Avaliação 01

Programando o AVR Atmel ATmega2560 utilizando *C* e a biblioteca AVR-LibC

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

Aluno
Paulo Fylippe Sell
152000502-4

Março de 2019

1 Introdução

Este relatório apresenta a implementação de cinco diferentes exercícios para um primeiro contato com o AVR **Atmel ATmega2560**. Segundo (LIMA; VILLAçA, 2012), as principais características de um **AVR** são:

- Executam a maioria das instruções em um ou dois ciclos de *clock*;
- Alta integração e grande número de periféricos com efetiva compatibilidade com toda a família AVR
- Possuem vários modos para redução do consumo de energia
- Preço acessível

Os exercícios foram desenvolvidos através da biblioteca **AVR-LibC** e a linguagem de programação **C**. A IDE (*Integrated Development Environment*) **Eclipse neon**, modificada com um *plugin* para a injeção dos códigos no AVR, também foi utilizada para auxílio nos desenvolvimentos. As sessões seguintes irão apresentar o desenvolvimento das atividades, bem como quais registradores do AVR Atmel ATmega2560 foram utilizados em cada exercício.

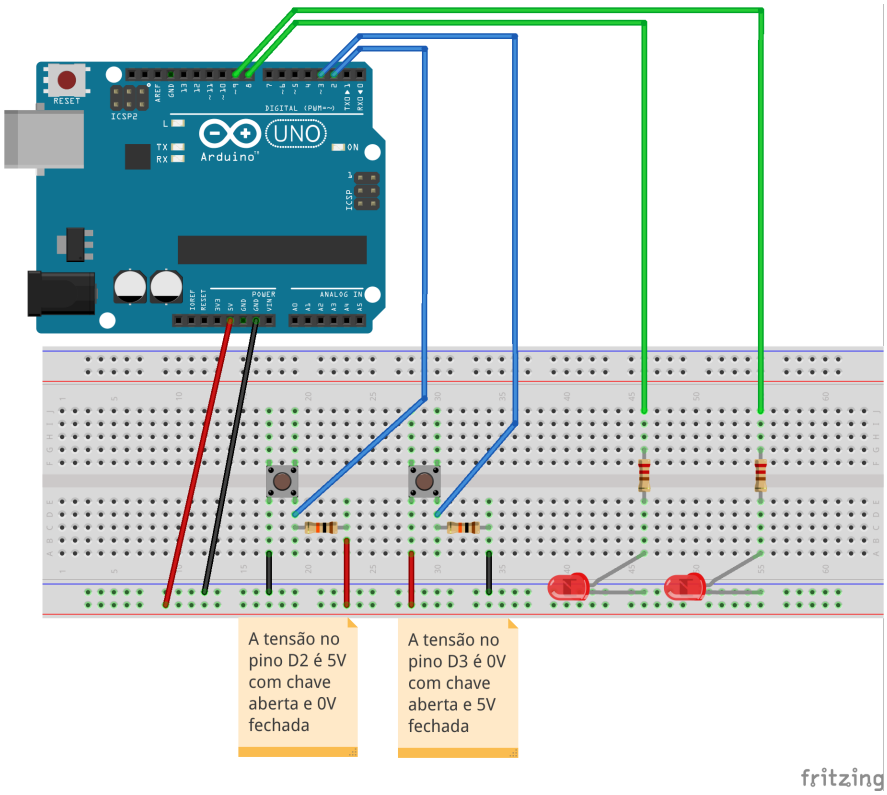
2 Desenvolvimento das atividades

2.1 GPIO

O primeiro exercício consistia em acender dois ***LEDS*** diferentes, a partir do acionamento de dois botões.

A figura 1 ilustra a maneira que o circuito desde exercício foi montado.

Figura 1: Circuito montado



O botão conectado ao pino **D2** do *Arduino* irá acender o *LED* conectado ao pino **D8** e o botão conectado ao pino **D3** irá acender o *LED* conectado ao pino **D9** do *Arduino*.

Para traduzir os pinos do *Arduino* para as portas reais do AVR devemos utilizar seu ***datasheet*** (ATMEL, 2014). Utilizou-se um conjunto de quatro registradores para a realização desta atividade. A tabela 1 descreve os registradores utilizados e suas funções.

