Avaliaçao 01

Re-implementação de atividades usando C++ e AVR-Libc

Curso: Engenharia de Telecomunicações
Disciplina: STE29008 – Sistemas Embarcados
Professor: Roberto de Matos

Aluna
Maria Fernanda Silva Tutui

Abril de 2019

1 Introdução

O seguinte relatório apresenta a re-implementação das seguintes atividades usando C++ e AVR-Libc:

- Lab 01: GPIO - *Hello* LED
- Lab 02: UART - *Serial Communication*
- Lab 04: UART - ADC *and Sensor Reading*
- Lab 06: GPIO *and External Interrupts - part 1*
- Lab 07: GPIO *and External Interrupts - part 2*

O relatório expõe separadamente cada uma das atividades. Contendo a descrição do funcionamento das mesmas e os principais registradores utilizados.

Todos os laboratórios foram desenvolvidas a partir do uso do microcontrolador AVR Atmel Atmega2560 juntamente com a biblioteca AVR-LibC por meio o ambiente de desenvolvimento Eclipse neon modificado.

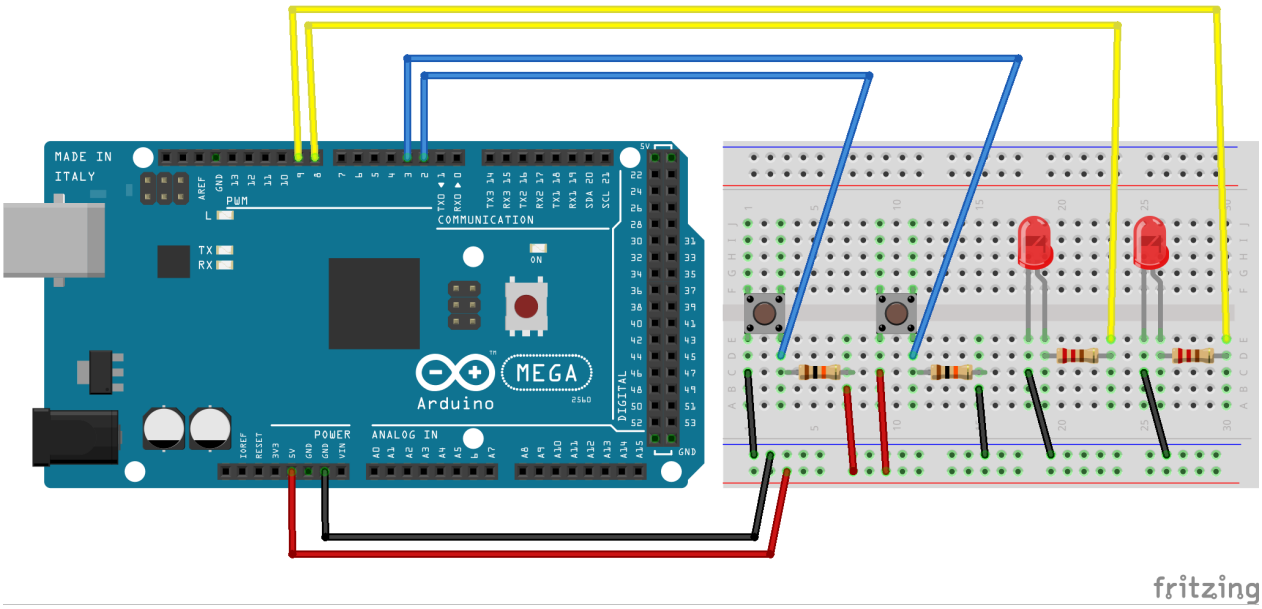
A seção seguinte é responsável pela apresentação detalhada das atividades, em seguida o relatório é finalizando com a conclusão.

