Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

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

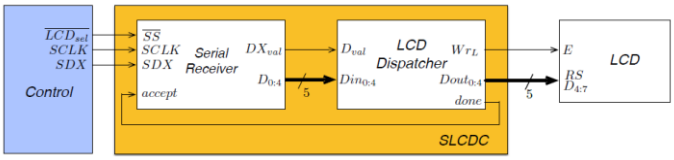


Figura 1 – Diagrama de blocos do módulo *Serial LCD Controller*

O módulo SLCDC recebe em série uma mensagem constituída por 5 *bits* de informação e um bit de paridade. A comunicação com este módulo realiza-se segundo o protocolo ilustrado na Figura 2, em que o bit RS é o primeiro bit de informação e indica se a mensagem é de controlo ou dados. Os seguintes 4 bits contêm os dados a entregar ao LCD. O último bit contém a informação de paridade ímpar, utilizada para detetar erros de transmissão.

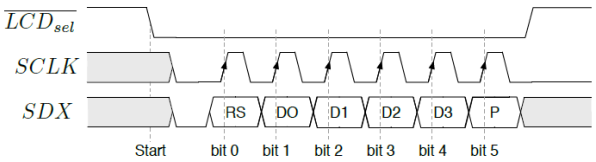


Figura 2 - Protocolo de comunicação com o módulo *Serial LCD Controller*

O código do módulo *Serial LCD Controller* em CUPL encontra-se no Anexo C para referência e o esquema elétrico encontra-se no Anexo D.

1 Serial Receiver

O bloco *Serial Receiver* do módulo SLCDC é constituído por quatro blocos principais: i) um bloco de controlo; ii) um bloco conversor série paralelo; iii) um contador de bits recebidos; e iv) um bloco de validação de paridade, designados por *Serial Control*, *Shift Register*, *Counter* e *Parity Check* respetivamente. O bloco *Serial Receiver* deverá ser implementado com base no diagrama de blocos apresentado na Figura 3.

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

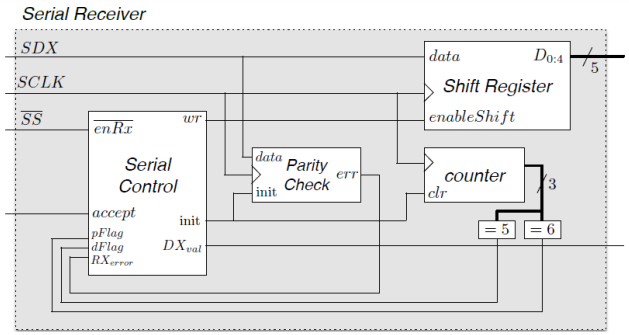


Figura 3 - Diagrama de blocos do bloco *Serial Receiver*

Tendo em conta as restrições de *clock* do módulo LCD *Dispatcher*, para garantir sincronização entre as máquinas de estados dos dois módulos usa-se um certo *clock* numa máquina e a sua respetiva negação na outra, com uma frequência máxima de 4MHz tal como está referido em LCD *Dispatcher* mais à frente.

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

Descrição dos estados posteriormente presentes no ASM-chart do bloco Serial Control:

'wait for input' – Visto que o ar (*asynchronous reset*) dos blocos *Counter* e *Parity Check* é o próprio init do *Serial Control*, inicia-se a máquina de estados com valor lógico 1 nessa variável.

'receive data' – O output wr fica ativo de modo a autorizar (*Enable*) o bloco *Shift Register* a registar os valores de data. Neste estado a condição apresentada por *dFlag* avança para o estado seguinte caso o valor de *Counter* seja 5, caso contrário permanece no mesmo estado.

'receive parity bit' – Um estado intermédio com o objetivo de verificação de erro. Com a variável *pFlag* verifica-se a partir do bloco *counter* se o bit de paridade já foi introduzido, e de seguida pondera-se a partir de DXval se o bit de paridade está correto.

'validate data' – Ativa-se o output DXval para informar o bloco *LCD Dispatcher* que a data foi validada e espera-se *accept* proveniente desse mesmo módulo para se poder prosseguir para o próximo estado.

