O módulo *LCD* implementado é constituído por dois blocos: *i*) o *Control* sendo uma arquitetura de software ; e *ii*) o bloco de *LCD*, conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em *software*, é a entidade de envio para o LCD.

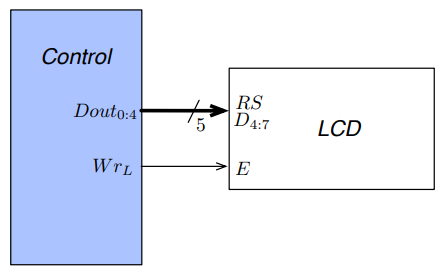
****

Figura 1 – Diagrama de blocos do módulo *LCD*

# LCD

Foi implementado o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na figura 2. No desenvolvimento do código, prestou-se especial atenção, ao *datasheet* fornecido pelos docentes, desenvolvendo o módulo com essa base. A figura 3 mostra o correspondente de cada *PIN*do *LCD* executa sobre o componente. A parte software permite então ter o controlo sobre o *LCD*, dado o exemplo como, a escrita, ficar a piscar ou até mesmo limpar o ecrã. A conexão entre o software e o *LCD* em si é feita através de um *UsbPort* que permite a conexão entre o *hardware* e o *software*.

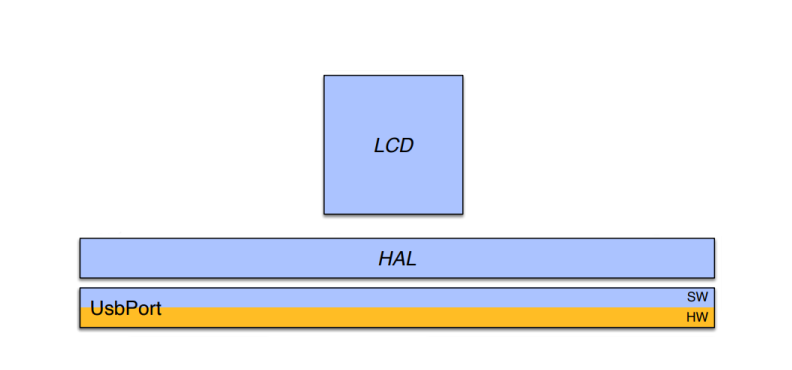
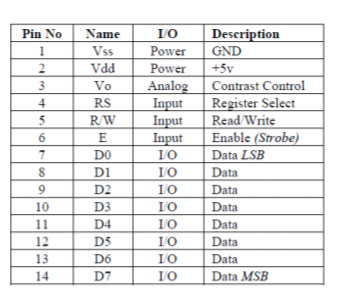
 Figura 2-Diagrama lógico do módulo *Control* de interface com o modulo *LCD*

Figura 3-Mapa de pinsdo *LCD*

O LCD em si, é um *display* de cristais líquidos, onde é permitido escrever em 2 linhas e 16 colunas. Do pin 1 ao 3 tem-se as alimentações e o contraste. No pin seguinte temos o *Register Select* (*RS*), onde é permitido dizer ao componente se os valores no barramento de dados, são de instrução ou de data. O pin 5 *Read/Write* (*R/!W*), onde a ativação do *Write* é com o valor lógico ‘0’, onde a mesma instrução permite a leitura ou escrita de dados no *LCD*. Neste projeto teve-se mais o uso da escrita, podendo este pin ficar ligado ao *ground* (valor lógico ‘0’). O pin 6 é o *Enable* do componente.

O LCD tem 8 pins de entrada de data, mas como no *UsbPort* só tem 8 pins tanto de input como de output, era fisicamente impossível, trabalhar com este a 8 *bits* e ter o resto do projeto funcional. Logo o LCD permite ser trabalhado com uma interface de 4 *bits* de dados e 3 *bits* de controlo, fornecendo a capacidade de realizar *Shift Left* da parte baixa da *word*, através do comando sobre o mesmo imposto.

