Documentação do código do projeto Game Genius

LUANA PALMA
DEIVIS STRIEDER
Versão 1.0
Julho de 2016

Game Genius

Autor:
Luana Palma & Deivis Strieder

Versão:
1.0

Data:
Julho de 2016

Lista de Arquivos

GameButtonsGenius.c
GameButtonsGenius.h
InterfaceGenius.c
InterfaceGenius.h
LCDGenius.c
LCDGenius.h
/LEDsGenius.c
LEDsGenius.h
main.c
MemoriaGenius.c
MemoriaGenius.h
TimerAndADCGenius.c
TimerAndADCGenius.h

Arquivos

Referência do Arquivo GameButtonsGenius.c

```
#include <avr/io.h>
#include "GameButtonsGenius.h"
```

Funções

- void **GameButtonsInit** ()
  Função que configura as interrupções ligadas ao botões e as portas as quais estes se ligam.

Funções

void GameButtonsInit ()

```
 9     {
10     BOTOES_CONTROLE = 0x00;     //Configura as interrupções
11     BOTOES_SELECT &= ~(0xFF);
12     /* Seleciona as portas de interrupção */
13     BOTOES_SELECT =
 (1<<BOTAO_Y_PIN)|(1<<BOTAO_B_PIN)|(1<<BOTAO_G_PIN)|(1<<BOTAO_R_PIN);
14 }
```

Referência do Arquivo GameButtonsGenius.h

Definições e Macros

- #define **BOTOES_CONTROLE** EICRA
  Registrador utilizado no controle de interrupções externas.
- #define **BOTOES_SELECT** EIMSK
  Registrador que seleciona quais interrupções externas serão ativadas.
- #define **BOTAO_Y_PIN** INT0
  Porta do processador utilizada na interrupção do botão amarelo.
- #define **BOTAO_B_PIN** INT1
  Porta do processador utilizada na interrupção do botão azul.
- #define **BOTAO_G_PIN** INT2
  Porta do processador utilizada na interrupção do botão verde.
- #define **BOTAO_R_PIN** INT3
  Porta do processador utilizada na interrupção do botão vermelho.
- #define **BOTAO_Y_FUNCT** INT0_vect
  Redefinição do nome da função de interrupção para a porta do botão amarelo.
- #define **BOTAO_B_FUNCT** INT1_vect
  Redefinição do nome da função de interrupção para a porta do botão azul.
- #define **BOTAO_G_FUNCT** INT2_vect
  Redefinição do nome da função de interrupção para a porta do botão verde.
- #define **BOTAO_R_FUNCT** INT3_vect
  Redefinição do nome da função de interrupção para a porta do botão vermelho.

Funções

- void **GameButtonsInit** ()
  Função que configura as interrupções ligadas ao botões e as portas as quais estes se ligam.

Referência do Arquivo InterfaceGenius.c

```
#include <avr/io.h>
#include <util/delay.h>
#include "LCDGenius.h"
```

Definições e Macros

- #define **F_CPU** 16000000UL

Funções

- void **LimpaTela** ()
  Função auxiliar que limpa a tela do display e posiciona o cursor na primeira posição da primeira linha.
- void **ExibeMenuPrincipal** ()
  Função que exibe a tela do menu principal no display.
- void **ExibeJogar** ()
  Função que exibe a tela de confirmação de jogo no display.
- void **ExibeGameOver** (uint8_t pontuacao)
  Função que exibe a tela de fim de jogo no display.
- void **ExibeGenius** ()
  Função que exibe a tela de preparação de jogo no display.
- void **ExibeTelaName** ()
  Função que exibe o layout inicial da tela de inserção de nome no display.
- void **ExibeTelaRanking** ()
  Função que exibe layout inicial do Ranking no display.

Definições e macros

#define F_CPU 16000000UL

Funções

void LimpaTela ()

```
14                {
15     lcd_write_instruction(lcd_Clear);
16     _delay_us(2000);
17     lcd_write_instruction(lcd_SetCursor|lcd_LineOne);
18      delay us(80);
19 }
```

void ExibeMenuPrincipal ()

```
21                          {
22     uint8_t menu[] = "       Menu";
23     uint8_t playRanking[] = "<-Play Ranking->";
24     LimpaTela();
25     lcd_write_string(menu);
26     lcd_write_instruction(lcd_SetCursor|lcd_LineTwo);
27      delay us(80);
28     lcd write string(playRanking);
29 }
```

void ExibeJogar ()

```
   32                 {
   33     uint8_t play[] = "     Play?";
   34     uint8_t yesNo[] = "<-Yes       No->";
   35     LimpaTela();
   36     lcd_write_string(play);
   37     lcd write instruction(lcd SetCursor|lcd LineTwo);
   38     _delay_us(80);
   39     lcd_write_string(yesNo);
   40 }
```

void ExibeGameOver (uint8_t *pontuacao*)

Parâmetros:

pontuacao	Pontuação final do jogador a ser exibida na tela.

```
   44                                    {
   45     uint8_t gameOver[] = "   Game Over";
   46     uint8_t strScore[] = "  Score - ";
   47     uint8_t dig1Score = (pontuacao/100);                              //Separa o
algarismo de centena da pontuação.
   48     uint8_t dig2Score = ((pontuacao-(dig1Score*100))/10);             //Separa o
algarismo de dezena da pontuação.
   49     uint8_t dig3Score = (pontuacao-(dig1Score*100)-(dig2Score*10)); //Separa o
algarismo de unidade da pontuação.
   50     dig1Score+=48;                                                    //O offset de
um número em relação ao valor do seu
   51     dig2Score+=48;                                                    //caractere em
ASCII é 48.
   52     dig3Score+=48;
   53     LimpaTela();
   54     lcd_write_string(gameOver);
   55     lcd_write_instruction(lcd_SetCursor|lcd_LineTwo);
   56      delay us(80);
   57     lcd write string(strScore);
   58     lcd_write_character(dig1Score);
   59     lcd_write_character(dig2Score);
   60     lcd write character(dig3Score);
   61 }
```

void ExibeGenius ()

```
   63                    {
   64     uint8 t genius[] = "     Genius";
   65     uint8 t tres[] = " 3"; //1
   66     uint8_t dois[] = "   2";//3
   67     uint8_t um[] = "   1";//3
   68     uint8_t go[] = "   GO!";//2
   69     LimpaTela();
   70     lcd_write_string(genius);
   71     lcd write instruction(lcd SetCursor|lcd LineTwo);   //Os delays entre escritas
servem
   72     _delay_ms(800);                                     //para o jogador ter a
sensação
   73     lcd write string(tres);                             //de uma contagem
regressiva.
   74      delay ms(800);
   75     lcd_write_string(dois);
   76     _delay_ms(800);
   77     lcd_write_string(um);
   78      delay ms(800);
   79     lcd_write_string(go);
   80 }
```

void ExibeTelaName ()

```
   82                     {
   83     uint8 t nameMenu[] = "      Name";
   84     uint8_t endName[] = "        [end]";
```

```
85      LimpaTela();
86      lcd_write_string(nameMenu);
87      lcd_write_instruction(lcd_SetCursor|lcd_LineTwo);
88      _delay_us(80);
89      lcd write string(endName);
90      lcd_write_instruction(lcd_SetCursor|lcd_LineTwo+2);
91      _delay_us(80);
92  }
```

void ExibeTelaRanking ()

```
94                          {
95      uint8_t rankNameScore[] = "Rank Name Score";
96      LimpaTela();
97      lcd write string(rankNameScore);
98  }
```