2 Desenvolvimento

2.1 Lab 01: GPIO - *Hello* LED

A fim de usar o GPIO, grupo de pinos responsáveis pela comunicação de sinais digitais de entrada e saída, o primeiro exercício propôs que um LED fosse ligado e desligado a partir do pressionamento de um botão ou não, respectivamente. O esquemático para a montagem do circuito pode ser vista na figura 1, a seguir.

Figura 1: Montagem Lab01



O botão conectado ao pino 2 no Arduino MEGA é responsável pelo acionamento do LED conectado ao pino 8 e o botão conectado ao pino 3, pelo LED no pino 9.

A primeira função função presente no código, chamada de *setup()* diz respeito a inicialização para o uso dos pinos, a fim de estabelece-los como pinos de entrada ou de saída.

```
1  void setup(){
2    DDRH |=  (1 << PH5); // LED   1 - Pino 8 - Escreve 1 - Saida
3    DDRE &= ~(1 << PE4); // Botao 1 - Pino 2 - Escreve 0 - Entrada
4
5    DDRH |=  (1 << PH6); // LED   2 - Pino 9 - Escreve 1 - Saida
6    DDRE &= ~(1 << PE5); // Botao 2 - Pino 3 - Escreve 0 - Entrada
7  }
```

Como pode ser observado na linha de número 2, o primeiro registrador utilizado, DDRH, que tem como função a definição do modo de operação dos pinos da porta H recebeu como escrita ”1” em PH5, quinto bit do registrador. Dessa forma o pino 8, regido por esse registrador, foi definido como um pino de saída, ou seja, o LED que encontra-se nessa porta poderá ser aceso.

O mesmo acontece na linha 5, o pino 9, administrado pelo registrador DDRH, quanto a função de operação, foi igualmente setado como ”1” na posição 6, sexto bit do registrador, para que pudesse suportar o LED, ou seja, como um pino de saída.

Nas linhas 3 e 6 são feitas as configurações dos bits 4 (PE4) e 5 (PE5) do registrador DDRE, também um registrador capaz de definir o modo de operação dos pinos, dessa vez na porta E. Em ambos os casos, por tratarem de pinos para a entrada de dados, diferentemente da situação dos LEDs, os pinos receberam a escrita de "0". Agora os botões são capazes de enviar informações para a placa.

```
8  int main(){
9    setup();
10
11   while (true){
12     // Botao e Pino 1
13     if (PINE & (1 << PE4))
14       PORTH &= ~(1 << PH5); // Escreve 0 no LED   1
15     else
16       PORTH |=  (1 << PH5); // Escreve 1 no Botao 1
17
18     // Botao e Pino 2
19     if (PINE & (1 << PE5))
20       PORTH |=  (1 << PH6); // Escreve 1 no LED   2
21     else
22       PORTH &= ~(1 << PH6); // Escreve 0 no Botao 2
23     }
24   }
25 }
```

Feitas as configurações dos pinos, a próxima função *main()* é responsável pelo controle dos botões e acendimento dos LEDs. Como pode ser visto no código acima dentro de um laço infinito podem ser vistos dois laços com *if()*, em cada um deles é checado se o botão está sendo pressionado. O primeiro *if()* usa o registrador PINE, para verificar o botão, nesse caso, o botão do pino 2 usando o quarto bit do registrador na porta E. O segundo *if()*, que também faz uma verificação no registrador PINE, porém para o quinto bit trata do botão na porta 3.

O registrador PORTH, utilizado para a escrita, foi responsável por escrever nos bits 5 e 6, LED do pino 8 e 9, respectivamente, para acender e apagar os LEDs.

2.2 Lab 02: UART - *Serial Communication*

Nesse laboratório a proposta foi de usar a UART, um tipo de comunicação serial assíncrona onde apenas um bit de informação é enviado por vez.

O exercício propunha o recebimento de um byte usando uma das entradas seriais, em seguida, o byte deveria ser incrementado e após, transmitido usando uma das saídas seriais. O código a baixo mostra como a implementação dessa funcionalidade foi executada.

