Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

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 Buffer*), conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em *software*, é a entidade consumidora.

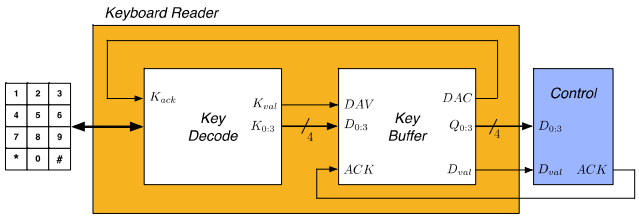
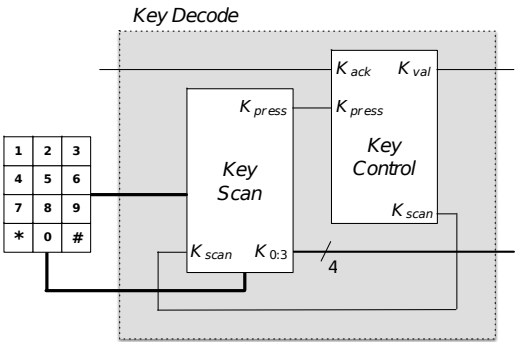


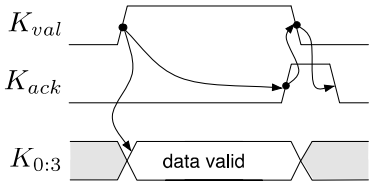
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

1 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x3 por *hardware*, sendo constituído por três sub-blocos: *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. O nosso grupo inicialmente escolheu a versão 1, mas foi-nos pedido para fazer a implementação 3 de modo a por à prova as nossas capacidades.

O bloco *Key Scan* faz o varrimento do teclado. Para este varrimento acontecer é necessário 5 elementos do diagrama de blocos sendo estes um contador, um *decoder 2x3*, um *priority encoder* de 4x2, um *register* e um teclado matricial de 4x3.

O contador é feito a partir de 4 *flip-flops*, no entanto este apresenta uma particularidade, este só conta até ao 11 pois só serão usadas 12 teclas (0-11), logo seria ineficiente fazer a contagem até ao último número disponível (16), sendo que cada flip-flop representa um bit. Para isso foi colocado um *priority encoder* e um registo em que o *priority encoder* irá verificar os bits que estão ativos e o registo guarda qual tecla foi detetada posteriormente e apresentá-la na saída

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

Começando pelo primeiro estado (000) este apresenta o *KscanCounter* pronto para receber uma tecla estando por isso ativo e permanece neste estado enquanto nenhuma tecla for ativa, quando uma tecla for pressionada este avança para o segundo estado (001).

No segundo estado (001). verificamos que o *Kscan Register* começa ativo pois no estado anterior foi detetado que uma tecla foi pressionada passando assim logo para o terceiro estado (010).

Chegando ao terceiro estado este inicia com o *Kval* ativo já que foi detetado a pressão de uma tecla nos estados anteriores e irá avançar para o estado seguinte quando o *Kack* for ativo, ou seja quando o Buffer já recebeu a tecla, se não irá voltar para o segundo estado e manterá este loop até que o *Kack* seja *true*.

Passando agora para o quarto(011) e último estado este tem a saída a 0, sem nenhum elemento ativo, pois é um estado no qual a tecla já foi aceite no *Key Buffer*, mas o *Key Scan* não pode ainda fazer o varrimento do teclado o que faz com que se

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

o *Kack* ou *Kpress* estiverem ativos o *Key Control* não consiga avançar para outro estado, sendo assim este só avança para outro estado quando estão os dois desativados, significando assim que tecla já não se encontra premida e que o seu código já foi aceite permitindo a volta ao estado inicial, recomeçando assim o processo.

