O módulo de interface com o LCD (Serial LCD Interface, SLCDC) implementa a receção em série da informação enviada pelo módulo de controlo, entregando-a posteriormente ao LCD, conforme representado na Figura 1.

**Uma imagem com texto, captura de ecrã, file, Tipo de letra

Descrição gerada automaticamente**

Figura 1 – Diagrama de blocos do Serial LCD Controller

O SLCDC recebe em série uma mensagem constituída por cinco bits de informação. A comunicação com o SLCDC realiza-se segundo o protocolo ilustrado na Figura 2, tendo como primeiro bit de informação, o bit RS que indica se a mensagem é de controlo ou dados, os restantes bits contêm os dados a transmitir ao LCD.

Uma imagem com captura de ecrã, texto, Tipo de letra, file

Descrição gerada automaticamente

Figura 2 – Protocolo de comunicação com Serial LCD Controller

O emissor, realizado em software, quando pretende enviar uma trama para o SLCDC promove uma condição de início de trama (Start), que corresponde a uma transição descendente na linha de nLCSsel. Após a condição de início, o SLCDC armazena os bits da trama nas transições ascendentes do sinal SCLK.

# Serial Receiver

O bloco Serial Receiver do SLCDC é constituído por três blocos principais: i) um bloco de controlo; ii) um contador de bits recebidos; e iii) um bloco conversor série paralelo, designados respetivamente por Serial Control, Counter, e Shift Register. O Serial Receiver deverá ser implementado com base no diagrama de blocos apresentado na Figura 3. Uma imagem com texto, diagrama, Esquema, captura de ecrã

Descrição gerada automaticamente

Figura 3 – Diagrama de blocos do Serial Receiver

O bloco *Shift Register* foi implementado de acordo com o diagrama de blocos representado na figura 4.

O bloco *Counter* foi implementado de acordo com o diagrama de blocos representado na figura 5.

O bloco *Serial Control* foi implementado pela máquina de estados representada em *ASM-chart* na figura 6.

A máquina de estados do Serial Control é composta por 4 estados: **WAITING**, **RECEIVING**, **END** **e** **WAITING\_ACCEPT**.

No 1º estado (**WAITING**), como o nome indica ficámos à espera que uma trama pudesse começar a ser recebida através da verificação do sinal **EnRx**; caso este tome o valor lógico ‘1’ significa que continuamos no estado **WAITING** e por sua vez o sinal **clr** continua com o valor lógico ‘1’ de modo a que o bloco Counter continue com o valor da contagem com o valor ‘0’ até que uma trama possa começar a ser recebida. Caso contrário, pode-se começar a receber a trama e então prosseguimos para o estado **RECEIVING**. Chegando ao 2º estado (**RECEIVING**), é ativada a saída wr ativando o sinal de **enable** do bloco Shift Register e de seguida espera-se que os 5 bits da trama sejam recebidos através da verificação dos sinais **EnRx** e eq5, respetivamente; caso o sinal **EnRx** tenha o valor lógico ‘0’ significa que ficamos no estado **RECEIVING**, caso contrário iremos fazer a verificação do sinal **eq5**. Por sua vez, se o sinal **eq5** tiver o valor lógico ‘0’ voltaremos para o estado **WAITING**, caso contrário prosseguimos para o estado **END**. Chegando ao 3º estado (**END**), é ativada a saída **DXval** ativando o sinal **Dval** do bloco Dispatcher indicando-lhe que a trama a receber está “pronta” e é ativado também a saída **Busy**, de forma a informar ao Control que ainda estamos a processar a trama , neste caso, a enviá-la. De seguida espera-se que o bloco Dispatcher tenha recebido a trama através da verificação do sinal **Accept**. Caso o valor lógico deste seja ‘0’ mantemo-nos no mesmo estado visto que significa que o bloco Dispatcher ainda não recebeu de facto a trama, caso contrário prosseguimos para o estado **WAITING** **ACCEPT**. Por fim, no estado **WAITING** **ACCEPT** visto que o bloco Serial Receiver e o bloco Dispatcher possuem diferentes clocks e de modo a verificar se o bloco Dispatcher recebeu de facto a trama, é feita novamente uma verificação do sinal **Accept**. Se este tiver o valor lógico ‘1’ significa que pode ainda não ter acabado de receber a trama em questão e por isso permanece no mesmo estado, caso contrário, voltaremos para o estado inicial **WAITING**. Neste estado, mantemos o sinal de **Busy** ativo uma vez que não temos a certeza de o bloco Dispatcher já tratou de facto da trama, como referido anteriormente, daí a necessidade de informar o Control de que ainda não é possível enviar uma nova trama.