A primeira função *USART_Init()*, programa o comportamento na inicialização da USART. O primeiro registrador a ser usado pode ser visto na linha 28, UBRR0H onde recebe a velocidade de transmissão escolhida. Logo na linha abaixo o registrador BRR0L faz o controle da taxa de transmissão.

```
26 void USART_Init(unsigned int ubrr){
27   UBRR0H = (unsigned char)(ubrr>>8); // Taxa de transmissao
28   UBRR0L = (unsigned char)ubrr;
29   UCSR0B = (1<<RXEN0)|(1<<TXEN0);    // Habilita RX e TX
30
31   UCSR0C &= ~(1<<USBS0);             // Formato do frame
32   UCSR0C |= (3<<UCSZ00);
33 }
```

Para a habilitação de envio e recebimento de dados os bits RXEN0 e TXEN0 foram classificados como ”1”, sendo eles os bits 4 e 3, respectivamente o registrador UCSR0B, responsável por essa habilitação.

Ao fim da função, o registrador UCSR0C recebe as definições para o formato do *frame*, tamanho e também a quantidade de *stop-bits*. Para isso foram definidos os bits em USBS0 em ”0” para a transmissão de 8 bits e UCSZ00 em ”3” para 1 *sotp-bit*, respectivamente.

As duas funções seguintes são responsáveis pela recepção (*USART_Receive()*) e transmissão (*USART_Transmit()*) dos dados, ambas usando o registrador UCSR0A.

```
34 unsigned char USART_Receive(){
35   while ( !(UCSR0A & (1<<RXC0)) );   // Espera dados
36   return UDR0;                       // Retorna dado recebido
37 }
38
39 void USART_Transmit( uint8_t data ){
40   while ( !( UCSR0A & (1<<UDRE0)) ); // Espera para transmitir
41   UDR0 = data;                       // Envia dado
42 }
```

Para a o recebimento de dados o registrador UCSR0A é verificado no sétimo bit (RXC0) e o dado o recebido é gravado no registrador UDR0. Já para a transmissão é verificado o quinto bit (UDRE0) do mesmo registrador e então o registrador UDR0 envia a informação.

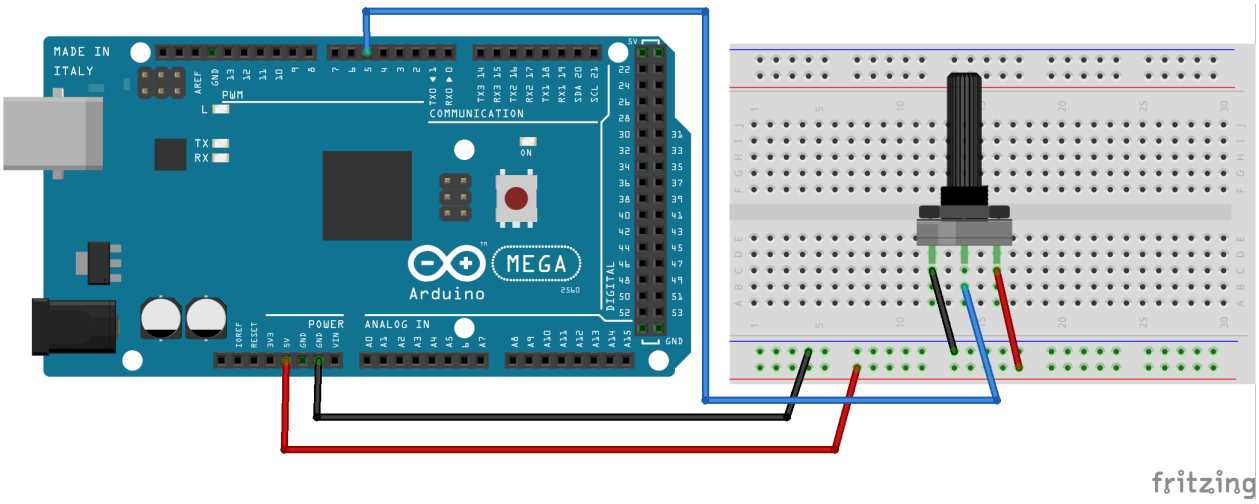
```
43 int main( void ){
44   USART_Init(MYUBRR);
45
46   while(true){
47     uint8_t data;
48     data = USART_Receive();
49     data = data+1;
50     USART_Transmit(data);
51   }
```