Como é 4 *bits*, é usado *nibbles* para a transferência de *bits*, primeiro enviando os 4 bits de maior peso e a seguir os restantes 4, como é possível ver na figura 4. Para efetuar uma escrita é preciso que o R/!W esteja com o valor lógico ´0´, provoca-se uma descida no sinal de *Enable* para “prender” o *nibble*, depois volta-se a ascender o *clock*, para fazer “captura” do próximo *nibble*. Este processo é uma instrução de escrita de acordo com o sinal de RS.

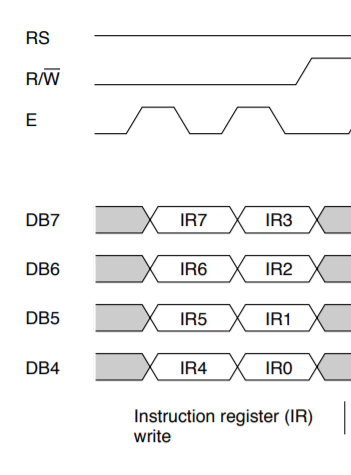


Figura 4 – Exemplo de uma transferência a 4 bits

## Classe LCD

A classe LCD implementada foi obtida através de oito funções, excluindo a inicialização do ecrã, onde quatro dessas não estão disponíveis ao utilizador, sendo as mesmas privadas. As funções que estão disponíveis a publico, foram desenvolvidas através das quatros primeiras.

A função *writeNibbles*, permite nos trabalhar a com quatros bits, onde se tem um “caminho” diferente dependendo do valor do *RS*. O seu desenvolvimento foi aplicado a várias funções do *HAL*. Escrever ao byte (*writeByte*), chama a função *writeNibble*, duas vezes, dando um pequeno espaço temporal entre ambos. Escrever um comando ou data tem, tem uma implementação muito parecida, só varia na chamada à função sobre o valor booleano que *RS* representa.

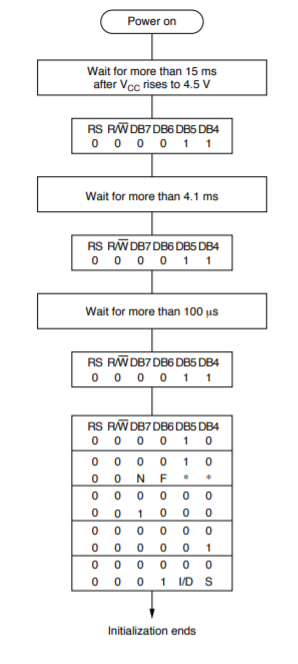
Nas funções disponíveis ao utilizador, o *init* é o que permite inicializar a classe, e o código foi todos baseado na figura 6. O valor escolhido para *N* foi 1 porque quer-se escrever em duas linhas, o de *F* = 0 para se obter o tamanho de 5 por 8 em cada caracter e assim usufruir mais do tamanho do display, o de *I/D* =1 porque quer-se que o cursor incremente ao escrever e o de *S*=0 para não haver um *Shift* no *display* logo na sua inicialização.

Figura 6 – Inicialização do *LCD*

É possível também ter controlo sobre o cursor escolhendo, a posição onde se quer escrever no *LCD,* utilizando a função para o mesmo efeito.

Também é possível escrever um caracter de cada vez, ou uma *string* inteira, dependendo da função escolhida. Para a otimização da APP para a realização de vários pedidos em sequências foi criado o *clear* que nos permite ir limpando o ecrã. Tudo isto é possível ser observado no anexo B.

