INSTITUTO FEDERAL
Santa Catarina

Avaliação 02

Análise de classes *GPIO* no 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

Abril de 2019

1 Introdução

Este relatório apresenta quatro diferentes implementações de classe para o uso dos pinos de entrada e saída do AVR **Atmel ATmega2560** na placa *Arduino Mega*. 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

As implementações foram desenvolvidas 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 as implementações feitas, bem como uma análise de desempenho de cada implementação. Todas as análises serão feitas com base no mesmo programa de teste.

2 Implementações

2.1 GPIO versão 1.1

A primeira versão desenvolvida da classe GPIO tem como atributos apenas a identificação do pino na placa *Arduino Mega* e a posição do *bit* do pino para o uso dos registradores de controle de GPIO do AVR. Desta forma, no construtor do objeto da classe e em cada método da classe, faz-se necessário implementar uma estrutura de *switch case*, afim de verificar qual pino do *Arduino mega* está sendo utilizado e associá-lo ao respectivo registrador. O código abaixo mostra uma parte da implementação do construtor da classe.

```
GPIO::GPIO(uint8_t id, PortDirection_t dir):
_id(id)
{
  switch (_id) {
  case 0:
  case 1:
    _bit = id;
    if (dir)
      DDRE |= (1 << _bit);
    else
      DDRE &= ~(1 << _bit);
    break;

  case 2:
  case 3:
    _bit = id + 2;
    if (dir)
      DDRE |= (1 << _bit);
    else
      DDRE &= ~(1 << _bit);
    break;
    .
    .
    .

  case 13:
    _bit = id - 6;
    if (dir)
      DDRB |= (1 << _bit);
    else
      DDRB &= ~(1 << _bit);
    break;

    }
}
```

Como é possível verificar, na medida que a implementação se estende para todos os pinos da placa, o código aumenta gradualmente, bem como o uso de memória de programa do AVR.

Tabela 1: Uso de memória da versão implementada

GPIO_v1.1	
Construtor	978 *bytes* na memória de programa (aproximadamente)
Método set()	616 *bytes* na memória de programa (aproximadamente)
Objeto	2 *bytes* na memória de dados e 16 *bytes* na memória de programa
Memória de programa	3520 *bytes*
Memória de dados	10 *bytes*

O uso de memória de programa para esta implementação é bastante alto, uma vez que cada método precisa de muitas instruções para serem executados. Para um programa que execute

vários métodos desta classe, com vários objetos instanciados, a memória de programa do AVR pode não ser suficiente, dependendo do modelo do AVR a ser utilizado.

2.2 GPIO versão 1.2

Além do pino do *Arduino* e da posição do *bit* utilizado nos registradores, a segunda versão da classe GPIO tem como atributos ponteiros para os próprios registradores de controle de GPIO (**DDR**, **PIN** e **PORT**), a fim de diminuir o uso de memória de programa do AVR.

Desta forma, a estrutura *switch case* é utilizada apenas no construtor do objeto, facilitando a implementação e diminuindo o uso da memória de programa nos métodos da classe.

O código abaixo mostra parte de implementação do construtor da classe e a implementação do método set().

```
36 GPIO::GPIO(uint8_t id, PortDirection_t dir):
37 _id(id)
38 {
39   switch (_id) {
40   case 0:
41   case 1:
42     _bit = (1 << id);
43     _pin = &PINE;
44     _ddr = &DDRE;
45     _port = &PORTE;
46     break;
47
48   case 2:
49   case 3:
50     _bit = (1 << (id + 2));
51     _pin = &PINE;
52     _ddr = &DDRE;
53     _port = &PORTE;
54     break;
55   }
56
57   if (dir)
58     *_ddr |= _bit;
59   else
60     *_ddr &= ~_bit;
61   }
62
63 void GPIO::set(bool val) {
64   if (val)
65     *_port |= _bit;
66   else
67     *_port &= ~_bit;
68 }
```

Esta implementação diminui a memória de programa utilizada, tanto para o construtor quanto para os demais métodos. Entretanto, como cada objeto irá armazenar os registradores de GPIO, o uso da memória de dados aumentou.

