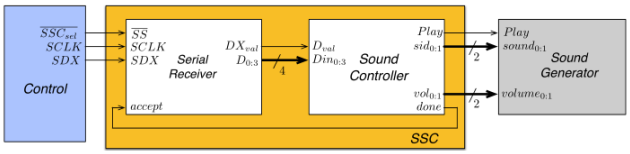
Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

O módulo *Serial Sound Controller* (SSC) implementa a interface com um possível *Sound Generator*, fazendo a receção em séria da informação enviada pelo módulo de controlo (desenvolvido em *software*) e entregando-a posteriormente ao tal dispositivo de som. Neste caso usaremos a *ATB* para simular o dispositivo de som.



- Figura 1 – Diagrama de blocos do módulo *Serial Sound Generator Controller*

- O módulo *SSC* recebe em série uma mensagem constituída por 4 *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 *Sound Generator*. O último *bit* contém a informação de paridade ímpar, utilizada para detetar erros de transmissão. A figura 2b representa as instruções possíveis que o *Sound Generator* pode realizar.

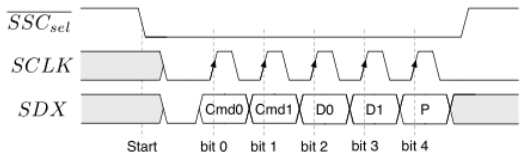


Figura 2a – Protocolo de comunicação com o módulo *SSC*

Cmd 1 0	*data* 1 0	*Function*
0 0	* *	*stop*
0 1	* *	*play*
1 0	s_1 s_0	*set sound*
1 1	v_1 v_0	*set volume*

Figura 2b – Comandos do módulo *Sound Generator*

1 Serial Receiver

O bloco *Serial Receiver* do SSC, sendo semelhante ao do SLCDC, é constituído por quatro blocos principais: i) um bloco de controlo; ii) um bloco conversor série-paralelo; iii) um contador de bits recebidos; 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.

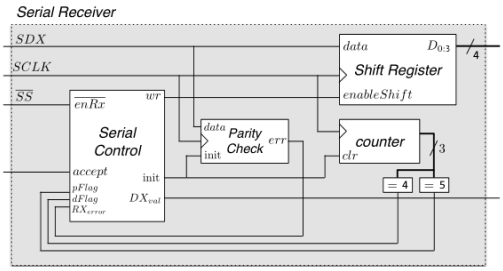


Figura 3 – Diagrama de blocos do bloco *Serial Receiver*

O bloco *Serial Control* foi implementado a partir do *ASM-chart* da Figura 4.

Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

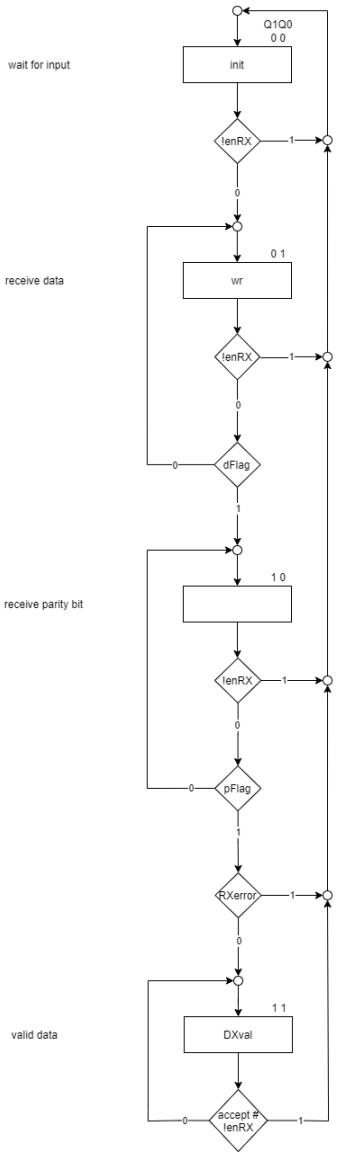


Figura 4 – *ASM-chart* do bloco *Serial Control*

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 C*ounter* seja 4, 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.