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

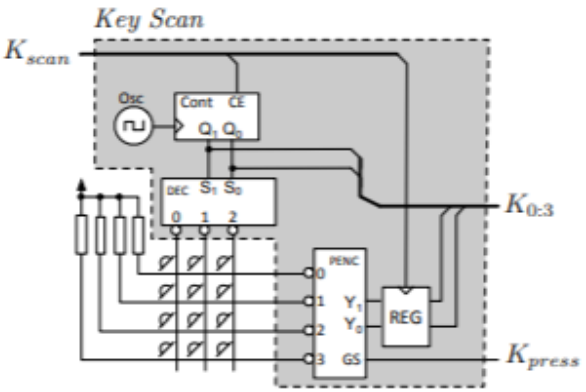


Figura 3 - Diagrama de blocos do bloco *Key Scan*

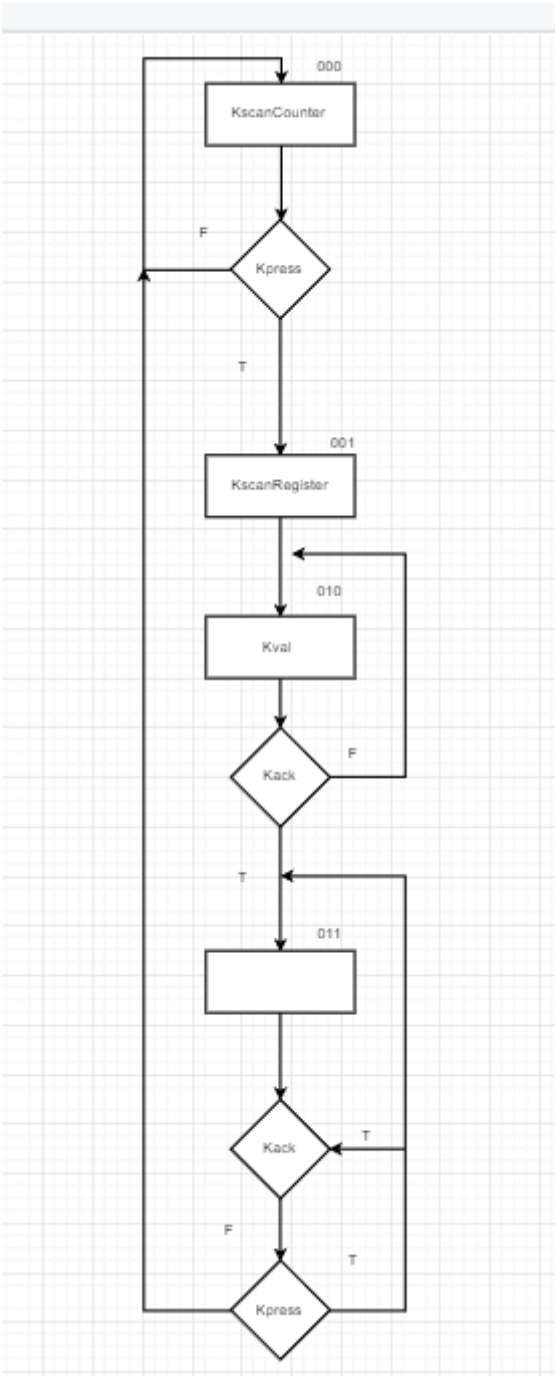


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo D.
A frequência escolhida para o *clock* foi de 1000Hz de modo a evitar o aparecimento do *bounce* o que provocaria uma leitura errada da tecla premida e para evitar também ter de usar mais componentes eletrónicos de modo a corrigir o *bounce.* No caso das resistências estas foram escolhidas com valores iguais de

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

modo a evitar qualquer diferença entre os sinais passados em cada fila do teclado mantendo assim os números corretos.

2 Key Buffer

O módulo *Key Buffer* implementa uma estrutura de armazenamento de dados, com capacidade de uma palavra de quatro bits. A escrita de dados no *Key Buffer* inicia-se com a ativação do sinal *DAV* (*Data Available*) pelo sistema produtor, neste caso pelo *Key Decode*, indicando que tem dados para serem armazenados. Logo que tenha disponibilidade para armazenar informação, o *Key Buffer* escreve os dados $D_{0:3}$ em memória. Concluída a escrita em memória, ativa o sinal *DAC* (*Data Accepted*) para informar o sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal *DAV* ativo até que *DAC* seja ativado. O *Key Buffer* só desativa *DAC* depois de *DAV* ter sido desativado.
A implementação do *key Buffer* deverá ser baseada numa máquina de controlo (*Key Buffer Control*) e num registo externo (*Output Register*), conforme o diagrama de blocos apresentado na Figura 5.