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

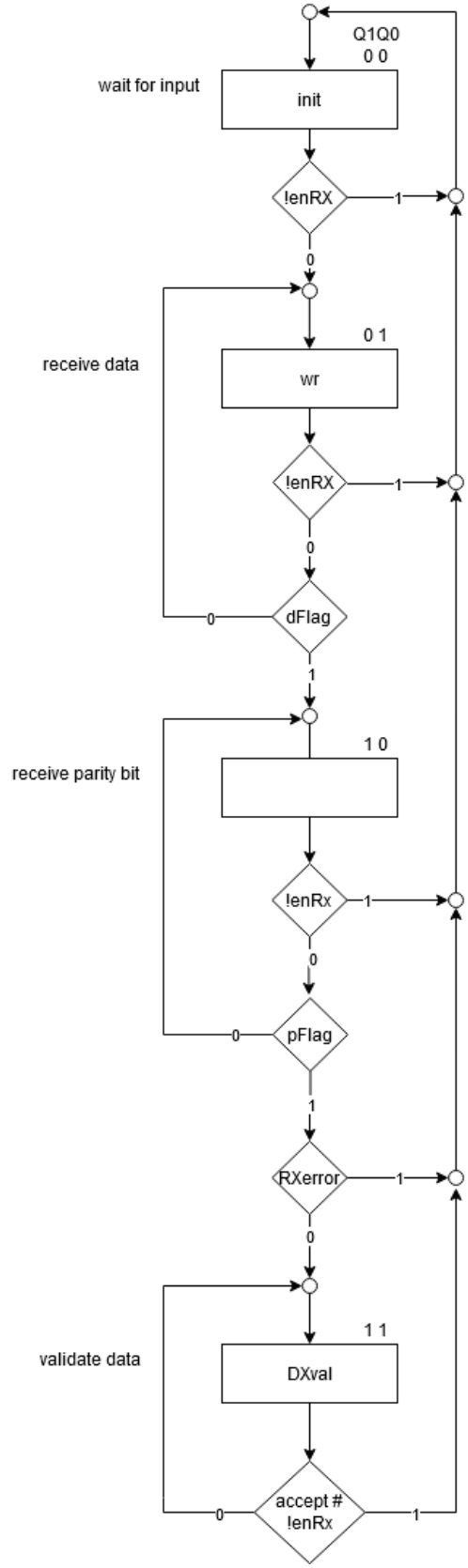


Figura 4 – ASM chart do bloco *Serial Control*

2 LCD Dispatcher

O bloco *LCD Dispatcher* é responsável pela entrega das tramas válidas recebidas pelo bloco *Serial Receiver* ao LCD, através da ativação do sinal *WrL*. A receção de uma trama válida é sinalizada pela ativação do sinal Dval.
O processamento das tramas recebidas pelo LCD respeita os comandos definidos pelo fabricante, não sendo necessário esperar pela sua execução para libertar o canal de receção série. Assim, o bloco LCD *Dispatcher* pode ativar, prontamente, o sinal done para notificar o bloco *Serial Receiver* que a trama já foi processada.
Para controlar o módulo LCD *Dispatcher* foi desenvolvida uma máquina de estados (representada na figura 5) que consiste na sua totalidade no próprio módulo *LCD Dispatcher*.

Descrição dos estados posteriormente presentes no ASM-chart do bloco LCD Dispatcher:

'wait val' – Este estado é responsável pela verificação de Dval que provém do bloco *Serial Control*, sendo essa variável responsável pela validação da data. Caso Dval tenha valor lógico 1 então avança-se para o próximo estado, caso contrário, permanece-se no mesmo.

'activate enable' – Este estado consiste exclusivamente para ativar o output *WrL* de modo a ativar *Enable* do LCD pela duração de um *clock* desta máquina de estados.

'ended' – Tendo este bloco (*LCD Dispatcher*) enviado a data e o sinal WrL, neste estado aciona-se o output *done* de forma a informar o módulo *Serial Receiver* que se recebeu e se enviou a respetiva trama.

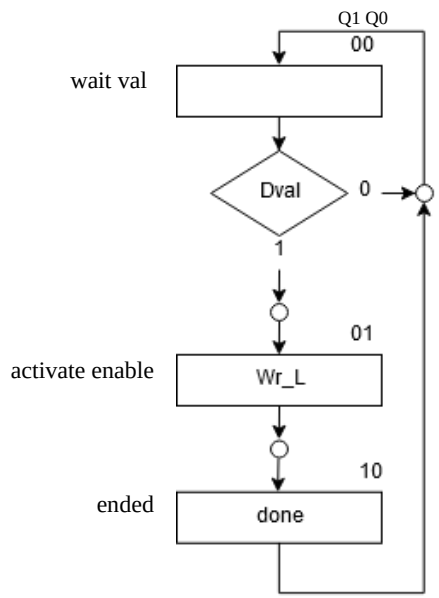


Figura 5 – ASM chart do bloco *LCD Dispatcher*

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

Dependendo do hardware e fabricante, neste caso o LCD, com que se está a operar deve ter-se em conta a frequência do *clock* desta máquina de estados. No caso apresentado anteriormente o foco está principalmente no estado '01', e com base nas figuras 6 e 7, '*tw*' que consiste no tempo em que *Enable* tem valor lógico 1, pode-se concluir que o *clock* desse estado não pode ter um tempo inferior a 230ns, mas '*tc*', que consiste em dois *clocks* da máquina de estados (em relação a um ciclo de *Enable*) não pode ser inferior a 500ns, podendo então chegar-se a uma conclusão final de que o tempo mínimo é de (500/2)ns = 250ns que corresponde a uma frequência máxima de *clock* de 4MHz. Com essa informação determinou-se que o *clock* deste módulo será a negação do *clock* do módulo *Serial Receiver* para se poder garantir sincronização.

