Programação em linguagem C
EA870 - FEEC - Unicamp

Introdução à linguagem C, para programação de microcontroladores HC11,
utilizando o compilador ICC11

Murillo Fernandes Bernardes
bernarde@fee.unicamp.br
http://www.fee.unicamp.br/~bernarde/

Setembro de 2002

"He who asks is a fool for five minutes, but he who does not ask remains a fool forever"
Provérbio Chinês

Sumário

1 Introdução

Estamos começando uma nova fase do curso de EA870, a utilização da linguagem C, no lugar do uso do assembly. Alguns podem até reclamar que quando estavam se acostumando muda tudo, tem que aprender uma linguagem nova, mas com certeza o uso do "C"vai facilitar bastante o trabalho.

Este tutorial tem o intuito de ser apenas um "pontapé"inicial no aprendizado da linguagem C, sendo que a maior parte do aprendizado ocorrerá no dia-a-dia, nas dificuldades enfrentadas a cada programa, e com o costume de programar em C.

A linguagem (vocabulário) utilizada neste tutorial é, na medida do possível, a mais simples, com a intenção de que todos entendam realmente como as coisas funcionam em C, e também com o intuito de tirar todos os medos que ainda possam existir. Infelizmente vão haver alguns termos meio estranhos, mas que com o tempo se tornarão comuns a todos.

"Programar é só isso, resolver problemas da vida real de forma "entendível"ao computador. Mostrar-lhe como você fez, em uma linguagem que ele entenda."

1.1 Histórico do C

Na década de 70, nos Laboratórios AT&T Bell (EUA), a linguagem C foi criada, tendo como principal motivação o desenvolvimento do sistema UNIX em uma linguagem de alto nível. O "C"se tornou ferramenta inseparável/indispensável em todos os sistemas UNIX, e com este foi sendo bastante difundida.

Com o passar do tempo havia diversos compiladores "C", mas com diversas incompatibilidades entre eles. Então em 1983 o Insituto Norte Americano de Padronização (ANSI) criou um comitê para a padronização da linguagem C. O resultado desta padronização foi concluido em 1990 e é hoje o padrão internacional, chamado de "C ANSI". Nesta padronização foram definidas algumas funções básicas de suporte, que deveriam ser fornecidas pelos compiladores, apesar delas não fazerem parte da linguagem.

Uma das grandes vantagens da linguagem C é a união harmoniosa entre diversas características de linguagens de baixo nível, e as caracteristicas das linguagens de alto nível. Algumas linguagens antecessoras ao C tinham certas características muito favoráveis, mas no geral eram muito restritas ou muito complicadas. Já o C conseguiu unir tanto simplicidade e quanto potencialidade em suas aplicações. Com isso mesmo tarefas muito complexas podem ser realizadas com uma relativa simplicidade.

2 Compilador ICC11

No curso de EA870 o compilador utilizado é o ICC11, que apesar de não atender todos os quesitos do C ANSI, atende ao menos os quesitos básicos. Por tanto qualquer livro que trate do C ANSI pode (e deve) ser utilizado como referência.

Ao longo deste tutorial mostraremos tanto definições do C padrão, quanto definições específicas do ICC11 (onde houverem diferenças).

2.1 Sintaxe da linha de comando

A sintaxe do comando para compilar um programa utilizando o ICC11 é a seguinte:

icc11 [opções] nomedoarquivo.c

As opções são as seguintes:

-v verbose, mostra na tela as ações que estão sendo executadas
-o Nome do aqtuivo de saida, padrão é iccout.s19
-l Cria o arquivo contendo a listagem em assembly, com o nome ¡arquivo_de_saida¿.lst
-E Para a execução após o pre-processamento, gerando um arquivo .i
-S Compiles para assembly, mas não monta o programa (não gera arquivo objeto)
-s Modo silencioso. Não mostrará na tela as ações

Assim para compilar o programa contido no arquivo prog1.c, gerando o arquivo objeto com o nome de prog1.s19, a listagem com prog1.lst, e visualizando as ações do compilador utilizamos a seguinte linha de comando:

icc11 -v -l -o prog1.s19 prog1.c

2.2 Processo de compilação

O processo de compilação de um programa não é direto, são necessários vários passos até que se chegue ao código objeto (.s19).