2 Sound Controller

O bloco *Sound Controller* é responsável pela entrega da informação válida que neste caso é constituída por: i) estar no modo *Play/Stop;* ii) qual o *id* do som escolhido; iii) qual o nível do volume. Estas três informações são permanentes até nova instrução que substitua as mesmas. Para isso, este bloco foi desenvolvido de acordo com a Figura 5.

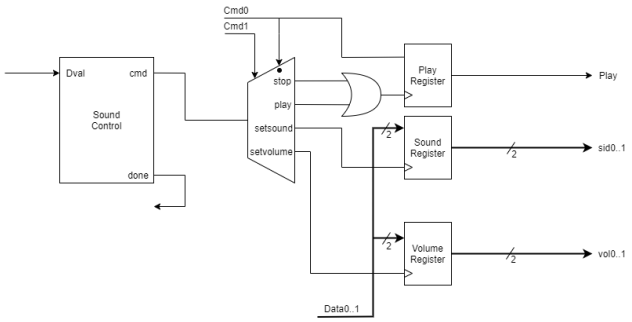


Figura 5 – Diagrama de blocos do bloco *Sound Controller*

O bloco *Sound Control* foi implementado a partir do *ASM-chart* da Figura 6.

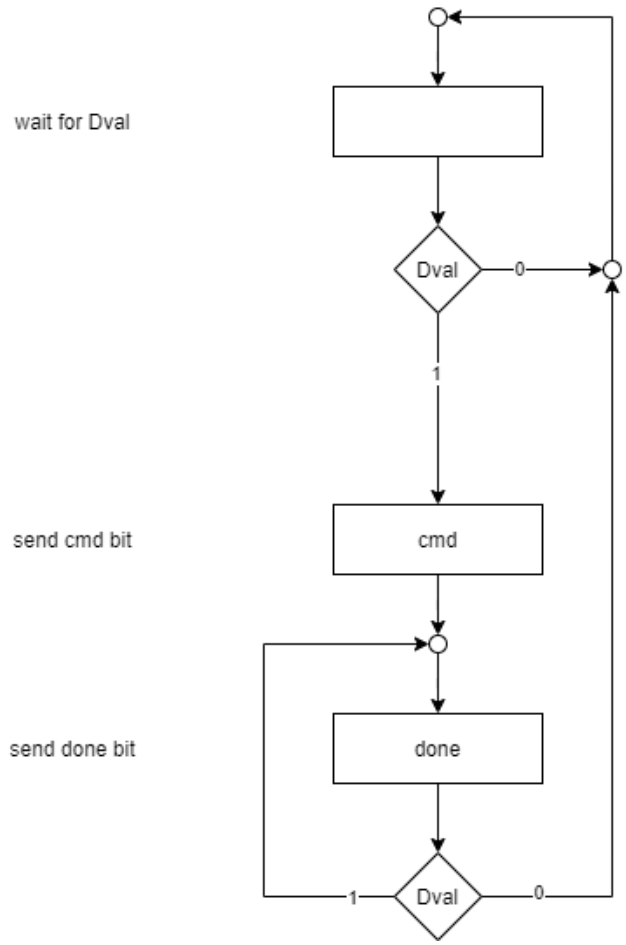


Figura 6 – *ASM-chart* do bloco *Sound Control*

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

"*wait for Dval*" – o próprio nome diz. Espera pelo valor de uma trama válido vindo do *Serial Receiver*

"send cmd bit" – envia um "enable" para o *Decoder* de modo a ativar a seleção de qual o comando a realizar. Sendo que existem só quatro comandos possíveis, um *Decoder 1x4* com os *bits Cmd0..1* seleciona corretamente o comando a processar.