Tabela 1: Registradores para controle de GPIO

Registrador	Descrição
DDRE	Define o modo de operação dos pinos da porta E
DDRH	Define o modo de operação dos pinos da porta H
PINE	Registra o estado de cada pino da porta E
PORTH	Define o estado de cada pino da porta H

A função *setup1()* mostra de que maneira os pinos utilizados foram configurados (pinos D2, D3, D8 e D9 do *Arduino*, respectivamente):

```
1  void setup1(){
2        DDRE &= ~(1 << PE4); // Pino D2 do Arduino como entrada
3        DDRE &= ~(1 << PE5); // Pino D3 do Arduino como entrada
4
5        DDRH |= (1 << PH5);  // Pino D8 do Arduino como saída
6        DDRH |= (1 << PH6);  // Pino D9 do Arduino como saída
7
8  }
```

Ao escrever '0' nos *bits* 4 e 5 (**PE4** e **PE5**, respectivamente) do registrador **DDRE**, estamos informando ao AVR que estes pinos irão funcionar como pinos de entrada, ou seja, irão receber os sinais dos botões.

Já ao escrever '1' nos *bits* 5 e 6 (**PH5** e **PH6**, respectivamente) do registrador **DDRH**, estamos configurando estes pinos como saída, ou seja, eles irão acender os *LEDS* quando os botões (*bits* 4 e 5 da porta E) forem pressionados.

Após a configuração dos pinos, o desenvolvimento do código de controle dos botões e iluminação dos *LEDS* não foi difícil. A função abaixo mostra a programação realizada. O registrador **PINE** foi utilizado para verificação dos estados dos botões em cada *bit* (4 e 5) utilizado da porta E. Já o registrador **PORTH** foi utilizado para escrever nos *bits* 5 e 6 da porta H os valores para acender e apagar os *LEDS*.

```
9  void exec1(){
10   setup1();
11   while(true){
12     if ((PINE & (1<<PE5))){   // O led conectado no bit 6 da PORTH
13       PORTH |= (1 << PH6);    // irá acender quando o quinto bit
14     } else {                  // da PORTE estiver em 1
15       PORTH &= ~(1 << PH6);
16     }
17
18     if (PINE & (1<<PE4)){     // O led conectado no bit 5 da PORTH irá
19       PORTH &= ~(1 << PH5);   // acender quando o quarto bit da PORTE
20     } else {                  // estiver em 0
21       PORTH |= (1 << PH5);
22     }
23   }
24 }
```

2.2 UART - Comunicação serial

Esta atividade consistia em receber um *byte* por uma das entradas seriais do AVR ATmega2560, incrementá-lo e, em seguida, transmiti-lo através de uma das saídas seriais do microcontrolador.

Apesar desta atividade não possuir montagem de circuito, a mesma exigiu a manipulação de um número maior de registradores do que a atividade anterior. A tabela 2 indica os registradores utilizados.

Tabela 2: Registradores para comunicação serial

Registrador	Descrição
UBRR0	Define a quantidade de *bits* por segundo que a comunicação utilizará (*baudrate*)
UCSR0B	Entre outras funções, habilita os pinos TX e RX da comunicação serial
UCSR0C	Entre outras funções, define o número de *bits* do quadro e o número de *stop-bits*
UCSR0A	Entre outras funções, indica se uma troca de mensagem foi completada
UDR0	Registrador que guarda o *byte* a ser enviado ou *byte* recebido

A função abaixo mostra a preparação do microcontrolador para a utilização de sua *UART* (Universal Asynchronous serial Receiver and Transmitter). As linhas 31 e 32 mostram o registrador **UBRR0** recebendo a taxa de transmissão escolhida (9600 bps). O microcontrolador possui um *timer* que recebe a informação contida no registrador **UBRR0** e faz o controle da taxa de transmissão.