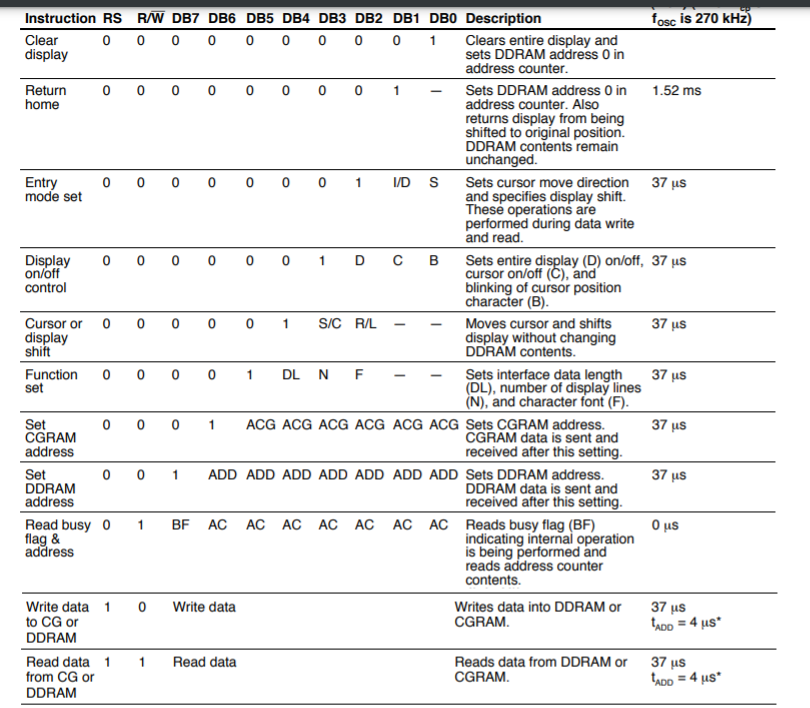
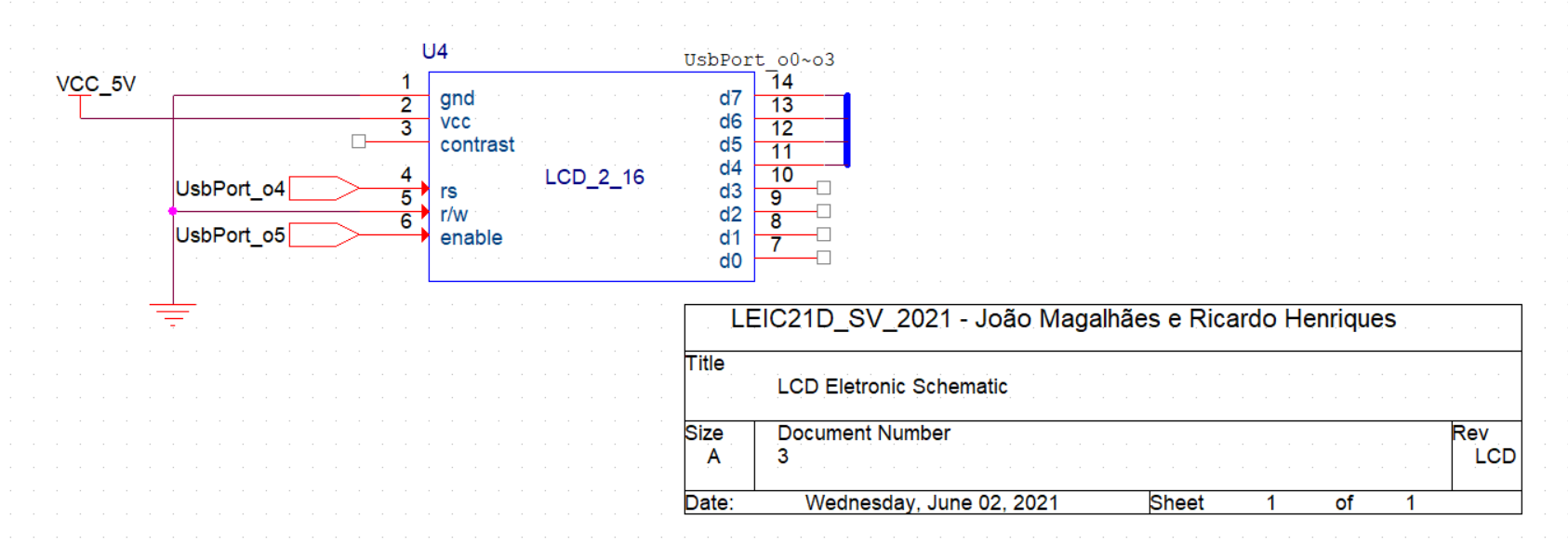


Figura 7 – Comandos para o *LCD*

## Classe TUI

# Conclusões

O modulo do *LCD,* é implementado só através duma vertente software, onde o seu conteúdo permite o controlo sobre o componente *LCD,* e alcançar os objetivos propostos. Com este controlo é possível, escrever frases ou um simples caracter, limpar o ecrã, ou até mesmo escrever em vários sítios do componente. Com a ajuda da classe TUI (bla bla bla)