Comparando a segunda versão com a primeira, é perceptível a diminuição do uso de memória de programa. Ainda assim, esta implementação utiliza oito *bytes* de memória de dados para cada objeto instanciado e os mesmos ponteiros para os registradores podem ser armazenados até oito vezes, caso todos os pinos de uma porta do AVR esteja sendo utilizado, ou seja, esta versão armazena dados redundantes.

Tabela 2: Uso de memória da versão implementada

GPIO_v1.2	
Construtor	728 *bytes* na memória de programa (aproximadamente)
Método set()	118 *bytes* na memória de programa (aproximadamente)
Objeto	8 *bytes* na memória de dados e 16 *bytes* na memória de programa
Memória de programa	2724 *bytes*
Memória de dados	40 *bytes*

2.3 GPIO versão 1.3

A terceira implementação de classe GPIO visa contornar o problema de armazenamento de dados redundantes encontrado na versão anterior.

Esta versão possui uma classe auxiliar (chamada de **GPIO_Port**) que armazena os endereços dos registradores de controle de GPIO. Desta forma, para instanciar um pino de uma porta qualquer, serão utilizados seis *bytes* da memória de dados. Entretanto, quando um segundo pino for instanciado, apenas mais quatro *bytes* serão utilizados da memória de dados, uma vez que a porta, comum para os dois pinos, já foi guardada em memória. Em outras palavras, o primeiro pino de uma porta ”custa” seis *bytes* da memória de dados e cada pino instanciado a mais ”custará” quatro *bytes* (um *byte* para a identificação do pino no *Arduino*, um *byte* para a posição do *bit* do pino nos registradores e dois *bytes* para o ponteiro que guarda as informações dos registradores da porta) para armazenar seus atributos.

É importante salientar que esta versão não resolve o problema de utilizar *switch case* no construtor da classe, uma vez que esta implementação visa diminuir apenas o uso de memória de dados.

```
70 GPIO::GPIO(uint8_t id, PortDirection_t dir):
71 _id(id)
72 {
73   switch (_id) {
74   case 0:
75   case 1:
76     _bit = (1 << _id);
77     _Px = GPIO_PORT::PE;
78     break;
79   }
80   _Px->dir(_bit, dir);
81
82 void GPIO::set(bool val) {
83   _Px->set(_bit, val);
84 }
85
86 void GPIO_Port::dir(uint8_t p, bool io) {
87     if (io)
88         ddr |= p;
89     else
90         ddr &= ~p;
91 }
92
93 void GPIO_Port::set(uint8_t p, bool val) {
94     if (val)
95         port |= p;
96     else
97         port &= ~p;
98 }
```

Como visto no código acima, a implementação dos métodos da classe ocorrem de fato no na classe auxiliar implementada, a partir do atributo que indica qual porta cada pino do *Arduino*

está associado (variável **_Px** do código).

Tabela 3: Uso de memória da versão implementada

GPIO_v1.3	
Construtor	568 *bytes* na memória de programa (aproximadamente)
Método set()	132 *bytes* na memória de programa (aproximadamente)
Objeto	4 a 6 *bytes* na memória de dados e 16 *bytes* na memória de programa
Memória de programa	2738 *bytes*
Memória de dados	28 *bytes*

Apesar de utilizar mais memória de programa, esta versão é mais vantajosa que a anterior, uma vez que haverá economia no uso de memória de dados para cada pino do *Arduino* utilizado.