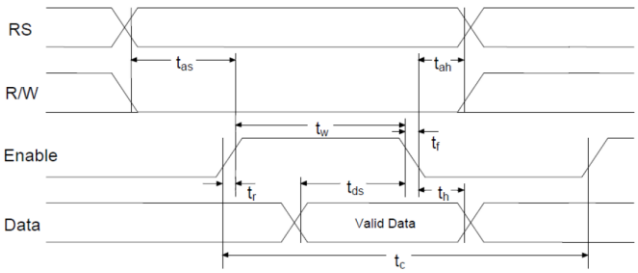


Figura 6 – Diagrama temporal do ciclo de leitura do LCD

Parameter	Symbol	Min (1)	Typ (1)	Max (1)	Unit
Enable Cycle Time	t_c	500	-	-	ns
Enable Pulse Width (High)	t_w	230	-	-	ns
Enable Rise/Fall Time	t_r,t_f	-	-	20	ns
Address Setup Time	t_{as}	40	-	-	ns
Address Hold Time	t_{ah}	10	-	-	ns
Data Setup Time	t_{ds}	80	-		ns
Data Hold Time	t_h	10	-	-	ns

Figura 7 – Tabela de tempos do Diagrama temporal do ciclo de leitura do LCD

A respetiva descrição hardware do bloco *LCD Dispatcher* em CUPL encontra-se no Anexo B.

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

2.1 Classe *LCD*

Esta classe realiza a inicialização do LCD em uso (LCD-HD44780U) e facilita o uso do mesmo.

Os métodos *writeNibbleParallel*(), *writeNibbleSerial*(), *writeNibble*(), *writeByte*(), *writeCMD*() e *writeDATA*() são internos, isto é, que traduzem o código binário para o LCD no modo correto (8/4 bits, série/paralelo, comando/data).

Temos também *write*() que escreve um caracter ou uma *String* na posição atual do cursor. E por fim *cursor*(), que altera a posição do cursor e *clear*() que apaga o ecrã todo e posiciona o cursor na posição (0,0).

Esta classe conta também com uma classe auxiliar chamada LCDCode que guarda todos os valores e as possibilidades de código para os comandos/datas (eliminando assim qualquer hipótese de "código *hardcoded*").

2.2 Classe *SerialEmitter*

Esta classe envia os bits em série para o módulo correspondente (neste momento: SLCDC).

Para realizar um envio correto é necessário, a partir do método *send()*, obter o bit *Serial Select*, o tamanho da trama e a data a enviar.

clockPulse() realiza somente um pulso do *SCLK* de modo a validar o bit atual.

2.3 Classe *CustomCharacter*

Esta nova classe serve unicamente para criar carateres customizados no *LCD*. Funciona com objetos, isto é, para criar um caráter, cria-se o seu padrão (como exemplo, ver código Java da classe *LCD*, o padrão da *spaceship* e do *invader*) e a seguir cria-se um objeto do tipo *CustomCharacter* que, mais tarde, pode ser adicionado à memória *CGRAM* do *LCD* com o comando: i.e: *CustomCharacter.add(spaceship)*. Estes dois padrões são inicializados no método *init()* da classe *LCD*.

3 Conclusões

Em *software:* Da 1ª avaliação, a classe LCD sofreu a alteração necessária para retirar a parte "*hardcoded*" do código. Para isso criámos uma classe chamada LCDCode para guardar todos os códigos possíveis a realizar com o LCD, sendo possível selecioná-los por simples booleanos dentro dos métodos correspondentes. Outro aspeto a ter em conta é, o SCLK controlado na classe *SerialEmitter* não tem qualquer intervalo na transição do valor *high* para *low*; isto poderá trazer problemas mais tarde na montagem do projeto em laboratório.