A descrição hardware do bloco *Serial Receiver* em VHDL encontra-se no Anexo A.

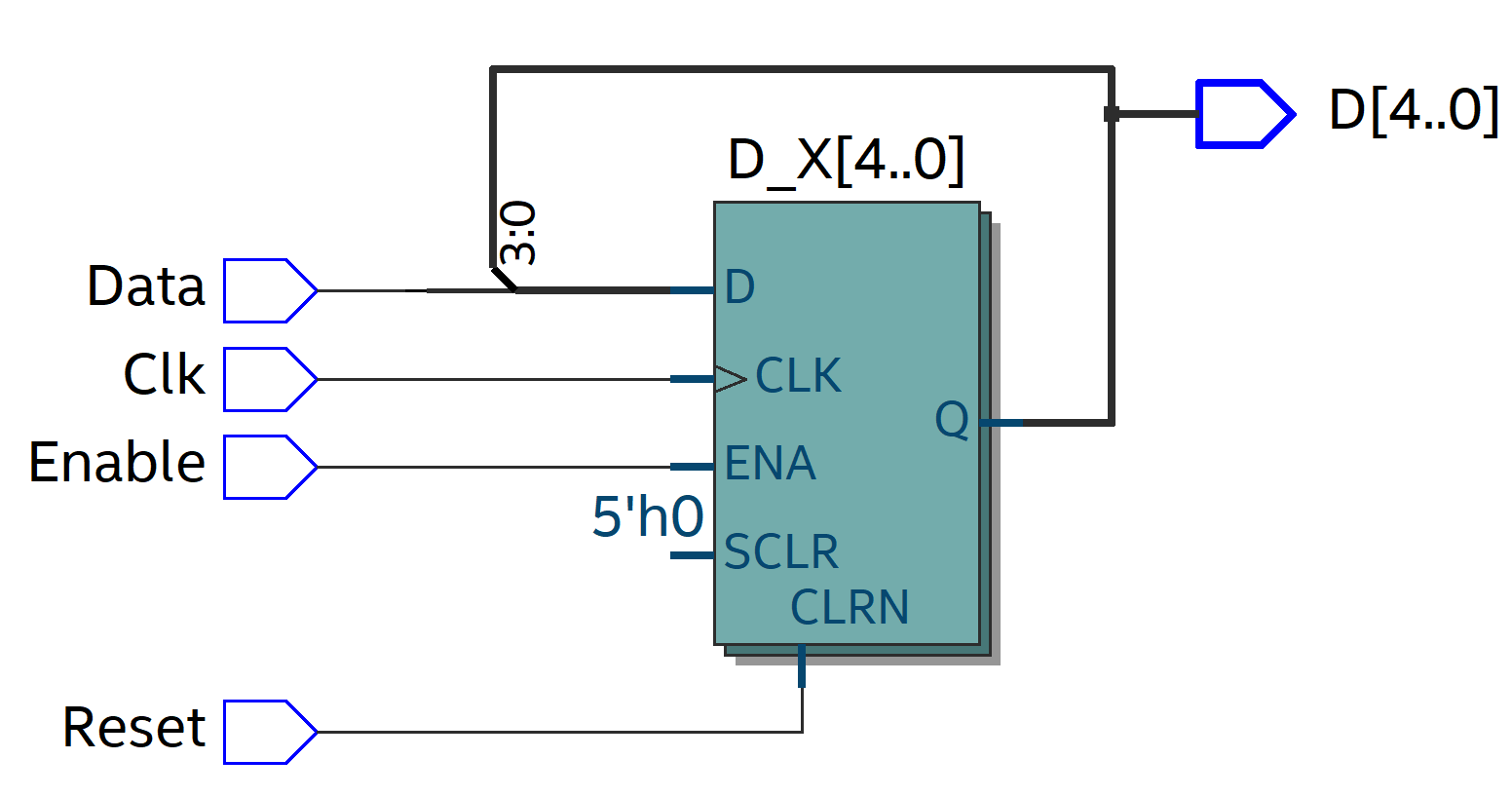


Figura 4 - Diagrama de blocosdo bloco *Shift Register*

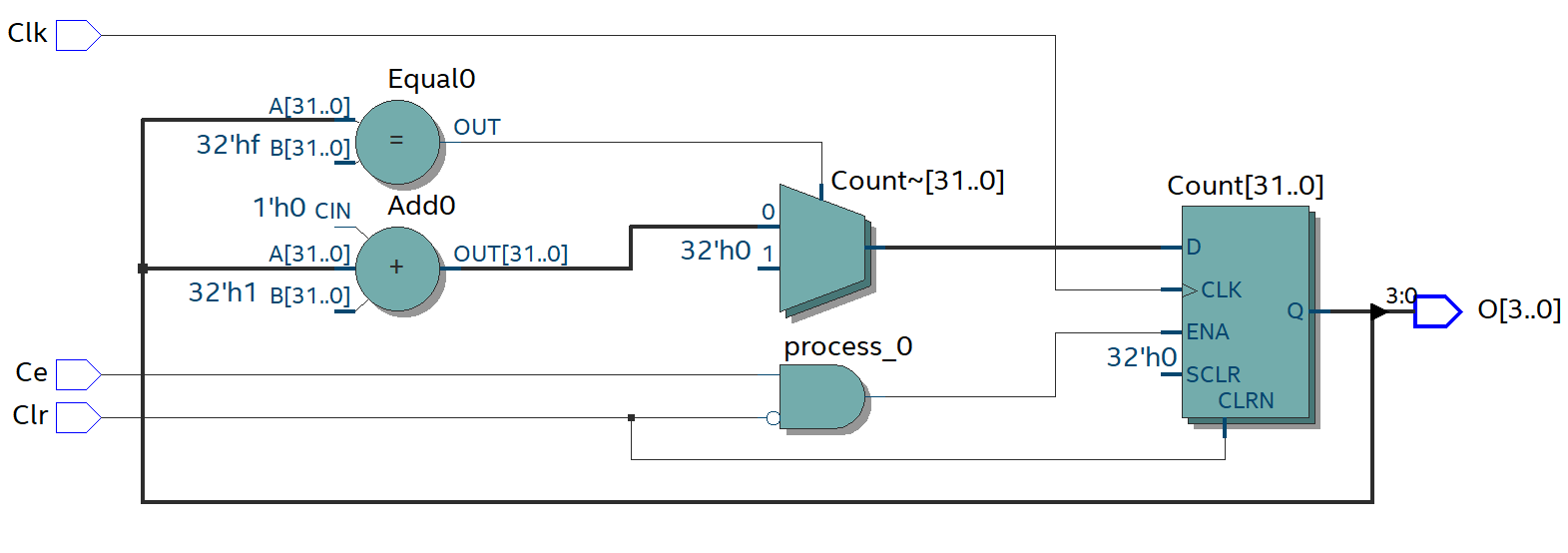


Figura 5 - Diagrama de blocosdo bloco *Counter*

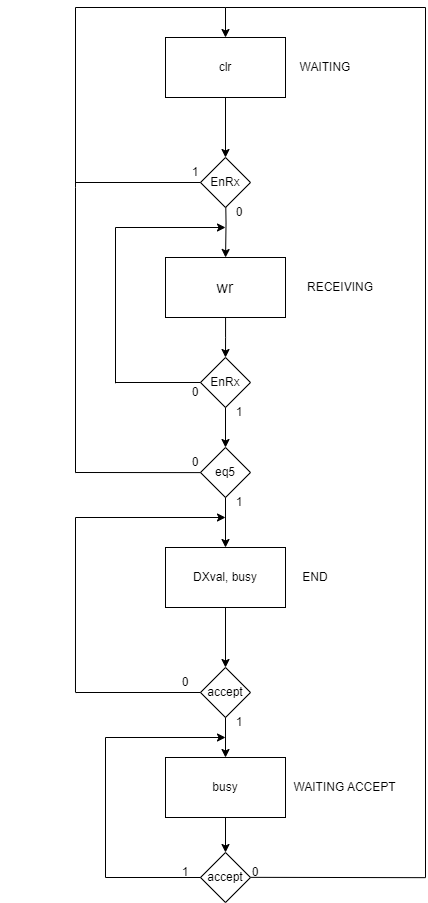


Figura 6 – Máquina de estados do bloco *Serial Control*

1.2 LCD Dispatcher

O bloco Dispatcher entrega a trama recebida pelo Serial Receiver ao LCD através da ativação do sinal WrL, após este ter recebido uma trama válida, indicado pela ativação do sinal DXval. O LCD processa as tramas recebidas de acordo com os comandos definidos pelo fabricante, não sendo necessário esperar pela sua execução para libertar o canal de receção série. Assim, o Dispatcher pode sinalizar ao Serial Receiver que a trama foi processada, ativando o sinal done.

O bloco LCD *Dispatcher* foi implementado pela máquina de estados representada em *ASM-chart* na figura 7. A máquina de estados do LCD Dispatcher é composta por 3 estados: **WAITING**, **TRAMA\_RECEBIDA** E **DONE**.  
