Microprocessadores - 2010-1 10/06/2010

Aula 29: Linguagem C para Microcontroladores

Professor: Eraldo Silveira e Silva eraldo.silveira@gmail.com

1 Introdução

O objetivo desta aula é iniciar o aluno na programação em C para microcontroladores. Presume-se que o mesmo já possua as noções básicas da linguagem embora um breve revisão será efetuada.

2 A Linguagem C

A linguagem C apresenta facilidades para manipulação a baixo nível que a tornam interessante para uso em microcontroladores. Além disto, normalmente os compiladores conseguem gerar código de máquina com muita eficiência. Isto faz com que na maior parte das aplicações, o programador opte por trabalhar em C e, eventualmente, programe algumas rotinas em assembly.

Note que programar em alto nível não significa que o microprocessador/microcontrolador irá executar o código de alto nível. Da mesma forma que no assembly, o código é traduzido para linguagem de máquina com o assemblador, o código C será traduzido para linguagem de máquina através de um processo de compilação.

Um programa em C se apresenta como uma coleção de funções (subrotinas) e estruturas de dados. Estas funções podem estar em um ou mais arquivos com a terminação ".c". Em adição, para facilitar que os arquivos possam referenciar as funções ou estruturas de dados de outros arquivos, checando tipos de parâmetros, por exemplo, pode-se criar arquivos de cabeçalhos com terminação ".h".

Por exemplo, um usuário pode criar um arquivo "piscaP1.c" que usa funções e definições de uma biblioteca já previamente desenvolvida:

```
1  #include <8051.h>
2
3  void main(void)
4  {
5     int i;
6     for (i=0;i<10000;i++) {
7        P1_7=1;
8        P1_7=0;
9     }
10 }
```

Neste programa o usuário inclui um arquivo cabeçalho ("header") chamado "8051.h". Este arquivo é do tipo texto que contém algumas definições a serem usadas ao longo do programa.

3 O processo de Geração do Código em Linguagem de Máquina

4 Compilação usando o SDCC

Neste curso vamos utilizar o compilador livre SDCC sobre o linux. Este compilador permite gerar código para vários microcontroladores. Por "default" ele gera código para o microcontrolador 8051.

Para compilar o programa acima basta fazer:

```
sdcc piscaP1.c
```

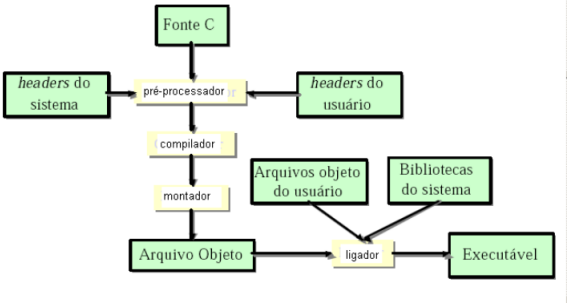

Figura 1: Etapas de Geração de Código

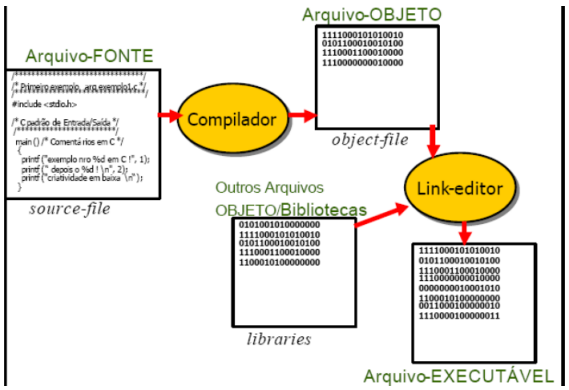

Figura 2: Detalhe de Geração de Código

Vários arquivos são gerados como subprodutos desta operação. Podemos destacar o piscaP1.lst (listagem em assembly), piscaP1.rel (código objeto) e piscaP1.ihx (arquivo hexa no formato intel).

Caso o projeto envolva vários arquivos c, então pode-se primeiramente gerar os códigos objeto, fazendo:

```
sdcc -c arquivo1.c
sdcc -c arquivo2.c
sdcc -c arquivo3.c
```

e depois linká-los da forma:

```
sdcc arquivo1.rel arquivo2.rel arquivo3.rel -o arquivo_final.ihx
```

5 Exemplo Completo

O exemplo a seguir mostra um programa para acionar um LED com temporização baseada em interrupção e timers.

```
1
2  #include <at89x52.h>
3
4  #define OSCILADOR 11059200
5
6  // timer 0 used for systemclock
7  #define TIMER0_RELOAD_VALUE 0x10000-OSCILADOR/12/1000 // 0.999348ms for
8  11.059Mhz
9
10 static long data miliSegundos;
11 volatile unsigned char UmSeg;
12
13 // Rotina de Tratamento de Interrupcao
14
15 void ClockIrqHandler (void) interrupt 1 using 3
16 {
17   TL0 = TIMER0_RELOAD_VALUE&0xff;
18   TH0 = TIMER0_RELOAD_VALUE>>8;
19   miliSegundos++;
20   if (miliSegundos==1000) {
21         UmSeg = 1;
22   }
23 }
24
25 void delay(void)
26 {
27   ET0 = 0;
```

```
28    miliSegundos = 0;
29    UmSeg = 0;
30    ET0 = 1;
31    while (!UmSeg);
32    return;
33  }
34
35  void init_timer() {
36    // initialize timer0 for system clock
37    TR0=0; // stop timer 0
38    TMOD =(TMOD&0xf0)|0x01; // T0=16bit timer
39    // timeout is xtal/12
40    TL0 = TIMER0_RELOAD_VALUE&0xff;
41    TH0 = TIMER0_RELOAD_VALUE>>8;
42
43    TR0=1; // start timer 0
44    ET0=1; // enable timer 0 interrupt
45    EA=1; // enable global interrupt
46
47    return;
48  }
49
50  void main()
51  {
52     init_timer();
53     for (;;) {
54          P1_1 = 1;
55          delay();
56          P1_1 = 0;
57          delay();
58   }
59  }
```