O primeiro passo é a análise do código, pelo pré-processador C, em busca de erros, e executando diretivas do pré-processador, como *include* e *define*. Ao final deste passo tem-se um novo arquivo, com extensão *.i*, que será utilizado no próximo passo.

O segundo passo é o processo de tradução para assembly, irá converter o código em C para um código assembly, e unir o conteúdo do arquivo *crt.s* com o código assembly gerado, criando o arquivo *.s*.

O terceiro passo é o processo de montagem, que irá traduzir o código assembly gerado, para um arquivo objeto, num formato (s19) que já pode ser carregado no HC11 através do BUFALLO.

Ao final do processo os arquivos *.i* e *.s* são apagados, por padrão, mas há opções para que isso não aconteça.

2.3 C run-time (crt.s)

No segundo passo foi citado o uso de um arquivo chamado *crt.s*, que é um pequeno arquivo contendo código assembly, que contém as funções básicas definidas no compilador (por exemplo: divisão, resto, multiplicação e putchar). O nome *crt* vem de *C run-time*.

Para um programa seja compilado, deve-se ter um arquivo *crt.s* no mesmo diretório do programa.

É importante também estar atento a dois *ORG* que estão no início deste arquivo, e que definem onde começam as áreas de dados, e de programa. Na placa utilizada (PDHC11-FEEC) a região de memória disponível para programas do usuário é de $6000 até $7FFF, compreendendo 8Kbytes. Portanto estes dois ORGs devem estar dentro desta região. Qualquer problema verifiquem no arquivo *.lst* se não há código (ou variáveis) gerado fora desta região.

Para a placa utilizada, um exemplo de como ficariam os ORGs é:

```
* define starting addresses
sect 0 * code
org $6000
sect 1 * data
org $7000
```

3 Visão Geral de um programa em C

Para começarmos, vamos ao exemplo (C ANSI) que não pode faltar em nenhum tutorial, e que simplesmente mostraria na tela a frase "Hello World!":

```
/************************************************************/
/* Exemplo 1 - Tutorial C - EA870     */
/************************************************************/

#include <stdio.h>  /* biblioteca padrão de I/O */

/************************************************************/

void main ()  /* Comentarios */
{
   printf("Hello World!"); /* Dizendo olá mundo! */
}
```

Com este simples programinha podemos mostrar a "base mais básica"do C, que é a função *main*. Esta função é que será executada ao se chamar o programa criado em C, portanto todo programa deve ter sua própria função *main*.

Este programa também mostra a chamada a uma função externa (printf) ao programa, mas que está definida numa das bibliotecas do C ANSI. Esta função está sendo referenciada dentro do arquivo stdio.h, que está sendo incluído no programa através da diretiva *#include <stdio.h>*.

Outra coisa básica e muito importante que podemos ver neste exemplo é como se colocar um comentário num programa em C. Utiliza-se sempre "/*"para início do comentário e "*/"para indicar o fim de um comentário, assim pode-se ter comentários de várias linhas.

Linhas que iniciam com "#"são geralmente diretivas ao pre-processador C.

3.1 Funções

Em C todo algorítmo deve ser escrito na forma de funções. E funções são simplesmente conjuntos de comandos que podem incluir declarações, chamadas a outras funções. A organização básica de uma função em C é a seguinte:

```
tipo funcao(parametros) {
   declarações;
   comandos;
}
```

O *tipo* indica o tipo do valor de retorno da função, e pode ser qualquer um dos tipos definidos em C. (Mostrados na próxima seção)

funcao é o nome dado à função, e atualmente não se tem limitação quanto ao tamanho do nome, mas recomenda-se utilizar nomes que lembrem o que a função irá executar.

parametros correspondem na verdade à definição do tipo do parâmetro a ser passado para a função, e o nome com o qual a função referenciará o valor passado.

Em C é necessário se declarar todas as variáveis antes que as utilize, assim a parte de *declarações* é onde isso deve ser feito.

Logo após as declarações pode-se colocar a chamada a todos os *comandos* a serem executados pela função.