Em *hardware:* De início deparámo-nos com uma complicação que consistia no *LCD Dispatcher*, o módulo *SLCDC* pode funcionar perfeitamente sem *LCD Dispacher* se as respetivas mudanças forem feitas no bloco *Serial Receiver*, mas esse não é o objetivo. O bloco *Serial Receiver* tem como objetivo ter uma estrutura flexível com objetivo de ter compatibilidade em outros módulos e por fim, ter um bloco de adaptação, que neste caso será o bloco *LCD Dispatcher*. Tendo-se chegado a essa conclusão prosseguiu-se então à implementação desse bloco em forma de máquina de estados, e então dessa forma surge outro benefício que consiste no respetivo *clock* da máquina de estados. Esse *clock* dá a possibilidade de controlar e adaptar o bloco *LCD Dispatcher* às regras temporais dos respetivos LCDs.

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

A. Descrição CUPL do bloco *Serial Receiver*

Tendo em conta o diagrama de blocos da figura 3 e o seu próprio diagrama de blocos, o bloco *Serial Receiver* é composto pelos seguintes módulos:
- *Serial Control*;
- *Shift Register*;
- *Parity Check*;
- *Counter*.

- Input:

-SCLK (PIN 1) que consiste num *clock* proveniente do módulo de controlo que vai atuar sobre os blocos *Shift Register*, *Parity Check* e *Counter* como os seus respetivos *clocks*;

-SDX (PIN 2), representado por 'data' no código CUPL, que consiste nas tramas de informação enviadas bit a bit com objetivo de serem entregues ao LCD de forma correta;

- $\overline{SS}$(PIN 3), representado por *enRx* no código CUPL contabilizando a sua respetiva negação, que corresponde a um bit que seleciona este módulo *register*;

- *accept* (como ambos os módulos de SLCDC estão contidos no mesmo chip, não é necessário um Pin específico) é uma variável com origem no módulo *LCD Dispatcher* que informa a confirmação do envio da trama que o módulo *Serial Receiver* enviou ao *LCD Dispatcher* e o despachamento da mesma trama tal como o respetivo *enable* para o LCD.

- *Serial Control*:

Uma máquina de estados (respetivo ASM *chart* na figura 4) que usufrui de 2 flip-flops (E0..1) que disponibiliza 4 estados. Tem como objetivo orientar os restantes blocos presentes neste módulo. Como *clock* deste bloco tem-se CLK (PIN 11), um *clock* à parte com objetivo de ter uma frequência mais elevada que SCLK para prevenir perda de informação e *bugs* mas que não pode exceder uma frequência de 4MHz devido à sua sincronização com a máquina de estados do módulo LCD *Dispatcher*.

- *Shift Register*:

Bloco de registos com shift right, possui 5 flip-flops (SR0..4), com clock SCLK(PIN 1). Tem uma singularidade que consiste num *enable* não relacionado com *clock*, dessa forma a solução foi implementar um *multiplexer* à entrada de cada flip-flop com entrada de data do flip-flop anterior (no caso do primeiro, será a variável 'data') e do seu próprio output, com *enable* sendo a variável de saída do bloco *Serial Control* denominada 'wr', de forma a prevenir que *clocks* de SCLK provoquem um "*shift*" intencionalmente.

- *Parity Check*:

Bloco de verificação de erro, usa 1 flip-flop PB, tem como clock SCLK e como ar (*asynchronous reset*) *init* proveniente do bloco *Serial Control*. Recebe o valor de SDX no input (bit de paridade) e no output err é apresentado valor lógico 1 caso haja erro, e valor lógico 0 caso contrário. Para determinar esse erro verificam-se os bits todos de SDX e caso deem um número impar não houve erro, caso deem par então houve interferência em algum bit portanto há erro.

- *Counter*:

Contador de 3 bits, usando os *flip-flops* C0..2, com clock SCLK e ar (*asynchronous reset*) init proveniente do bloco *Serial Control*. A sua particularidade consiste em informar o bloco *Serial Control* quando o mesmo estiver a apresentar os números naturais 5 e 6, "101" e "110" respetivamente.

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

B. Descrição CUPL do bloco *LCD Dispatcher*

O módulo *LCD Dispatcher* é apresentado como uma máquina de estados cujo seu *clock* é nCLK, uma negação do clock usado pela máquina de estados *Serial Control* presente no módulo *Serial Receiver*, que tem que respeitar uma frequência não superior a 4MHz.

Este módulo apesar de ter como entrada os valores D0 até D4 (trama de informação), esse valor sai diretamente em $D_{out\ 0:4}$

para o LCD, logo, assim pode ter-se sempre em consideração que a data está sempre presente nos inputs do LCD, o que é um benefício em relação a Dout0 que corresponde a RS, que tem de estar presente no LCD antes de *Enable*, e depois também. Respeitando todas as regras do diagrama temporal de leitura deste LCD (figuras 6 e 7) a data fica também presente ainda depois de *Enable* ficar inativo. Obedecendo o diagrama, t_c que corresponde a um ciclo de *Enable*, não pode ser inferior a 500ns, logo dividindo esse valor por 2 *clocks* da máquina de estados, 250ns é o tempo mínimo que um *clock* pode ter, que corresponde a uma frequência máxima de 4MHz tal como mencionado acima. Se se tiver em conta esta conclusão respetiva ao *clock*, todos os tempos a cumprir dos parâmetros presentes no diagrama temporal serão cumpridos pois terão todos um tempo mínimo de 1 *clock* (‘<4MHz’ ou ‘>250ns’).

