O módulo *Keyboard Reader* é constituído por três blocos principais: *i*) o descodificador de teclado (*Key Decode*); *ii*) o bloco de armazenamento (designado por *Ring Buffer*); e *iii*) o bloco de entrega ao consumidor (designado por *Output Buffer*). Neste caso o módulo *Control*, implementado em *software*, é a entidade consumidora.

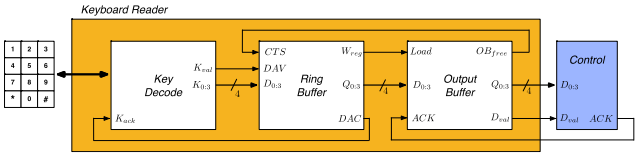
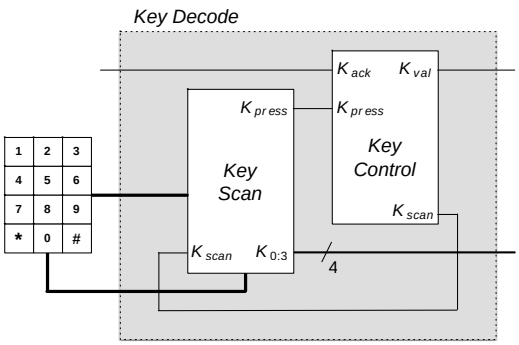


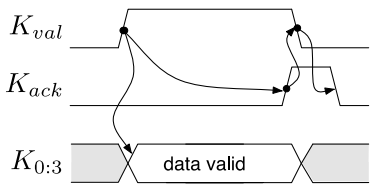
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

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 2a. 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 2b.



a) Diagrama de blocos



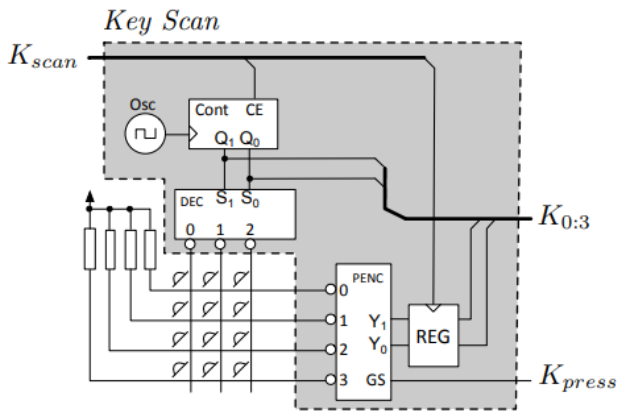
b) Diagrama temporal

Figura 2 – Bloco *Key Decode*

O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado na Figura 3. Optamos pela escolha da versão 3 do Kscan por se tratar da solução mais eficiente, mais eficiente entendemos como sendo a solução que mais rapido nos apresenta o resultado pelo que comparando com as versões anteriores este versão apenas necessita de 3 ciclos de clock para encontrar a tecla premida ao contrario da primeira versão que precisa de 12 ciclos e a segunda versão que precisa de 7 ciclos. Tambem vemos vantagens na flag de Kpress que é imediatamente ativa quando uma tecla é premida, ao contrario das outras soluções que apenas ativam esta flag quando encontram a tecla premida. Outra vantagem desta implementação é tambem o facto de que o resultado só ser produzido quando habilitamos o registo.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. No estado '00', o Key Control espera pelo sinal Kpress enviado pelo Key Scan, que indica quando uma tecla é premida. Assim que recebe o valor lógico '1', avança para o estado seguinte '01', onde efetua o varrimento do código da tecla premida e aguarda o sinal Kack, que deve estar em nível lógico '1', indicando que os bits foram recebidos. O estado '10' é utilizado para confirmar que tanto o sinal Kack como o sinal Kpress retornaram a nível lógico '0', evitando assim que, quando o sistema retornar ao estado '00', esses sinais estejam em nível lógico '1' incorretamente.

