Autores: Ricardo Henriques (A48322@alunos.isel.pt)
João Magalhães (A48348@alunos.isel.pt)

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.

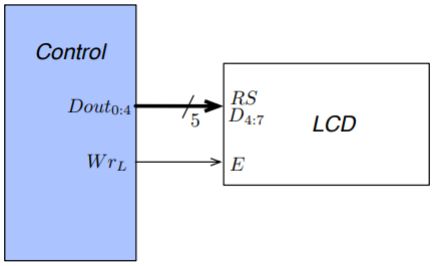


Figura 1 – Diagrama de blocos do módulo *LCD*

1 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*.

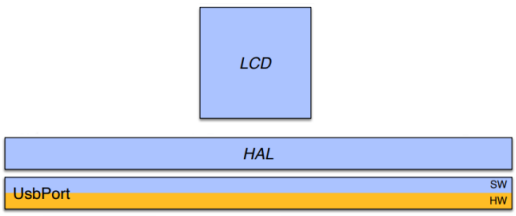


Figura 2-Diagrama lógico do módulo *Control* de interface com o modulo *LCD*

Pin No	Name	I/O	Description
1	Vss	Power	GND
2	Vdd	Power	+5v
3	Vo	Analog	Contrast Control
4	RS	Input	Register Select
5	R/W	Input	Read/Write
6	E	Input	Enable *(Strobe)*
7	D0	I/O	Data *LSB*
8	D1	I/O	Data
9	D2	I/O	Data
10	D3	I/O	Data
11	D4	I/O	Data
12	D5	I/O	Data
13	D6	I/O	Data
14	D7	I/O	Data *MSB*

Figura 3-Mapa de pins do *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.

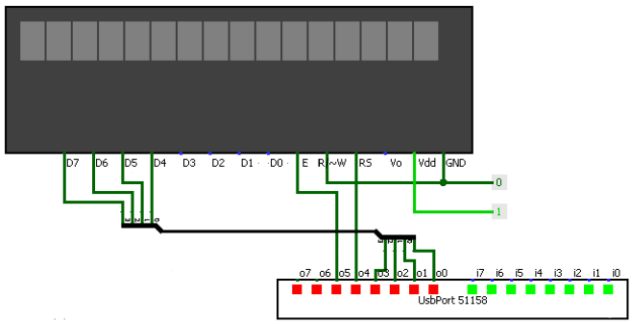


Figura 4 – Arquitetura em *logisim* do *LCD*

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.

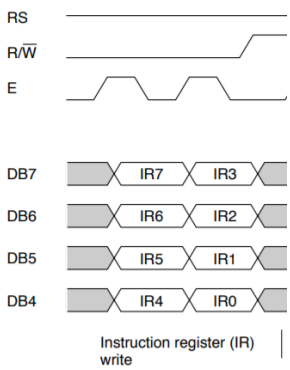


Figura 4 – Exemplo de uma transferência a 4 bits

1.1 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.

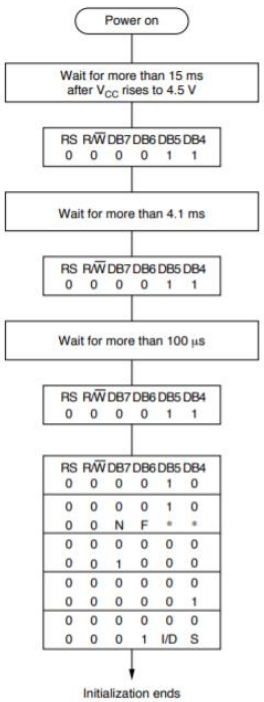


Figura 6 – Inicialização do *LCD*

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.

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	t_{osc} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	–	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	–	–	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	–	–	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*

Figura 7 – Comandos para o *LCD*

1.2 Classe *TUI*

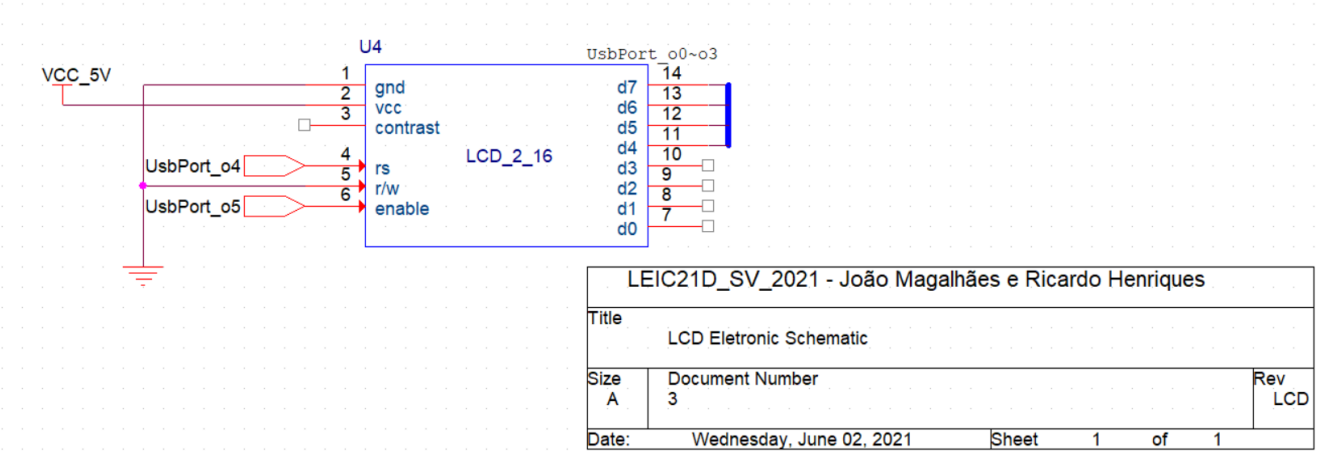
A classe *TUI* permite ao utilizador uma panóplia de funções que permitem um controlo ainda maior sobre o *LCD*. Tem uma função *key*, que concede ao utilizador a opção de quer que o seu código seja exposto, ou escondido, aparecendo no lugar do código '*'. Também foi realizado em 3 funções que permitem escolher de onde se quer começar a escrever no *display*. Com ajuda da biblioteca *java* (*java.time.LocalDataTime*), o utilizador pode ter o dia e a hora exata no seu *display*. Todas as funções implementadas são provenientes e obedecem à cadeira hierárquica, neste caso necessitando da classe *LCD*.

2 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*, o utilizador, tem opções mais concretas para trabalhar com o *LCD*, não estando limitado só a escrever, e possibilitando assim vastas opções para implementações futuras, podendo serem aproveitadas ao máximo conforme a criatividade do utilizador.

A. Esquema elétrico do módulo *LCD*



Autores: Ricardo Henriques (A48322@alunos.isel.pt)
João Magalhães (A48348@alunos.isel.pt)

B. 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)
    }
```

Autores: Ricardo Henriques (A48322@alunos.isel.pt)
João Magalhães (A48348@alunos.isel.pt)

C. 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(5000)
            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)
    }


}
```