Tutorial C para MCU - Capitulo 1

Neste tutorial utilizaremos a arquitetura AVR, as justificativas para tal são:

- Ele possui todo o ferramental disponibilizado pela Atmel em open source pelo GCC/GDB (diferente da arquietetura PIC)
- Caso você queira relizar uma implementação real você encontra um chip Atmel em qualquer lugar e barato, visto que eles são utilizados nas placas Arduino
- Se desejar aprender usando um Arduino sinta-se avontade, a ideia de utilizar o avr-libc para programar é para um questão didatica de manter você o mais próximo do hardware, o que acontece na maioria dos casos de programação de MCU. Raros casos você encontrará uma plataforma de programação amigável como o Arduino, pois ela não tão otimizado

A arquitutura AVR é basesada no RISC de Harvard e sua definição é disponibilizada pela Atmel(Microchip atual proprietária). Eventualmente posso publicar algo mais especifico sobre estas arquiteturas.

Para este tutorial que visa a familiarização da parte técnica com programação em C de baixissimo nível de abstração será apresentado mais o passo a passo técnico.

O documento a seguir esta organizado em:

- Instalação
- Ligar um LED
    - Registradores
- avr-gcc
    - Assembly
    - Executável

Instalação

Para fazer o dowload do ferramental utilizado aqui você pode seguir o tutorial presente no: original documentation of avr lib. Ou usar os seguintes comandos.

```
sudo apt update
sudo apt install binutils-avr
sudo apt install gcc-avr
sudo apt install avr-libc
sudo apt install gdb-avr
```

Depois deste itens instalados estamos com um ambiente pronto para fazer cross compile de códigos para arquitetura AVR e realizar uma simulação com o gdb, caso queira fazer o upload para um chip será necessário outros pacotes

LED On

Para entendermos como programar um Atmega vamos usar como exemplo um código simples e usaremos engenharia reversa para entender um pouco mais.

```
#ifndef F_CPU
#define F_CPU 1000000UL
#endif
#include <avr/io.h>


int main(void){

    DDRB = 0xff;

    while (1){
        PORTB = 0xff;
    }

    return (0);
}
```

Este código simples ajuda a entender bastante a proximidade que devemos ter a placa para conseguir fazer o código funcionar.

Inicialmente fazermos uma definição do clock do MCU:

```
#ifndef F_CPU
#define F_CPU 1000000UL
#endif
```

Incluimos a biblioteca do avr, descrita na documentação original

Finalmente definimos a main. Nela são realizadas as seguintes atividades:

- inserção de 0xff em DDRB
- inserção de 0xff em PORTB

```
    DDRB = 0xff;

    while (1){
        PORTB = 0xff;
    }
```

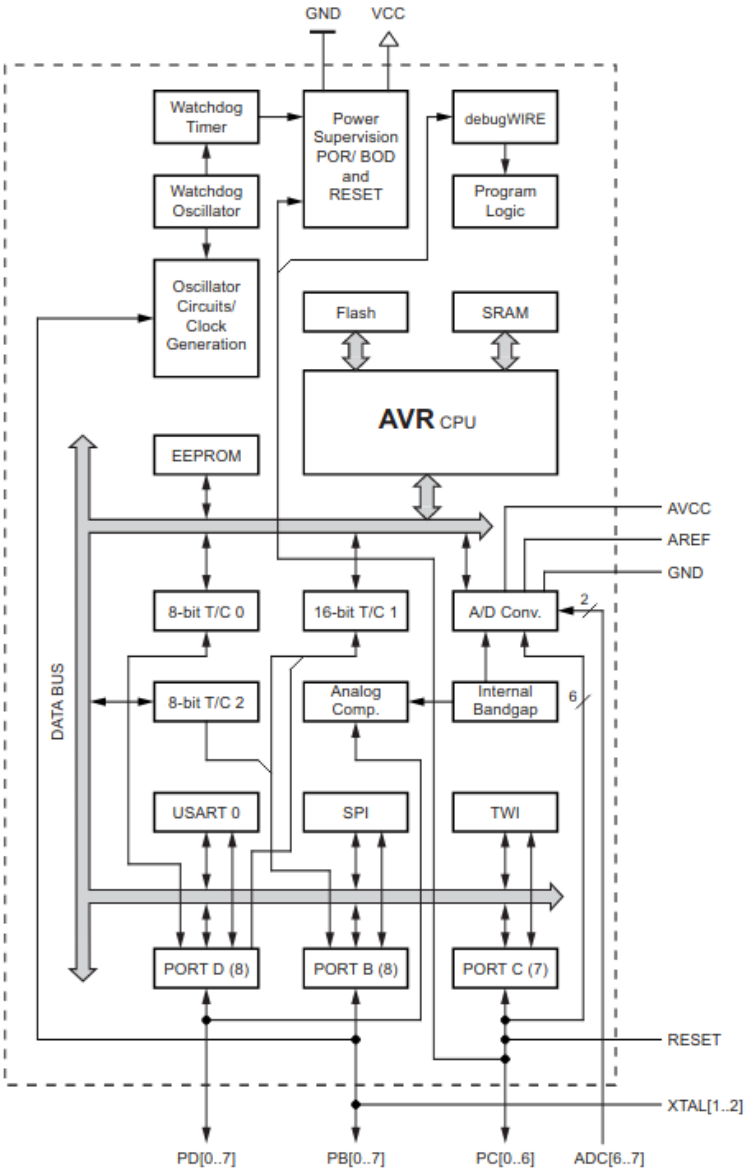
Para entendermos estes comandos vamos explorar um pouco a arquitetura AVR e seus registradores