No 1º estado, WAITING, como o nome indica ficámos à espera que tenha sido enviada uma trama através da verificação do sinal **DXval**; caso este tome o valor lógico ‘1’ significa que foi recebida uma trama e então prosseguimos para o estado **TRAMA\_RECEBIDA**, caso contrário continuamos no estado **WAITING**. Chegando ao 2º estado, **TRAMA\_RECEBIDA** é ativada a saída **WrL** ativando o sinal de **enable** do **LCD** e de seguida avançamos para o 3º e último estado. Por fim, no estado **DONE** ativamos a saída **Done** sinalizando que a trama foi entregue ao **LCD** e verificamos se o sinal DXval já se encontra a nível lógico ‘0’ e só assim voltamos ao estado inicial **WAITING**.

A descrição hardware do bloco *LCD Dispatcher* em VHDL encontra-se no Anexo A.

Com base nas descrições do bloco *Serial Receiver* e *LCD Dispatcher* implementou-se o módulo *Serial LCD Controller* de acordo com o esquema elétrico representado no Anexo B. Uma vez que o bloco dispatcher é mais lento que o Serial Receiver, foi necessário usar um bloco extra chamado ClkDiv de forma a melhorar a precisão de sincronização entre estes dois blocos.O sinal de Clk original destes dois é o da placa DE10-Lite com cerca de 50MHz.

A descrição hardware do bloco *Serial LCD Controller* em VHDL encontra-se no Anexo A.

Uma imagem com captura de ecrã, design

Descrição gerada automaticamente

Figura 7 – Máquina de estados do *LCD Dispatcher*

# Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 8.

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Figura 8 – Diagrama lógico do módulo *Control* de interface com o módulo *Serial LCD Controller*

*HAL*, *Serial Emitter e LCD* desenvolvidos são descritos nas secções 3.1., 3.2 e 3.3, e o código fonte desenvolvido nos Anexos C, D e E, respetivamente.

## *HAL*

O objeto HAL, tem como objetivo servir de ‘ponto de ligação’ entre o UsbPort e o Hardware, ou seja, permitir o funcionamento entre os diversos blocos de código implementados com o Hardware e o software desenvolvido em kotllin. Para isso foram desenvolvidas várias funções como a **isBit**, **readBits**, **writeBits**, **setBits** e **clearBits** que manipulam os bits de entrada e saída do UsbPort. Para se ter noção da última escrita nos bits de saída foi também adicionada uma variável **lastWriting**.

## *Serial Emitter*

O objeto Serial Emmiter tem como função gerar uma trama para o LCD (de acordo com a figura 2). Para tal criou-se 2 funções, **Send** e **isBusy** que para este bloco (Serial LCD Controller) não é necessária. Uma vez que tem como objetivo gerar a trama do LCD é necessário interagir com os bits do UsbPort usando então as funções do HAL.

