Operações bit a bit e Campos de bit

Módulo 9 Aula 3

Linguagem C, o Curso Definitivo WR Kits

Autor: Dr. Eng. Wagner Rambo

Operadores bit a bit (técnica de operação *bitwise*)

A Linguagem C conversa muito bem com os dois mundos – conforme já exaltamos diversas vezes – sendo o alto nível de abstração, trabalhando muito bem com funções que se aproximam bastante da linguagem do ser humano; e, talvez o mais importante, funciona nos baixos níveis de abstração, conversando intimamente com a máquina. As operações bit a bit comprovam que podemos nos aproximar bastante da linguagem de máquina, por permitir a manipulação dos mesmos de forma individual. Este recurso é especialmente útil quando precisamos programar algum microcontrolador, que apresentará registradores organizados em bits (ainda temos muitos processadores excelentes com arquitetura de 8 bits). Na Tabela 1 apresentamos os operadores que C disponibiliza para manipulação de bits.

Operador	Descrição
~	Inversão de bits, complementa todos os bits, o que é 0 vira 1 e o que é 1 vira 0.
>>	Deslocamento à direita (shift right). Desloca todos os bits para direita, de acordo com o número utilizado.
<<	Deslocamento à esquerda (shift left). Desloca todos os bits para esquerda, de acordo com o número utilizado.
&	Operação E (AND) bit a bit.
^	Operação OU-Exclusivo (XOR) bit a bit.
|	Operação OU (OR) bit a bit.

Tabela 1 - Operadores de manipulação bit a bit em ordem de precedência.

Na Tabela 1, os operadores aparecem na ordem de precedência para expressões, sendo o ~ o de maior precedência e o | o de menor. O código do Box 1 demonstra a utilização do operador de complemento.

```
main()
{
  unsigned char registro1 = 0x48,  /* reg1: 01001000b */
                registro2;

  registro2 = ~registro1;          /* reg2: 10110111b */

  printf("reg1: %X\n", registro1);
  printf("reg2: %X\n", registro2);

}
```

Box 1 - Teste do operador de complemento ~.

Confira no Box 2, um exemplo para teste dos operadores >> e <<.

```
main()
{
  unsigned char registro1 = 0x35,   /* reg1: 00110101b */
                registro2,
                registro3;

  registro2 = registro1 >> 1;       /* reg2: 00011010b */

  registro3 = registro1 << 2;       /* reg3: 11010100b */

  printf("reg1: %X\n", registro1);
  printf("reg2: %X\n", registro2);
  printf("reg3: %X\n", registro3);

}
```

Box 2 - Exemplo para teste de shift right e shift left.

O programa do Box 3, serve para testar os operadores &, ^ e |.

```
main()
{
  unsigned char registro1 = 0xC3,   /* reg1: 11000011b */
                registro2 = 0x6B,   /* reg2: 01101011b */
                registro3,
                registro4,
                registro5;

  registro3 = registro1&registro2;  /* reg3: 01000011b */
  registro4 = registro1^registro2;  /* reg4: 10101000b */
  registro5 = registro1|registro2;  /* reg5: 11101011b */

  printf("reg1: %X\n", registro1);
  printf("reg2: %X\n", registro2);
  printf("reg3: %X\n", registro3);
  printf("reg4: %X\n", registro4);
  printf("reg5: %X\n", registro5);

}
```

Box 3 - Exemplo para teste de &, ^ e |.

Os operadores bit a bit podem ser combinados em expressões e também utilizados no contexto de funções. A função apresentada no Box 4 recebe um byte e imprime ele em formato binário no console.

```
void print_bin(unsigned char reg)
{
  register int i;

  for(i=7;i>=0;i--)
    (reg>>i)&1 ? putchar('1') : putchar('0');

  putchar('b');
  putchar('\n');

}
```

Box 4 - Função para mostrar o valor binário do byte passado como parâmetro.

Como sugestão de exercício, experimente testar a função nos códigos exemplo dos boxes anteriores, para visualizar melhor as operações bit a bit.