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

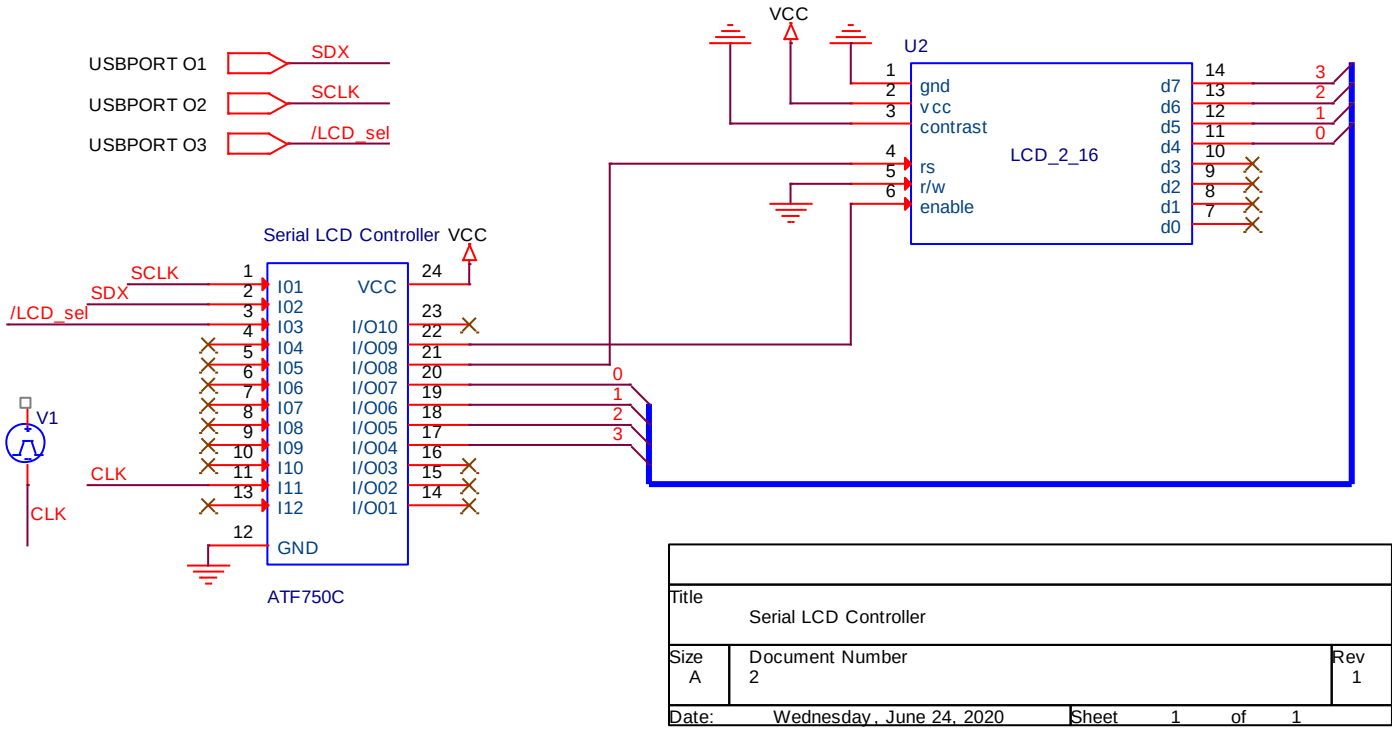
C. Código CUPL do módulo *Serial LCD Controller*

```
/* INPUT PINS */
PIN 1 = SCLK;
PIN 2 = SDX;     /*SDX*/
PIN 3 = SS;      /*SS=!(!SS)*/
PIN 11 = CLK;

/* OUTPUT PINS */
PIN 14 = T0;
PINNODE 33 = T1;
PINNODE 29 = E0;
PIN 15 = nCLK;
PIN 16 = E1;
PIN 17 = SR4;            /*D7 LCD input pin */
PIN 18 = SR3;            /*D6 LCD input pin */
PIN 19 = SR2;            /*D5 LCD input pin */
PIN 20 = SR1;            /*D4 LCD input pin*/
PIN 21 = SR0;            /*RS LCD input pin */
PIN 22 = Wr_L;
PIN 23 = init;

PINNODE 28 = PB;
PINNODE 30 = C2;
PINNODE 31 = C1;
PINNODE 32 = C0;

/******* BODY  *********/
/*----*** SERIAL RECEIVER *----***/
/** Serial Control **/
enRx = !SS ;
RXerror = err ;

[E0..1].ck = CLK;
[E0..1].ar = 'b'0;
[E0..1].sp = 'b'0;
SEQUENCE [E1 , E0] {
        PRESENT 0
                OUT init;
                IF enRx Next 1;
                DEFAULT Next 0;
        PRESENT 1
                OUT wr;
                IF !enRx Next 0;
                IF dFlag Next 2;
                DEFAULT Next 1;
        PRESENT 2
                IF pFlag & !RXerror Next 3;
