INSTITUTO FEDERAL
Santa Catarina

Avaliaçao 02

Análise prática de versões da classe GPIO usando C++ e AVR-Libc

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

Aluna
Maria Fernanda Silva Tutui

Maio de 2019

1 Introdução

O seguinte relatório apresenta uma análise de versões classes de GPIO. Todas as classes possuem a mesma função, implementar o acesso das portas no microcontrolador AVR Atmel Atmega2560 usando a placa Arduino Mega. Todas as versões foram feitas utilizando o ambiente de desenvolvimento Eclipse neon modificado e o mesmo código para teste (*main()*). Cada uma das 5 versões será apresentada separadamente em subseções na seção a seguir a fim de expor suas particularidades para que seja feita uma análise comparativa de desempenho. Ao fim, a ultima seção apresentará as conclusões.

2 Implementações

2.1 GPIO_v1.1

A primeira versão da implementação da classe GPIO faz o uso de uma estrutura de condição. Essa estrutura, o *switch/case*, define o código a ser executado com base em uma comparação de valores. Nela, o número do pino correspondente na placa Arduino Mega, chamado de *_bit*, e é o valor responsável por definir a posição do *_bit* no registrador de controle do AVR específico responsável pelo pino.

A classe trata a variável *_bit* de forma a armazenar o *byte* deslocado para que o mesmo seja tratado futuramente como um valor decimal, isso otimiza a utilização da memória na execução do programa. A mesma implementação foi replicada para todos os métodos, porém, o mapeamento dos pinos deixa o código extenso influenciando no consumo da memória de programa do AVR. Uma análise dos dados referentes a essa primeira implementação pode ser observada na Tabela 1 abaixo.

GPIO_v1.1		
Construtor	942 *bytes*	
Método *set()*	616 *bytes*	
Objeto	36 *bytes**	2 *bytes***
Memória de programa	3520 *bytes*	
Memória de dados	10 *bytes*	
*memória de programa **memória de dados		

Tabela 1: Uso aproximado de memória na versão 1.1.

2.2 GPIO_v1.2

A fim de diminuir o uso da memória de programa do AVR, a segunda versão da classe GPIO utiliza a estrutura de *switch/case* somente para o construtor do objeto por adicionar a técnica de ponteiros para os próprios registradores. Essa abordagem facilita a implementação do código e diminui o uso da memória nos métodos da classe.

Em comparação com a primeira implementação pode ser observada uma diminuição da quantidade de memória de programa de 3520 *bytes* para 2724 *bytes* (valores aproximados). Esse e os demais dados sobre essa versão podem ser conferidos na Tabela 2, apresentada na página a seguir.

GPIO_v1.2		
Construtor	724 *bytes*	
Método *set()*	126 *bytes*	
Objeto	36 *bytes**	8 *bytes***
Memória de programa	2724 *bytes*	
Memória de dados	40 *bytes*	
*memória de programa **memória de dados		

Tabela 2: Uso aproximado de memória na versão 1.2.

Essa versão apresenta um aumento na memória de dados do objeto, uma vez que o mesmo armazenará os registradores de GPIO. Ao armazenar os registradores de GPIO, caso haja a utilização de mais de um pino de uma porta do AVR dados redundantes estarão presentes, o que não é interessante para a implementação.

2.3 GPIO_v1.3

A terceira versão da implementação da classe GPIO ainda faz o uso do *switch/case* porém, apresenta uma melhora quanto ao tamanho da memória de dados em relação a segunda versão. Ainda que exista um aumento na memória de programa, de 2724 *bytes* para 2738 *bytes* (aproximadamente), para a terceira versão, ele se torna ínfimo quando comparado ao benefício obtido com a diminuição da memória de dados de objeto. Todos os dados sobre o uso aproximado de memória podem ser conferidos abaixo na Tabela 3.

GPIO_v1.3		
Construtor	532 *bytes*	
Método *set()*	140 *bytes*	
Objeto	36 *bytes**	4 *bytes***
Memória de programa	2738 *bytes*	
Memória de dados	28 *bytes*	
*memória de programa **memória de dados		

Tabela 3: Uso aproximado de memória na versão 1.3.