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



c) versão III

Figura 3 - Diagrama de blocos do bloco *Key Scan*

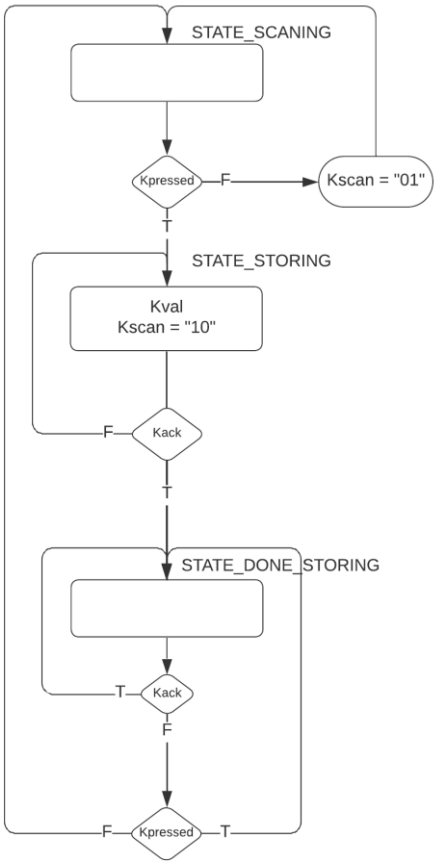


Figura 4 – Máquina de estados do bloco *Key Control*

2 Ring Buffer

O bloco Ring Buffer que foi desenvolvido é uma estrutura de dados usada para armazenar teclas em uma ordem específica, seguindo o princípio FIFO (First In First Out). Ele tem a capacidade de armazenar até oito palavras de quatro bits cada.

A escrita de dados no Ring Buffer começa quando o sistema produtor, o Key Decode neste caso, ativa o sinal DAV (Data Available), indicando que há dados a serem armazenados. Assim que houver espaço disponível para armazenar informações, o Ring Buffer escreve os dados D0:3 na memória. Após a conclusão da escrita na memória, ele ativa o sinal DAC (Data Accepted) para informar ao sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal DAV ativo até que o DAC seja ativado. O Ring Buffer só desativa o DAC depois que o DAV for desativado.

A implementação do Ring Buffer é baseada em uma memória RAM (Random Access Memory). O endereço de escrita/leitura, selecionado por meio de uma combinação de sinais, é determinado pelo bloco Memory Address Control (MAC), que é composto por dois registros chamados putIndex e getIndex. Esses registros armazenam os endereços de escrita e leitura, respectivamente. O MAC suporta operações de incremento (incPut e incGet) e gera informações indicando se a estrutura de dados está cheia (Full) ou vazia (Empty). O bloco Ring Buffer envia os dados para a entidade consumidora sempre que ela indicar que está pronta para recebê-los, por meio do sinal Clear To Send (CTS). O diagrama de blocos para uma estrutura do bloco Ring Buffer é mostrado na Figura 5.

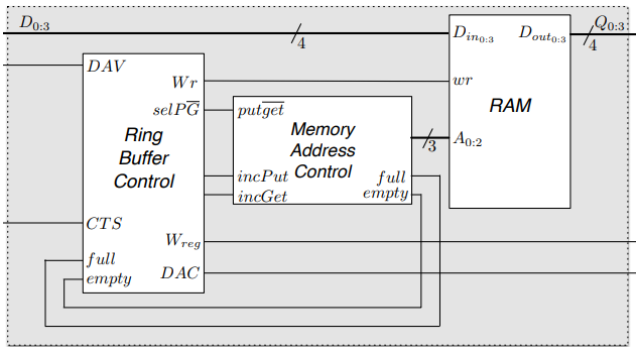


Figura 5 - diagrama de blocos para uma estrutura do bloco Ring Buffer

2.1 Ring Buffer Control

O bloco Ring Buffer Control é responsável pela comunicação entre a entidade produtora e consumidora (Key Decode e Output Buffer respetivamente). O mesmo recebe a indicação de que há dados para serem armazenados e indica à entidade consumidora que já pode ler os mesmo da memória. Este bloco foi implementado pela máquina de estados representada pela Figura 6.