Como exemplo iremos criar uma função que escreverá na tela a frase "Eu faço EA870", e um programa principal que escreve "Oi pessoal", e logo após chama a função anteriormente criada:

```
#include <stdio.h>

void escreve() { /* função para escrever "Eu faço EA870"*/
   printf("Eu faço EA870");
}

void main() { /* função principal */
   printf("Oi pessoal"); /* imprime "Oi pessoal" */
```

```
   escreve(); /* chama função que escreve "Eu faço EA870" */
}
```

Vale lembrar que o ICC11 não provê as bibliotecas de funções definidas pelo C ANSI, assim para o ICC11 a diretiva *#include <stdio.h>* não pode ser utilizada. A única função que o ICC11 fornece é a printf (com algumas limitações), que para ser utilizada o arquivo printf.c deve ser compilado juntamente com o programa. Utilizando o exemplo da linha de comando utilizado na seção anterior teriamos que ter agora esta linha de comando, para que o programa pudesse utilizar a função printf (com o printf.c no mesmo diretório do programa prog1.c):

icc11 -l -o prog1.s19 prog1.c printf.c

4 Tipos de Dados

O C padrão define os seguintes tipos de dados:

char	Caractere (1 byte)
int	Inteiro (em geral 32 bits)
float	Número real
double	Número real precisão dupla
void	Sem valor

Além dos tipos, o C também define alguns modificadores, que simplismente mudam a forma com que o número será tratado ou armazenado. Alguns modificadores são: unsigned, signed, short, long

Na verdade há mais alguns tipos e modificadores que não iremos citar aqui, mas que podem ser encontrados em qualquer livro de C ANSI. Além disso o C padrão permite que novos tipos de dados sejam criados.

Já o ICC11 conta basicamente com os seguintes tipos:

char	Caracter unitário (1 byte)
int	um inteiro (2 bytes)
void	sem valor

E também com alguns modificadores, como o **unsigned** que será muito utilizado durante o curso.

Pode até parecer que são poucos tipos de dados, mas esta é uma das "mágicas"do C, fazer muito com pouco. Com apenas estes tipos de dados somos capazes de representar tudo que é necessário numa máquina como o HC11. Por exemplo: Como temos o tipo *char* (caractere) podemos facilmente ter *strings*, já que estas são simplesmente vetores de caracteres.

4.1 Representação Hexadecimal, decimal, octal, ASCII e caracteres de Escape

Em C números em hexadecimal são representados sempre iniciados com "0x", ou seja, para representar o número $FF deve-se escrever 0xFF.

Números decimais são representados normalmente (nunca com zeros à esquerda), quando negativos, com o sinal de menos (-) à sua esquerda.

Números na base octal devem ser iniciados com "0", assim o número 023 está na base 8.

Caracteres ASCII são representados entre aspas simples ('), assim *'m'* é o caractere ASCII *m*.

Strings (vetores de caracteres) são representados entre aspas duplas , assim "EA870"é a representação da string *EA870*.

4.2 Declaração de variáveis

Agora que já conhecemos os tipos de dados possíveis, podemos realmente começar a fazer as declarações de variáveis num programa.

A declaração das variáveis é da forma:

tipo nome_da_variável;

ou

tipo nome_var1, nome_var2, nome_var3;

Por exemplo:

```
int h, m, s;
char a;
```

Com estas duas linhas acima declaramos três variáveis do tipo inteiro, e uma do tipo char.

Pode-se também inicializar as variáveis no momento da declaração, da seguinte forma:

```
int h=5;
char a='j';
```

Assim declaramos a variável h como inteiro, e definimos seu valor inicial como 5. Já a variável a foi definida como caractere, e é inicializada com o caractere *'j'*.

4.2.1 Variáveis Locais x Globais

Quando-se declara variáveis dentro de uma função (mesmo que seja a main), esta variável só fica acessível de dentro daquela função, temos então uma variável local. Pode-se declarar variáveis fora de qualquer função, o que faz com que a variável seja global, ou seja acessível apartir de qualquer função do programa. Por exemplo:

```
char c;

void main() { /* função principal */
   int a=5;

   c='m';
}
```

No trecho de código apresentado acima, temos a variável *c* que é global, e a variável *a* que é local (função main).