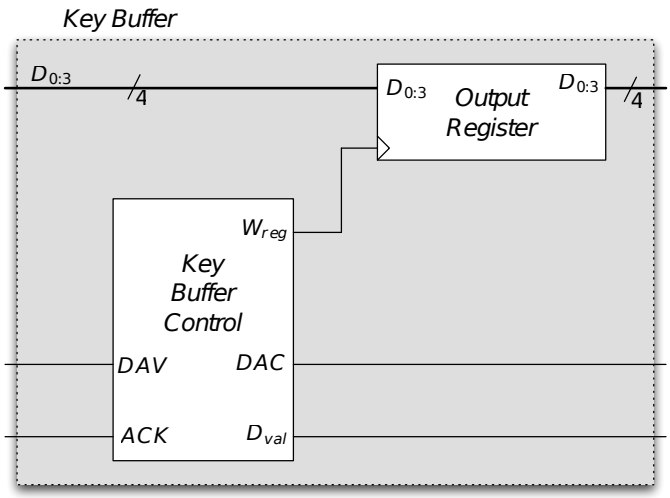


Figura 5 – Diagrama de blocos do *Key Buffer*

O bloco *Key Buffer Control* do *Key Buffer* é também responsável pela interação com o sistema consumidor, neste caso o módulo *Control*. O *Control* quando pretende ler dados do *Key Buffer*, aguarda que o sinal D_{val} fique ativo, recolhe os dados e ativa o sinal *ACK* indicando que estes já foram consumidos.
O *Key Buffer Control*, logo que o sinal *ACK* fique ativo, deve invalidar os dados baixando o sinal D_{val}, só deverá voltar a armazenar uma nova palavra depois do *Control* ter desativado o sinal *ACK*.

O bloco *Key Buffer Control* foi implementado de acordo com o diagrama de blocos representado na . Para implementar o *Key Buffer Control* foi feito uma máquina de estados com 5 estados como se pode observar no ASM.

No começo o primeiro estado encontra-se a 0, sem nenhum elemento ativo, pois não existem dados para serem guardados. Caso *DAV* não seja ativo este permanece no primeiro estado pois não existe a entrada de nenhum dado válido, caso *DAV* seja ativo este passa para o segundo estado.

Passando para o segundo estado este apresenta a ativação do *Wreg* que permite registar o código da tecla e depois de um *clock* avança para o estado seguinte.

No terceiro estado a saída apresenta *DAC*, pois os dados foram aceites pelo *Key Buffer*, este estado mantém-se enquanto existir dados disponíveis.

No quarto estado o *Dval* permanece ativo, significando que os dados recebidos são válidos e estão atualizados, e que com a ativação do *ACK* estes dados deixam de ser válidos pois já foram utilizados.

Chegando ao último estado este apresenta a saída vazia pois é impossível para o *Key* Buffer aceitar dados e os dados recebidos são inválidos sendo por isso só um estado de transição no qual o se espera que o sinal de *ACK* se baixe para voltar para o primeiro estado reiniciando todo o processo.

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

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

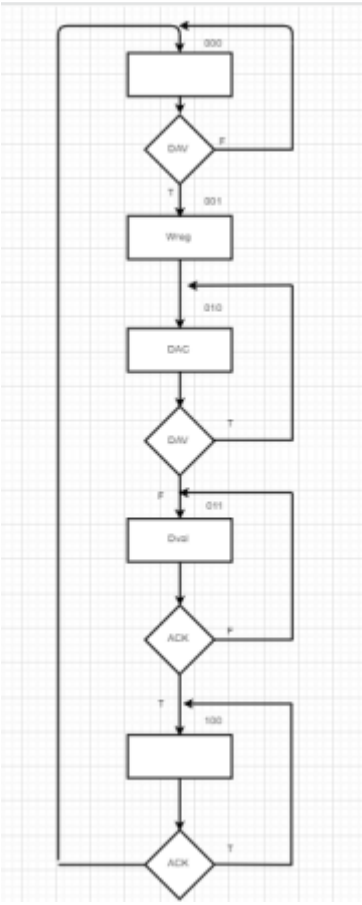


Figura 6 -Máquina de estados do bloco *Key Buffer Control*

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo D. A escolha do clock é a mesma referida antes servindo assim para evitar leituras erradas das teclas premidas.

