EA075 – Material Suplementar ao Projeto Final

Segundo Semestre de 2019

Autora: Wu Shin-Ting

No tutorial de RoboCore, “Robô Seguidor de Linha” [1], são apresentados os códigos usando as funções das bibliotecas disponíveis no IDE (Integrated Development Environment) do Arduino [2], mas, como vimos ao longo do curso, o *hardware* que está por baixo destas funções é um microcontrolador composto de um (ou mais) processador, diferentes unidades de memória e vários circuitos dedicados integrados num mesmo *chip*. Os registradores de controle, de estado e de dados destes módulos são mapeados nos espaços de memória e podemos implementar funcionalidades equivalentes, porém mais eficientes, acessando diretamente os registradores como justifica em [3]. Em [3] é detalhado um projeto de controle de pinos GPIO (*General Purpose I/O*) das três portas D, B e C do microcontrolador Atmega328, com a finalidade de ilustrar esta equivalência.

Neste material suplementar, vou dissecar os códigos apresentados em [1], mostrando como as funções das bibliotecas Arduino são implementadas e explicando alguns operadores da linguagem C usuais na programação de microcontroladores. O principal objetivo deste material é mostrar a conexão dos dois modos de programação em *baremetal* (sem sistema operacional): o modo de linguagem de Arduino, que vou denominar modo IDE [2], e o modo via registradores, que chamarei de modo REG [3,10]. Utilizarei na minha explicação as convenções de pinagem apresentadas na Figura 1 e uma implementação oficial das funções do núcleo Arduino AVR, disponíveis em https://github.com/arduino/ArduinoCore-avr.