Referência do Arquivo InterfaceGenius.h

Funções

- void **ExibeMenuPrincipal** ()
  Função que exibe a tela do menu principal no display.
- void **ExibeJogar** ()
  Função que exibe a tela de confirmação de jogo no display.
- void **ExibeGenius** ()
  Função que exibe a tela de preparação de jogo no display.
- void **ExibeGameOver** (uint8_t *pontuacao*)
  Função que exibe a tela de fim de jogo no display.
- void **ExibeTelaName** ()
  Função que exibe o layout inicial da tela de inserção de nome no display.
- void **ExibeTelaRanking** ()
  Função que exibe layout inicial do Ranking no display.

Referência do Arquivo LCDGenius.c

```
#include <avr/io.h>
#include <util/delay.h>
#include "LCDGenius.h"
```

Definições e Macros

- #define **F_CPU** 16000000UL

Funções

- void **lcd_init** (void)
  Função que faz a configuração inicial do LCD.
- void **lcd_write_string** (uint8_t theString[])
  Função que escreve uma string na tela do LCD.
- void **lcd_write_character** (uint8_t theData)
  Função que escreve um caractere na tela do LCD.
- void **lcd_write_instruction** (uint8_t theInstruction)

Função que envia uma instrução para o LCD.

- void **lcd_write** (uint8_t theByte)
  Função que escreve 4 bits nos pinos do LCD (D7,D6,D5 e D4).

Definições e macros

#define F_CPU 16000000UL

Funções

void lcd_init (void)

```
   11 {
   12     /* Configuração dos pinos do processador que o LCD utiliza como saída. */
   13     lcd_D7_ddr |= (1<<lcd_D7_bit);
   14     lcd_D6_ddr |= (1<<lcd_D6_bit);
   15     lcd_D5_ddr |= (1<<lcd_D5_bit);
   16     lcd_D4_ddr |= (1<<lcd_D4_bit);
   17     lcd_E_ddr |= (1<<lcd_E_bit);
   18     lcd_RS_ddr |= (1<<lcd_RS_bit);
   19
   20     _delay_ms(100);                                  //Delay inicial de 100ms.
   21
   22     lcd_RS_port &= ~(1<<lcd_RS_bit);                 //Seleciona o Instruction
Register.
   23     lcd_E_port &= ~(1<<lcd_E_bit);                   //Coloca E em nível baixo.
   24
   25     /* Início da sequência de resets do LCD. */
   26     lcd_write(lcd_FunctionReset);
   27     _delay_ms(10);
   28
   29     lcd_write(lcd_FunctionReset);
   30     _delay_us(200);
   31
   32     lcd_write(lcd_FunctionReset);
   33     _delay_us(200);
   34     /* Fim da sequência de resets. */
   35
   36     lcd_write(lcd_FunctionSet4bit);                  //Configura modo de 4 bits.
   37      delay us(80);
   38
   39     lcd_write_instruction(lcd_FunctionSet4bit);      //Configura modo, linhas e
fonte.
   40      delay us(80);
   41
   42     lcd_write_instruction(lcd_DisplayOff);           //Desliga o display.
   43     _delay_us(80);
   44
   45     lcd_write_instruction(lcd_Clear);                //Limpa a RAM do display.
   46      delay ms(4);
   47
   48     lcd_write_instruction(lcd_EntryMode);            //Configura o modo de
deslocamento.
   49     _delay_us(80);
   50
   51     lcd_write_instruction(lcd_DisplayOn);            //Liga o display.
   52      delay us(80);
   53 }
```

void lcd_write_string (uint8_t *theString*[])

Parâmetros:

theString[]	Array de caracteres a serem escritos no display.

```
   58 {
   59     volatile int i = 0;
   60     while (theString[i] != 0)   //Loop que escreve a string caractere por caractere
até que chegue ao fim desta.
   61     {
   62         lcd_write_character(theString[i]);
   63         i++;
   64          delay_us(80);
   65     }
   66 }
```

void lcd_write_character (uint8_t *theData*)

Parâmetros:

theData	Caractere a ser escrito no display.

```
   71 {
   72     lcd_RS_port |= (1<<lcd_RS_bit);                 //Seleciona o Data Register.
   73     lcd_E_port &= ~(1<<lcd_E_bit);                  //Coloca Enable em nível baixo.
   74     lcd_write(theData);                             //Escreve a parte alta (4 bits)
do caractere.
   75     lcd_write(theData << 4);                       //Escreve a parte baixa (4 bits)
do caractere.
   76 }
```

void lcd_write_instruction (uint8_t *theInstruction*)

Parâmetros:

theInstruction	Instrução a ser enviada para o LCD.

```
   81 {
   82     lcd_RS_port &= ~(1<<lcd_RS_bit);                //Seleciona o Instruction
Register.
   83     lcd_E_port &= ~(1<<lcd_E_bit);                  //Coloca Enable em nível baixo.
   84     lcd_write(theInstruction);                      //Escreve a parte alta (4 bits)
da instrução.
   85     lcd_write(theInstruction << 4);                //Escreve a parte baixa (4 bits)
da instrução.
   86 }
```

void lcd_write (uint8_t *theByte*)