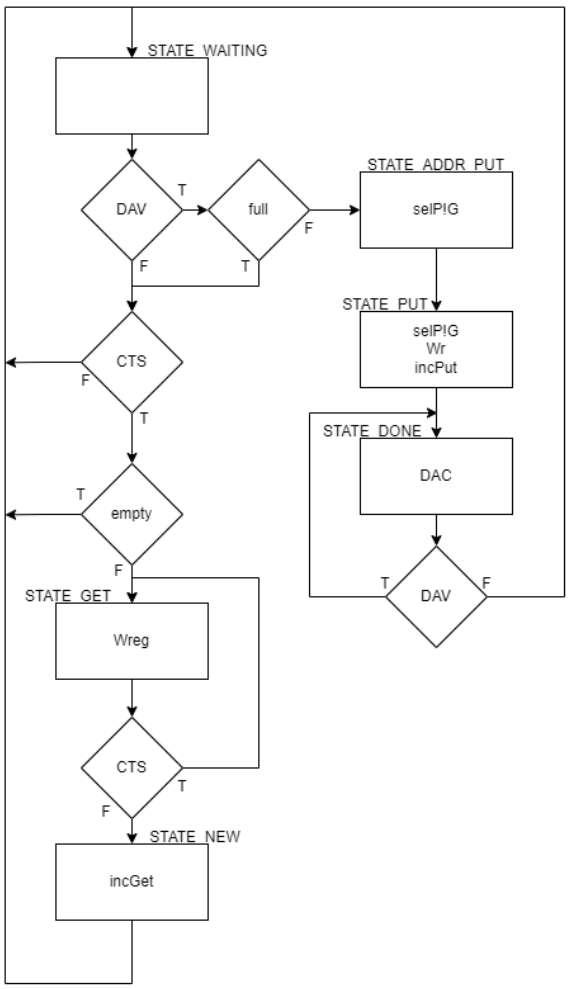


Figura 6- Máquina de estados do bloco Ring Buffer Control

2.2 Memory Address Control

O bloco Memory Address Control é responsável por selecionar os endereços da memória correspondente à escrita e à leitura, consoante o sinal put!get. O mesmo incrementa os registos incPut e incGet à medida que são escritos ou lidos dados, que servem de índices para a próxima posição de memória a ser escrita (incPut) e a próxima a ser lida (incGet) . A memória consegue armazenar até 8 tramas de dados, conseguindo indicar ao Ring Buffer Control quando a RAM está cheia ou vazia através de um registo a 4 bits, que conta o número de tramas na memória. Este bloco foi implementado pelo diagrama representado na Figura 7.

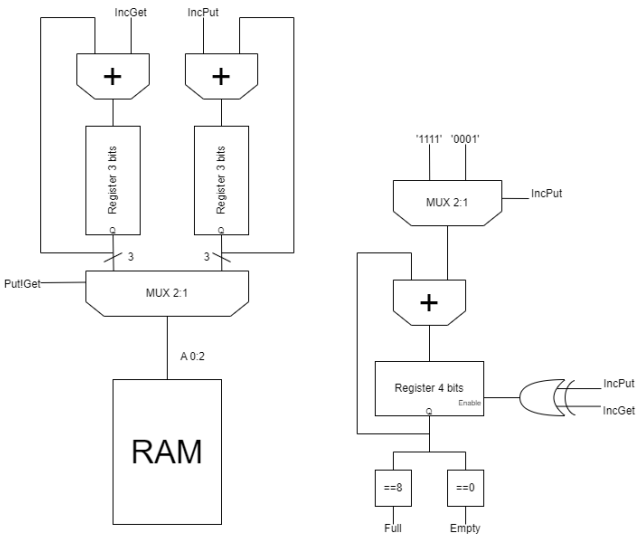


Figura 7- Diagrama do bloco Memory Address Control

3 Output Buffer

O bloco Output Buffer do Keyboard Reader é responsável pela interação com o sistema consumidor, neste caso o módulo Control. O Output Buffer indica que está disponível para armazenar dados através do sinal OBfree. Assim, nesta situação o sistema produtor pode ativar o sinal Load para registar os dados. O Control quando pretende ler dados do Output Buffer, aguarda que o sinal Dval fique ativo, recolhe os dados e pulsa o sinal ACK indicando que estes já foram consumidos. O Output Buffer, logo que o sinal ACK pulse, deve invalidar os dados baixando o sinal Dval e sinalizar que está novamente disponível para entregar dados ao sistema consumidor, ativando o sinal OBfree. Na Figura 8, é apresentado o diagrama de blocos do Output Buffer.