                IF pFlag & RXerror # !enRx NEXT 0;
                DEFAULT Next 2;
        PRESENT 3
                OUT DXval;
                IF accept # !enRx NEXT 0;
                DEFAULT Next 3;
}
```

```
/** Shift Register **/
enableShift = wr ;
[SR0..4].ckmux = SCLK ;
[SR0..4].sp = 'b'0 ;
[SR0..4].ar = 'b'0 ;

SR4.d = SDX & enableShift # SR4 & !enableShift;
[SR3..0].d = [SR4..1] & enableShift # [SR3..0] &
!enableShift ;

/** Parity Check **/
PB.ckmux = SCLK;
PB.sp = 'b'0;
PB.ar = init;

PB.T = SDX;

err = !PB;

/** Counter **/

[C0..2].ck = SCLK;
[C0..2].sp = 'b'0;
[C0..2].ar = init;

C0.T = 'b'1;
C1.T = C0;
C2.T = C0 & C1;

dFlag = C0 & !C1 & C2; /* =5 */
pFlag = !C0 & C1 & C2; /* =6 */


/*----*** LCD Dispatcher ***----*/
accept = done;
nCLK = !CLK;

[T0..1].ck = nCLK;
[T0..1].ar = 'b'0;
[T0..1].sp = 'b'0;

SEQUENCE [T1 , T0] {

        PRESENT 0
                IF DXval NEXT 1;
                DEFAULT NEXT 0;

        PRESENT 1
                OUT Wr_L;
                DEFAULT NEXT 2;

        PRESENT 2
                OUT done;
                DEFAULT NEXT 0;

        PRESENT 3
                DEFAULT NEXT 0;
}
```