Finalmente a ultima função *main()* é faz o controle do acesso as funções de recepção incremento do dado recebido e transmissão do mesmo.

2.3 Lab 04: UART - ADC *and Sensor Reading*

O seguinte exercício tem como objetivo realizar a conversão analógico-digital (gerar uma representação digital a partir de uma grandeza analógica) de um sensor, no caso um potenciômetro. O esquemático para a montagem pode ser visto na figura 2.

Figura 2: Montagem Lab04



Primeiramente, é necessário lembrar que o ARV utilizado é baseado em 8 bits e que o conversor do ATmega2560 trabalha com uma resolução de 10 bits.

A mesma abordagem feita nos exercícios anteriores foi aplicada a este iniciando o programa com funções de inicialização das funcionalidades utilizadas.

```
52 void adc_init(void){
53   ADMUX |= (1<<REFS0);        // Tensao de referencia
54   ADCSRA |= (1<<ADEN);        // ADC
55 }
```

Nesse caso, a primeira função, apresentada acima, *adc_init()* seleciona a tensão de referência utilizada escrevendo ”1” no sexto bit (REFS0) do registrador ADMUX, 5 Volts. Logo abaixo, o registrador ADCSRA é escrito com o valor ”1” no sétimo bit (ADEN) habilitando a conversão AD.

```
56 uint16_t read_adc(uint8_t channel){
57   ADMUX &= 0xE0;              // Limpa MUX
58   ADMUX |= channel&0x07;      // Canal a ser lido
59   ADCSRB = channel&(1<<3);    // MUX5
60   ADCSRA |= (1<<ADSC);        // Inicia nova conversão
61   while(ADCSRA & (1<<ADSC)); // Espera conversão
62   return ADCW;                // Retorna o valor para o canal
63 }
```

O código mostrado acima possui as configurações necessárias para o uso do conversor. Iniciando com a limpeza do registrador ADMUX, em seguida expondo o canal a ser lido, no caso o pino A5, usando o valor correspondente a ele ”0x07”. O valor selecionado para o registrador ADCSRB diz respeito ao uso do canal sem ganhos. Assim, é iniciada uma nova conversão a partir da adição de ”1” no sexto bit do registrador ADCSRA. A partir do laço de *while()* é aguardada uma conversão e o retorno da mesma no registrador que ADCW, o qual armazena esse resultado.

Assim como no exercício anterior a interface serial é utilizada para que os valores das conversões sejam vistos. Da mesma forma foi declarada uma função para a transmissão dos dados (*USART_Transmit()*).

```
64 int RMS ( int repeat ){
65   float accumulated = 0;
66   float average;
67
68   for (int i = 0; i < repeat; i++){
69     digital_value = read_adc(channel);
70     accumulated = accumulated + (digital_value * digital_value);
71   }
72   average = accumulated/repeat;
73   return sqrt(average);
74 }
```

A próxima função chamada de *RMS()*, mostrada a cima, é necessária devido a inúmeras conversões feitas a cada segundo pelo AVR, os 30 primeiros valores capturados são acumulados e ao final é retornada a raiz do valor quadrático médio.