Parâmetros:

theByte	Dado a ser escrito nas portas do LCD. Somente serão utilizados os 4 bits mais significativos.

```
   91 {
   92     lcd_D7_port &= ~(1<<lcd_D7_bit);                      //Assume o dado é '0'.
   93     if (theByte & 1<<7) lcd_D7_port |= (1<<lcd_D7_bit);   //Em caso negativo, seta
o bit para '1'.
   94
   95     lcd_D6_port &= ~(1<<lcd_D6_bit);                      //Processo repetido para
os outros bits.
   96     if (theByte & 1<<6) lcd_D6_port |= (1<<lcd_D6_bit);
   97
   98     lcd_D5_port &= ~(1<<lcd_D5_bit);
   99     if (theByte & 1<<5) lcd_D5_port |= (1<<lcd_D5_bit);
  100
  101     lcd_D4_port &= ~(1<<lcd_D4_bit);
  102     if (theByte & 1<<4) lcd_D4_port |= (1<<lcd_D4_bit);
  103
  104     /* Este processo dá um pulso em Enable e escreve os 4 bits no LCD. */
```

```
105     lcd_E_port &= ~(1<<lcd_E_bit);
106     _delay_us(1);
107     lcd_E_port |= (1<<lcd_E_bit);
108     _delay_us(1);
109     lcd E port &= ~(1<<lcd E bit);
110     _delay_us(100);
111 }
```

Referência do Arquivo Genius/Genius/Genius/LCDGenius.h

Definições e Macros

- #define **lcd_D7_port** PORTH
  Definição da porta utilizada pelo pino D7.
- #define **lcd_D7_bit** DDH4
  Definição do bit que o pino D7 ocupa na porta declarada.
- #define **lcd_D7_ddr** DDRH
  Registrador de controle da porta do pino D7.
- #define **lcd_D6_port** PORTH
  Definição da porta utilizada pelo pino D6.
- #define **lcd_D6_bit** DDH3
  Definição do bit que o pino D6 ocupa na porta declarada.
- #define **lcd_D6_ddr** DDRH
  Registrador de controle da porta do pino D6.
- #define **lcd_D5_port** PORTE
  Definição da porta utilizada pelo pino D5.
- #define **lcd_D5_bit** DDE3
  Definição do bit que o pino D5 ocupa na porta declarada.
- #define **lcd_D5_ddr** DDRE
  Registrador de controle da porta do pino D5.
- #define **lcd_D4_port** PORTG
  Definição da porta utilizada pelo pino D4.
- #define **lcd_D4_bit** DDG5
  Definição do bit que o pino D4 ocupa na porta declarada.
- #define **lcd_D4_ddr** DDRG
  Registrador de controle da porta do pino D4.
- #define **lcd_E_port** PORTH
  Definição da porta utilizada pelo pino Enable.
- #define **lcd_E_bit** DDH6
  Definição do bit que o pino Enable ocupa na porta declarada.
- #define **lcd_E_ddr** DDRH
  Registrador de controle da porta do pino Enable.
- #define **lcd_RS_port** PORTH
- #define **lcd_RS_bit** DDH5
  Definição do bit que o pino Register Select ocupa na porta declarada.
- #define **lcd_RS_ddr** DDRH

Registrador de controle da porta do pino Register Select.

- #define **lcd_LineOne** 0x00
  Definição do offset relativo ao início da primeira linha do display.
- #define **lcd_LineTwo** 0x40
  Definição do offset relativo ao início da segunda linha do display.
- #define **lcd_Clear** 0b00000001
  Instrução que substitui todos os caracteres por 'espaços' ASCII.
- #define **lcd_Home** 0b00000010
  Instrução que retorna o cursor para a primeira posição da primeira linha.
- #define **lcd_EntryMode** 0b00000110
  Instrução que move o cursor da esquerda para a direita quando há leitura/escrita.
- #define **lcd_DisplayOff** 0b00001000
  Instrução que desliga o display.
- #define **lcd_DisplayOn** 0b00001100
  Instrução que configura o display ligado, cursor desligado e caractere estático.
- #define **lcd_FunctionReset** 0b00110000
  Instrução que reseta o LCD.
- #define **lcd_FunctionSet4bit** 0b00101000
  Instrução que configura o LCD para receber dados de 4 bits, exibir fonte 5x7 e saber que possui duas linhas.
- #define **lcd_SetCursor** 0b10000000
  Instrução que seta a posição do cursor.

Funções

- void **lcd_write** (uint8_t)
  Função que escreve 4 bits nos pinos do LCD (D7,D6,D5 e D4).
- void **lcd_write_instruction** (uint8_t)
  Função que envia uma instrução para o LCD.
- void **lcd_write_character** (uint8_t)
  Função que escreve um caractere na tela do LCD.
- void **lcd_write_string** (uint8_t *)
  Função que escreve uma string na tela do LCD.
- void **lcd_init** (void)
  Função que faz a configuração inicial do LCD.

Referência do Arquivo LEDsGenius.c

```
#include <util/delay.h>
#include <avr/io.h>
#include "LEDsGenius.h"
```

Definições e Macros

- #define **F_CPU** 16000000UL

Funções

- void **YBlink** ()

Função que pisca o LED amarelo.

- void **BBlink** ()
  Função que pisca o LED azul.
- void **GBlink** ()
  Função que pisca o LED verde.
- void **RBlink** ()
  Função que pisca o LED vermelho.
- void **LEDsInit** ()
  Função que configura os pinos utilizados pelos LEDs.
- void **PiscaLeds** ()
  Função que pisca os 4 LEDs ao mesmo tempo.

Definições e macros

#define F_CPU 16000000UL

Funções

void YBlink ()

```
   11 {
   12     PORTA_LEDS |=(1<<LED_Y_PIN);    //Acende
   13     _delay_ms(DELAY_BLINK);
   14     PORTA_LEDS &=~(1<<LED_Y_PIN);   //Apaga
   15 }
```

void BBlink ()

```
   18 {
   19     PORTA_LEDS |=(1<<LED_B_PIN);    //Acende
   20     _delay_ms(DELAY_BLINK);
   21     PORTA_LEDS &=~(1<<LED_B_PIN);   //Apaga
   22 }
```

void GBlink ()

```
   24 {
   25     PORTA_LEDS |=(1<<LED_G_PIN);    //Acende
   26     _delay_ms(DELAY_BLINK);
   27     PORTA_LEDS &=~(1<<LED_G_PIN);   //Apaga
   28 }
```

void RBlink ()

```
   30 {
   31     PORTA_LEDS |=(1<<LED_R_PIN);    //Acende
   32     _delay_ms(DELAY_BLINK);
   33     PORTA_LEDS &=~(1<<LED_R_PIN);   //Apaga
   34 }
```

void LEDsInit ()

```
   36 {
   37     /* Configura como saída os pinos utilizados pelos LEDs. */
   38     PORTA_LEDS_CONTROLE =
(1<<LED_Y_PIN)|(1<<LED_B_PIN)|(1<<LED_G_PIN)|(1<<LED_R_PIN);
   39 }
```

void PiscaLeds ()

```
41 {
42     uint8_t i;
43     for(i = 0; i < QUANT_PISCA; i++){
44         /* Acende todos os LEDs. */
45         PORTA_LEDS |=(1<<LED_Y_PIN);
46         PORTA_LEDS |=(1<<LED_B_PIN);
47         PORTA_LEDS |=(1<<LED_G_PIN);
48         PORTA_LEDS |=(1<<LED_R_PIN);
49         _delay_ms(DELAY_PISCA);
50         /* Apaga todos os LEDs. */
51         PORTA_LEDS &=~(1<<LED_Y_PIN);
52         PORTA_LEDS &=~(1<<LED_B_PIN);
53         PORTA_LEDS &=~(1<<LED_G_PIN);
54         PORTA_LEDS &=~(1<<LED_R_PIN);
55         _delay_ms(DELAY_PISCA);
56     }
57 }
```