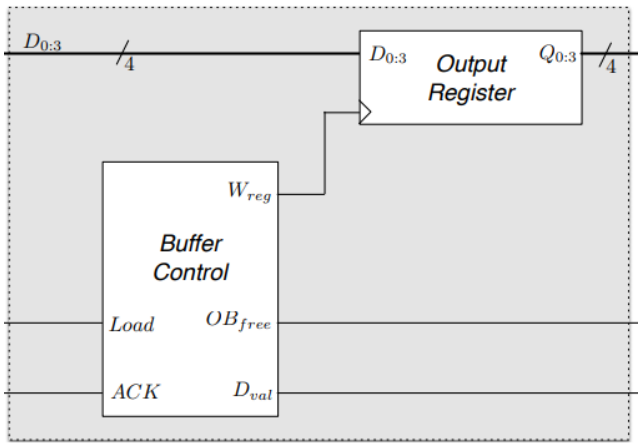


Figura 8-Diagrama de blocos do bloco Output Buffer

3.1 Buffer Control

O bloco Buffer Control é responsável por enviar a trama de dados para o software, recebe a trama da entidade produtora e despacha a mesma através do Output Register, e só volta a receber uma nova trama quando tiver a confirmação de que a mesma foi recebida na entidade consumidora. Este bloco foi implementado pela máquina de estados representada na Figura 9.

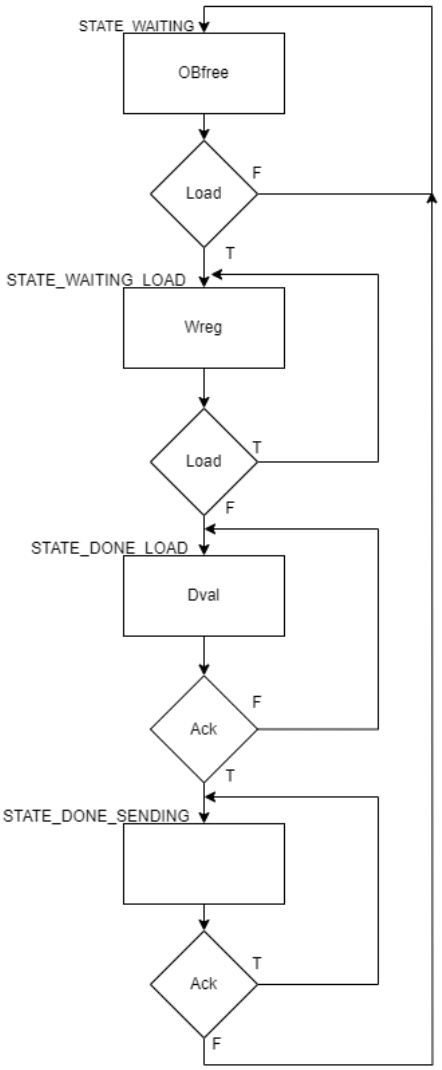


Figura 9- Máquina de estados do bloco Buffer Control

4 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 5.

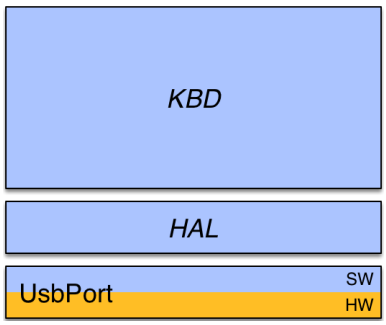


Figura 5 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

HAL e *KBD* desenvolvidos são descritos nas secções 4.1. e 4.2, e o código fonte desenvolvido nos Anexos E e F, respetivamente.

4.1 HAL

O bloco *HAL* virtualiza o acesso ao sistema UsbPort, que faz a ligação entre o *hardware* e o *software*. Utiliza máscaras para definir o significado da informação dos pinos do UsbPort. O módulo *Control* comunica com o hardware lendo os bits de input do UsbPort e escrevendo nos bits de output do mesmo.

4.2 KBD

O bloco *KBD* recebe o código da tecla premida através da função de leitura do *HAL*, e converte esse código no seu char. Também pode permitir a leitura de uma tecla apenas num tempo definido.

5 Conclusões

O módulo *Keyboard Reader* é essencial na implementação deste projeto, pois não só faz a ligação entre o teclado matricial e o *hardware*, como também a ligação entre o *hardware* e o *software*. No módulo *Control* podemos prever que tanto o bloco *HAL* como o bloco *KBD* vão ser uma base para os próximos blocos de *software*.