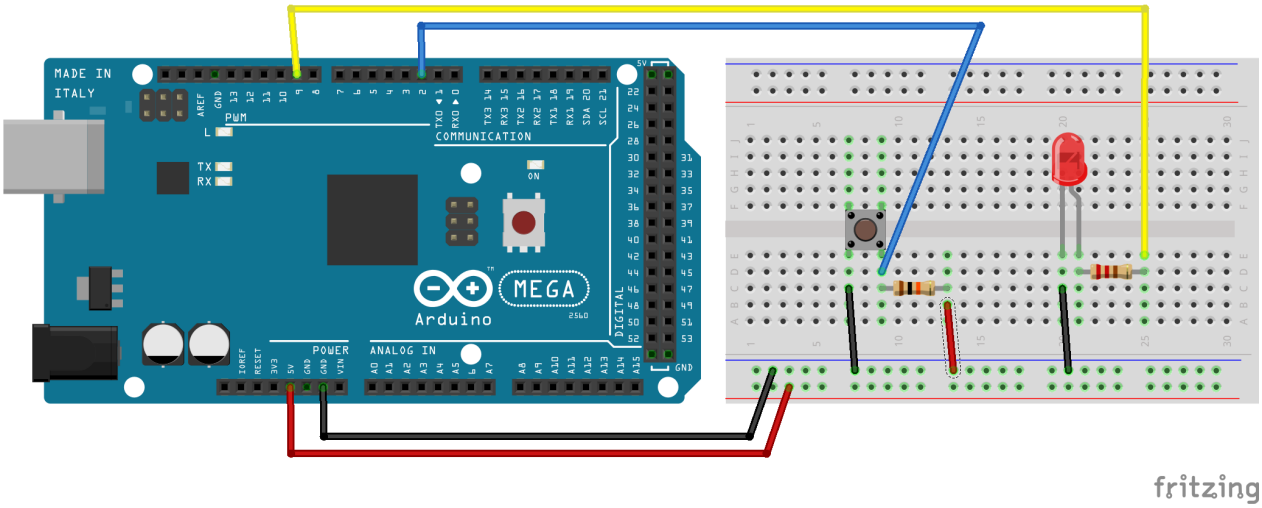
A última parte do exercício contém a função *main()*, mostrada a baixo. Ela faz uso da função *RMS()* e transforma o valor digital recebido em analógico novamente. Como já comentado, a conversão retorna um valor com 10 bits, por esse motivo uma conversão é feita para que sejam convertidos para 8 bits. Em seguida esses valores são mostrados usando a função *USART_Transmit()* em um for para que cada bit possa ser transmitido por vez.

```
75 int main( void ){
76   adc_init();
77   USART_Init(MYUBRR);
78
79   while(true){
80     digital_value = RMS(30);
81     _delay_ms(2000);
82     analog_value = (digital_value*5)/1024;
83     char digital[10];
84     char analog[50];
85
86     dtostrf(digital_value, 5, 0, digital);
87     dtostrf(analog_value, 3, 3, analog);
88
89     char label_d[20] = "Digital: ";
90     for (int i = 0; i < 9; i++){
91       USART_Transmit((uint8_t)label_d[i]);
92     }
93     for (int i = 1; i < 5; i++){
94       USART_Transmit((uint8_t)digital[i]);
95     }
96     USART_Transmit((uint8_t)32);
97
98     char label_a[13] = "Analogico: ";
99     for (int i = 0; i < 11; i++){
100      USART_Transmit((uint8_t)label_a[i]);
101    }
102    for (int i = 0; i < 4; i++){
103      USART_Transmit((uint8_t)analog[i]);
104    }
105    USART_Transmit((uint8_t)32);
106    USART_Transmit((uint8_t)124);
107    USART_Transmit((uint8_t)32);
108  }
109 }
```

2.4 Lab 06: GPIO *and External Interrupts - part 1*

Para esse exercício o objetivo foi tratar interrupções no microcontrolador. Para isso foi proposta a alteração do estado do LED a cada vez que um botão fosse pressionado. O esquemático utilizado pra a montagem do circuito para esse laboratório pode ser visto na figura 3 abaixo.