Para que essa diminuição do uso da memória de dados ocorra essa versão faz o uso de outra classe usada como suporte chamada GPIO_Port. Essa nova classe surge para tratar os dados redundantes citados na implementação anterior. Isso é feito armazenando os endereços dos registradores de controle de GPIO, dessa maneira, para instanciar um pino, 6 *bytes* da memória de dados serão utilizados, não 8 como anteriormente. Caso um segundo pino for instanciado, apenas mais 4 *bytes* serão utilizados na memória de dados, isso acontece pois a porta compartilhada pelos dois pinos já foi alocada em memória.

2.4 GPIO_v2

A versão chamada GPIO_v2, quarta versão geral a ser abordada no relatório, tem como objetivo a retirada do uso do *switch/case* do método construtor da classe. A classe auxiliar, GPIO_Port, guarda as informações de porta e posição do bit nos registradores da porta referente ao pino no Arduino por meio de dois vetores.

A utilização desses vetores impacta diretamente na alocação da memória *flash*, trazendo benefícios para o programa. A troca da estrutura de *switch/case* pelo uso dos vetores leva consigo outras estruturas de condição como if/else, por exemplo, compactando o programa e contribuindo também no uso da memória. Logo, a partir dessa substituição (*switch/case* por vetores) é observada uma enorme redução no uso da memória de programa, que pode ser observada na Tabela 4 a seguir.

GPIO_v2		
Construtor	216 *bytes*	
Método *set()*	90 *bytes*	
Objeto	36 *bytes**	3 *bytes***
Memória de programa	2444 *bytes*	
Memória de dados	51 *bytes*	
*memória de programa **memória de dados		

Tabela 4: Uso aproximado de memória na versão 2.

A memória de dados é utilizada assim que um pino é inicializado, pois os vetores que fazem a tradução e identificação pino no Arduino - registradores AVR são carregados na memória de dados. Tornando-se a versão com o menor tamanho para a memória de dados para objeto.

2.5 GPIO_v3

A quinta e ultima versão para a implementação da classe GPIO trata exatamente de uma amplificação da anterior, nela todos os 70 pinos presentes no Arduino Mega foram discriminados em GPIO_Port. Na Tabela mostrada abaixo essa versão pode ser identificada como *Sem PROGMEM*. A versão *Com PROGMEM* trata a estrutura de arrays na memória RAM e as força para a memória *flash*.

A melhoria utilizando o PROGMEM pode ser observada a partir da diminuição visível do uso da memória de dados. Isso foi possível pois, os vetores de tradução não precisam ser guardados na memória RAM, são estáticos, reduzindo a alocação da mesma.

GPIO_v3				
	Sem PROGMEM		Com PROGMEM	
Construtor	194 *bytes*		248 *bytes*	
Método *set()*	90 *bytes*		90 *bytes*	
Objeto	20 *bytes**	3 *bytes***	36 *bytes**	3 *bytes***
Memória de programa	2570 *bytes*		2606 *bytes*	
Memória de dados	177 *bytes*		37 *bytes*	
*memória de programa **memória de dados				

Tabela 5: Uso aproximado de memória na versão 3.

3 Conclusões

O desenvolvimento do relatório deixa claro que mudanças aparentemente pequenas na estrutura de código das classes podem gerar enormes impactos quanto ao uso das memórias de programa e dados no AVR. Vale ressaltar que todos os pinos foram implementados na versão GPIO_v3 porém, o mesmo não aconteceu nas versões anteriores. Uma expansão para todos os pinos nessas versões impactaria no tamanho da memória de programa e traria uma complexidade enorme à implementação.

Pode-se observar também que a implementação do método *clear()* é maior, em termos de alocação de memória de programa, do que a do método *set()*, isso ocorre devido a adição da instrução de zerar o registrador no *clear()* além de usar o próprio método *set()* na mesma. Uma possível abordagem para a diminuição no tamanho do *clear()* seria utilizar somente o método *set()*, onde ele seria o responsável por zerar o registrador.

Essas observações são importantes devido a escassez de recursos do microcontrolador e valiosas para que as análises das versões sejam feitas a fim de identificar as consequências das mudanças feitas.