As duas próximas linhas informam ao registrador **UCSR0B** que os *bits* 4 e 3 (chamados de **RXEN0** e **TXEN0** respectivamente) devem estar em estado lógico alto ('1'). Desta forma, tanto a recepção quanto a transmissão de *bytes* pela porta serial escolhida estão habilitadas.

```
25 #define FOSC 16000000
26 #define BAUD 9600
27
28 void setup2(){
29   unsigned int ubrr = (FOSC/16/BAUD)-1;   // Definindo frequencia de oscilamento
30
31   UBRR0H = (unsigned char) (ubrr >> 8);   // Definindo frequencia de
32   UBRR0L = (unsigned char) ubrr;          // oscilamento
33
34   UCSR0B |= (1 << RXEN0);                 // Habilitando RX0
35   UCSR0B |= (1 << TXEN0);                 // Habilitando TX0
36
37   UCSR0C &= ~(1 << USBS0);                // Definindo numerido de bits e
38   UCSR0C |= (3 << UCSZ00);                // Stop-bit
39
40 }
```

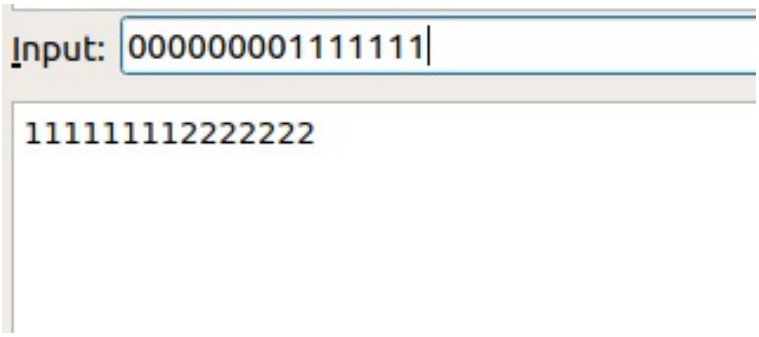
As duas últimas linhas desta função definem o número de *stop-bits* (1 *stop-bit*) e o tamanho dos quadros (8 *bits*) a serem trocados, ao definir que o *bit* 3 do registrador **UCSR0C** esteja em '0' (*bit* **USBS0**) e definindo que os *bits* 2 e 1 do registrador **UCSR0C** estejam em '1' (*bits* **UCSZ01** e **UCSZ00**)

Uma vez tendo o microcontrolador configurado e pronto para a troca de mensagens, podemos desenvolver uma função que receba *bytes* pela serial, faça o incremento e transmita os *bytes* novamente. É possível testar a função abaixo utilizando um *software* (*CuteCom*, *Minicom*, monitor serial da IDE do *Arduino*, etc) que também envie e receba dados pela porta serial (USB) de um computador.

```
42 void exec2(){
43   setup2();
44   while (true){
45     unsigned char byte;
46     if ((UCSR0A & (1<<RXC0))){        // Se receber algo pela serial
47       byte = UDR0;                    // incrementa o byte e envia
48       byte = byte + 1;                // o byte incrementado.
49       if ((UCSR0A & (1<<UDRE0))){     // Só envia se não ter nada
50         UDR0 = byte;                  // na fila de envio
51       }
52     }
53   }
54 }
```

A linha 46 da função faz a seguinte pergunta ao microcontrolador: "Existe algum dado recebido e que **não** foi lido?". Em termos mais técnicos, o *bit* 7 (**RXC0**) do registrador **UCSR0A** está em estado lógico '1'? Se a resposta for positiva, o microcontrolador armazena a informação contida no registrador **UDR0**, que é o registrador responsável por armazenar as mensagens trocadas. O AVR então incrementa o dado recebido e, caso o *bit* 5 (**UDRE0**) do registrador **UCSR0A** enteja em nível alto, o registrador **UDR0** pode então receber o *byte* incrementado e fazer o envio da informação.

Figura 2: Troca de mensagens pela *UART* do microcontrolador



A figure 2 mostra uma sequência de zeros e uns sendo recebidos pelo microcontrolador e, em seguida, o computador recebe do microcontrolador os *bytes* incrementados (captura feita pelo *software CuteCom*).

2.3 Conversão analógico-digital

O terceiro experimento consistia em utilizar o conversor analógico-digital do AVR Atmel ATmega2560. O conversor analógico-digital deste microcontrolador possui uma resolução de 10 *bits*. Desta forma, o resultado da conversão fica armazenado em dois diferentes registradores, uma vez que este AVR é baseado em 8 *bits*. Mais uma vez, foram utilizados diferentes registradores para a realização desta atividade. Uma vez que nós, como usuários, precisamos verificar se a conversão realizada está correta, esta atividade exigiu também a utilização da comunicação serial do microcontrolador. Como os registradores da comunicação serial já foram explicados na atividade anterior, a tabela 3 apresenta apenas os registradores necessários para a conversão analógico-digital.