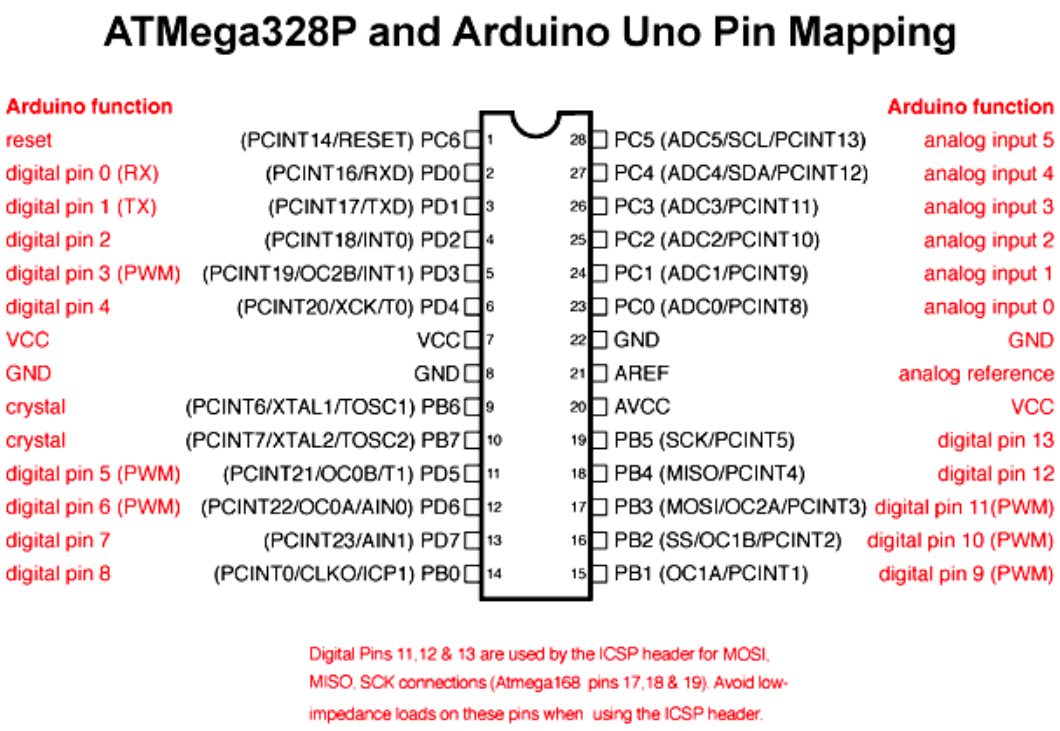

Figura 1: Mapeamento entre Atmega328P e Arduino Uno.

```arduino
const int motorA = 5; //velocidade motor A - de 0 a 255
const int motorB = 6; //velocidade motor B - de 0 a 255
const int dirA = 7; //direcao do motor A - HIGH ou LOW
const int dirB = 8; //direcao do motor B - HIGH ou LOW

void setup() {
  pinMode(motorA, OUTPUT);
  pinMode(motorB, OUTPUT);
  pinMode(dirA, OUTPUT);
  pinMode(dirB, OUTPUT);
}

void loop() {
  digitalWrite(dirA, HIGH); //SENTIDO DE ROTACAO
  digitalWrite(dirB, LOW);
  analogWrite(motorA, 150); //VELOCIDADE
  analogWrite(motorB, 150);
}
```

Figura 2: Primeiro bloco de códigos

I) **Declaração de Variáveis**: As variáveis são do tipo int com qualificador const, que significa que o conteúdo destas variáveis não será alterado. Neste caso, estão nomeados os pinos digitais 5 (PWM), 6 (PWM), 7 e 8 mostrados na Figura 1 com as variáveis (nomes) motorA, motorB, dirA e dirB.

```
const int motorA = 5; //velocidade motor A - de 0 a 255
const int motorB = 6; //velocidade motor B - de 0 a 255
const int dirA = 7; //direcao do motor A - HIGH ou LOW
const int dirB = 8; //direcao do motor B - HIGH ou LOW
```

II) **Inicialização dos módulos e dos pinos que servem os módulos**. Neste caso são os 4 pinos declarados, 5, 6, 7 e 8. Todos eles são configurados como pinos de saída (OUTPUT).

```
void setup() {
  pinMode(motorA, OUTPUT);
  pinMode(motorB, OUTPUT);
  pinMode(dirA, OUTPUT);
  pinMode(dirB, OUTPUT);
}
```

O que a função pinMode faz, com os seus dois argumentos de entrada?

Veja em [4] como a função pinMode é definida. Observe o tipo de dados uint8_t dos argumentos de pinMode, duas primeiras chamadas, o qualificador volatile, e mais duas chamadas usando o argumento port, cujo valor foi recuperado com a macro digitalPinToPort.

```
void pinMode(uint8_t pin, uint8_t mode)
{
        uint8_t bit = digitalPinToBitMask(pin);
        uint8_t port = digitalPinToPort(pin);
       volatile uint8_t *reg, *out;

        reg = portModeRegister(port);
        out = portOutputRegister(port);
:
:
```

```
    } else {
        uint8_t oldSREG = SREG;
        cli();
        *reg |= bit;
        SREG = oldSREG;
    }
```

O tipo de dado uint8_t é um tipo muito usado nos microcontroladores [5]. É *unsigned integer* de 8 *bits type*. Isso significa que todas as variáveis motorA, motorB, dirA e dirB são tratadas internamente como do tipo uint8_t ao invés do tipo int.

II.A. As duas primeiras chamadas são as macros definidas em [6]:

```
#define digitalPinToPort(P) ( pgm_read_byte( digital_pin_to_port_PGM + (P) ) )
#define digitalPinToBitMask(P) ( pgm_read_byte( digital_pin_to_bit_mask_PGM + (P) ) )
```

que nos permitem achar a correpondência entre a nomenclatura do Arduino Uno e a porta e o pino da porta do microcontrolador Atmega328 através dos vetores digital_pin_to_port_PGM[] e digital_pin_to_bit_mask_PGM[]. Em [7] podemos ver como estes vetores são definidos.

```
const uint8_t PROGMEM digital_pin_to_port_PGM[] = {
        PD, /* 0 */
    PD,
    PD,
    PD,
    PD,
    PD,
    PD,
    PD,
    PB, /* 8 */
    PB,
    PB,
    PB,
    PB,
    PB,
    PC, /* 14 */
    PC,
    PC,
    PC,
    PC,
    PC,
};const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[] = {

    _BV(0), /* 0, port D */
    _BV(1),
    _BV(2),
    _BV(3),
    _BV(4),
    _BV(5),
    _BV(6),
    _BV(7),
    _BV(0), /* 8, port B */
    _BV(1),
    _BV(2),
    _BV(3),
```

```
        _BV(4),
        _BV(5),
        _BV(0), /* 14, port C */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
};
```

Os colchetes são notações aceitas pela linguagem C para designar um vetor com inicialização. O compilador alocará automaticamente o espaço de memória necessário para acomodar todos os elementos declarados na inicialização. Neste caso específico, deverão ser alocados automaticamente 20 elementos para cada vetor.