3 Interface com o *Control*

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

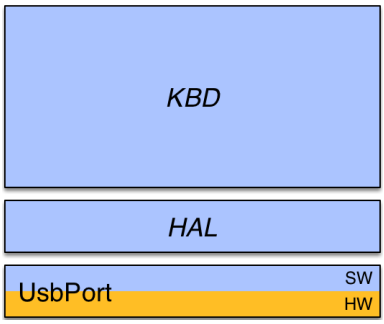


Figura 7 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

Os módulos de *software HAL* e *KBD* desenvolvidos são descritos nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos X e SSSS, respetivamente.

3.1 *HAL*

Esta classe tem a função descodificar as informações vindas do *hardware* para o *software* sendo por isso a ponte de ligação entre o *hardware* e *software*.

Para esta classe foram criadas 7 funções com o propósito de ler os bits recebidos na entrada e entregar um valor na saída. Na entrada de bits que queremos ler ou o valor que queremos apresentar na saída é estabelecido por uma máscara.

3.2 *KBD*

Na classe *KBD* apresenta apenas 3 funções que servem para realizar a leitura do teclado e entregar um *Char* correspondente à tecla premida. A primeira função é *init()* que serve para limpar o *HAL* a partir da informação do *ACK*.

A função *getKey* irá retorna a tecla *primida* caso exista alguma tecla premida.

Por fim, a função *waitKey* tem como objetivo retornar a tecla quando esta for pressionada ou não retornar nenhuma quando chegar ao fim do intervalo permitido (t*imeout*).

4 Conclusões

Na implementação do módulo *Keyboard Reader* foi utilizado uma PAL ATF750C e 4 resistências de valores iguais estando

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

estas ligadas à saída do *KeyBoard* e ao *Vcc* tendo assim sido selecionado uma frequência de 1khz para o *clock*.

O Key Decode e o Key Buffer fazem a leitura do teclado matricial e a transmissão das teclas premidas para os módulos seguintes.

Observando a latência dos módulos verificamos que o *key Decode* apresenta apenas 4 *clocks* e o *key buffer* tem 5 *clocks* não ultrapassando e foram todos usados a uma frequência de 1000Hz.

No caso do *HAL* e o *KBD* estes recebem a informação do *hardware* e traduzem-na de modo a ser possível o *software* conseguir processá-la.

Desta forma podemos concluir que o varrimento do teclado, a sua leitura e descodificação de *hardware* para *software* foi finalizada com sucesso.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

A. Descrição CUPL do bloco *Key Decode*

```
/* Key Decode */

[Qc0..1].AR = 'b'0;
[Qc0..1].SP = 'b'0;
[Qc0..1].CKMUX = Osc;

sequence [Qc0..1]{
    present 0
        out KscanCounter;
        if Kpress next 1;
        if !Kpress next 0;
    present 1
        out KscanRegister;
        default next 2;
    present 2
        out DAV;
        if DAC next 3;
        if !DAC next 2;
    present 3
        if !DAC & !Kpress next 0;
        default next 3;
}
```