Referência do Arquivo LEDsGenius.h

Definições e Macros

- #define **PORTA_LEDS** PORTK
  Definição da porta do processador utilizada pelos LEDs.
- #define **PORTA_LEDS_CONTROLE** DDRK
  Registrador de controle da porta utilizada pelos LEDs.
- #define **LED_Y_PIN** DDK7
  Definição do pino da porta utilizado pelo LED amarelo.
- #define **LED_B_PIN** DDK6
  Definição do pino da porta utilizado pelo LED azul.
- #define **LED_G_PIN** DDK5
  Definição do pino da porta utilizado pelo LED verde.
- #define **LED_R_PIN** DDK4
  Definição do pino da porta utilizado pelo LED vermelho.
- #define **DELAY_BLINK** 700
  Definição do tempo, em ms, que um LED deve piscar durante o jogo.
- #define **DELAY_PISCA** 500
  Definição do tempo, em ms, que os LEDs devem piscar quando chamada a função PiscaLeds.
- #define **QUANT_PISCA** 3
  Quantidade de vezes que os LEDs devem piscar quando chamada a função PiscaLeds.

Funções

- void **PiscaLeds** ()
  Função que pisca os 4 LEDs ao mesmo tempo.
- void **YBlink** ()
  Função que pisca o LED amarelo.
- void **BBlink** ()
  Função que pisca o LED azul.

- void **GBlink** ()
  Função que pisca o LED verde.
- void **RBlink** ()
  Função que pisca o LED vermelho.
- void **LEDsInit** ()
  Função que configura os pinos utilizados pelos LEDs.

Referência do Arquivo main.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdbool.h>
#include <stdlib.h>
#include "LCDGenius.h"
#include "LEDsGenius.h"
#include "InterfaceGenius.h"
#include "GameButtonsGenius.h"
#include "TimerAndADCGenius.h"
#include "MemoriaGenius.h"
```

Estruturas de Dados

- struct **Posicao**
  Tipo de estrutura que comporta as informações de uma posição de ranking.
- struct **StateMachine**
  Estrutura que representa a máquina de estados do jogo.

Definições e Macros

- #define **F_CPU** 16000000UL
  Definição da frequência da CPU para utilização da biblioteca delay.h.
- #define **MAX_SEQ** 100
  Tamanho máximo de elementos na sequência de luzes.
- #define **Y_NUM** 1
  Definição do número correspondente à cor amarela.
- #define **B_NUM** 2
  Definição do número correspondente à cor azul.
- #define **G_NUM** 3
  Definição do número correspondente à cor verde.
- #define **R_NUM** 4
  Definição do número correspondente à cor vermelha.
- #define **POSICOES_RANKING** 10
  Definição do número total de posições no ranking.
- #define **TAM_LINHA** 4
  Definição da quantidade de posições da memória ocupadas por uma linha do ranking.
- #define **ULTIMA_LINHA** (**POSICOES_RANKING**-1)***TAM_LINHA**
  Definição do endereço inicial na memória da última linha de ranking.
- #define **A** 65

Definição do valor da letra A referente à tabela ASCII.

- #define **Z** 90
  Definição do valor da letra Z referente à tabela ASCII.
- #define **PRIM_CARACT_NOME** 3
  Definição do espaçamento inicial do nome no estado Insere Nome.

Definições de Tipos

- typedef void(* **Action**) (void)
  Definição do tipo ponteiro de função.
- typedef struct **Posicao pos**
  Tipo de estrutura que comporta as informações de uma posição de ranking.

Enumerações

- enum **States** { **sMenuInicial** = 0, **sRanking**, **sInsereNome**, **sConfirmaJogo**, **sPlayGenius**, **sGameOver** }*Tipo utilizado para definir o nome dos estados da FSM.*

Funções

- void **InitStateMachine** ()
  Função que inicializa a máquina de estados configurando os ponteiros de função e o estado inicial.
- void **IniciaGenius** ()
  Função que inicializa os periféricos e portas.
- void **PrintLinha** (**pos** pPosicao, uint8_t ind)
  Função que exibe uma linha de ranking no display.
- bool **InsereElemento** ()
  Função insere um elemento de luz na sequência de jogo.
- void **MostraSequencia** ()
  Função que exibe a sequência de luzes nos LEDs.
- void **MenuInicial** ()
  Função para o estado Menu Inicial.
- void **Ranking** ()
  Função para o estado Ranking.
- void **InsereNome** ()
  Função para o estado Insere Nome.
- void **ConfirmaJogo** ()
  Função para o estado Confirma Jogo.
- void **PlayGenius** ()
  Função para o estado Play Genius.
- void **GameOver** ()
  Função para o estado Game Over.
- int **main** (void)
  Função principal.
- **ISR** (**BOTAO_Y_FUNCT**)
  Função do vetor de interrupções para o botão amarelo.
- **ISR** (**BOTAO_B_FUNCT**)
  Função do vetor de interrupções para o botão azul.
- **ISR** (**BOTAO_G_FUNCT**)

Função do vetor de interrupções para o botão verde.

- **ISR** (**BOTAO_R_FUNCT**)
  Função do vetor de interrupções para o botão vermelho.
- **ISR** (**TIMER_RAND_FUNCT**)
  Função do vetor de interrupções para o timer de overflow.

Variáveis

- struct **StateMachine sm**
- uint8_t **name** [3]
  Variável que guarda nome do jogador.
- uint8_t **score** = 0
  Variável que guarda a pontuação do jogador.
- uint8_t **sequencia** [**MAX_SEQ**]
  Variável que guarda a sequência de luzes durante o jogo.
- uint8_t **indexSequencia** = 0
  Variável que guarda a posição da sequência a ser verificada.
- uint8_t **numBotaoPressionado** = 0
  Variável que o número do botão de jogo pressionado.
- bool **botaoPressionado** = false
  Flag que indica se ouve uma interrupção causada por um botão de jogo.

Estruturas

struct Posicao

Campos de Dados:

uint8_t	pName[3]	Campo utilizado para o nome.
uint8_t	pScore	Campo utilizado para a pontuação.

struct StateMachine

Campos de Dados:

States	state	Variável que guarda o estado atual da FSM.
Action	action[6]	Vetor de ponteiros de função que recebem os ponteiros das funções referentes aos estados.

Enumerações

enum States

Valores de enumerações

sMenuInicial
sRanking
sInsereNome

sConfirmaJogo
sPlayGenius
sGameOver