Operações com vetores como
digital_pin_to_port_PGM + (P)
e
digital_pin_to_bit_mask_PGM + (P)
correspondem a soma do endereço inicial do vetor, digital_pin_to_port_PGM ou digital_pin_to_bit_mask_PGM, e a quantidade de bytes de P elementos do vetor. Por exemplo, para motorA (=5), motorB (=6), dirA (=7) e dirB (=8), o valor acessado é o quinto até o oitavo elemento, respectivamente, isto é,

digital_pin_to_port_PGM + (5) = digital_pin_to_port_PGM[5]
digital_pin_to_port_PGM + (6) = digital_pin_to_port_PGM[6]
digital_pin_to_port_PGM + (7) = digital_pin_to_port_PGM[7]
digital_pin_to_port_PGM + (8) = digital_pin_to_port_PGM[8]
digital_pin_to_bit_mask_PGM + (5) = digital_pin_to_bit_mask_PGM[5]
digital_pin_to_bit_mask_PGM + (6) = digital_pin_to_bit_mask_PGM[6]
digital_pin_to_bit_mask_PGM + (7) = digital_pin_to_bit_mask_PGM[7]
digital_pin_to_bit_mask_PGM + (8) = digital_pin_to_bit_mask_PGM[8]

O que são os valores PD, PB e PC do vetor digital_pin_to_port_PGM? São as macros cuja definição pode ser encontrada em [6].

```
#ifdef ARDUINO_MAIN
#define PA 1
#define PB 2
#define PC 3
#define PD 4
```

E o que significa _BV(*bit*)? _BV(*bit*) é uma macro cuja definição pode ser vista em [8]. Ela converte o argumento *bit* num *byte* com 1 somente na posição *bit*.

```
#define _BV(bit) (1 << (bit))
```

Por exemplo, _BV(4) é substituído pelo pré-processador (antes da compilação) por 1<<4 = 0b00010000.

Em [9] podemos ver a definição da macro

```
#define pgm_read_byte(addr) (*(const unsigned char *)(addr)) ,
```

onde (const unsigned char *)(addr) é uma operação que "*casting*" o endereço addr para o endereço do tipo de dados (const unsigned char), do tipo de 8 *bits* sem sinal. E o operador * antes de (const unsigned char) instrui o processador retornar o conteúdo do endereço acessado. Em resumo, a macro pgm_read_byte retorna o conteúdo do endereço acessado no tamanho de um *byte*.

Portanto, ao aplicarmos as operações mostradas nos códigos, teremos como retorno os valores apresentados no extremo direito das expressões

bit =

digitalPinToBitMask(5) = pgm_read_byte (digital_pin_to_bit_mask_PGM + (5)) = pgm_read_byte (digital_pin_to_bit_mask_PGM[5]) = _BV(5) = 0b00100000

digitalPinToBitMask(6) = pgm_read_byte (digital_pin_to_bit_mask_PGM + (6)) = pgm_read_byte (digital_pin_to_bit_mask_PGM[6]) = _BV(6) = 0b01000000

digitalPinToBitMask(7) = pgm_read_byte (digital_pin_to_bit_mask_PGM + (7)) = pgm_read_byte (digital_pin_to_bit_mask_PGM[7]) = _BV(7) = 0b10000000

digitalPinToBitMask(8) = pgm_read_byte (digital_pin_to_bit_mask_PGM + (8)) = pgm_read_byte (digital_pin_to_bit_mask_PGM[8]) = _BV(0) = 0b00000001

port =

digitalPinToPort(5) = pgm_read_byte (digital_pin_to_port_PGM + (5)) = (*(const unsigned char *) (digital_pin_to_port_PGM[5])) = PD

digitalPinToPort(6) = pgm_read_byte (digital_pin_to_port_PGM + (6)) = (*(const unsigned char *) (digital_pin_to_port_PGM[6])) = PD

digitalPinToPort(7) = pgm_read_byte (digital_pin_to_port_PGM + (7)) = (*(const unsigned char *) (digital_pin_to_port_PGM[7])) = PD

digitalPinToPort(8) = pgm_read_byte (digital_pin_to_port_PGM + (8)) = (*(const unsigned char *) (digital_pin_to_port_PGM[8])) = PB

Podemos certificar os resultados das operações através da Figura 1. Veja que os pinos digitais 5, 6, 7 e 8 do Arduino correspondem aos pinos 5, 6 e 7 da porta PD e o pino 0 da porta PB.