2.4 GPIO versão 2

A implementação desta versão da classe GPIO visa retirar do construtor a estrutura *switch case*, que é responsável por uma boa parte do uso da memória de programa. A classe auxiliar GPIO_Port possui dois vetores que guardam a informação de porta e posição do *bit* nos registradores da porta a qual o pino do *Arduino* está relacionado. Desta maneira, toda a estrutura do *switch case* feita nas versões anteriores é substituída por duas buscas em dois vetores e o tamanho do construtor da classe é reduzido a aproximadamente metade do tamanho da versão apresentada anteriormente, além de deixar o código mais ”limpo”.

```
99  GPIO::GPIO(uint8_t id, PortDirection_t dir)
100 {
101   _bit = GPIO_PORT::id_to_bit[id];
102   _port = GPIO_PORT::AllPorts[GPIO_PORT::id_to_port[id]];
103   _port->dir(_bit, dir);
104 }
```

Esta classe armazena apenas a posição do *bit* nos registradores do AVR e um ponteiro para os registradores que controlam o pino escolhido.

Tabela 4: Uso de memória da versão implementada

GPIO_v2	
Construtor	218 *bytes* na memória de programa (aproximadamente)
Método set()	132 *bytes* na memória de programa (aproximadamente)
Objeto	3 a 39 *bytes* na memória de dados e 16 *bytes* na memória de programa
Memória de programa	2444 *bytes*
Memória de dados	51 *bytes*

Ao inicializar um pino, os vetores que fazem a tradução de identificação do pino no *Arduino* para os registradores do AVR são carregados na memória de dados. Desta forma, obrigatoriamente 39 *bytes* são utilizados. A partir disto, cada pino instanciado irá utilizar três *bytes* da memória de dados, uma vez que as informações dos registradores já estão armazenadas. Esta versão é a versão que menos utiliza memória de programa e a que menos utiliza memória de dados (a partir do primeiro pino instanciado).

2.5 GPIO versão 3

A última versão implementada da classe GPIO é uma extensão da versão anterior, expandida para a utilização de todos os pinos de entrada e saída do *Arduino mega*. Esta versão

tem duas implementações: uma que armazena os vetores de tradução dos pinos do *Arduino* na memória de dados e outra que armazena os vetores na memória de programa. A tabela 5 mostra a diferença no uso das memórias.

Tabela 5: Uso de memória da versão implementada (dados aproximados)

GPIO_v3		
	PROGMEM	SEM PROGMEM
Construtor	254 *bytes* na memória de programa	224 *bytes* na memória de programa
Método set()	132 *bytes* na memória de programa	132 *bytes* na memória de programa
Objeto	3 a 25* *bytes* 16** *bytes*	3 a 165* *bytes* 16** *bytes*
Memória de programa	2606 *bytes*	2570 *bytes*
Memória de dados	37 *bytes*	177 *bytes*
* memória de dados ** memória de programa		

Cada vetor de tradução possui setenta *bytes*. Desta forma, fica evidente a diferença de uso da memória de dados. A diferença na quantidade de *bytes* da memória de dados utilizada é exatamente a soma dos dois vetores de tradução de pinos do *Arduino*. Como os vetores de tradução são estáticos, não existe razão para guardá-los na memória de dados, poupando-a para outros usos.

3 Conclusão

Este relatório apresentou quatro diferentes implementações para o uso de GPIO no AVR Atmel ATmega2560. O desenvolvimento mostrou que, um uso a principio simples de portas de entrada e saída pode gerar inúmeras implementações, com suas respectivas vantagens e desvantagens. Os recursos escassos que um microcontrolador possuí deve ser levado em consideração na hora de decidir qual a melhor implementação para determinado uso. Alguns métodos das classes implementados podem ser melhorados, como o método *clear()*, que tem como função zerar um pino do *Arduino*. Este método utiliza dentro de sua implementação o método *set()*. Desta forma, o método *clear()* é maior que o método *set()*, uma vez que o compilador gera instruções de dois métodos ao mesmo tempo (*clear()* e *set()*). Uma forma de melhorar este método é não utilizar o método *set()* e alterar os registradores diretamente no método *clear()*. Estas mudanças sutis podem fazer diferença no uso de memória de programa e memória de dados e melhorar a eficiência de microcontroladores e sistemas embarcados.

Referências

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