```
 95             {
 96     sMenuInicial = 0, sRanking, sInsereNome, sConfirmaJogo, sPlayGenius, sGameOver
 97 }States;
```

Funções

void InitStateMachine ()

```
232 {
233     sm.state = sMenuInicial;
234     sm.action[sMenuInicial] = MenuInicial;
235     sm.action[sRanking] = Ranking;
236     sm.action[sInsereNome] = InsereNome;
237     sm.action[sConfirmaJogo] = ConfirmaJogo;
238     sm.action[sPlayGenius] = PlayGenius;
239     sm.action[sGameOver] = GameOver;
240 }
```

void IniciaGenius ()

```
203 {
204
205     uint16_t i, endLinha;
206     uint8_t nomeNulo[3] = {'X','X','X'};    //Valor "nulo" para todos os nomes.
207
208     /* Configuração do display LCD. */
209     lcd_init();
210
211     /* Configuração do timer utilizado para gerar um número aleatório. */
212     TimerInit();
213
214     /* Configuração do conversor A/D utilizado nos botões de menu. */
215     ADCInit();
216
217     /* Configuração das interrupções ligadas aos botões de jogo. */
218     GameButtonsInit();
219
220     /* Configuração das portas que acionam os leds. */
221     LEDsInit();
222
223     /* Escreve ranking nulo na memória. */
224     for(i=0;i<POSICOES RANKING;i++)
225     {
226         endLinha = i*TAM LINHA;
227         WriteLine(END_INICIAL+endLinha,nomeNulo,0);
228     }
229 }
```

void PrintLinha (pos *pPosicao*, uint8_t *ind*)

```
 483 {
 484     uint8_t dig1Score = (pPosicao.pScore/100);
//Separa o algarismo de centena do score.
 485     uint8_t dig2Score = ((pPosicao.pScore-(dig1Score*100))/10);
//Separa o algarismo de dezena do score.
 486     uint8_t dig3Score = (pPosicao.pScore-(dig1Score*100)-(dig2Score*10));
//Separa o algarismo de unidade do score.
 487     uint8_t dig1Pos = (ind/100);
//Separa o algarismo de centena do número de posição.
 488     uint8_t dig2Pos = ((ind-(dig1Pos*100))/10);
//Separa o algarismo de dezena do número de posição.
 489     uint8_t dig3Pos = (ind-(dig1Pos*100)-(dig2Pos*10));
//Separa o algarismo de unidade do número de posição.
```

```
  490     dig1Score+=48;                                                    //O
offset de um número em relação ao valor do seu
  491     dig2Score+=48;
//caractere em ASCII é 48.
  492     dig3Score+=48;
  493     dig2Pos+=48;
  494     dig3Pos+=48;
  495     lcd_write_instruction(lcd_SetCursor|(lcd_LineTwo + 1));
//Mostra o número de posição na segunda
  496     _delay_us(80);
//linha do display utilizando dois algarismos.
  497     lcd_write_character(dig2Pos);
  498     _delay_us(80);
  499     lcd_write_character(dig3Pos);
  500      delay us(80);
  501     lcd_write_instruction(lcd_SetCursor|(lcd_LineTwo + 6));
//Mostra os três caracteres do nome.
  502     _delay_us(80);
  503     lcd_write_character(pPosicao.pName[0]);
  504     _delay_us(80);
  505     lcd write character(pPosicao.pName[1]);
  506     _delay_us(80);
  507     lcd write character(pPosicao.pName[2]);
  508     _delay_us(80);
  509     lcd_write_instruction(lcd_SetCursor|(lcd_LineTwo + 11));
//Mostra o score com 3 algarismo.
  510      delay us(80);
  511     lcd write character(dig1Score);
  512     _delay_us(80);
  513     lcd_write_character(dig2Score);
  514     _delay_us(80);
  515     lcd write character(dig3Score);
  516     _delay_us(80);
  517 }
```

bool InsereElemento ()

Retorna:

Flag que sinaliza se a sequencia foi preenchida com sucesso ou não.

```
  524 {
  525     uint8_t novo = ((TIMER_RAND+rand()+(rand()%MAX_SEQ))%4)+1;  //Captura do valor do
timer e geração do valor de luz.
  526     if(score >= MAX_SEQ)                                        //Caso para a
sequencia completamente preenchida.
  527         return false;
  528     sequencia[score] = novo;                                    //Insere o novo
elemento na sequência.
  529     score++;                                                    //Incrementa o
score.
  530     return true;
  531 }
```

void MostraSequencia ()

```
  535 {
  536     uint8_t i;
  537     for(i=0;i<score;i++)            //Loop que faz as luzes da sequência piscarem
  538     {                               //uma de cada vez.
  539         switch(sequencia[i])
  540         {
  541             case Y_NUM:
  542                 _delay_ms(800);
  543                 YBlink();
  544                 break;
  545             case B_NUM:
  546                 _delay_ms(800);
  547                 BBlink();
  548                 break;
```

```
549             case G_NUM:
550                 _delay_ms(800);
551                 GBlink();
552                 break;
553             case R_NUM:
554                 _delay_ms(800);
555                 RBlink();
556                 break;
557             default:
558                 PiscaLeds();
559                 break;
560         }
561     }
562 }
```

void MenuInicial ()

```
244 {
245     uint16_t opcao;
246     ExibeMenuPrincipal();               //Mostra a tela do menu inicial.
247     opcao = lerBotaoMenu();             //Tratamento do debounce inicial.
248     _delay_ms(500);                     //
249     while(sm.state==sMenuInicial)       //Polling para os botões de menu.
250     {
251         opcao = lerBotaoMenu();         //Leitura da porta analógica.
252          delay ms(150);                 //Debounce.
253         if(opcao==LEFT)
254         {
255             sm.state = sInsereNome;     //Mudança de estado -> Insere Nome
256         }
257         else if(opcao==RIGHT)
258         {
259             sm.state = sRanking;        //Mudança de estado -> Ranking
260         }
261     }
262 }
```

void Ranking ()