4.3 Cast

Cast ou *coerção* é simplesmente a conversão de tipo de um determinado valor, ou variável. Podemos falar que seria o modelamento daquele valor para que ele fique compatível com o que se quer.

No seguinte trecho de código, um inteiro está recebendo o valor de um ponteiro pra inteiro (ponteiros serão detalhados mais a frente), ou seja, está recebendo o endereço, mas estes dois tipos não são iguais, e portanto o compilador deverá mostrar um aviso do que está sendo feito.

```
int a;    /* declara um inteiro */
int *p;   /* declara um ponteiro para inteiro */

a = p;    /* o endereço apontado pelo p é "jogado" em a, ou seja,
             a será um inteiro que contém o valor do endereço */
```

Para mostrar ao compilador que é realmente aquilo que se quer fazer, e também evitar possíveis erros, o melhor seria utilizar um *cast*, convertendo o tipo, de ponteiro para inteiro, para simplesmente inteiro, e isso seria feito da seguinte forma:

```
int a;    /* declara um inteiro */
int *p;   /* declara um ponteiro para inteiro */

a = (int) p;    /* o endereço apontado pelo p é "jogado" em a, ou seja,
                   a será um inteiro que contém o valor do endereço */
```

O *cast* é feito simplesmente colocando o tipo que se quer, entre parenteses antes do valor, ou variável. Assim poderia utilizar também:

```
int *p;   /* declara um ponteiro para inteiro */

p = (int *) 0x1000;
```

Neste exemplo, modifica-se o tipo inteiro do valor 0x1000 para ponteiro para inteiro. *Casts* desta forma podem ser utilizados para qualquer tipo de dados suportado. Vale prestar atenção ao tamanho em bytes que cada tipo ocupa, para que não ocorra truncamentos indesejados, ou coisas parecidas.

5 Expressões

Como já dito anteriormente "funções são simplesmente conjuntos de comandos"válidos, que serão executados de acordo com uma determinada lógica.

Para que esta lógica possa ser expressa tem-se as expressões aritméticas, as expressões condicionais, e as expressões de manipulação de bits.

5.1 Expressões Aritméticas

Atribuições em C são indicadas pelo simbolo =. Assim:

```
void main() {
   int a, b;
   a = 10;  /* a recebe valor 10 */
   b = a;  /* b recebe o valor de a (10) */
}
```

Além da atribuição as expressões aritméticas em C podem ser: soma(+), subtração(-), multiplicação (*), divisão (/), módulo (%), incremento(++), e decremento(–). Por exemplo:

```
void main() { /* função principal */
   int a=2, b, c, d, e; /* declaração de variáveis */

   b = 5*a; /* b recebe 10 */
   c = a+b; /* c recebe 12 */
   d = b/a; /* d recebe 5  */
   e = b%3; /* e recebe 1  */
   e++; /* incrementa e, e fica com 2 */
   d--; /* decrementa d, d fica com 4 */
}
```

Números negativos são representados com o simbolo - a esquerda do número.

5.2 Expressões Condicionais

As expressões condicionais são das mais importantes no desenvolvimento de um programa, pois nelas se baseiam todos os testes, e verificações necessárias num programa.

A linguagem C não suporta nativamente o tipo *booleano*, mas utiliza inteiros pra representar *verdadeiro* ou *falso*. Quando o resultado de uma expressão condicional for 0 (zero) é interpretado como *falso*, e quando for diferente de 0 é *verdadeiro*. Assim podemos utilizar qualquer expressão inteira como expressão condicional, ou seja, a sintaxe: `if (a) printf('Verdadeiro'); else print ('Falso');` é perfeitamente válida, e mostrará na tela "verdadeiro", caso *a* seja diferente de zero, e mostrará "Falso"caso *a* seja igual a zero.