## LCD

O objeto LCD serve para interagir com o bloco LCD, desde à sua inicialização até á interação com o mesmo. Para isso foram criadas funções que nos permitam realizar esses mesmos, tais como: **writeNibbleParallel**, **writeNibbleSerial**, **writeNibble**, **writeByte**, **writeCMD**, **writeDATA**, **write**, **cursor** e **clear**. Como o bloco LCD não foi implementado por nós, estas funções tiveram de ser baseadas na documentação do mesmo, inclusiva para o desenvolvimento da função **init**, que como o nome indica, inicializa o LCD.

# Conclusões

No módulo Serial LCD Controller foram implementados os módulos Serial Receiver e LCD Dispatcher, cada um com os seus submódulos. Para tal foi necessário realizar testes na placa DE10-Lite e também no simulador do quartus usando os módulos test bench desenvolvidos para cada um dos componentes dos módulos. Decidiu-se, para certos módulos usar-se um estilo de programação em vhdl, como por exemplo, no SerialReceiverCounter, uma vez que reduz o uso de submódulos e torna o código mais visível, uma vez que temos também outras disciplinas de programação, relaciona o projeto com outras disciplinas. Como já referido anteriormente, uma vez que o bloco dispatcher é mais lento que o Serial Receiver (visto que o LCD pode demorar a executar a trama), foi necessário usar um bloco extra chamado ClkDiv de forma a melhorar a precisão de sincronização entre estes dois blocos.



A Descrição VHDL do bloco *Serial LCD Controller*

library ieee;

use ieee.std\_logic\_1164.all;

entity SerialLCDController is

port

(

-- Input ports

nLCDsel : in std\_logic;

SClk : in std\_logic;

Clk : in std\_logic;

SDX : in std\_logic;

Reset : in std\_logic;

-- Output ports

D : out std\_logic\_vector(4 downto 0);

WrL : out std\_logic

);

end SerialLCDController;

architecture structural of SerialLCDController is

component SerialReceiver is

port

(

-- Input ports

Clk : in std\_logic;

SDX : in std\_logic;

SClk : in std\_logic;

nSS : in std\_logic;

Accept : in std\_logic;

Reset : in std\_logic;

-- Output ports

D : out std\_logic\_vector(4 downto 0);

DXval : out std\_logic;

Busy : out std\_logic

);

end component;

component LCDDispatcher is

port

(

-- Input ports

Clk : in std\_logic;

Reset : in std\_logic;

DXval : in std\_logic;

Din : in std\_logic\_vector(4 downto 0);

-- Output ports

WrL : out std\_logic;

Dout : out std\_logic\_vector(4 downto 0);

Done : out std\_logic

);

end component;

component ClkDiv is

generic (div: natural := 50000000);

port

(

Clk\_in : in std\_logic;

Clk\_out : out std\_logic

);

end component;

signal Done\_X, Dxval\_X, Clk\_X, Busy\_X : std\_logic;

signal Din\_X : std\_logic\_vector(4 downto 0);

begin

U0: SerialReceiver port map (Clk => Clk, SDX => SDX, SClk => SClk, nSS => nLCDsel, Accept => Done\_X, Reset => Reset,

D => Din\_X, DXval => Dxval\_X, Busy => Busy\_X);

U1: LCDDispatcher port map (Clk => Clk\_X, Reset => Reset, Dxval => Dxval\_X, Din => Din\_X,

WrL => WrL, Dout => D, Done => Done\_X);

U2: ClkDiv generic map(500) port map (Clk\_in => Clk, Clk\_out => Clk\_X);

end structural;

1. Descrição VHDL do bloco *Serial Receiver*

library ieee;

use ieee.std\_logic\_1164.all;

entity SerialReceiver is

port

(

-- Input ports

Clk : in std\_logic;

SDX : in std\_logic;

SClk : in std\_logic;

nSS : in std\_logic;

Accept : in std\_logic;

Reset : in std\_logic;

-- Output ports

D : out std\_logic\_vector(4 downto 0);

DXval : out std\_logic;

Busy : out std\_logic

);

end SerialReceiver;

architecture structural of SerialReceiver is

component SerialControl is

port

(

-- Input ports

Clk : in std\_logic;

EnRx : in std\_logic;

Accept : in std\_logic;