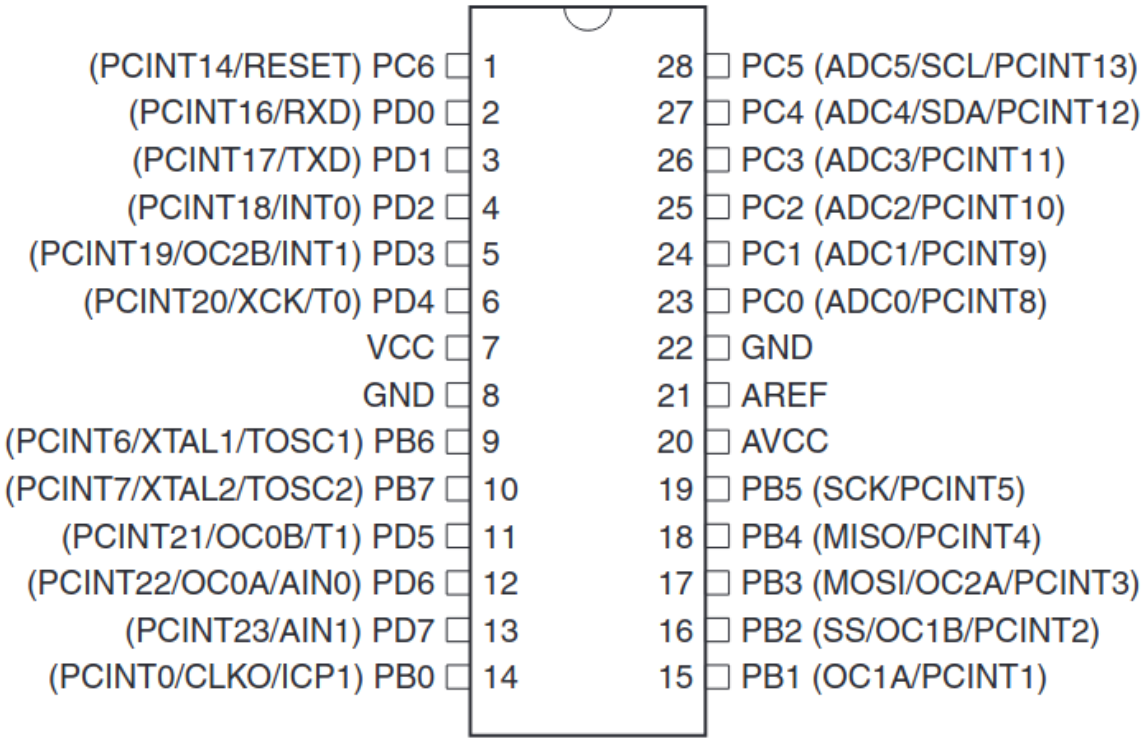
Um pouco da arquitetura AVR

Como exemplo o chip do popular ATmega328P, cuja o diagrama de blocos e o encapsulamento, retirados de datasheets da atmel estão apresentados a seguir (datasheet 1 datasheet 2).

2.1 Block Diagram

Figure 2-1. Block Diagram





Podemos ver através das figuras que o PORTB e o DDRB tem uma relação bem ínitima com o pinout do chip, mas como eles se relacionam?