II.B A partir dos nomes das portas P são recuperados os endereços dos registradores físicos mapeados no espaço de memória através das duas chamadas.

```
reg = portModeRegister(port);
out = portOutputRegister(port);
```

Essas macros fazem acesso ao elemento P dos vetores port_to_output_PGM[] e port_to_mode_PGM[], como mostram as seguintes definições [6]:

```
#define portOutputRegister(P) ( (volatile uint8_t *)( pgm_read_word( port_to_output_PGM + (P))) )
#define portModeRegister(P) ( (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM + (P))) )
```

Ou seja, esses vetores mapeiam as portas aos registradores especificados na folha técnica do microcontrolador ATMega328P (Seção 13.4 em [10]), como mostram as suas declarações em [7]:

```
const uint16_t PROGMEM port_to_mode_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        (uint16_t) &DDRB,
```

```
        (uint16_t) &DDRC,
        (uint16_t) &DDRD,
}

const uint16_t PROGMEM port_to_output_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        (uint16_t) &PORTB,
        (uint16_t) &PORTC,
        (uint16_t) &PORTD,
};
```

Em [13] são encontradas as definições dos valores PORT* e DDR* encontrados nos dois vetores, como macros:

```
#define PINB _SFR_IO8(0x03)
#define DDRB _SFR_IO8(0x04)
#define PORTB _SFR_IO8(0x05)
#define PIND _SFR_IO8(0x09)
#define DDRD _SFR_IO8(0x0A)
#define PORTD _SFR_IO8(0x0B)
```

Tanto a macro como o endereço-base dos registradores tem as suas definições encontradas em [8]

```
#define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)

# if __AVR_ARCH__ >= 100
# define __SFR_OFFSET 0x00
# else
# define __SFR_OFFSET 0x20
# endif
```

Exemplificando, se a versão da arquitetura AVR é maior ou igual a 100, o endereço do registrador PORTB no espaço de memória é _SFR_IO8(0x05) = (0x05 + 0x20) = 0x25 e o de PORTD é _SFR_IO8(0x0B) = (0x0B+0x20) = 0x2B.

Quanto ao qualificador volatile, ele indica que o conteúdo de uma variável pode ser alterado por eventos externos, como um sinal digital de entrada.

Assim,

reg=

```
portModeRegister(PD) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM + (PD))) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM[PD])) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM[4])) = (uint16_t)&DDRD
portModeRegister(PB) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM + (PB))) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM[PB])) = (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM[2])) = (uint16_t)&DDRB
```

II.C Tendo os endereços dos registradores de controle, mais especificamente dos registradores de direção DDRx e dos registradores de saída PORTx, é possível configurar a direção dos sinais dos pinos a serem utilizados:

```
uint8_t oldSREG = SREG;  //salvar o conteudo do registrador de estado
cli(); // macro para desabilitar a interrupção [6]
*reg |= bit; // configurar o sentido do sinal nos pinos para OUTPUT
SREG = oldSREG;  //restaurar o conteudo do registrador de estado
```

Considerando que as macros dos endereços dos registradores sejam definidas, podemos reduzir as chamadas do modo IDE, que vimos até agora, para as do modo REG, que acessam diretamente os registradores especificados na folha técnica [10], como mostra nas próximas 4 linhas:

```
pinMode(motorA, OUTPUT) = pinMode(5,OUTPUT): *reg |= bit → DDRD |= 0b00100000
pinMode(motorB, OUTPUT) = pinMode(6,OUTPUT): *reg |= bit → DDRD |= 0b01000000
pinMode(dirA, OUTPUT)=pinMode(7,OUTPUT): *reg |= bit → DDRD |= 0b10000000
pinMode(dirB, OUTPUT)=pinMode(8,OUTPUT): *reg |= bit → DDRB |= 0b00000001
```