```
265 {
266     uint8_t i, endLinha;
267     uint16_t opcao;
268     pos posicoes[POSICOES RANKING];             //Vetor que manipula as posições.
269     ExibeTelaRanking();                         //Mostra a tela inicial do ranking.
270     for(i=0;i<POSICOES RANKING;i++)             //Laço que lê todas posições
271     {                                           //do ranking da memória.
272         endLinha = i*TAM_LINHA;
273         ReadLine((END_INICIAL+endLinha),&posicoes[i]);
274          delay us(10);
275     }
276     i = 0;
277     PrintLinha(posicoes[i],i+1);                //Exibe a primeira posição no
display.
278     opcao = lerBotaoMenu();                     //Debounce inicial da entrada
analógica.
279      delay ms(500);                             //
280     while(sm.state==sRanking)                   //Polling para os botões de menu.
281     {
282         opcao = lerBotaoMenu();
283          delay ms(150);
284         if((opcao==UP) & (i>0))                 //Exibe posição acima.
285         {
286             i--;
287             PrintLinha(posicoes[i],i+1);
288         }
289         if((opcao==DOWN) & (i<POSICOES RANKING-1))
290         {
291             i++;
292             PrintLinha(posicoes[i],i+1);        //Exibe posição abaixo.
293         }
294         else if(opcao==CANCEL)
```

```
295         {
296             sm.state = sMenuInicial;            //Mudança de estado -> Menu Inicial
297         }
298     }
299 }
```

void InsereNome ()

```
302 {
303     uint16_t opcao;
304     uint8_t indName = 0;
305     uint8_t letra = A;
306     bool primeiraMudanca = true;
307     ExibeTelaName();                            //Mostra a tela de inserção de nome.
308     opcao = lerBotaoMenu();                     //Debounce inicial da entrada
analógica.
309     _delay_ms(500);                             //
310     while(sm.state==sInsereNome)                //Polling para os botões de menu.
311     {
312         opcao = lerBotaoMenu();
313         _delay_ms(150);
314         if(opcao!=NONE)
315         {
316             if(opcao==RIGHT)                    //Confirmação da letra escolhida.
317             {
318                 if(primeiraMudanca)
319                     letra=A;
320                 name[indName]=(char)letra;
321
lcd_write_instruction((lcd_SetCursor)|(lcd_LineTwo+PRIM_CARACT_NOME+indName));
322                 _delay_us(80);
323                 lcd write character(letra);
324                 _delay_us(80);
325                 indName++;
326                 primeiraMudanca = true;
327                 if(indName==3){
328                     _delay_ms(1000);
329                     sm.state = sConfirmaJogo;   //Mudança de estado -> Confirma Jogo
330                 }
331             }
332             else if(opcao==UP)                  //Navega pelo alfabeto em direção
crescente.
333             {
334                 if((letra==Z) | (primeiraMudanca))
335                 {
336                     letra=A;
337                     primeiraMudanca = false;
338                 }
339                 else letra++;
340
341
lcd_write_instruction((lcd_SetCursor)|(lcd_LineTwo+PRIM_CARACT_NOME+indName));
342                 _delay_us(80);
343                 lcd_write_character(letra);
344                  delay us(80);
345             }
346             else if(opcao==DOWN)                //Navega pelo alfabeto em direção
descrescente.
347             {
348                 if((letra==A) | (primeiraMudanca))
349                 {
350                     letra=Z;
351                     primeiraMudanca = false;
352                 }
353                 else letra--;
354
355
lcd write instruction((lcd SetCursor)|(lcd LineTwo+PRIM CARACT NOME+indName));
356                 _delay_us(80);
357                 lcd_write_character(letra);
358                 _delay_us(80);
```

```
  359             }
  360             else if(opcao==CANCEL)
  361             {
  362                 sm.state = sMenuInicial;        //Mudança de estado -> Menu Inicial
  363             }
  364         }
  365     }
  366 }
```

void ConfirmaJogo ()

```
  369 {
  370     uint16_t opcao;
  371     ExibeJogar();                           //Mostra a tela para confirmação de jogo.
  372     opcao = lerBotaoMenu();                 //Debounce inicial da entrada analógica.
  373      delay ms(500);                         //
  374     sm.state = sConfirmaJogo;
  375     while(sm.state==sConfirmaJogo)          //Polling para os botões de menu.
  376     {
  377         opcao = lerBotaoMenu();
  378         _delay_ms(150);
  379         if(opcao==LEFT)
  380         {
  381             sm.state = sPlayGenius;         //Mudança de estado -> Play Genius
  382         }
  383         else if((opcao==RIGHT)|(opcao==CANCEL))
  384         {
  385             sm.state = sMenuInicial;        //Mudança de estado -> Menu Inicial
  386         }
  387         else
  388         {
  389             sm.state = sConfirmaJogo;
  390         }
  391     }
  392 }
```

void PlayGenius ()

```
  395 {
  396     bool sucesso;
  397     ExibeGenius();                          //Mostra a tela de preparação para o
jogo.
  398     score = 0;                              //Zera o score.
  399     botaoPressionado = false;               //Desabilita a flag dos botões de jogo.
  400     InsereElemento();                       //Insere o primeiro elemento da
sequencia.
  401     MostraSequencia();                      //Mostra a primeira luz para o jogador.
  402     sei();                                  //Botões ativos.
  403     while(sm.state==sPlayGenius)                                        //Polling para
os botões de jogo.
  404     {
  405         _delay_ms(100);                                                 //Debounce.
  406         if(botaoPressionado)                                            //Detecção da
interrupção.
  407         {
  408             cli();                                                      //Desabilita
os botões.
  409                 if(numBotaoPressionado==sequencia[indexSequencia])  //Caso de
acerto do jogador.
  410                 {
  411                     if(indexSequencia==(score-1))                   //Caso para
entrada do último elemento da sequencia.
  412                     {
  413                         sucesso = InsereElemento();
  414                         if(!sucesso)                                //Caso para o
jogador atingir o máximo de pontos.
  415                         {
  416                             sm.state = sGameOver;                   //Mudança de
estado -> Game Over
  417                             score--;
  418                         }
```

```
  419                         MostraSequencia();                          //Exibe
novamente a sequencia de luzes.
  420                         indexSequencia = 0;                         //Reinicia o
indexador de verificação da sequência.
  421                     }
  422                     else                                            //Caso para a
entrada de um elemento inicial ou de meio
  423                     {                                               //da
sequência.
  424                         indexSequencia++;
  425                     }
  426                     botaoPressionado = false;                       //Desabilita a
flag dos botões de jogo.
  427                     sei();                                          //Botões
ativos.
  428                 }
  429                 else                                                //Caso para
erro do jogador.
  430                 {
  431                     sm.state = sGameOver;                           //Mudança de
estado -> Game Over
  432                     indexSequencia = 0;
  433                     score--;
  434                 }
  435         }
  436     }
  437 }
```

void GameOver ()