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

D. Esquema elétrico do módulo *SLCDC*



Title: Serial LCD Controller		
Size: A	Document Number: 2	Rev: 1
Date: Wednesday, June 24, 2020	Sheet 1 of 1	

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

E. Código Java da classe *LCD*

```
package edu.isel.lic.peripherals.lcd;

import edu.isel.lic.link.SerialEmitter;
import isel.leic.utils.*;
import edu.isel.lic.link.HAL;

public class LCD // Escreve no LCD usando a interface a 4 bits.
{
  public static final int LINES = 2, COLS = 16;        // Dimensão do
display.
  public static final int SERIAL_DATA_SIZE = 5;        // Dimensão da
data a enviar para SDX
  public static final int RS_MASK = 0x10;              // Máscara para
selecionar entre Data/Command
  public static final int PARALLEL_ENABLE_MASK = 0x20;   //
Mascara do bit de enable em modo paralelo
  private static final int DDRAM_LINE = 0x40;

  private static final int[]
      spaceship_pattern = {
      0B11110,
      0B11000,
      0B11100,
      0B11111,
      0B11100,
      0B11000,
      0B11110,
      0B00000},
      invader_pattern = {
      0B11111,
      0B11111,
      0B10101,
      0B11111,
      0B11111,
      0B10001,
      0B10001,
      0B00000};
  public static final CustomCharacter spaceship = new
CustomCharacter(spaceship_pattern);
  public static final CustomCharacter invader = new
CustomCharacter(invader_pattern);

  // Define se a interface com o LCD é série ou paralela
  private static final boolean SERIAL_INTERFACE = true;

  // Escreve um nibble de comando/dados no LCD em paralelo
  private static void writeNibbleParallel (boolean rs, int data) {
    HAL.setBits(rs ? RS_MASK : 0);                  //rs on/off
    HAL.setBits(data);                          //data on/off
    HAL.setBits(PARALLEL_ENABLE_MASK);                  //enable on
    HAL.clrBits(PARALLEL_ENABLE_MASK);              //enable off
    HAL.clrBits(HAL.MAX_BITS);                  //clear bits
  }

  // Escreve um nibble de comando/dados no LCD em série
  private static void writeNibbleSerial (boolean rs, int data) {
    data = (rs) ? (data<<1)+0x01 : data<<1;
    SerialEmitter.send(SerialEmitter.Destination.SLCD,
SERIAL_DATA_SIZE, data);
  }

  // Escreve um nibble de comando/dados no LCD
  private static void writeNibble (boolean rs, int data) {
    if(SERIAL_INTERFACE)
      writeNibbleSerial(rs, data);
    else
      writeNibbleParallel(rs, data);
  }

  // Escreve um byte de comando/dados no LCD
  private static void writeByte (boolean rs, int data) {
    int lowerBits = data & 0x0F, higherBits = data>>>4;
    writeNibble(rs, higherBits); writeNibble(rs, lowerBits);
  }

  // Escreve um comando no LCD
  public static void writeCMD (int data) { writeByte (false, data); }
  // Escreve um dado no LCD
  public static void writeDATA (int data) { writeByte (true, data); }
  // Escreve um caráter na posição corrente.
  public static void write (char c) { writeDATA (c); }
  // Escreve uma string na posição corrente.
  public static void write (String txt) {
    for (int i = 0; i < txt.length(); ++i)
      write(txt.charAt(i));
  }

  // Escreve um caráter customizado na posição corrente.
  public static void write (CustomCharacter custom_char) {
    writeDATA((char)(custom_char.getAddr()));
  }

  // Envia comando para posicionar cursor ('lin':0..LINES-1 ,
'col':0..COLS-1)
  public static void cursor (int lin, int col) {
    writeCMD(LCDCode.set_ddram_address(((lin * DDRAM_LINE)) |
col));
  }

  // Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
  public static void clear () { writeCMD (LCDCode.clear_display()); }

  // Envia a sequência de iniciação para comunicação a 4 bits.
  public static void init () {
    //init1 -> 0011 -> 0110 -> 110
    Time.sleep(20);
    writeNibble(false, 0x03);
    //init2 -> 0011 -> 0110 -> 110
    Time.sleep(5);
    writeNibble(false, 0x03);
    //init3 -> 0011-> 0110 -> 110
    Time.sleep(1);
    writeNibble(false, 0x03);
    //set 4bit mode -> 0010 -> 0100 -> 100
    writeNibble(false, 0x02);
    //number of display lines and character font -> 0010 -> 0100 -> 100
& 1000 -> 10000
    writeCMD(LCDCode.function_set(false, true, false));
    //display off -> 0000 -> 0 & 1000 -> 10000
    writeCMD(LCDCode.display_control(false, false, false));
    //display clear -> 0000 -> 0 & 0001 -> 0010 -> 10
    writeCMD(LCDCode.clear_display());
    //cursor direction and display shift mode -> 0000 -> 0 & 0110 ->
1100
    writeCMD(LCDCode.entry_mode_set(true, false));
    //display on (entire display, cursor on, cursor blinking on) -> 0000 -
> 0 & 1111 -> 11110
    writeCMD(LCDCode.display_control(true, false, false));
    //Custom Characters
    CustomCharacter.add(spaceship);
    CustomCharacter.add(invader);
    clear();
  }
}
```