Aplicando as propriedades de operadores lógicos, podemos compactar ainda mais a função setup() para inicilizar os pinos necessários ao projeto de robô seguidor de linha, usando as macros que já vimos:

```
void setup() {
        DDRD |= _BV(5) | _BV(6)| _BV(7);
        DDRB |= _BV(0);
}
```

que é equivalente a

```
void setup() {
        DDRD |= 0b00100000|0b01000000|0b10000000;
        DDRB |= 0b00000001;
}
```

III) Laço de iterações de execução. Neste caso, os dois motores são configurados com o mesmo sentido de rotação (HIGH) e a mesma velocidade (150).

```
void loop() {
  digitalWrite(dirA, HIGH); //SENTIDO DE ROTACAO
  digitalWrite(dirB, HIGH);
  analogWrite(motorA, 150); //VELOCIDADE
  analogWrite(motorB, 150);
}
```

III.A Uma implementação da rotina digitalWrte se encontra em [4]

```
void digitalWrite(uint8_t pin, uint8_t val)
{
        uint8_t timer = digitalPinToTimer(pin);
        uint8_t bit = digitalPinToBitMask(pin);
        uint8_t port = digitalPinToPort(pin);
        volatile uint8_t *out;
        if (port == NOT_A_PIN) return;
```

```
        // If the pin that support PWM output, we need to turn it off
        // before doing a digital write.
        if (timer != NOT_ON_TIMER) turnOffPWM(timer);
        out = portOutputRegister(port);
        uint8_t oldSREG = SREG;
        cli();
        if (val == LOW) {
            *out &= ~bit;
        else
            *out |= bit;
        }
        SREG = oldSREG;
}
```

Há duas chamadas novas na rotina digitalWrite:

```
uint8_t timer = digitalPinToTimer(pin);
if (timer != NOT_ON_TIMER) turnOffPWM(timer);
```

A macro digitalPinToTimer está definida em [6]

```
#define digitalPinToTimer(P) ( pgm_read_byte( digital_pin_to_timer_PGM + (P) ) )
```

que, pelo que já vimos, acessa o conteúdo do elemento P do vetor digital_pin_to_timer_PGM[] que está definido em [7]:

```
const uint8_t PROGMEM digital_pin_to_timer_PGM[] = {
        NOT_ON_TIMER, /* 0 - port D */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        TIMER2B,
        NOT_ON_TIMER,
        TIMER0B,
        TIMER0A,
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 8 - port B */
        TIMER1A,
        TIMER1B,
        TIMER2A,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 14 - port C */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
};
```

Os valores dos elementos dos vetores são definidos em [6]:

```
#define NOT_ON_TIMER 0
```

```
#define TIMER0A 1
#define TIMER0B 2
#define TIMER1A 3
#define TIMER1B 4
#define TIMER2A 7
#define TIMER2B 8
```

Portanto, lembrando que são atribuídos 7 e 8 às variáveis dirA e dirB, temos

timer =

digitalPinToTimer(7) = pgm_read_byte (digital_pin_to_timer_PGM + (7)) = (*(const unsigned char *) (digital_pin_to_port_PGM[7])) = NOT_ON_TIMER

digitalPinToTimer(8) = pgm_read_byte (digital_pin_to_timer_PGM + (8)) = (*(const unsigned char *) (digital_pin_to_port_PGM[8])) = NOT_ON_TIMER

Vimos em II que essas variáveis dirA e dirB são associadas aos pinos das portas D (PD) e B (PB), respectivamente. Ou seja, os valores das variáveis

out =

portOutputRegister(PD) = (volatile uint8_t *)(pgm_read_word(port_to_output_PGM + (PD))) = (volatile uint8_t *) (pgm_read_word(port_to_output_PGM[PD])) = (volatile uint8_t *)(pgm_read_word(port_to_output_PGM[4])) = (uint16_t)&PORTD

portOutputRegister(PB) = (volatile uint8_t *)(pgm_read_word(port_to_output_PGM + (PB))) = (volatile uint8_t *) (pgm_read_word(port_to_output_PGM[PB])) = (volatile uint8_t *)(pgm_read_word(port_to_output_PGM[2])) = (uint16_t)&PORTB

e das variáveis

bit =

0b10000000

0b00000001

Como o argumento de entrada de digitalWrite é HIGH, podemos reescrever a operação *out |= bit na forma

PORTD |= 0b10000000

PORTB |= 0b00000001

para setar em 1 a saída dos pinos 7 (PD) e 0 (PB), respectivamente.