Eq5 : in std\_logic;

Reset : in std\_logic;

-- Output ports

Clr : out std\_logic;

Wr : out std\_logic;

DXval : out std\_logic;

Busy : out std\_logic

);

end component;

component ShiftRegister is

port

(

-- Input ports

Data : in std\_logic;

Clk : in std\_logic;

Enable : in std\_logic;

Reset : in std\_logic;

-- Output ports

D : out std\_logic\_vector(4 downto 0)

);

end component;

component SerialReceiverCounter IS

port

(

-- Input ports

Clk : in std\_logic;

Ce : in std\_logic;

Clr : in std\_logic;

-- Output ports

O : out std\_logic\_vector(3 downto 0)

);

end component;

signal Clr\_X, Wr\_X, Eq5\_X : std\_logic;

signal O\_X : std\_logic\_vector(3 downto 0);

begin

Eq5\_X <= not O\_X(3) and O\_X(2) and not O\_X(1) and O\_X(0);

U0: SerialControl port map (Clk => Clk, EnRx => nSS, Eq5 => Eq5\_X, Accept => Accept, Reset => Reset,

Wr => Wr\_X, Clr => Clr\_X, DXval => DXval, Busy => Busy);

U1: ShiftRegister port map (Clk => SClk, Reset => Reset, Data => SDX, Enable => Wr\_X,

D => D);

U2: SerialReceiverCounter port map (Clk => SClk , Clr => Clr\_X, Ce => '1',

O => O\_X);

end structural;

A. Descrição VHDL do bloco *LCD Dispatcher*

library ieee;

use ieee.std\_logic\_1164.all;

entity LCDDispatcher is

port

(

-- Input ports

Clk : in std\_logic;

Reset : in std\_logic;

DXval : in std\_logic;

Din : in std\_logic\_vector(4 downto 0);

-- Output ports

WrL : out std\_logic;

Dout : out std\_logic\_vector(4 downto 0);

Done : out std\_logic

);

end LCDDispatcher;

architecture behavioral of LCDDispatcher is

type STATE\_TYPE is (STATE\_WAITING, STATE\_TRAMA\_RECEBIDA, STATE\_DONE);

signal CurrentState, NextState: STATE\_TYPE;

begin

--FLIP-FLOP'S

CurrentState <= STATE\_WAITING when Reset = '1' else NextState when rising\_edge(Clk);

--GENERATE NEXT STATE

GenerateNextState:

process (CurrentState, DXval)

begin

case CurrentState is

when STATE\_WAITING => if (Dxval = '1') then

NextState <= STATE\_TRAMA\_RECEBIDA;

else

NextState <= STATE\_WAITING;

end if;

when STATE\_TRAMA\_RECEBIDA => NextState <= STATE\_DONE;

when STATE\_DONE => if (Dxval = '1') then

NextState <= STATE\_DONE;

else

NextState <= STATE\_WAITING;

end if;

end case;

end process;

-- GENERATE OUTPUTS

WrL <= '1' when (CurrentState = STATE\_TRAMA\_RECEBIDA) else '0';

Dout <= Din;

Done <= '1' when (CurrentState = STATE\_DONE) else '0';

end behavioral;

1. Atribuição de pinos do módulo *Serial LCD Controller*

set\_location\_assignment PIN\_N5 -to Clk

set\_location\_assignment PIN\_C10 -to SDX

set\_location\_assignment PIN\_C11 -to SClk

set\_location\_assignment PIN\_D12 -to nLCDsel

set\_location\_assignment PIN\_A12 -to Reset

-set\_location\_assignment PIN\_A8 -to D[0]

-set\_location\_assignment PIN\_A9 -to D[1]

-set\_location\_assignment PIN\_A10 -to D[2]

-set\_location\_assignment PIN\_B10 -to D[3]

-set\_location\_assignment PIN\_D13 -to D[4]