```
  440 {
  441     uint8 t i,j=POSICOES RANKING, endLinha;
  442     pos posicoes[POSICOES_RANKING];
  443     ExibeGameOver(score);                                           //Mostra na tela o
score do jogador.
  444     PiscaLeds();                                                    //Pisca todos os
LEDs juntos.
  445     for(i=0;i<POSICOES RANKING;i++)                                 //Lê da memória
todas as posições do ranking.
  446     {
  447         endLinha = i*TAM_LINHA;
  448         ReadLine((END_INICIAL+endLinha),&posicoes[i]);
  449     }
  450     for(i=0;i<POSICOES_RANKING;i++)                                 //Loop que procura
uma posição para a inserir o novo score
  451     {
  452         if(score>=posicoes[i].pScore)                               //Caso para o
encontro de uma.
  453         {
  454             j=POSICOES_RANKING-1;
  455             while(i<j)                                              //Loop que desloca
as outras posições para baixo.
  456             {
  457                 posicoes[j].pName[0] = posicoes[j-1].pName[0];
  458                 posicoes[j].pName[1] = posicoes[j-1].pName[1];
  459                 posicoes[j].pName[2] = posicoes[j-1].pName[2];
  460                 posicoes[j].pScore = posicoes[j-1].pScore;
  461                 j--;
  462             }
  463             posicoes[i].pName[0] = name[0];                         //Guarda o nome e o
score na posição encontrada.
  464             posicoes[i].pName[1] = name[1];                         //
  465             posicoes[i].pName[2] = name[2];                         //
  466             posicoes[i].pScore = score;                             //
  467             i = POSICOES_RANKING;                                   //Necessário para
quebrar o loop.
  468         }
  469     }
  470     for(i=0;i<POSICOES_RANKING;i++)                                 //Escreve
novamente na memória todas as posições
  471     {                                                               //do ranking.
```

```
  472         endLinha = i*TAM_LINHA;
  473         WriteLine(END_INICIAL+endLinha,posicoes[i].pName,posicoes[i].pScore);
  474     }
  475     _delay_ms(1000);                                          //Tempo para que o
jogador visualize seu score.
  476     sm.state = sMenuInicial;                                  //Mudança de estado
-> Menu Inicial
  477 }
```

int main (void)

```
  191 {
  192     IniciaGenius();             // Configura os periféricos.
  193     InitStateMachine();         // Inicializa a máquina de estados.
  194     while(1)
  195     {
  196         sm.action[sm.state]();  // Roda a máquina de estados
  197     }
  198     return 0;
  199 }
```

ISR (BOTAO_Y_FUNCT)

```
  568 {
  569     botaoPressionado = true;        //Ativação da flag.
  570     numBotaoPressionado = Y_NUM;    //Indica que o botão pressionado foi o amarelo.
  571     YBlink();                       //Pisca o LED amarelo.
  572 }
```

ISR (BOTAO_B_FUNCT)

```
  577 {
  578     botaoPressionado = true;        //Ativação da flag.
  579     numBotaoPressionado = B_NUM;    //Indica que o botão pressionado foi o azul.
  580     BBlink();                       //Pisca o LED azul.
  581 }
```

ISR (BOTAO_G_FUNCT)

```
  586 {
  587     botaoPressionado = true;        //Ativação da flag.
  588     numBotaoPressionado = G_NUM;    //Indica que o botão pressionado foi o verde.
  589     GBlink();                       //Pisca o LED verde.
  590 }
```

ISR (BOTAO_R_FUNCT)

```
  595 {
  596     botaoPressionado = true;        //Ativação da flag.
  597     numBotaoPressionado = R_NUM;    //Indica que o botão pressionado foi o vermelho.
  598     RBlink();                       //Pisca o LED vermelho.
  599 }
```

ISR (TIMER_RAND_FUNCT)

```
  604 {
  605     return;
  606 }
```

Referência do Arquivo MemoriaGenius.c

```
#include <avr/io.h>
```

Funções

- void **EEPROM_write** (unsigned int uiAddress, unsigned char ucData)
  Função que escreve um dado de um byte na posição da EEPROM indicada.
- uint8_t **EEPROM_read** (unsigned int uiAddress)
  Função que lê um dado de um byte na posição da EEPROM indicada.
- void **WriteLine** (uint16_t wAddress, uint8_t wName[3], uint8_t wScore)
  Função que escreve a partir do endereço indicado uma linha com 4 bytes de dados referentes a uma posição de ranking.
- void **ReadLine** (uint16_t rAddress, uint8_t *rLine)
  Função que lê a partir do endereço indicado uma linha com 4 bytes de dados referentes a uma posição de ranking.

Funções

void EEPROM_write (unsigned int *uiAddress*, unsigned char *ucData*)

Parâmetros:

uiAddress	Endereço de 2 bytes da memória EEPROM onde será escrito o dado.
ucData	Dado de um byte que será escrito no endereço indicado.

```
13 {
14     while(EECR & (1<<EEPE));    //Aguarda término de escrita anterior.
15     EEAR = uiAddress;           //Configura o registrador de endereço.
16     EEDR = ucData;              //Configura o registrador de dado.
17     EECR |= (1<<EEMPE);         //Habilita possibilidade de escrita na memória.
18     EECR |= (1<<EEPE);          //Habilita a escrita na memória.
19 }
```

uint8_t EEPROM_read (unsigned int *uiAddress*)

Parâmetros:

uiAddress	Endereço de 2 bytes da memória EEPROM de onde será lido um dado de um byte.

Retorna:

Conteúdo de um byte do endereço passado para a função.

```
25 {
26     while(EECR & (1<<EEPE)) ;   //Aguarda término de escrita anterior.
27     EEAR = uiAddress;           //Configura o registrador de endereço.
28     EECR |= (1<<EERE);          //Habilita leitura.
29     return EEDR;                //Retorna o valor lido.
30 }
```

void WriteLine (uint16_t *wAddress*, uint8_t *wName*[3], uint8_t *wScore*)

Parâmetros:

wAddress	Endereço de 2 bytes da memória EEPROM onde iniciará a escrita da linha.
wName[3]	Nome de 3 caracteres a ser escrito na linha.

wScore	Score a ser escrito na memória logo após o nome.