Embora turnOffPWM(timer) não seja executada nas duas chamadas, vamos dar uma olhada nela. Uma implementação desta função é encontrada em [4]. Veja que, dependendo do tipo de timer, é chamada a macro cbi() para limpar um *bit* específico COM* do registrador TCCR*, ambos especificados na folha técnica fornecida pelos fabricantes [10], como ilustra o seguinte trecho de códigos:

```
switch (timer)
{
:
:
case TIMER0A: cbi(TCCR0A, COM0A1); break;
case TIMER0B: cbi(TCCR0A, COM0B1); break;
:
:
```

```
}
```

A especificação dos endereços dos registradores TCCR* e das macros COM* dos temporizadores é encontrada em [13], como mostra algumas das linhas de códigos transcritas abaixo:

```
#define TCCR0A _SFR_IO8(0x24)
#define WGM00 0
#define WGM01 1
#define COM0B0 4
#define COM0B1 5
#define COM0A0 6
#define COM0A1 7

#define TCCR0B _SFR_IO8(0x25)
#define CS00 0
#define CS01 1
#define CS02 2
#define WGM02 3
#define FOC0B 6
#define FOC0A 7

#define TCNT0 _SFR_IO8(0x26)
#define OCR0A _SFR_IO8(0x27)
#define OCR0B _SFR_IO8(0x28)
```

A definição da macro cbi() se encontra em [12]:

```
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
```

Essencialmente, a função desta macro consiste em limpar o *timer* sfr, resetando o *bit* correspondente no registrador de controle do *timer*, cujo endereço é obtido pela macro _SFR_BYTE(sfr).

```
#define _SFR_BYTE(sfr) _MMIO_BYTE(_SFR_ADDR(sfr))
#define _SFR_ADDR(sfr) _SFR_MEM_ADDR(sfr)
#define _SFR_MEM_ADDR(sfr) ((uint16_t) &(sfr))
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
```

Vimos em II.B que a macro _BV() gera um *byte* com um único *bit* em 1. O operador ~(_BV()) inverte todos os *bits*, ficando somente um *bit* em 0, correspondente ao *bit* a ser resetado.

III.B Uma implementação da rotina analogWrte se encontra em [11]. Essencialmente, ela consiste em setar no pino um valor digital se a entrada val for 0 (0) ou 255 (1), ou um valor analógico se o argumento val for diferente destes dois valores, como mostra o seguinte trecho de códigos:

```
void analogWrite(uint8_t pin, int val)
if (val == 0)
{
        digitalWrite(pin, LOW);
}
else if (val == 255)
```

```
{
        digitalWrite(pin, HIGH);
}
else
{
        switch(digitalPinToTimer(pin));
        {
        :
        case TIMER0A:
        sbi(TCCR0A, COM0A1);
        OCR0A = val; // set pwm duty
        break;
        case TIMER0B:
        sbi(TCCR0A, COM0B1);
        OCR0B = val; // set pwm duty
        break;
        :
        :
        }
}
```

A dissecação da macro digitalPinToTimer feita em III.A nos permite interpretar as seguintes chamadas:
digitalPinToTimer(5) = pgm_read_byte (digital_pin_to_timer_PGM + (5)) = (*(const unsigned char *) (digital_pin_to_port_PGM[5])) = Timer0B
digitalPinToTimer(6) = pgm_read_byte (digital_pin_to_timer_PGM + (6)) = (*(const unsigned char *) (digital_pin_to_port_PGM[6])) = Timer0A

Identificado o *timer* retornado pela macro digitalPinToTimer, Timer0A e Timer0B, é executada uma das duas possíveis alternativas envolvendo a macro sbi() cuja definição é dada em [12]:
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))

Exemplificando, para a chamada
sbi(TCCR0A, COM0A1), temos TCCR0A |= _BV(COM0A1) = _BV(7) = 0b10000000
sbi(TCCR0A, COM0B1), temos TCCR0A |= _BV(COM0B1) = _BV(5) = 0b00100000.
Confira os modos configurados, usando as Tab. 14-3 e 14-6 na Seção 14.9.1 em [10] (questão da segunda avaliação). Lembre que todos os *bits* do registrador TCCR0A estão em 0 na inicialização (*reset*) do microcontrolador (*Initial Value*).