Estes nomes **PORTx** e **DDRx** são os nomes dados para os registradores relacionados aos pinos **Px**, onde x pertence a [A, B, C, D]. Cada agrumento de pinos tem suas particularidades, que não convém apresentá-las a fundo agora, porém convém entender um pouco mais sobre os registradores, apresentados nesta figura.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x15 (0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14 (0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13 (0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12 (0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11 (0x31)	Reserved	–	–	–	–	–	–	–	–	
0x10 (0x30)	Reserved	–	–	–	–	–	–	–	–	
0x0F (0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E (0x2E)	Reserved	–	–	–	–	–	–	–	–	
0x0D (0x2D)	Reserved	–	–	–	–	–	–	–	–	
0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	101
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	101
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	101
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	100
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	100
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	101
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	100
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	100
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	100

A arquitetura AVR usada neste chuip é um arquitetura 8 bits,e de forma condizente os registradores são de 8 bits. Também é interessante observar que estes registradores estão mapeados no endereço de memória (0x25 para o PORTB).

O **DDRx** é o registrador responsável por dizer se um pino esta funcionando com input ou output. Cada pino (0, 1, 2...) esta associado a um bit do registrador (bits DDB0, DDB1...), e tem sua função setada como input quando seu bit é 0 e output 1.

Já o **PORTx** define o modo de operação do pino, para o modo de input o **PORTx** define: (0) modo de operação normal, e (1) pull-up. Por outro lado, quando o pino esta definido como output o valor colocado no respectivo bit é colocado no pino como saida, ou seja, se o bit PORTB3 = 1, o pino PB3 terá valor 1 como saída (tipicamente 5V), se ele estiver com 0 a saida será 0.

Portanto em nosso exemplo colocamos o todo os pinos PB em modo output e colocamos seus valores lógicos em 1. Se o devido circuito elétrico for montado você conseguirá acender 8 LEDs.

avr-gcc

Agora chegou a parte divertida, iremos executar nosso código.

O avr-gcc é um cross compilador que abrange um ampla gama de chips, portanto sempre é necessário definir o chip. Além disso, aqui colocaremos uma flag para o debbuger para permitir que possamos ler usar o gdb de forma mais tranquila.

```
avr-gcc -mmcu=atmega328 -W -g led.c -o led.out
```

-W e -g passam parametros para o gdb,"led.c" é o nome do arquivo que desejamos compilar. A flag -o permite você definir o nome do objeto gerado, caso n apresentado é gerado um arquivo a.out.

A flag "-mmcu=" é usada para definir o chip que vamos compilar, neste caso o atmege328.

Assembly

```
avr-gcc -mmcu=atmega328 -S led.c -o led.s
```

Este comando vai produzir um arquivo em linguagem assembly, fica a cargo do leitor ler ele para ter uma ideia de como é gerado código objeto. Para entender um pouco melhor consulte a ISA do AVR

Executável

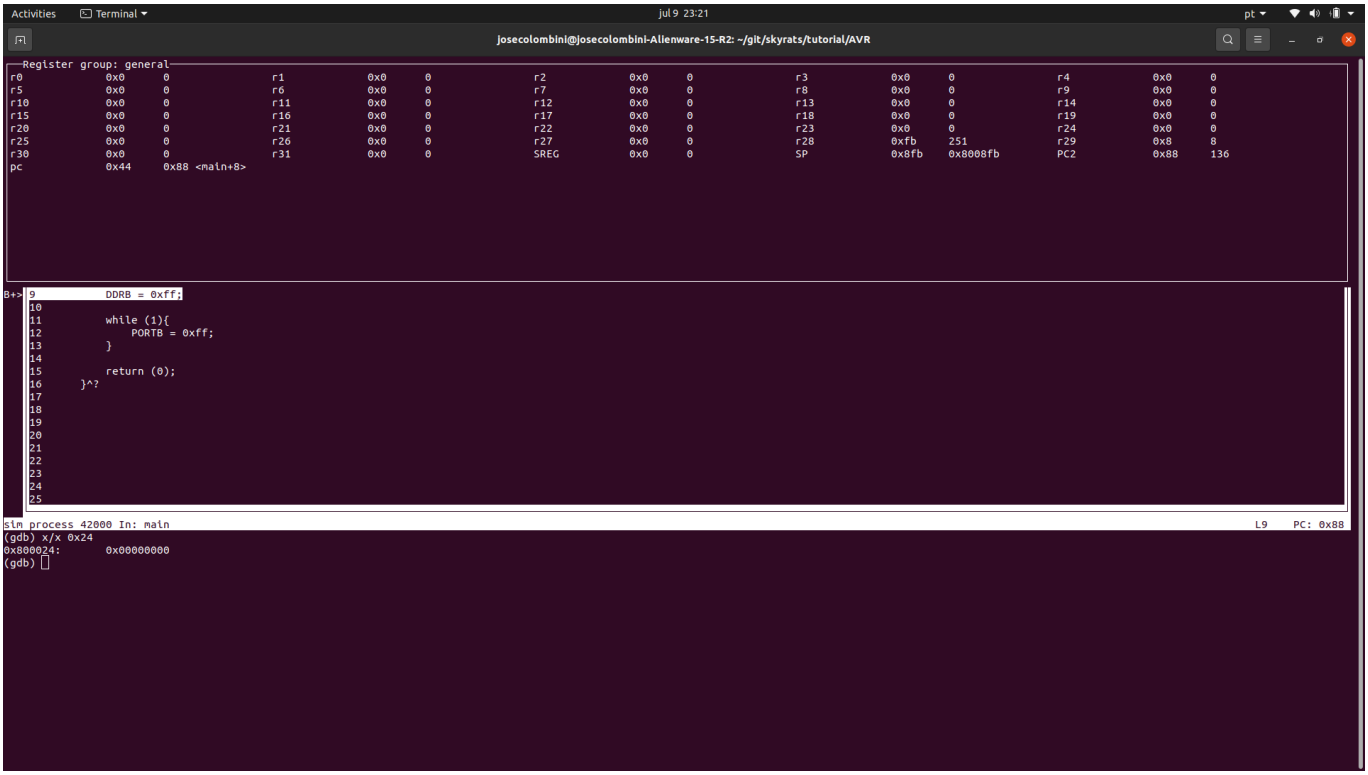
Para executar o código no gdb para podermos ver o esta acontecendo execute

```
avr-gdb led.out
```