1. Esquema elétrico do módulo *LCD*

1. Código *Kotlin* da classe *LCD*

object LCD {  
  
 private const val LINES = 2  
 private const val COLS = 16 *// Display dimension* private const val Enable = 0x20  
 private const val RS = 0x10  
 private const val LCDData = 0x0F  
 private const val LCDLine = 0x40 *//If wanted to write at the second line just need to add 0x40* private const val DisplayClear = 0x01  
 private const val CursorCMD = 0x80  
  
 private fun writeNibble(rs: Boolean, data: Int) {  
 *// RS -> UsbPort.i4* if (rs){  
 HAL.setBits(RS)  
 }else{  
 HAL.clrBits(RS)  
 }  
  
 *// EnableOn -> i5* HAL.setBits(Enable)  
  
 *//Data* HAL.writeBits(LCDData,data)  
  
 *// EnableOff -> i5* HAL.clrBits(Enable)  
 Time.sleep(2)  
 }  
  
  
 private fun writeByte(rs: Boolean, data: Int) {  
 writeNibble(rs,data/16) *// /16 == ShiftRight 4 times* Time.sleep(2)  
 writeNibble(rs,data)  
 }  
  
 private fun writeCMD(data: Int) {  
 writeByte(false,data)  
 }  
  
 private fun writeDATA(data: Int) {  
 writeByte(true,data)  
 }  
  
  
 fun init() {  
  
 */\*\*  
 \* All the "fly" variables, like 5 or 0x08.. It's for the LCD configuration  
 \* They're times and commands got in the manual  
 \*/* Time.sleep(80)  
  
 writeNibble(false,0x03)  
  
 Time.sleep(5)  
  
 writeNibble(false,0x03)  
  
 Time.sleep(1)  
  
 writeNibble(false,0x03)  
  
 writeNibble(false,0x02)  
 writeCMD(0x28) *// N=1 & F= 0* writeCMD(0x08)  
 writeCMD(0x01)  
 writeCMD(0x06) *// I/D=1 & S=0* writeCMD(0x0F)  
 }  
  
 *// Char write at the position.* fun write(c: Char) {  
 writeDATA(c.toInt())  
 }  
 *// String write at the position.* fun write(text: String) {  
 text.*forEach* **{** write(**it**) **}** }  
  
 fun cursor(line: Int, column: Int) {  
 val x = column + (line\* LCDLine)  
 writeCMD(x+ CursorCMD)  
 }  
  
 fun clear() {  
 writeCMD(DisplayClear)  
 }

1. Código *Kotlin* da classe *TUI*

object TUI {  
 const val LCDColuns = 16  
  
  
 fun key(l:Int, vis :Boolean):Int{  
 var s = 0.0  
 var i =0  
  
 do {  
 val x = KBD.waitKey(25000)  
 if ((s == 0.0 && x == '\*') || x == KBD.NONE.toChar()) return -1  
 if (x == '\*') {  
 LCD.clear()  
 i=0  
 } else {  
 if (!vis) {  
 LCD.write('\*')  
 } else {  
 LCD.write(x)  
 }  
  
 s += (x - '0') \* ((10.0).*pow*(l - i - 1))  
 i++  
 }  
 }while (i<l)  
  
 return s.toInt()  
 }  
  
 fun writeleft(s:String,line:Int){  
 LCD.cursor(line,0)  
 LCD.write(s)  
 }  
  
 fun writecenter(s:String,line:Int){  
 var size = s.length  
 size = (LCDColuns-s.length\*1.5-1).toInt()  
 LCD.cursor(line,size)  
 LCD.write(s)  
 }  
  
 fun writeright(s:String,line:Int){  
 var size = s.length  
 size = LCDColuns-s.length  
 LCD.cursor(line,size)  
 LCD.write(s)  
 }  
  
 fun time():String{  
 val time = LocalDateTime.now()  
 val format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm")  
 return time.format(format)  
 }  
  
  
}