"send done bit" **–** após o processamento do comando enviado para o *Sound Generator*, o *Sound Control* valida o fim da leitura ativando o *bit done* que comunica com o bloco *Serial Receiver* para este mesmo enviar novas tramas.

3 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 2.

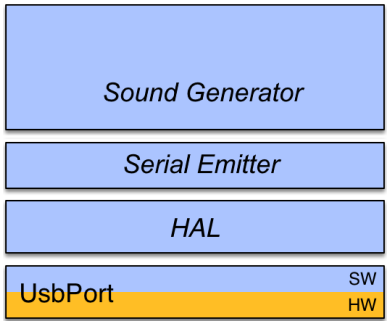


Figura 2 – Diagrama lógico do módulo *Control* de interface com o módulo *Sound Generator*

As classes *Sound Generator* e *HAL* desenvolvidas são descritas nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos E e F, respetivamente.

3.1 Classe *HAL*

Esta classe, resumidamente, traduz a entrada(*UsbPort.in()*) do UsbPort para fácil manuseamento dentro das outras classes desenvolvidas em *software*. Faz também a conversão devida para a saída (*UsbPort.out()*). É composta por métodos como *readBits()* e *isBit()* que leem a entrada do UsbPort e retornam, respetivamente, o valor dos bits a 1 e se o bit estiver a 1 ou a 0. Pertencem também os métodos: *writeBits()*, *setBits()* e *clrBits()*; que servem para manipulação dos bits de saída do *UsbPort*. Estes métodos estão devidamente comentados na secção onde o código-fonte desta classe se encontra. Por fim, *updateOutput()* atualiza UsbPort.out() para o valor atual da saída do dispositivo a ser utilizado (*UsbPort* ou *Uclix*) e *getInput()* atualiza a variável *input* com a entrada do dispositivo a ser utilizado (*UsbPort* ou *Uclix*).

3.2 Classe *Sound Generator*

Esta classe tem 3 métodos principais:

- play(): Seleciona o som e põe-no a tocar.
- stop(): para o som que está a tocar
- setVolume(): define o nível de volume

Estes métodos estão todos acompanhados de uma classe auxiliar, *SGCode* (*Sound Generator Code)*, que elimina qualquer código "*hardcoded"*. Esta classe auxiliar contém todas as possibilidades de instruções do *Sound Generator*, facilitando, assim, a utilização da classe *Sound Generator*.

Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

4 Conclusões

Em *software*: A classe *Sound Generator* sem a *SGCode* não faria sentido pois seria difícil implementar novas instruções.

Em *hardware*: Após o teste em laboratório conclui-se que o *clock* (*MCLK*) do *Sound Control* deve ser o máximo que a *ATB* fornece, assim a informação não se perde devido à velocidade que o *software* processa as tramas (*Serial Emitter)*.

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 3), representado por 'data' no código CUPL, que consiste nas tramas de informação enviadas bit a bit com objetivo de serem entregues ao *Sound Generator* de forma correta;

- $\overline{SS}$ (PIN 4), 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 *Sound Generator* que informa a confirmação do envio da trama que o módulo *Serial Receiver* enviou ao *Sound Controller* e o despachamento da mesma trama tal como o respetivo *enable* para o *Sound Generator*.

- Serial Control:

Uma máquina de estados (respetivo *ASM-chart* na figura 4) que usufrui de 2 flip-flops (MSR0..1) que disponibiliza 4 estados. Tem como objetivo orientar os restantes blocos presentes neste módulo. Como *clock* deste bloco tem-se MCLK (PIN 2), 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 *Sound Controller*.