A. Descrição VHDL do bloco *Key Decode*

```
  LIBRARY IEEE;
use IEEE.std_logic_1164.all;

entity KeyDecoder_v3 is
        port(
        Kack : in std_logic;
        CLK : in std_logic;
        Rst : in std_logic;
        Ln1,Ln2,Ln3, Ln4: in std_logic;

        Col1,Col2,Col3: out std_logic;
        Kval : out std_logic;
        K : out std_logic_vector(3 downto 0)
        );
end KeyDecoder_v3;

architecture structural of KeyDecoder_v3 is

component KeyScan_v3 is
        port
        (
        Kscan : in std_logic_vector(1 downto 0);
        CLK : in std_logic;
        Rst : in std_logic;
        Ln1,Ln2,Ln3, Ln4: in std_logic;

        Col1,Col2,Col3: out std_logic;
        Kpress : out std_logic;
        K : out std_logic_vector(3 downto 0)
        );
end component KeyScan_v3;

component KeyControl_v3 is
        port(
        Kpressed : in std_logic;
        Kack : in std_logic;
        Kscan : out std_logic_vector(1 downto 0);
        Kval : out std_logic;
        clk : in std_logic;
        reset: in std_logic
        );
end component KeyControl_v3;

signal Kscan_s: std_logic_vector(1 downto 0);
signal Kpress_s: std_logic;

begin

ks01: KeyScan_v3 port map(Kscan => Kscan_s, CLK => CLK, Rst => Rst, Col1 => Col1,
                                                        Col2 => Col2, Col3 =>
Col3, Kpress => Kpress_s, K => K, Ln1 => Ln1, Ln2 => Ln2, Ln3 => Ln3, Ln4 => Ln4);

kc01: KeyControl_v3 port map(Kpressed => Kpress_s, Kack => Kack, Kscan => Kscan_s,
```

```
                                                    Kval => Kval,
clk => CLK, reset => Rst);

end structural;
```

B. Descrição VHDL do bloco Ring Buffer

```
 library ieee;
use ieee.std_logic_1164.all;

entity RingBuffer is
        port(
        CTS, DAV, rst, clk : in std_logic;
        D: in std_logic_vector(3 downto 0);

        Wreg, DAC: out std_logic;
        Q: out std_logic_vector(3 downto 0)
        );
end RingBuffer;

architecture structural of RingBuffer is
component MAC is
        port(
        putget: in std_logic;
        incPut: in std_logic;
        incGet: in std_logic;
        clk: in std_logic;
        reset: in std_logic;

        full: out std_logic;
        empty: out std_logic;
        Aout: out std_logic_vector(2 downto 0)
        );
end component MAC;

component RingB_Control is
        port(
        DAV, CTS, full, empty, rst, clk: in std_logic;

        Wr, selPG, Wreg, DAC, incGet, incPut: out std_logic
        );
end component RingB_Control;

component RAM is
        generic(
                ADDRESS_WIDTH : natural := 3;
                DATA_WIDTH : natural := 4
        );
        port(
                address : in std_logic_vector(ADDRESS_WIDTH - 1 downto 0);
                wr: in std_logic;
                din: in std_logic_vector(DATA_WIDTH - 1 downto 0);
                dout: out std_logic_vector(DATA_WIDTH - 1 downto 0)
        );
end component RAM;

signal selPG_S, incPut_S, incGet_S, Wr_S, full_S, empty_s: std_logic;
signal A_S : std_logic_vector(2 downto 0);

begin
```

```
RBC1: RingB_Control port map(DAV => DAV, CTS => CTS, full => full_S, empty =>
empty_S, Wr => Wr_S,
                                                                    selPG =>
selPG_S, incPut => incPut_S, incGet => incGet_S, Wreg => Wreg, DAC => DAC,
                                                                    clk => clk,
rst => rst
                                                                    );

MAC1 : MAC port map(putGet => selPG_S, incPut => incPut_S, incGet => incGet_S, full
=> full_S,
                                                 empty => empty_S, Aout => A_S, clk
=> clk, reset => rst
                                                 );

RAM1: RAM port map(address => A_S, wr => Wr_S, din => D, dout => Q);


end structural;
```