Tabela 3: Registradores para conversão analógico-digital

Registrador	Descrição
ADMUX	Registrador de definição de pino de conversão e tensão de referência
ADCSRA	Entre outras funções, habilita e desabilita a conversão analógico-digital
ADCSRB	Entre outras funções, define o pino de conversão e o ganho que a leitura irá receber

Como nas atividades anteriores, precisamos preparar o microcontrolador para seu uso. A função *setup3()* apresenta a inicialização dos registradores necessários. O *bit* número 6 (**REFS0**) do registrador **ADMUX** é configurado com valor alto, para que a tensão de referência do conversor seja 5 Volts. Em seguida, a conversão é habilitada, ao iniciar o sétimo *bit* (**ADEN**) do registrador **ADCSRA** com o valor '1'.

```
55 void setup3(){
56   ADMUX  |= (1 << REFS0); // Definindo tensão de referencia do conversor
57   ADCSRA |= (1 << ADEN);  // Habilitando conversor
58 }
```

A partir de agora, é possível iniciar uma conversão. O pino escolhido para a leitura dos valores analógicos foi o pino **ADC5**. Desta forma, devemos informar aos registradores **ADMUX** (que também tem a função de habilitar os pinos a serem lidos) e **ADCSRB** que usaremos o pino ADC5. Para isto, devemos escrever nos *bits* de 4 a 0 (**MUX4:0**) do registrador o valor em binário do pino escolhido (ADC5, 00101) e escrever '0' no *bit* 3 (**MUX5**) do registrador **ADCSRB**, uma vez que não queremos que a conversão sofra nenhum ganho. A função *read_adc()* faz estas definições e em seguida inicia a conversão ao escrever '1' no *bit* 6 (**ADSC**) do registrador **ADCSRA**. A função abaixo tem como retorno um valor digital de 16 *bits*, sendo que nos 10 *bits* menos significativos está o resultado da conversão analógico-digital, uma vez que o *bit* 5 (**ADLAR**) do registrador **ADMUX**, que define se serão utilizados os dez primeiros ou dez últimos *bits* do registrador que armazena o resultado da conversão (**ADCW**), não foi alterado para o valor '1'.

```
60 uint16_t read_adc(uint8_t channel){
61   ADMUX |= channel&0x07;      // Definindo pino de entrada ADC5
62   ADCSRB = channel&(1<<3);    // Definindo bit MUX5 como 0
63   ADCSRA |= (1<<ADSC);        // Inicia nova conversão
64   while(ADCSRA & (1<<ADSC));  // Aguarda a conversão finalizar e retorna
65   return ADCW;                // dois bytes com a conversão
66 }
```

A linha 64 da função acima mostra que o microcontrolador aguarda que a conversão finalize para continuar sua execução. O próprio microcontrolador altera o valor do *bit* **ADSC** para '0' via *hardware* quando a conversão é finalizada.

Como o AVR faz inúmeras conversões por segundo, é importante acumular estes valores e extrair a raiz quadrada do seu valor médio quadrático (ou valor RMS, função *RMS()* abaixo), para ter uma melhor precisão do valor que está sendo convertido.