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

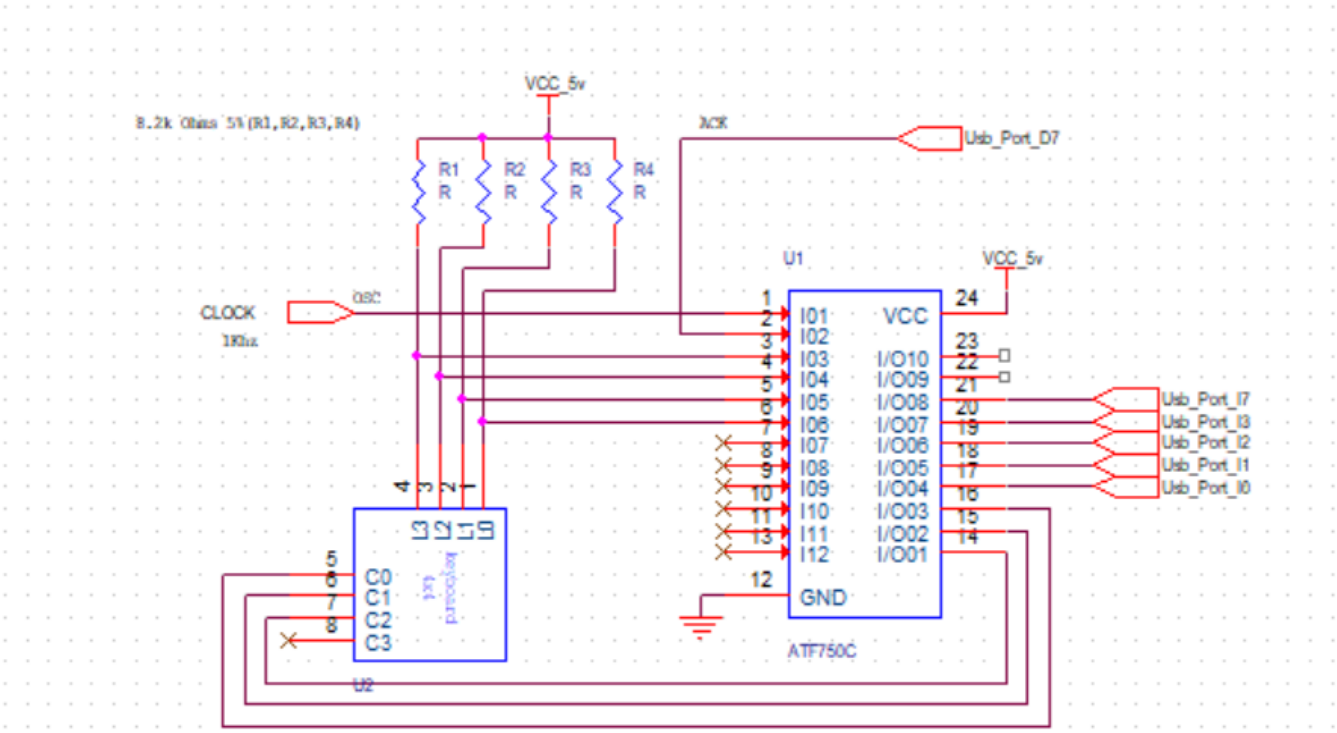
B. Descrição CUPL do bloco *Key Buffer*

```
/* Key Buffer Control */

[Qb0..2].AR = 'b'0;
[Qb0..2].SP = 'b'0;
[Qb0..2].CKMUX = Osc;

sequence [Qb0, Qb1, Qb2]{
    present 0
        if DAV  next 1;
        default next 0;
    present 1
        out WR;
        default next 2;
    present 2
        out DAC;
        if DAV next 2;
        if !DAV next 3;
    present 3
        out Dval;
        if !ACK next 3;
        if ACK next 4;
    present 4
        if ACK next 4;
        if !ACK next 0;
}
```

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

C.

D. Esquema elétrico do módulo *Keyboard Reader*



E.
F.
G. Figura 9- Esquema elétrico do Keyboard Reader, criado no OrCAD.
H.
I.
J.
K.
L.
M.
N.
O.
P.
Q.
R.
S.
T.
U.
V.

W.

X. Código *Kotlin - HAL*

Y.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

Z. AA.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

```
BB.    CC. object HAL {   // Virtualiza o acesso ao sistema UsbPort
       DD. // Inicia a classe
EE.    FF.     private var usbPortOut = 0x00
GG.    HH.
       II. fun init() = doOutput()
JJ.    KK.
                                                                    LL.
MM.    NN. // Retorna true se o bit tiver o valor lógico '1'
OO.    PP. fun isBit(mask: Int): Boolean = readBits(mask) == mask
QQ.    RR.
                                                                    SS.
TT.    UU. // Retorna os valores dos bits representados por mask presentes no
           UsbPort
VV.    WW. fun readBits(mask: Int): Int = (UsbPort.`in`().inv() and mask)
XX.    YY.
                                                                    ZZ.
AAA.   BBB.       // Escreve nos bits representados por mask o valor de value
CCC.   DDD.          fun writeBits(mask: Int, value: Int) {
EEE.   FFF.              usbPortOut = (usbPortOut and mask.inv()) or (value and
           mask)
GGG.   HHH.              doOutput()
III.   JJJ.          }
KKK.   LLL.
                                                                    MMM.
NNN.   OOO.          // Coloca os bits representados por mask no valor lógico '1'
PPP.   QQQ.          fun setBits(mask: Int) {
RRR.   SSS.              usbPortOut = usbPortOut or mask
TTT.   UUU.              doOutput()
VVV.   WWW.          }
XXX.   YYY.
                                                                    ZZZ.
AAAA.  BBBB.         // Coloca os bits representados por mask no valor lógico '0'
CCCC.  DDDD.         fun clrBits(mask: Int) {
EEEE.  FFFF.             usbPortOut = usbPortOut and mask.inv()
GGGG.  HHHH.             doOutput()
IIII.  JJJJ.         }
KKKK.  LLLL.
                                                                    MMMM.
NNNN.  OOOO.         private fun doOutput() = UsbPort.out(usbPortOut.inv())
PPPP.  QQQQ.     }
RRRR.
```