Máscara de bits

Os operadores bit a bit podem ser utilizados no conceito de máscara de bits. Basicamente, aplicamos eles quando desejamos preservar/remover, concatenar ou inverter bits pontuais. Para preservar/remover, utilize o operador AND (&), como no exemplo do Box 5. Pelo fato da lógica E sempre resultar em 0 quando uma das entradas for 0, utilizaremos 1 quando o desejo for preservar determinado bit e 0 quando o desejo for remover.

```
main()
{
  unsigned char registro1 = 0x32;

  print_bin(registro1);  /* 00110010b */
  registro1 &= 0xF0;     /* preserva os 4 bits mais significativos*/
  print_bin(registro1);  /* 00110000b */
}
```

Box 5 - Aplicando máscara de bits com &.

Observe que o código do Box 5 requer o suporte da função *print_bin* do Box 4 para vermos cada bit dos bytes envolvidos. Salientamos também que é possível de se utilizar o C reduzido com os operadores bit a bit, a linha

```
registro1 &= 0xF0;
```

Poderia ser escrita assim

```
registro1 = registro1 & 0xF0;
```

O operador OU (|) é aplicável quando desejamos concatenar dois bytes. Vamos supor que queremos reunir em um único byte o *nibble* menos de um determinado byte com o mais significativo de outro. O código do Box 6 realizará esta tarefa perfeitamente.

```
main()
{
  unsigned char byte1, byte2, byte3;


  byte1 = 0x45;              /* byte1: 01000101b */
  byte2 = 0x6E;              /* byte2: 01101110b */
  puts("Bytes originais:");
  printf("byte1: ");
  print_bin(byte1);
  printf("byte2: ");
  print_bin(byte2);

  byte1 &= 0x0F;             /* byte1: 00000101b */
  byte2 &= 0xF0;             /* byte2: 01100000b */
  puts("Bytes depois da marcara:");
  printf("byte1: ");
  print_bin(byte1);
  printf("byte2: ");
  print_bin(byte2);

  byte3 = byte1|byte2;       /* byte3: 01100101 */
  puts("Bytes concatenados:");
  printf("byte3: ");
  print_bin(byte3);
  printf("byte3: %Xh\n",byte3);

}
```

Box 6 - Concatenando bytes com |.

A operação XOR também pode ser entendida como de "inversão controlada", uma vez que podemos utilizá-la para inverter bits pontuais em um determinado byte. Vamos supor que desejamos inverter apenas o bit 2 de um determinado byte, sem afetar o conteúdo dos demais bits. O código do Box 7 ilustra como isso pode ser feito.

```
main()
{
  unsigned char reg = 0xAB;

  printf("\n%Xh\n",reg);
  printf("reg: ");
  print_bin(reg);            /* reg: 10101011b */

  reg ^= 0x04;   /*  reg = reg ^ 00000100b */

  printf("\n%Xh\n",reg);
  printf("reg: ");
  print_bin(reg);            /* reg: 10101111b */
}
```

Box 7 - Inversão de bits pontuais com ^.

Sempre que você desejar inverter um único bit, utilize uma máscara XOR onde os bits em que se deseja a inversão devem estar em 1. A máscara usada no Box 7 foi 04h, que em binário é igual a 00000100b, em outras palavras, o bit 2 está com valor 1 e é este que será invertido.

Campos de bits (*bit fields*)

Os campos de bits são uma extensão das funcionalidades das estruturas em C e sua aplicação se dá quando desejamos mapear registradores em microcontroladores ou mesmo ter informações de um único bit em nosso código. No Box 8 vemos a sintaxe da declaração de um campo de bits.

```
struct reg
{
  unsigned nome : tamanho_em_bits;
  unsigned nome : tamanho_em_bits;
  .
  .
  .

} registro;
```

Box 8 - Sintaxe para declaração de um campo de bits.