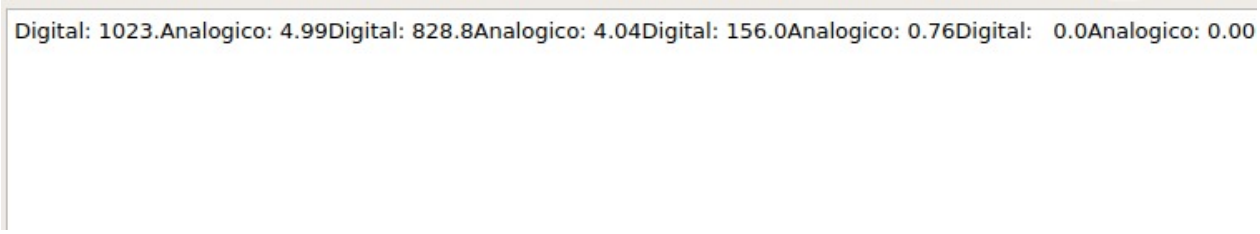
```
67  float RMS (int repeat){
68
69    float accumulated = 0;
70    float average;
71    float digital_value;
72    uint8_t channel = 5;
73    for (int i = 0; i < repeat; i++){
74      digital_value = read_adc(channel);
75      accumulated = accumulated + (digital_value*digital_value);
76    }
77
78    average = accumulated/repeat;
79    return sqrt(average);
80  }
81
82  void exec3(){
83    setup3();
84    setup2();
85    while (true){
86      _delay_ms(2000);
87      float val = RMS(300);
88      float analog_val = (val*5)/1024;    // Regra de 3 simples para obter o valor analógico lido
89      printf("%.f", double(val));
90      char a[10] = "Digital: ";
91      char b[12] = "Analogico: ";
92      char digital[50];
93      char analog[50];
94
95      for (int i = 0; i < 9; i ++){       // Loop para escrever na serial
96        while (!( UCSR0A & (1<<UDRE0)));  // a palavra "Digital: "
97        UDR0 = (uint8_t) a[i];
98      }
99
100     dtostrf(val,5,1, digital);
101     dtostrf(analog_val, 3,3,analog);
102
103     for (int i = 0; i < 5; i ++){       // Loop para escrever os bytes
104       while (!( UCSR0A & (1<<UDRE0)));  // da conversão em valores
105       UDR0 = (uint8_t) digital[i];      // digitais
106     }
107
108     for (int i = 0; i < 11; i ++){      // Loop para escrever na serial
109       while (!( UCSR0A & (1<<UDRE0)));  // a palavra "Analogico: "
110       UDR0 = (uint8_t) b[i];
111     }
112
113     for (int i = 0; i < 4; i ++){       // Loop para escrever os bytes
114       while (!( UCSR0A & (1<<UDRE0)));  // da conversão em valores
115       UDR0 = (uint8_t) analog[i];       // analógicos
116     }
117   }
118
119 }
```

O bloco de código acima nos mostra a manipulação na linguagem *C* dos dados recebidos da conversão analógico-digital. A cada dois segundos, o microcontrolador irá fazer 30 conversões e enviar o resultado da conversão pela porta serial (a partir da linha 94). Novamente é possível observar os valores transmitidos pela porta serial com auxílio de um *software* que receba dados pela porta serial (USB) de um computador (*CuteCom*, *Minicom*, monitor serial da IDE do *Arduino*, etc).

É importante observar que, como o resultado da conversão analógico-digital possui 10 *bits* e a *UART* do AVR possui registradores de 8 *bits*, é necessário fazer uma conversão destes valores para que seja possível transmitir os dados para a porta serial do microcontrolador (linhas 100 e 101 do bloco de código acima)