- Shift Register:

Bloco de registos com *shift right*, possui 5 flip-flops (SR0..3), 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 ímpar não houve erro, caso deem par então houve interferência num *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 4 e 5, "100" e "101" 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 *Sound Controller*

O módulo *Sound Controller* tem como objetivo processar e guardar a última instrução fornecida pelo módulo anterior, *Serial Receiver*. Deste modo é necessário compreender os seguintes aspetos:

- Para usarmos a função de *Play/Stop* é necessário, de alguma forma, guardar essa informação mesmo depois de instanciar a sua instrução. Para isso, um *register* de 1 *bit* é usado para guardar a informação atual dessa tal ação *Play/Stop*. Para haver atualização dessa instrução é necessário utilizar o *clock* do *register*. Um simples *OR* entre as duas possíveis instruções (*play* ou *stop*) resolve este problema.
- Para as restantes situações (mudar o som ou o volume) é de modo semelhante ao do *Play/Stop* mas desta vez, utilizando *registers* de 2 *bits* cada um. O *clock* destes mesmos é atuado a partir do sinal *setsound* ou *setvolume* para o *register* correspondente.

Por fim falta falar sobre as saídas para obtermos os valores de cada instrução. Para isso temos:

- *Play* – PIN 14
- *sid0..1* (*id* do som) – PIN 17 e 18
- *vol0..1* (nível do volume) – PIN 15 e 16

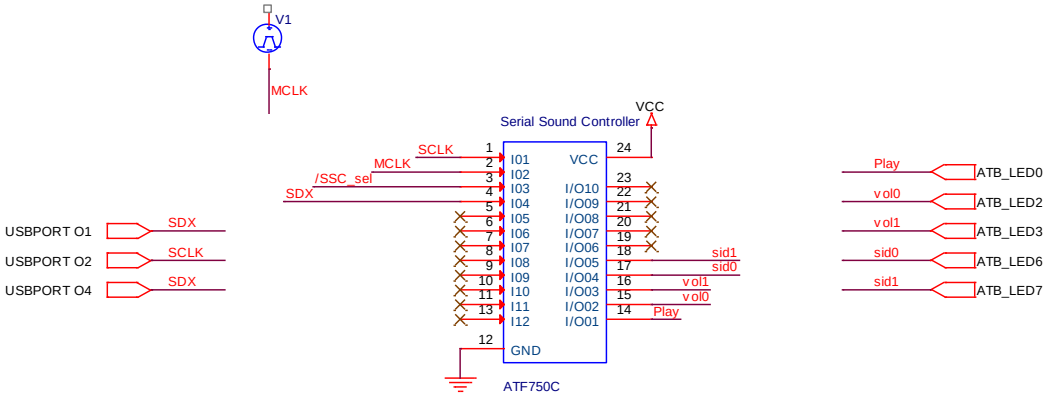
Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