Observe também que o valor que controla a largura do pulso, neste caso 150, é setado nos registradores OCR0A (canal A) (Seção 14.9.4 [10]) e OCR0B (canal B) (Seção 14.9.5 [10]) dentro da rotina analogWrite() após a configuração da função dos pinos para a do *timer* TCCR0.

Note que a função da macro sbi é, de fato, uma função inversa a da macro cbi.

III.C Sintetizando, as instruções da rotina loop(), do modo IDE, podem ser substituídas por acessos diretos aos registradores de controle do modo REG:

```
void loop() {
        PORTD |= 0b10000000;
        PORTB |= 0b00000001;
        TCCR0A  |= 0b10000000;
        OCR0A = 150;
        TCCR0A  |= 0b00100000;
        OCROB = 150;
}
```

Não é muito mais simples, mais eficiente e com menos ocupação de memória?

Segundo Bloco

```
const int motorA = 5; //velocidade motor A - de 0 a 255
const int motorB = 6; //velocidade motor B - de 0 a 255
const int dirA = 7; //direcao do motor A - HIGH ou LOW
const int dirB = 8; //direcao do motor B - HIGH ou LOW

void setup() {
  pinMode(motorA, OUTPUT);
  pinMode(motorB, OUTPUT);
  pinMode(dirA, OUTPUT);
  pinMode(dirB, OUTPUT);
}

void loop() {

  digitalWrite(dirA, HIGH); //SENTIDO DE ROTACAO
  digitalWrite(dirB, HIGH);
  analogWrite(motorA, 150); //VELOCIDADE
  analogWrite(motorB, 150);

  delay(1000);

  analogWrite(motorA, 0);
  analogWrite(motorB, 0);

  delay(1000);

  digitalWrite(dirA, LOW);
  digitalWrite(dirB, LOW);
  analogWrite(motorA, 150);
  analogWrite(motorB, 150);

  delay(1000);

  analogWrite(motorA, 0);
  analogWrite(motorB, 0);

  delay(1000);
}

```

Neste segundo bloco de códigos os sentidos de rotação são alternados periodicamente num espaçamento de tempo de 1000 milisegundos. Analisando as funções desse segundo bloco de códigos, a única nova função é a função delay() cuja implementação é encontrada em [14]:

```
void delay( unsigned long ms )
{
	if (ms == 0)
	{
		return;
	}
	uint32_t start = micros();
	while (ms > 0)
	{
		yield(); // não tem efeito no Uno, pois só ha um thread.
		while (ms > 0 && (micros() - start) >= 1000)
		{
		ms--;
		start += 1000;
		}
	}
}
```

Por esta implementação, podemos concluir que a função delay() ocupa desnecessariamente o processador, verificando periodicamente (por *polling*) a contagem de um contador do sistema através da função micros(), e compare-a com a quantidade de contagem desejada.

O que acham de gerar interrupções periódicas [15] e alternar o sentido e a velocidade dentro das rotinas de interrupção?

Referências:

[1] Robô Seguidor de Linha. https://www.robocore.net/tutoriais/robo-seguidor-de-linha

[2] Arduino Programming Language. https://www.arduino.cc/reference/en/#page-title

[3] Tutorial: Arduino Port Manipulation. https://tronixstuff.com/2011/10/22/tutorial-arduino-port-manipulation/

[4] wiring_digital.c. https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/wiring_digital.c

[5] stdint.h. https://github.com/esp8266/Arduino/blob/master/tools/sdk/libc/xtensa-lx106-elf/include/stdint.h

[6] Arduino.h. https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/Arduino.h

[7] pins_arduino.h. https://github.com/arduino/ArduinoCore-avr/blob/master/variants/standard/pins_arduino.h

[8] sfr_defs.h. https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/sfr_defs.h

[9] pgmspace.h. https://github.com/arduino/ArduinoCore-arc32/blob/master/cores/arduino/avr/pgmspace.h

[10] Atmega328P Datasheet. ftp://ftp.dca.fee.unicamp.br/pub/docs/ea075/datasheet/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[11] wiring_analog.c. https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/wiring_analog.c

[12] wiring_private.h. https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/wiring_private.h

[13] iom328p.h. https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom328p.h

[14] delay.c. https://github.com/arduino/ArduinoCore-samd/blob/master/cores/arduino/delay.c

[15] Arduino 101: Timers and Interrupts. https://www.robotshop.com/community/forum/t/arduino-101-timers-and-interrupts/13072