-set\_location\_assignment PIN\_B11 -to WrL

set\_location\_assignment PIN\_W8 -to D[0]

set\_location\_assignment PIN\_AA15 -to D[1]

set\_location\_assignment PIN\_W13 -to D[2]

set\_location\_assignment PIN\_AB13 -to D[3]

set\_location\_assignment PIN\_Y11 -to D[4]

set\_location\_assignment PIN\_V5 -to WrL

1. Código *Kotlin* - *HAL*

import isel.leic.UsbPort

// Virtualiza o acesso ao sistema UsbPort

object HAL {

private var lastWriting = 0

// Inicia a classe

fun init() {

UsbPort.write(lastWriting)

}

// Retorna true se o bit tiver o valor lógico ‘1’

fun isBit(mask: Int): Boolean {

val temp = mask and UsbPort.read()

return mask == temp

}

// Retorna os valores dos bits representados por mask presentes no UsbPort

fun readBits(mask: Int): Int = mask and UsbPort.read()

// Escreve nos bits representados por mask o valor de value

fun writeBits(mask: Int, value: Int) {

val a = mask and value

val b = mask.inv() and lastWriting

val c = a or b

UsbPort.write(c)

lastWriting = c

}

// Coloca os bits representados por mask no valor lógico ‘1’

fun setBits(mask: Int) {

writeBits(mask,0xFF)

}

// Coloca os bits representados por mask no valor lógico ‘0’

fun clearBits(mask:Int) {

writeBits(mask,0x00)

}

}

fun main() {

val mask = 0b00001111

HAL.init()

HAL.setBits(mask)

Thread.sleep(2000)

HAL.clearBits(mask)

Thread.sleep(2000)

println(HAL.isBit(mask))

Thread.sleep(2000)

HAL.writeBits(mask, 6)

Thread.sleep(2000)

// Change the value of the input port bits

val currentBits = HAL.readBits(mask)

println(currentBits)

}

1. Código *Kotlin* – *Serial Emitter*

// Envia tramas para os diferentes módulos Serial Receiver.

object SerialEmitter {

private const val clk: Long = 50

enum class Destination {

LCD,

DOOR

}

// Inicia a classe

fun init() {

HAL.init()

HAL.clearBits(nSDCsel\_MASK)

HAL.clearBits(nLCDsel\_MASK)

HAL.clearBits(SCLK\_MASK)

HAL.clearBits(SDX\_MASK)

}

// Envia uma trama para o SerialReceiver identificado o destino em addr e os bits de dados em ‘data’.

fun send(addr: Destination, data: Int) {

Thread.sleep(clk)

var nSSMask = nLCDsel\_MASK

if (addr == Destination.DOOR) {

nSSMask = nSDCsel\_MASK

while (isBusy()) {

Thread.sleep(clk \* 10)

}

}

HAL.clearBits(nSSMask)

for (i in 4 downTo 0) {

HAL.clearBits(SCLK\_MASK)

val sdx = (data shr i) and 1

if (sdx == 1) HAL.setBits(SDX\_MASK) else HAL.clearBits(SDX\_MASK)

HAL.setBits(SCLK\_MASK)

}

HAL.clearBits(SCLK\_MASK)

HAL.setBits(nSSMask)

Thread.sleep(clk)

}

// Retorna true se o canal série estiver ocupado

fun isBusy(): Boolean = HAL.isBit(BUSY\_MASK)

}

fun main() {

SerialEmitter.init()

for (i in 0..31) {

SerialEmitter.send(SerialEmitter.Destination.LCD, i)

Thread.sleep(250)

}

}

1. Código Kotlin – LCD

// Escreve no LCD usando a interface a 4 bits.

**object LCD {**

// Dimensão do display.

**private const val LINES = 2**

**private const val COLS = 16**

// Escreve um nibble de comando/dados no LCD em paralelo

**private fun writeNibbleParallel(rs: Boolean, data: Int) {**

if (rs) HAL.setBits(LCD\_RS\_MASK) else HAL.clearBits(LCD\_RS\_MASK)

HAL.writeBits(LCD\_DATA\_MASK, data)

Thread.sleep(1)

HAL.setBits(LCD\_E\_MASK)

Thread.sleep(1)

HAL.clearBits(LCD\_E\_MASK)

**}**

// Escreve um byte de comando/dados no LCD em série

**private fun writeNibbleSerial(rs: Boolean, data: Int) {**

val rsValue = if (rs) 1 else 0

SerialEmitter.send(SerialEmitter.Destination.LCD, data shl 1 or rsValue)