```
   38 {
   39     EEPROM_write(wAddress, wName[0]);   //Escreve primeiro caractere.
   40     EEPROM write(wAddress+1, wName[1]); //Escreve o segundo caractere na próxima
posição.
   41     EEPROM_write(wAddress+2, wName[2]); //Escreve o terceiro caractere na posição
seguinte.
   42     EEPROM write(wAddress+3, wScore);   //Escreve o score na última posição da linha.
   43 }
```

void ReadLine (uint16_t *rAddress*, uint8_t * *rLine*)

Parâmetros:

wAddress	Endereço de 2 bytes da memória EEPROM onde iniciará a leitura da linha.
rLine	Endereço da memória de dados do processador onde deverá ser guardada a linha lida.

```
   49 {
   50     rLine[0] = EEPROM_read(rAddress);   //Leitura do primeiro caractere do nome.
   51     rLine[1] = EEPROM_read(rAddress+1); //Leitura do segundo caractere do nome.
   52     rLine[2] = EEPROM read(rAddress+2); //Leitura do terceiro caractere do nome.
   53     rLine[3] = EEPROM_read(rAddress+3); //Leitura do score.
   54 }
```

Referência do Arquivo MemoriaGenius.h

Definições e Macros

- #define **END_INICIAL** 0x0000
  Definição do endereço inicial da EEPROM.

Funções

- void **EEPROM_write** (unsigned int uiAddress, unsigned char ucData)
  Função que escreve um dado de um byte na posição da EEPROM indicada.
- uint8_t **EEPROM_read** (unsigned int uiAddress)
  Função que lê um dado de um byte na posição da EEPROM indicada.
- void **WriteLine** (uint16_t wAddress, uint8_t wName[3], uint8_t wScore)
  Função que escreve a partir do endereço indicado uma linha com 4 bytes de dados referentes a uma posição de ranking.
- void **ReadLine** (uint16_t rAddress, uint8_t *rLine)
  Função que lê a partir do endereço indicado uma linha com 4 bytes de dados referentes a uma posição de ranking.

Referência do Arquivo TimerAndADCGenius.c

```
#include <avr/io.h>
#include "TimerAndADCGenius.h"
```

Funções

- void **ADCInit** ()
  Função que inicializa o conversor A/D.

- void **TimerInit** ()
  Função que configura o timer.
- uint16_t **lerBotaoMenu** ()
  Função que lê o valor analógico da porta dos botões de menu e retorna o valor correspondente a UP, DOWN, LEFT, RIGHT, CANCEL ou NONE.

Funções

void ADCInit ()

```
    9               {
   10     ADMUX = 0x0F;                          //Usa o canal em 0V.
   11     ADMUX |= (1 << REFS0);                 //Usa AVcc como referencia.
   12     ADMUX &= ~(1 << ADLAR);                //Resolução de 10 bits, com alinhamento
à direita.
   13     ADCSRA |= (1 << ADPS1) | (1 << ADPS0);  //Ajusta clock do ADC para 125 kHz (1MHz
com prescala de 8).
   14 }
```

void TimerInit ()

```
   15                 {
   16     /* Configuração dos três registradores com seus respectivos comandos. */
   17     TIMER_RAND_CTRL_A = NORMAL_MODE_OPERATION;
   18     TIMER_RAND_CTRL_B = NO_PREESCALER;
   19     TIMER RAND MASK |= OVF INT ENABLE;
   20 }
```

uint16_t lerBotaoMenu ()

Retorna:

Valor correspondente ao botão pressionado.

```
   26 {
   27     uint8 t canal = 0;
   28     uint16_t ADC_res;
   29     ADMUX |= (canal & 0x0F);            //Define o canal.
   30     ADCSRA |= (1 << ADEN);              //Habilita o ADC.
   31     ADCSRA |= (1 << ADSC);              //Inicia conversao do ADC
   32     while(ADCSRA && (1 << ADSC)==1);    //Aguarda fim da conversao.
   33     ADC res = ADCL;
   34     ADC_res = (ADCH << 8) + ADC_res;    //Leitura do resultado.
   35     ADMUX &= ~0x0F;                     //Retorna para o canal 0V.
   36
   37     /*  A comparação precisa aqui precisa seguir
   38         uma ordem crescente para que nenhum botão
   39         sombreado por outro.                        */
   40
   41     if(ADC_res<RIGHT)                   //Valor da porta menor que o limite superior
de RIGHT.
   42         return RIGHT;
   43     if(ADC res<UP)                      //Valor da porta menor que o limite superior
de UP.
   44         return UP;
   45     if(ADC_res<DOWN)                    //Valor da porta menor que o limite superior
de DOWN.
   46         return DOWN;
   47     if(ADC res<LEFT)                    //Valor da porta menor que o limite superior
de LEFT.
   48         return LEFT;
   49     if(ADC_res<CANCEL)                  //Valor da porta menor que o limite superior
de CANCEL.
   50         return CANCEL;
```

```
   51     if(ADC_res>=CANCEL)                //Valor da porta maior que qualquer outro
valor testado.
   52         return NONE;
   53  }
```

Referência do Arquivo TimerAndADCGenius.h

Definições e Macros

- #define **TIMER_RAND** TCNT0
  Redefine o nome do registrador onde pode ser lido o valor do timer.
- #define **TIMER_RAND_CTRL_A** TCCR0A
  Registrador de controle A do timer do processador.
- #define **TIMER_RAND_CTRL_B** TCCR0B
  Registrador de controle B do timer do processador.
- #define **TIMER_RAND_MASK** TIMSK0
  Registrador máscara do timer.
- #define **OVF_INT_ENABLE** 0x01
  Comando a ser dado ao controle para que o timer funcione como uma interrupção dada por overflow.
- #define **NO_PREESCALER** 0x01
  Comando que define que o clock base não receberá divisão para o timer.
- #define **NORMAL_MODE_OPERATION** 0x00
  Comando que define o modo normal de operação do timer.
- #define **TIMER_RAND_FUNCT** TIMER0_OVF_vect
  Redefinição do nome da função de interrupção por overflow do timer.
- #define **RIGHT** 50
  Limite superior do valor analógico que representa botão direito do menu.
- #define **UP** 195
  Limite superior do valor analógico que representa botão superior do menu.
- #define **DOWN** 380
  Limite superior do valor analógico que representa botão inferior do menu.
- #define **LEFT** 555
  Limite superior do valor analógico que representa botão esquerdo do menu.
- #define **CANCEL** 790
  Limite superior do valor analógico que representa botão select do menu.
- #define **NONE** 1000
  Valor analógico representativo para nenhum botão pressionado.

Funções

- void **ADCInit** ()
  Função que inicializa o conversor A/D.
- void **TimerInit** ()
  Função que configura o timer.
- uint16_t **lerBotaoMenu** ()
  Função que lê o valor analógico da porta dos botões de menu e retorna o valor correspondente a UP, DOWN, LEFT, RIGHT, CANCEL ou NONE.