Ao abrir o gdb vocẽ precisará definir como simulação e carrear o código no simulador. Coloque um break point no main e de um run.

```
(gdb) target sim
(gdb) load
(gdb) b main
(gdb) r
```

Agora aperte "ctrl+x" e depois 2, desta forma vai aparecer a tabela de registradores de propósito geral e o código (apertar varias vezes essa combinação de botões para apresentar diferentes telas)



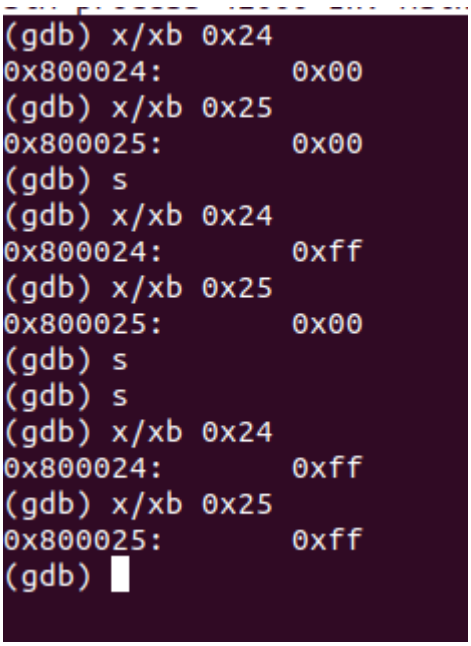
Agora com esta tela em visível vamos utilizar o comando para verificar o que esta no registrador DDRB e PORTB. Como eles são de apenas 1 byte (2 hexadecimais). Vamos ve-los com o seguintes comando x/xb [endereço], que permite ver o em hexadecimal o byte do endereço escolhido.

Para executar um passo use o comando step (s) e para executar a linha marcada.

Se dermos a seguinte lista de comandos teremos o resultado apresentado na figura a seguir

```
(gdb) x/xb 0x24
(gdb) x/xb 0x25
(gdb) s
(gdb) x/xb 0x24
(gdb) x/xb 0x25
(gdb) s
(gdb) s
(gdb) x/xb 0x25
```

O endereço 0x24 apresenta o DDRB e o 0x25 PORTB (como apresentado pela figura dos registrados la em cima). O primeiro step executa a linha **DDRB = 0xFF**, portanto 0x24 se torna FF e somente depois de executar o laço temos que 0x25 (**PORTB**) com FF


(gdb) x/xb 0x24
0x800024: 0x00
(gdb) x/xb 0x25
0x800025: 0x00
(gdb) s
(gdb) x/xb 0x24
0x800024: 0xff
(gdb) x/xb 0x25
0x800025: 0x00
(gdb) s
(gdb) s
(gdb) x/xb 0x24
0x800024: 0xff
(gdb) x/xb 0x25
0x800025: 0xff
(gdb)