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

F. Código Java da classe *LCDCode*

```
package edu.isel.lic.peripherals.lcd;

import edu.isel.lic.link.HAL;

public class LCDCode // Baseado no pdf QuickReference do
{
  private static int value = 0; // valor hexadecimal do código
correspondente
  private static final int
    CLEAR_DISPLAY = 0x01,
    RETURN_HOME = 0x02,
    ENTRY_MODE_SET = 0x04,
    DISPLAY_CONTROL = 0x08,
    CURSOR_DISPLAY_SHIFT = 0x10,
    FUNCTION_SET = 0x20,
    SET_CGRAM_ADDR = 0x40,
    SET_DDRAM_ADDR = 0x80;

  // Para efeitos de teste - needs work
  public static void main (String[] args) {
    HAL.init();
    LCD.init();
  }

  // Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
  public static int clear_display() { return CLEAR_DISPLAY; }

  // Retorna o ecrã e o cursor para a posição original (endereço 0)
  public static int return_home() { return RETURN_HOME; }

  /**
   * Define a direção do movimento do cursor e especifica a deslocação
do ecrã
   * @param cursor_move_to_right - true: cursor move-se para a
direita; false - cursor move-se para a esquerda
   * @param cursor_follows_display_shift - true: ecrã desloca-se com
o cursor; false: ecrã não se desloca com o cursor
   * @return - valor hexadecimal do código correspondente
   */
  public static int entry_mode_set(boolean cursor_move_to_right,
boolean cursor_follows_display_shift) {
    return value =
        ENTRY_MODE_SET +
        ((cursor_move_to_right) ? 0x02 : 0) +
        ((cursor_follows_display_shift) ? 0x01 : 0);
  }

  /**
   * Liga/desliga o ecrã, liga/desliga cursor e liga/desliga cursor a
piscar
   * @param display_on - true: liga o ecrã; false - desliga o ecrã
   * @param cursor_on - true: liga o cursor; false - desliga o cursor
   * @param cursor_blink - true: liga o modo cursor a piscar; false -
desliga o modo cursor a piscar
   * @return - valor hexadecimal do código correspondente
   */
  public static int display_control (boolean display_on, boolean
cursor_on, boolean cursor_blink) {
    return value =
        DISPLAY_CONTROL +
        ((display_on) ? 0x04 : 0) +
        ((cursor_on) ? 0x02 : 0) +
        ((cursor_blink) ? 0x01 : 0);
  }

  /**
   * Define o modo de deslocação do cursor e do ecrã
   * @param display_shift - true: o ecrã desloca-se; false: o cursor
desloca-se
   * @param shift_to_right - true: deslocamento para a direita; false -
deslocamento para a esquerda
   * @return - valor hexadecimal do código correspondente
   */
  public static int cursor_or_display_shift (boolean display_shift,
boolean shift_to_right) {
    return value =
        CURSOR_DISPLAY_SHIFT +
        ((display_shift) ? 0x08 : 0) +
        ((shift_to_right) ? 0x04 : 0);
  }

  /**
   * Define a dimensão da interface, número de linhas a mostrar e a
fonte das letras
   * @param interface_data_length_8bits - true: interface a 8 bits;
false: interface a 4 bits
   * @param number_of_display_lines_2 - true: 2 linhas no ecrã;
false: 1 linha no ecrã
   * @param character_font_5x10 - true: fonte do tipo 5x10; false:
fonte do tipo 5x8
   * @return - valor hexadecimal do código correspondente
   */
  public static int function_set (boolean interface_data_length_8bits,
boolean number_of_display_lines_2, boolean character_font_5x10) {
    return value =
        FUNCTION_SET +
        ((interface_data_length_8bits) ? 0x10 : 0) +
        ((number_of_display_lines_2) ? 0x08 : 0) +
        ((character_font_5x10) ? 0x04 : 0);
  }

  /**
   * Defina endereço para o módulo CGRAM
   * @param addr - endereço (6 bits)
   * @return - valor hexadecimal do código correspondente
   */
  public static int set_cgram_address (int addr) {
    return value =
        SET_CGRAM_ADDR + addr;
  }

  /**
   * Define endereço para o módulo DDRAM
   * @param addr - endereço (7 bits)
   * @return - valor hexadecimal do código correspondente
   */
  public static int set_ddram_address (int addr) {
    return value =
        SET_DDRAM_ADDR + addr;
  }
}
```

Autores: João Nunes (A47220@ alunos.isel.pt)
Alexandre Luís (A47222@ alunos.isel.pt)
Miguel Marques (A47204@ alunos.isel.pt)

G. Código Java da classe *CustomCharacter*

```
package edu.isel.lic.peripherals.lcd;

import java.util.ArrayList;

public class CustomCharacter {

  ArrayList<Integer> char_pattern = new ArrayList<Integer>();

  private final int ddram_addr, fontHeight;
  private static int numOfChars = 0;
  private static final int MAX_5x8_numOfChars = 8, MAX_5x10_numOfChars = 4;

  public CustomCharacter (int[] pattern_array) {
    for (int index : pattern_array)
        this.char_pattern.add(index);

    ddram_addr = numOfChars; // Font=5x8 -> cgram_addr={0,1,2,3,4,5,6,7} | Font=5x10 -> cgram_addr={0,1,2,3}
    fontHeight = pattern_array.length; // Font=5x8 -> fontHeight=8 | Font=5x10 -> fontHeight=10
    numOfChars++;

    // Algoritmo para dar ciclo aos endereços na CGRAM caso encher até ao fim o espaço disponível
    if (fontHeight == 8)
      numOfChars = (numOfChars == MAX_5x8_numOfChars) ? 0 : numOfChars++;
    else
      numOfChars = (numOfChars == MAX_5x10_numOfChars) ? 0 : numOfChars++;
    }

  public static void add (CustomCharacter char_object) {

    for (int i = 0; i < char_object.fontHeight; ++i) {
      LCD.writeCMD(LCDCode.set_cgram_address(char_object.ddram_addr * char_object.fontHeight + i));
      LCD.writeDATA(char_object.char_pattern.get(i));
    }
    LCD.writeCMD(LCDCode.set_ddram_address(0));
  }

  public int getAddr () {
    return ddram_addr;
  }
}
```