        DAV => KVAL_LINK,
        TXclk => TXclk,
        D => K_LINK,
        DAC => DAC_LINK,
        TXD => TXD );

end arq_KBD;
```

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

D. Código *Kotlin – HAL*

```
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)
    }
```

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

```
    // 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)
    }
}
```

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

E. Código Kotlin – KeyReceiver

```
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++
    }
}
```

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

F. Código *Kotlin - 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
```

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

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