C. Código CUPL do módulo *Serial Sound Controller*

```
Name     SerialSoundController ;
PartNo   00 ;
Date     09/06/2020 ;
Revision 01 ;
Designer JAM ;
Company  ISEL ;
Assembly None ;
Location  ;
Device   v750c ;
/* INPUT PINS */
PIN 1 = SCLK ;
PIN 2 = MCLK ;
PIN 3 = SS ;
PIN 4 = SDX ;
/* OUTPUT PINS */
PIN 14 = RP ;
PIN [15..16] = [RV0..1] ;
PIN [17..18] = [RS0..1] ;
PIN [20..21] = [MSR0..1] ;
PIN [22..23] = [MSC0..1] ;
/* PINNODES */
PINNODE 25 = PB ;
PINNODE 34 = SR0 ;
PINNODE 26 = SR1 ;
PINNODE 33 = SR2 ;
PINNODE 27 = SR3 ;
PINNODE 30 = C0 ;
PINNODE 28 = C1 ;
PINNODE 31 = C2 ;
/* BODY */
/* SERIAL RECEIVER */
accept = done ;
/* Serial Control */
enRX = !SS ;
RXerror = err ;
[MSR0..1].ck = MCLK ;
[MSR0..1].sp = 'b'0 ;
[MSR0..1].ar = 'b'0 ;
SEQUENCE [MSR1, MSR0] {
        PRESENT 0
                OUT init;
                IF !enRX NEXT 0 ;
                DEFAULT NEXT 1 ;
        PRESENT 1
                OUT wr ;
                IF !enRX NEXT 0 ;
                IF enRX & !dFlag NEXT 1 ;
                DEFAULT NEXT 2 ;
        PRESENT 2
                IF !enRX NEXT 0 ;
                IF enRX & !pFlag NEXT 2 ;
                IF enRX & pFlag & RXerror NEXT 0 ;
                DEFAULT NEXT 3 ;
        PRESENT 3
                OUT DXval ;
                IF accept # !enRX NEXT 0 ;
                DEFAULT NEXT 3 ;
}

/* Parity Check */
PB.ckmux = SCLK ;
PB.sp = 'b'0 ;
PB.ar = init ;
PB.t = SDX ;
err = !PB ;
/* Shift Register */
enableShift = wr ;
[SR0..3].ckmux = SCLK ;
[SR0..3].sp = 'b'0 ;
[SR0..3].ar = 'b'0 ;
SR3.d = SDX & enableShift # SR3 & !enableShift;
[SR2..0].d = [SR3..1] & enableShift # [SR2..0] &
!enableShift ;
[Din0..3] = [SR0..3] ;
/* 3 Bit Counter */
clr = init ;
[C0..2].ckmux = SCLK ;
[C0..2].sp = 'b'0 ;
[C0..2].ar = clr ;
C0.t = 'b'1 ;
C1.t = C0 ;
C2.t = C0 & C1 ;
dFlag = C2 & !C1 & !C0 ;
pFlag = C2 & !C1 & C0 ;
/* SOUND CONTROLLER */
Dval = DXval ;
[Cmd0..1] = [Din0..1] ;
[Data0..1] = [Din2..3] ;
/* Sound Control */

[MSC0..1].ck = MCLK ;
[MSC0..1].sp = 'b'0 ;
[MSC0..1].ar = 'b'0 ;

SEQUENCE [MSC1, MSC0] {
        PRESENT 0
                IF Dval NEXT 1 ;
                DEFAULT NEXT 0 ;
        PRESENT 1
                OUT cmd ;
                DEFAULT NEXT 2 ;
        PRESENT 2
                OUT done ;
                IF Dval NEXT 2 ;
                DEFAULT NEXT 0 ;
        PRESENT 3
                DEFAULT NEXT 0 ;
}
/* De-Mux 1x4 */
enableCmd = cmd ;
stop = enableCmd & !Cmd1 & !Cmd0 ;
play = enableCmd & !Cmd1 & Cmd0 ;
set_sound = enableCmd & Cmd1 & !Cmd0 ;
set_volume = enableCmd & Cmd1 & Cmd0 ;
/* 2 Bit Register - Sound */
RSClock = set_sound ;
[RS0..1].ck = RSClock ;
[RS0..1].sp = 'b'0 ;
[RS0..1].ar = 'b'0 ;
[RS0..1].d = [Data0..1] ;
/* 2 Bit Register - Volume */
RVClock = set_volume ;
[RV0..1].ck = RVClock ;
[RV0..1].sp = 'b'0 ;
[RV0..1].ar = 'b'0 ;
[RV0..1].d = [Data0..1] ;
/* 1 Bit Register - Play */
RPClock = play # stop ;
RP.ck = RPClock ;
RP.sp = 'b'0 ;
RP.ar = 'b'0 ;

RP.d = Cmd0 ;
```

Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

D. Esquema elétrico do módulo *Serial Sound Controller*



Title: Serial Sound Controller		
Size: A	Document Number: <Doc>	Rev: <RevCode>
Date: Thursday, July 02, 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 *HAL*

```
package edu.isel.lic.link;

import isel.leic.*;
import isel.leic.utils.Time;

import java.util.Scanner;

public class HAL // Virtualiza o acesso ao sistema UsbPort
{
  public static int input, output;
  public static final int MAX_BITS = 0xFF;
  private static final boolean ULICX = false; // mudar para true caso estiver a ser usado uLICx

  // Inicia a classe
  public static void init() {
    clrBits(MAX_BITS);
  }

  // Retorna true se o bit tiver o valor lógico '1'
  public static boolean isBit(int mask) {
    return (mask == readBits(mask));
  }

  // Retorna os valores dos bits representados por mask presentes no UsbPort
  public static int readBits(int mask) {
    getInput();
    return mask & input;
  }

  // Escreve nos bits representados por mask o valor de value
  public static void writeBits(int mask, int value) {
    output = mask & value | ~mask & output;
    updateOutput();
  }

  // Coloca os bits representados por mask no valor lógico '1'
  public static void setBits(int mask) {
    output = output | mask;
    updateOutput();
  }

  // Coloca os bits representados por mask no valor lógico '0'
  public static void clrBits(int mask) {
    output = output & ~mask;
    updateOutput();
  }

  // Atualiza a saída no UsbPort com o valor da variável output
  private static void updateOutput() {
    UsbPort.out(ULICX ? output : ~output);
  }

  // Atualiza a variável input com a entrada do UsbPort
  private static void getInput() {
    input = (ULICX) ? UsbPort.in() : ~UsbPort.in();
  }
}
```

Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

F. Código Java da classe *Serial Generator*

```
package edu.isel.lic.link.sound;

import edu.isel.lic.link.SerialEmitter;

public class SoundGenerator { // Controla o Sound Generator.

  public static final int SERIAL_DATA_SIZE = 4;

  // Envia comando para reproduzir um som, com a identificação deste
  public static void play(SGCode.Sound sound) {
    SerialEmitter.send(SerialEmitter.Destination.SSC, SERIAL_DATA_SIZE, SGCode.set_sound(sound));
    SerialEmitter.send(SerialEmitter.Destination.SSC, SERIAL_DATA_SIZE, SGCode.play());
  }

  // Envia comando para parar o som
  public static void stop() {
    SerialEmitter.send(SerialEmitter.Destination.SSC, SERIAL_DATA_SIZE, SGCode.stop());
  }

  // Envia comando para definir o volume do som
  public static void setVolume(SGCode.Volume volume) {
    SerialEmitter.send(SerialEmitter.Destination.SSC, SERIAL_DATA_SIZE, SGCode.set_volume(volume));
  }

  // Inicia a classe, estabelecendo os valores iniciais.
  public static void init() {
    stop();
  }
}
```

Autores: João Nunes (A47220@alunos.isel.pt
Alexandre Luís (A47222@alunos.isel.pt
Miguel Marques (a47204@alunos.isel.pt

A. Código Java da classe *SGCode*

```
package edu.isel.lic.link.sound;

public class SGCode
{
  private static int value = 0;

  private static final int
    STOP = 0x0,
    PLAY = 0x1,
    SET_SOUND = 0x2,
    SET_VOLUME = 0x3;

  public enum Sound {
    GAME_OVER(0x1),
    SOUND1(0x0),
    SOUND2(0x2),
    SOUND3(0x3);

    private final int sound;

    Sound(int sound) { this.sound = sound; }

    private int getValue() { return sound; }
  }

  public enum Volume {
    MUTE(0x0),
    LOW(0x1),
    MED(0X2),
    HIGH(0x3);

    private final int volume;

    Volume(int volume) { this.volume = volume; }

    private int getValue() { return volume; }
  }

  public static int stop() {
    return STOP;
  }

  public static int play() {
    return PLAY;
  }

  public static int set_sound (Sound sound) {
    return value =
        SET_SOUND +
        (sound.getValue() << 2);
  }

  public static int set_volume (Volume volume) {

    return value =
        SET_VOLUME +
        (volume.getValue() << 2);
  }

}
```