**}**

// Escreve um nibble de comando/dados no LCD

**private fun writeNibble(rs: Boolean, data: Int) {**

writeNibbleSerial(rs, data)

**}**

// Escreve um byte de comando/dados no LCD

**fun writeByte(rs: Boolean, data: Int) {**

writeNibble(rs, data shr 4)

writeNibble(rs, data)

**}**

// Escreve um comando no LCD

**fun writeCMD(data: Int) {**

writeByte(false, data)

**}**

// Escreve um dado no LCD

**fun writeDATA(data: Int) {**

writeByte(true, data)

**}**

// Envia a sequência de iniciação para comunicação a 4 bits.

**fun init() {**

SerialEmitter.init()

Thread.sleep(16) // Esperar x ms

writeNibble(false, 3)

Thread.sleep(5) // Esperar x ms

writeNibble(false, 3)

Thread.sleep(1) // Esperar x ms

writeNibble(false, 3)

writeNibble(false, 2)

writeCMD(40)

writeCMD(8)

writeCMD(1)

writeCMD(6)

writeCMD(15)

**}**

// Escreve um caráter na posição corrente.

**fun write(c: Char) {**

writeDATA(c.code)

**}**

// Escreve uma string na posição corrente.

**fun write(text: String) {**

for (c in text)

write(c) // Escreve no LCD usando a interface a 4 bits.

object LCD {

// Dimensão do display.

private const val LINES = 2

private const val COLS = 16

// Escreve um nibble de comando/dados no LCD em paralelo

private fun writeNibbleParallel(rs: Boolean, data: Int) {

if (rs) HAL.setBits(LCD\_RS\_MASK) else HAL.clearBits(LCD\_RS\_MASK)

HAL.writeBits(LCD\_DATA\_MASK, data)

Thread.sleep(1)

HAL.setBits(LCD\_E\_MASK)

Thread.sleep(1)

HAL.clearBits(LCD\_E\_MASK)

}

// Escreve um byte de comando/dados no LCD em série

private fun writeNibbleSerial(rs: Boolean, data: Int) {

val rsValue = if (rs) 1 else 0

SerialEmitter.send(SerialEmitter.Destination.LCD, data shl 1 or rsValue)

}

// Escreve um nibble de comando/dados no LCD

private fun writeNibble(rs: Boolean, data: Int) = writeNibbleSerial(rs, data)

// Escreve um byte de comando/dados no LCD

fun writeByte(rs: Boolean, data: Int) {

writeNibble(rs, data shr 4)

writeNibble(rs, data)

}

// Escreve um comando no LCD

fun writeCMD(data: Int) = writeByte(false, data)

// Escreve um dado no LCD

fun writeDATA(data: Int) = writeByte(true, data)

// Envia a sequência de iniciação para comunicação a 4 bits.

fun init() {

SerialEmitter.init()

Thread.sleep(16) // Esperar x ms

writeNibble(false, 3)

Thread.sleep(5) // Esperar x ms

writeNibble(false, 3)

Thread.sleep(1) // Esperar x ms

writeNibble(false, 3)

writeNibble(false, 2)

writeCMD(40)

writeCMD(8)

writeCMD(1)

writeCMD(6)

writeCMD(15)

}

// Escreve um caráter na posição corrente.

fun write(c: Char) = writeDATA(c.code)

// Escreve uma string na posição corrente.

fun write(text: String) {

for (c in text) write(c)

}

// Envia comando para posicionar cursor (‘line’:0..LINES-1 , ‘column’:0..COLS-1)

fun cursor(line: Int, column: Int) = writeCMD((line \* 0x40 + column) or 0x80)

// Envia comando para limpar o ecrã e posicionar o cursor em (0,0)

fun clear() {

writeCMD(1)

cursor(0,0)

}

}

fun main() {

LCD.init()

println(" LCD INITIALIZED ")

var count = 0

while (true) {

LCD.write("LCD COUNT: $count")

Thread.sleep(1000)

LCD.clear()

count++

if (count == 10) {

LCD.clear()

LCD.cursor(1, 0)

LCD.write("WE REACHED 10!")

LCD.clear()

count = 0

}

}

}