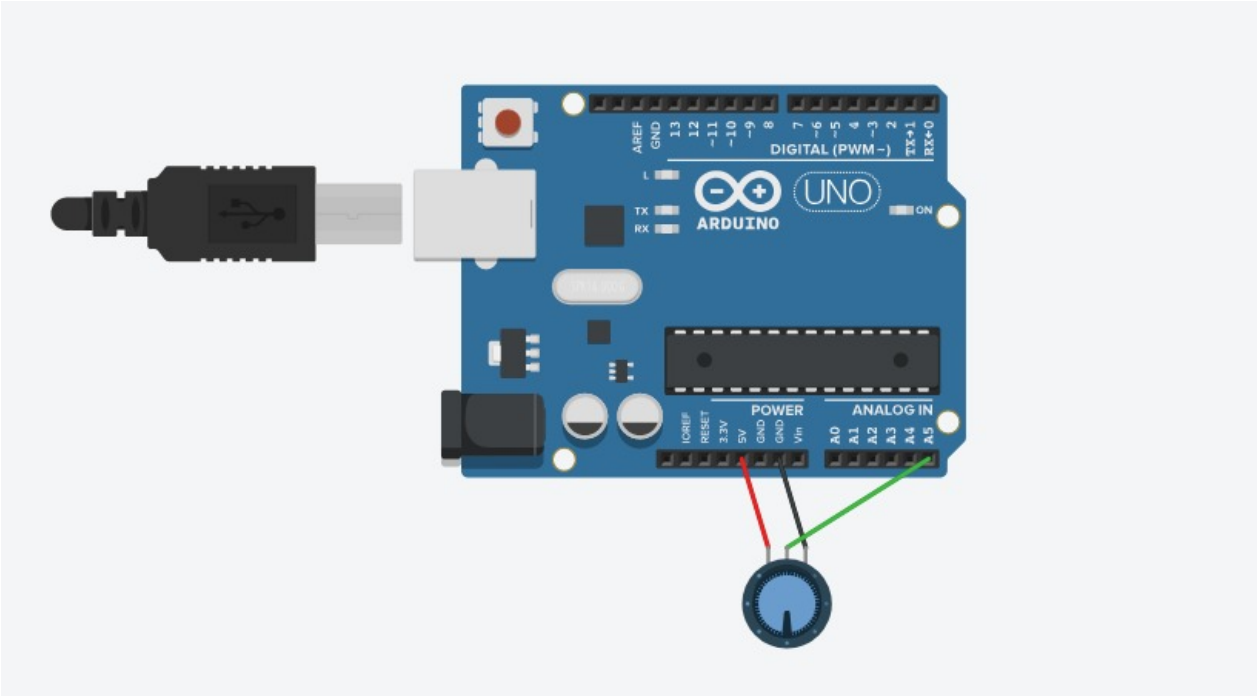
A figura 3 mostra os valores de tensão lidos pelo pino **ADC5** em valores digitais (0 a 1023) e analógicos (0.0V a 4.99V) através do *software CuteCom*.

Figura 3: Conversão analógico-digital



A figura 4 mostra a montagem do circuito, utilizando apenas um potênciometro conectado ao pino A5 (ADC5) do *Arduino*.