Figura 3: Montagem Lab06



Inicialmente, foi escrita uma função com as configurações de inicialização chamada *setup()*.

```
110 void setup(){
111   DDRH |=  (1 << PH6); // LED   - Pino 9 - Escreve 1 - Saida
112   DDRE &= ~(1 << PE4); // Botao - Pino 2 - Escreve 0 - Entrada
113
114   EICRB |= (1<<ISC41) | (1<<ISC40); // Habilita interrupcao na borda de subida
115   EIMSK =  (1<<INT4);               // Interrupcao INT4
116   sei();                            // Habilita interrupcoes globais, ativando bit I no SREG
117 }
```

Da mesma forma como no primeiro laboratório, Lab 01, onde LEDs e botões foram configurados, dessa vez, um LED e um botão são configurados da mesma forma, usando os registradores DDRH e DDRE, respectivamente.

Porém, para o uso da interrupção outros registradores foram utilizados. Ainda na função *setup()* pode ser observado na linha 115 o uso do registrador EICRB. Nele foi definido através da escrita do primeiro (ISC41) e segundo bits (ISC40) a interrupção externa na borda de subida do clock. Na linha seguinte, o registrador EIMSK recebe um ”1” no quarto bit (INT4) a fim de habilitar essas interrupções.

A última linha invocando a função *sei()* da biblioteca AVR-LibC serve para que as interrupções sejam habilitadas globalmente.

Uma função chamada *debounce()* foi criada para a captura do estado do botão, como pode ser visto abaixo.

```
118 bool debounce(){          // Funcao simulando millis
119   _delay_ms(300);
120   if (PINE & (1 << PE4))
121     return true;
122   else
123     return false;
124 }
```

A função verifica o se o botão continua sendo pressionado depois de 300 milissegundos, o que é feito a partir do quarto bit (PE4) do registrador PINE, assim como no primeiro laboratório. Caso a resposta seja verdadeira, o botão deve mudar de estado. Assim como mostrado na função abaixo, *ISR(INT4_vect)*.

```
125 ISR(INT4_vect){       // Trata interrupcao externa
126   if (debounce())
127     PINH |= (1<<PH6); // Muda o estado do bit
128 }
```

Essa função é responsável por tratar as interrupções, usando a função *debounce()*, onde, como já mencionado acima a partir do resultado muda o estado do LED, apagando ou acendendo.

```
129 int main(){
130   setup();
131   while(true);
132 }
```

Por fim, a ultima função faz invoca as configurações para a atividade e usa um laço infinito para que o programa continue verificando ininterruptamente.

2.5 Lab 07: GPIO *and External Interrupts - part 2*

Para o último exercício proposto, assim como no anterior a partir da interrupção externa gerada pelo botão. Dessa vez o LED deverá manter-se acesso enquanto o botão estiver pressionado. Para que isso ocorresse a única função diferente da parte 1 deve ser a pertencente ao comportamento ao ser detectada uma interrupção logo, a função ISR(INT4_vect()).

```
133 ISR(INT4_vect){                // Trata interrupcao externa
134   if (debounce())
135     while(PINE & (1 << PE4)){ // Verifica botao (5V com chave aberta e 0V fechada)
136       PORTH &= ~(1 << PH6);   // Escreve 0 no LED
137     }
138     PORTH |= (1<< PH6);       // Escreve 1 no LED
139   }
140 }
```

A nova modelagem da função pode ser vista acima. Para que o LED ascenda somente enquanto o botão estiver sendo pressionado o registrador PINE é verificado, caso a resposta seja verdadeira o LED é aceso, caso contrário, mantem-se apagado.

3 Conclusões

A partir da execução do trabalho, pode ser concluído que manipular o microcontrolador utilizando a biblioteca AVR-LibC é muito mais complexo pois demanda uma bagagem maior de conhecimento técnico e abstração, porém disponibiliza infinitas configurações diferentes possibilitando uma maior customização no programa a ser criado.