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

SSSS. Código *Kotlin - KBD*

TTTT.

UUUU. VVVV.

WWWW.

XXXX. YYYY. **object KBD** { // Ler teclas. Métodos retornam '0'..'9','#','*' ou NONE.

ZZZZ. AAAAA. private const val NONE = 0

BBBBB. CCCCC. private const val DEFAULT_KEY = NONE.toChar()

DDDDD. EEEEE. private const val KEY = 0x0F

FFFFF. GGGGG. private const val VAL = 0x80

HHHHH. IIIII. private const val ACK = 0x80

JJJJJ. KKKKK.

LLLLL.

MMMMM NNNNN. // Inicia a classe

OOOOO. PPPPP. **fun init()** {

QQQQQ. RRRRR. HAL.clrBits(ACK)

SSSSS. TTTTT. }

UUUUU. VVVVV.

WWWWW.

XXXXX YYYYY. // Retorna de imediato a tecla premida ou NONE se não há tecla premida.

ZZZZZ. AAAAAA. **fun getKey(): Char** {

BBBBBB CCCCCC. var key = DEFAULT_KEY

DDDDDD EEEEEE. if (HAL.isBit(VAL)) {

FFFFFF GGGGGG. key = when (HAL.readBits(KEY)) {

HHHHHH IIIIII. 0x0 -> '1'

JJJJJJ KKKKKK. 0x1 -> '2'

LLLLLL MMMMMM. 0x2 -> '3'

NNNNNN OOOOOO. 0x4 -> '4'

PPPPPP QQQQQQ. 0x5 -> '5'

RRRRRR SSSSSS. 0x6 -> '6'

TTTTTT UUUUUU. 0x8 -> '7'

VVVVVV WWWWWW. 0x9 -> '8'

XXXXXX YYYYYY. 0xa -> '9'

ZZZZZZ AAAAAAA. 0xc -> '*'

BBBBBBB CCCCCCC. 0xd -> '0'

DDDDDDD EEEEEEE. 0xe -> '#'

FFFFFFF GGGGGGG. else -> DEFAULT_KEY

HHHHHHH IIIIIII. }

JJJJJJJ KKKKKKK. HAL.setBits(ACK)

LLLLLLL MMMMMMM. while (HAL.isBit(VAL)) {}

NNNNNNN OOOOOOO. HAL.clrBits(ACK)

PPPPPPP QQQQQQQ. }

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

```
RRRRRRRSSSSSSS.          return key
TTTTTTTUUUUUUU.      }
VVVVVVVWWWWWWW.
                                                                XXXXXXX.
YYYYYYYZZZZZZZ.
      AAAAAAAA.
      BBBBBBBB. // Retorna quando a tecla for premida ou NONE após decorrido 'timeout'
         milisegundos.
CCCCCCCDDDDDDD.     fun waitKey(timeout: Long): Char {
EEEEEEEFFFFFFF.         var key = DEFAULT_KEY
GGGGGGGHHHHHHH.         val timeToStop = Time.getTimeInMillis()+timeout
IIIIIIIJJJJJJJ.         while(Time.getTimeInMillis()<timeToStop){
KKKKKKKLLLLLLL.             key = getKey()
MMMMMMMNNNNNNN.             if (key!= DEFAULT_KEY) break
OOOOOOOPPPPPPP.         }
QQQQQQQRRRRRRR.         return key
SSSSSSSTTTTTTT.     }
UUUUUUUVVVVVVV. }
WWWWWWWWW.
```