Figura 4: Conversão analógico-digital



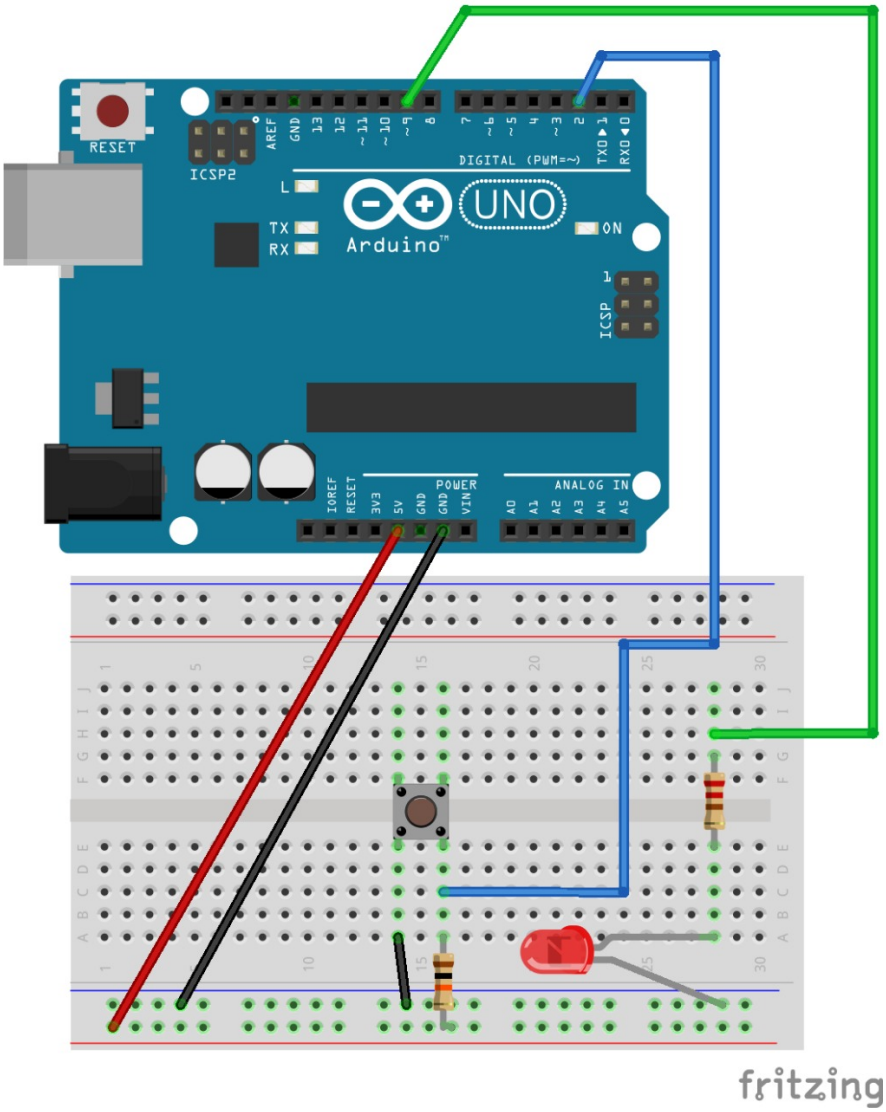
2.4 Interrupções

A parte final deste relatório irá tratar de interrupções no AVR Atmel ATmega2560. Tanto a quarta quanto a quinta atividade são bastante parecidas. Ambas compartilham os mesmos registradores, tendo como diferença apenas o tratamento da interrupção gerada (alterar o estado de um *LED* e manter o estado do *LED* enquanto um botão é pressionado).

A importância de uma interrupção se deve ao fato de que o microcontrolador pode ter que verificar e gerenciar interrupções externas no momento em que elas ocorrem e não fazer essa verificação a cada período de tempo. Caso a ultima opção seja escolhida, o microcontrolador pode não perceber que algum evento externo ocorreu.

Para as duas próximas atividades aqui descritas, o circuito montado foi o mesmo e a figura 5 o apresenta.

Figura 5: Conversão analógico-digital



Seguindo o padrão das outras atividades, uma função de preparação do AVR foi criada, escrevendo os valores necessários nos registradores utilizados. A tabela 4 apresenta os registradores. Como citado acima, estes registradores foram utilizados tanto para quarta como para a quinta atividade.

Tabela 4: Registradores para interrupções

Registrador	Descrição
DDRE	Define o modo de operação dos pinos da porta E
DDRH	Define o modo de operação dos pinos da porta H
PINE	Registra o estado de cada pino da porta E
EICRB	Define de que modo a interrupção deve ser acionada
EIMSK	Habilita as interrupções externas
SREG	Registrador de controle geral do AVR. Seu último *bit* habilita as interrupções globalmente

A função *setupINT()* faz a preparação do microcontrolador para a utilização das interrupções.

```
120 void setupINT(){
121   DDRE &= ~(1 << PE4);  // Pino D2 do arduino como entrada
122   DDRH |= (1 << PH6);   // Pino D9 do arduino como saida
123
124   EICRB = (1 << ISC41); // Definindo a interrupção externo INT4 para ser
125   EICRB = (1 << ISC40); // Acionando na borda de subida do sinal
126   EIMSK = (1 << INT4);  // Habilitando a interrupção externa INT4
127   sei();                // Habilitando o pino de interrupção global
128
129 }
```

Assim como na primeira atividade descrita neste relatório, devemos configurar os pinos de entrada (botão, porta D2 do *Arduino*) e saída (*LED*, porta D9 do *Arduino*). Traduzindo para as portas reais do AVR, devemos escrever '0' no quarto *bit* do registrador **DDRE** e escrever '1' no sexto *bit* do registrador **DDRH**.

Em seguida, definimos que a interrupção irá acontecer na borda de subida do sinal do botão pressionado, ao escrever '1' nos primeiros dois *bits* (**ISC41** e **ISC40**) do registrador **EICRB**. Este registrador é quem define o modo de operação das interrupções externas de 4 a 7. Uma vez configurado o modo de operação da interrupção, devemos habilitá-la, escrevendo '1' no *bit* 4 (**INT4**) do registrador **EIMSK**. A utilização destes *bits* (ISC41, ISC40 e INT4) deve-se ao fato de termos escolhido como entrada o pino D2 do *Arduino*, que representa a interrupção **INT4** do AVR. Por fim, a função *sei()* da biblioteca AVR-LibC é invocada para que o *bit* mais significativo do registrador **SREG** esteja em nível lógico alto, fazendo assim que as interrupções possam ser habilitadas globalmente. Caso contrário, nenhuma interrupção será capturada pelo microcontrolador.