5.1 Rotinas de Intrerrupção

Note a forma como é declarada uma rotina de interrução:

```
1  void interrupt_identifier (void) interrupt interrupt_number using bank_number
2  {
3      ...
4  }
```

O SDCC se preocupa em preparar a entrada da rotina de interrupção, salvando devidamente na pilha os valores de registradores. Variáveis acessadas dentro e fora da interrupção devem ser decladas como "voláteis":

```
1  volatile int contador;
```

5.2 Inserindo código assembly

Código em assembly pode ser inserido no programa usando a diretiva *_asm*:

```
1  void main(void)
2  {
3      while (1)
4      {
5
6          a = P0;
7
8          _asm
9              nop
10             nop
11             nop
```

```
12              inc _a
13          _endasm;
14
15          P1 = a;
16      }
17  }
```

6 Classes de Armazenamento

Ao longo do curso verificamos que os dados manipulados por um programa podem estar armazenados em diferentes áreas de memória. Por exemplo, dados que são constantes ao longo da execução do programa podem ser armazenados na própria memória de programa. Dados voláteis podem estar na memória RAM interna de acesso direto e indireto (primeiros 128 bytes), memória RAM interna de acesso indireto, área de SFR e memória de dados externa. Cada um destes tipos de acesso implica em uso de diferentes instruções.

No caso do C para microcontroladores é necessário usar diretivas para que o compilador possa gerar código adequadamente para cada tipo de classe de armazenamento. A seguir será mostrado como pode se fazer isto no SDCC.

6.1 Declarando bits

```
1   bit esc_char_flag = 0;
2
3   void main(void)
4   {
5       P1 = 0x00;
6
7       while (!esc_char_flag)
8       {
9           if (P0 == ESCAPE)
10              esc_char_flag = 1;
11      }
12
13      P1 = 0xFF;
14
15      while (1); // program loop
16  }
```

6.2 Classe data/near

Esta classe contempla dados na área de RAM interna de acesso direto. A declaração é da forma:

```
1   __data unsigned char test_data;
2   main()
3   {
4     test_data = 1;
5   }
```

O acesso a este dado será traduzido para:

```
mov  _test_data,#0x01
```

6.3 Classe xdata/far

Esta classe contempla dados na área de RAM externa:

```
__xdata unsigned char test_xdata;
main()
{
  test_xdata = 1;
}
```

O acesso a este dado será traduzido para:

```
mov  dptr,#_test_xdata
mov  a,#0x01
movx @dptr,a
```

6.4 Classe idata

Esta classe contempla dados na área de RAM interna de acesso indireto:

```
__idata unsigned char test_idata;
main()
{
  test_idata = 1;
}
```

O acesso a este dado será traduzido para:

```
mov  r0,#_test_idata
mov  @r0,#0x01
```

6.5 Classe code

Esta classe contempla dados na área de programa:

```
__code unsigned char test_code = 'A';
__data unsigned char test_data;
main()
{
  test_data = test_code;
}
```

Note que não teria sentido escrever em test_code pois ela está na área de memória de código. O código gerado para este acesso seria:

```
mov dptr,#_test_code
clr a
movc a,@a+dptr
mov  _test_data,a
```

6.6 Classe bit

Esta classe contempla dados armazenados na área com mapeamento por bit:

```
__bit test_bit_um;
__bit test_bit_dois;
main()
{
  test_bit_um = 1;
  test_bit_dois = 0;
}
```

O código gerado para este acesso seria:

```
setb _test_bit_um
clr  _test_bit_dois
```

6.7 Declarando ponteiros para áreas diversas

A declaração de ponteiros é realizada da seguinte forma:

```
/* Ponteiro em RAM interna apontando para objeto na RAM externa */
__xdata unsigned char * __data p;

/* ponteiro na RAM externa apontando para objeto na RAM interna */
__data unsigned char * __xdata p;

/* ponteiro na área de código apontando para objeto na RAM externa */
__xdata unsigned char * __code p;

/* ponteiro na área de código apontando para objeto na área de código */
__code unsigned char * __code p;

/* ponteiro genérico localizado na área RAM externa */
unsigned char * __xdata p;

/* ponteiro genérico localizado na área de memória default */
unsigned char * p;

/* ponteiro para função localizado na área de dados interna */
char (* __data fp)(void);
```

7 Endereçamento Absoluto

Normalmente o compilador, em função das variáveis declaradas, asssocia automaticamente endereços para as mesmas, segundo alguns critérios. No entanto, é possível forçar o endereçamento da forma:

```
__xdata __at (0x7ffe) unsigned int chksum;
__code __at (0x7ff0) char Id[5] = ''SDCC'';
```

A variável chksum será alocada nas posições 0x7ffe e 0x7fff da RAM externa. Já a variável Id será posicionada em 0x7ff0 até 0x7ff4.