Utilizamos a palavra reservada *struct* e dentro dela o tipo *unsigned* é aplicado, pois os bits devem ser sem sinal. Após, definimos um nome, seguindo as regras de sempre e em seguida o operador dois-pontos (:). À direita dos dois-pontos, definimos o tamanho em bits. No Box 9, apresentamos o exemplo de um registrador de 8 bits, que poderão ser acessados pelo usuário.

```
struct out1
{
  unsigned po0 : 1;
  unsigned po1 : 1;
  unsigned po2 : 1;
  unsigned po3 : 1;
  unsigned po4 : 1;
  unsigned po5 : 1;
  unsigned po6 : 1;
  unsigned po7 : 1;

} PORT1;
```

Box 9 - Declaração de um registrador de IOs.

Em alguns compiladores, o primeiro bit (no nosso caso po0) é o menos significativo, mas isso não é uma regra. É conveniente que você mesmo defina esta ordem quando for utilizar os bits do campo de bits. Quando o tamanho for de apenas 1 bit, obviamente você só poderá armazenar 0 ou 1. Para acessar, utilize o operador ponto (.), tal como é praticável com estruturas convencionais. No Box 10 um exemplo de teste do campo de bits com registrador PORT1.

```
main()
{
  unsigned char outp;
  register int i;

  PORT1.po0 = 1;
  PORT1.po6 = 1;

  outp = PORT1.po0      |
         PORT1.po1 << 1 |
         PORT1.po2 << 2 |
         PORT1.po3 << 3 |
         PORT1.po4 << 4 |
         PORT1.po5 << 5 |
         PORT1.po6 << 6 |
         PORT1.po7 << 7;

  printf("%Xh\n",outp);
}
```

Box 10 - Teste para o campo de bits.

Utilizando os operadores << e | podemos combinar todos os bits e armazenar em uma variável de 8 bits. Inicializamos po0 e po6 com 1. Por padrão, os bits não inicializados recebem o valor 0, portanto teremos 01000001b. Será armazenado 41h em outp.

Também é possível declarar registradores com alguns bits não implementados, vamos supor um registrador de UART, que conterá apenas os bits RX, TX e mais 2 bits para determinar os bits por segundo da transmissão. Teremos 4 bits ociosos e podemos escolher não implementá-los.

```
struct reg
{
  unsigned     : 4;  /* 4 bits não implementados */
  unsigned RX  : 1;
  unsigned TX  : 1;
  unsigned bps : 2;

} UART;
```

Box 11 - Campo de bits com bits não implementados.

Por fim, também convém mencionar que podemos combinar estruturas com tipos de dados comuns e campo de bits, como no exemplo do Box 12.

```
struct sensor
{
  float analog;
  short digital;
  unsigned ready : 1;

} light;
```

Box 12 - Combinando estruturas normais com campos de bits.

No exemplo, temos um sensor hipotético de luminosidade, que poderá ter um valor analógico com números reais, um digital discreto convertido e um bit de status para indicar quando o sensor está operante ou não.

Exercício proposto: Um sensor digital de temperatura pode ler valores na faixa de -255 a +255 (Celsius ou Fahrenheit) e envia dados seriais a partir do protocolo ilustrado na Figura 1, no formato de 2 bytes. Os primeiros 4 bits do byte mais significativo consistem na informação do protocolo em si, padrão definido em Ah. Os 2 bits seguintes consistem nos indicadores de erro (sensor fora do range, falha de comunicação, etc) e estarão em 0 quando estiver tudo ok. O próximo bit é o de sinal, que será 0 para positivo e 1 para negativo. Após, vem o bit de unidade, que será 0 para Celsius e 1 para Fahrenheit. O byte menos significativo é reservado para o valor da temperatura em si.

protocolo				erro		sig	uni	temperatura							
1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0

Figura 1 - Protocolo ilustrando sensor de temperatura ok, igual a +18°C.

Projete um software para ler os dois bytes no formato hexadecimal, onde você entra com o formato A012h e no console será impresso 18 graus Celsius, ou a temperatura que for verificada, de acordo com os bytes recebidos do sensor hipotético.

Bibliografia:

DAMAS, Luís; Linguagem C, décima edição.

Disponível em: https://amzn.to/3nGdIbN

SCHILDT, Herbert; C Completo e Total, terceira edição.

Disponível em: https://amzn.to/3xxi3l8