C. Descrição VHDL do bloco Output Buffer

```
library ieee;
use ieee.std_logic_1164.all;

entity OutputBuffer is
        port(
        D: in std_logic_vector(3 downto 0);
        Load, ACK, clk, rst: in std_logic;

        Q: out std_logic_vector(3 downto 0);
        OBfree, Dval: out std_logic
        );
end OutputBuffer;

architecture structural of OutputBuffer is
component BufferControl is
        port(
        Load, ACK, clk, rst: in std_logic;

        Wreg, OBfree, Dval: out std_logic
        );
end component BufferControl;

component Register4 is
        port
        (
        clk: in std_logic;
        reset: in std_logic;
        en: in std_logic;
        d : in std_logic_vector(3 downto 0);
        q : out std_logic_vector(3 downto 0)
        );
end component Register4;

signal Wreg_s:std_logic;
begin

BC1: BufferControl port map(Load => Load, ACK => ACK, Wreg => Wreg_s, OBfree =>
OBfree,
                                                                Dval => Dval,
clk => clk, rst => rst
                                                                );

Register1: Register4 port map(clk => Wreg_s, reset => rst, en => '1', d => D, Q=>
Q);

end structural;
```

D. Atribuição de pinos do módulo *Keyboard Reader*

```
set_location_assignment PIN_A8 -to K[0]
set_location_assignment PIN_A9 -to K[1]
set_location_assignment PIN_A10 -to K[2]
set_location_assignment PIN_B10 -to K[3]
set_location_assignment PIN_D13 -to Kpress
set_location_assignment PIN_W5 -to Ln1
set_location_assignment PIN_AA14 -to Ln2
set_location_assignment PIN_W12 -to Ln3
set_location_assignment PIN_AB12 -to Ln4

set_location_assignment PIN_C10 -to Rst
set_location_assignment PIN_C11 -to Kack
set_location_assignment PIN_P11 -to CLK
```

E. Código *Kotlin – HAL*

```
import isel.leic.UsbPort
import isel.leic.utils.*
object HAL { // Virtualiza o acesso ao sistema UsbPort
    // Inicia a classe
    var lastoutput=0x00
    fun init(){
        UsbPort.write(lastoutput)
    }
    // Retorna true se o bit tiver o valor lógico '1'
    fun isBit(mask: Int): Boolean {
        return (UsbPort.read() and mask) != 0
    }
    // Retorna os valores dos bits representados por mask presentes no UsbPort
    fun readBits(mask: Int): Int {
        return (UsbPort.read() and mask)
    }
    // Escreve nos bits representados por mask o valor de value
    fun writeBits(mask: Int, value: Int){
        val newvalue=(lastoutput and mask.inv())or(mask and value)
        UsbPort.write(newvalue)
        lastoutput=newvalue
    }
    // Coloca os bits representados por mask no valor lógico '1'
    fun setBits(mask: Int){
        writeBits(mask,mask)
    }
    // Coloca os bits representados por mask no valor lógico '0'
    fun clrBits(mask: Int){
        writeBits(mask, 0)
    }
}
```

F. Código *Kotlin - KBD*

```
import isel.leic.utils.Time

object KBD { // Ler teclas. Métodos retornam ‘0’..’9’,’#’,’*’ ou NONE.
    const val ACK=0x80
    const val DVAL=0x10
    const val CODE=0x0F
    const val NONE = 0.toChar()
    private val DIGITS = arrayOf('1','4','7','*','2','5','8','0','3','6','9','#')

    // Inicia a classe
    fun init() {
        HAL.clrBits(ACK)//Coloca o Kack a '0'

    }
    // Retorna de imediato a tecla premida ou NONE se não há tecla premida.
    fun getKey(): Char {
        var digit = NONE
        if(HAL.isBit(DVAL)){
            if(HAL.readBits(CODE)<12) {
                digit = DIGITS[HAL.readBits(CODE)]
            }
            HAL.setBits(ACK)
            while(HAL.isBit(DVAL));
            HAL.clrBits(ACK)
        }
        return digit
    }

// Retorna a tecla premida, caso ocorra antes do ‘timeout’ (representado em
milissegundos), ou NONE caso contrário.

    fun waitKey(timeout: Long): Char {
        val timelimit=Time.getTimeInMillis()+timeout
        var key=NONE
        while(Time.getTimeInMillis()<timelimit){
            if(HAL.isBit(DVAL)) {
                key = getKey()
            }
            if(key!=NONE){
                break
            }
        }
        return key
    }
}
```