Para ambas atividades, uma função de controle do botão (*debounce()*) foi utilizada para garantir que o pressionamento do botão seja capturado de forma correta.

```
131 bool debounce(){
132   _delay_ms(100);
133   if(PINE & (1 << PE4)){
134     return true;
135   } else {
136     return false;
137   }
138 }
```

Quando invocada, esta função aguarda 100 milissegundos e, caso o botão continue sendo pressionado (o registrador **PINE** com valor '1' no seu *bit* 4, **PE4**) a função retorna *true*, e caso contrário, retorna *false*.

A primeira atividade para o trabalho com as interrupções definia que, quando um botão fosse pressionado, um *LED* deveria alterar seu estado (ligado-desligado).

Cada interrupção deve ser tratada com sua rotina de serviço (ISR). Nesta atividade, as interrupções alteram o valor de saída do *LED* (*bit* 6 da porta H).

```
139 ISR(INT4_vect){
140   if(debounce()){
141     handler_exec4();
142   }
143 }
144
145 void handler_exec4(){
146   PORTH ^= (1 << PH6);
147 }
148
149 void exec4(){
150   setupINT();
151   PORTH &= ~(1 << PH6);
152   while(true){};
153 }
```

Como podemos observar no bloco de código acima, a função *exec4()* inicializa a saída do *LED* com o valor 0 e entra em um laço infinito. Caso o botão de interrupção seja pressionado e a interrupção seja validada pela função *debounce()*, a função *handler_exec4()* irá alterar o valor de saída do *bit* 6 da porta H (*bit* **PH6** do registrador **PORTH**, pino D9 do *Arduino*) através da operação lógica ***XOR***. Ou seja, cada vez que o botão é pressionado, o *LED* apaga e acende.

Concluindo o relatório, a segunda atividade com interrupção exigia que, enquanto o botão permanecesse pressionado, o *LED* continuasse aceso. A diferença para a atividade anterior é apenas a função de tratamento da interrupção (função *handler()*). Os registradores utilizados foram os mesmos que a atividade anterior (tabela 4) e suas inicializações estão na função *setupINT()* descrita anteriormente.

```
154 ISR(INT4_vect){
155   if(debounce()){
156     handler_exec5();
157   }
158 }
159
160 void handler_exec5(){
161   while((PINE & (1 << PE4))){
162     PORTH &= ~ (1 << PH6);
163   }
164   PORTH |= (1 << PH6);
165 }
166
167 void exec5(){
168   setupINT();
169   PORTH &= ~ (1 << PH6);
170   while(true){};
171 }
```

Como na primeira atividade envolvendo interrupções, a função principal *exec5()* inicializa o *LED* apagado e entra em um laço infinito. Quando o botão é pressionado e a interrupção é validada pela função *debounce()*, enquanto o botão permanecer pressionado o *LED* se manterá acesso, escrevendo '1' no sexto *bit* (**PE6**) do registrador **PORTH** (enquanto o quarto *bit* do registrador **PINE** estiver em '1', o *LED* se mantêm apagado, uma vez que o circuito está montado com o circuito em *Pull-Up*).

3 Conclusão

Este relatório apresentou cinco diferentes atividades para um primeiro contato com o AVR Atmel ATmega2560, programando-o com a linguagem de programação *C* através da biblioteca AVR-LibC. Através desta biblioteca, é possível ter um controle total do microcontrolador, diferentemente de utilizar, por exemplo, a plataforma *Arduino* para programar o AVR, uma vez que ficamos presos às funções criadas pela plataforma. As atividades foram desenvolvidas com a IDE Eclipse neon, para facilitar a injeção dos códigos no microcontrolador e com o auxílio do *datasheet* do microcontrolador para verificar a funcionalidade de cada registrador utilizado.

Referências

ATMEL. *Atmel ATmega 640/V-1280/V-1281/V-2560/V-2561/V Datasheet*. [S.l.: s.n.], 2014.

LIMA, C.; VILLAçA, M. *AVR e Arduino: técnicas de projeto*. 2. ed. [S.l.: s.n.], 2012.