Para comparações de valores, tem-se os operadores relacionais, que são:

`>`	maior que
`>=`	maior que ou igual a
`<`	menor que
`<=`	menor que ou igual a
`==`	**igual a**
`!=`	diferente de

Alem destes operadores há também mais três, que são operadores booleanos comuns:

&&	AND, o resultado é verdadeiro se ambas expressões envolvidas forem verdadeiras
||	OR, o resultado é verdadeiro se uma das expressões envolvidas for verdadeira.
!	NOT, o resultado é o inverso da expressão, se esta era verdadeira o resultado é falso, i.e.

5.3 Expressões de manipulação de bits

Além dos operadores já vistos, o C oferece também operadores para trabalhar diretamente com bits das variáveis de um programa, e podem ser utilizadas com qualquer tipo de variáveis.

&	AND bit-a-bit
|	OR bit-a-bit
^	XOR bit-a-bit
`<<`	deslocamento de bits à esquerda
`>>`	deslocamento de bits à direita
~	complemento de um (inverte cada bit)

6 Controle de Fluxo

Os programas consistem basicamente de um conjunto de comandos que são executados em sequência, mas se consegue um poder muito maior controlando a ordem de execução de certos trechos do programa, através de algumas condições. Para isso temos comandos que nos permitem fazer testes e mudar õ rumo da execução do programa, dependendo do resultado destes testes.

Estes comandos são: if, operador ternário, switch, while, do while, for, break, e continue. Todos já conhecem a idéia de cada um destes tipos de controle de fluxo do programa, pois foram vistos em MC102, aqui simplesmente será mostrado a sintaxe de cada um deles, alguns exemplos no final.

6.1 IF

O comando IF é um dos mais básicos e úteis comandos de qualquer linguagem de programação de alto nível. No C ele tem a seguinte estrutura:

```
if (condição) {
   comandosIf;
}
```

ou

```
if (condição) {
   comandosIf;
}
else {
   comandosElse;
}
```

Por exemplo:

6.2 Operador Ternário

O operador ternário é simplesmente uma forma simplificada do IF..ELSE, e é mais eficiente para testes simples. Sua estrutura é:

```
expressão1 ? expressão2 : expressão3
```

Que é na verdade:

```
if expressão1 then expressão2 else expressão3
```

6.3 Switch

O comando *switch* é bastante semelhante ao comando *case* do Pascal. Este comando permite a execução de uma determinada ação para cada valor da expressão condicional, sem que seja necessário fazer um "aninhamento"de IFs. Sua estrutura é:

```
switch (expressão) {
   case item1:
        comandos1;
        break;
   case item2:
        comandos2;
        break;
   case itemn:
        comandosn;
        break;
   default:
        comandos;
        break;
}
```

Antem-se ao fato de que para cada *case* há um *break*, e o break é necessário para que os comandos dos outros cases não sejam executados. Por exemplo, se a expressão era igual a item1, os comandos representados por *comandos1* serão executados, caso não houvesse o *break*, *comandos2*, até *comandosn*, e *comandos* seriam executados também. Com a presença do *break* o switch é "abandonado"apartir daquele ponto, não executando nenhum comando de outro item.

6.4 While

A estrutura do while é:

```
while (expressão) {
   comandos;
}
```

6.5 Do - While

O do-while do C, é praticamente igual ao *repeat until* do Pascal, pois sempre executa uma vez aí faz o teste. Sua estrutura é:

```
  do
 comandos;
 while (expressão);
```

6.6 For

Sua estrutura é a seguinte:

```
for(expressão1; expressão2; expressão3) {
    comandos;
}
```

Onde *expressão1* é onde se faz a atribuição do valor inicial do contador utilizado. *expressão2* é a condição ("enquanto") de continuar no loop. *expressão3* é a forma com que o contador será modificado a cada iteração. A *expressão3* é comumente algo como "contador++"ou "contador–", mas pode ser qualquer operação.

6.7 Break

O C provê dois comandos de controle de como será o loop, um deles é o *break* e o outro é o *continue*.

O *break* é simplesmente um comando de saída de um loop, ou switch.

6.8 Continue

O *continue* permite que uma determinada iteração (ou várias delas) sejam puladas, e o loop continue daí pra frente.

6.9 Exemplos

```
if(x > 5)
   x = 0;
else ++x;

while(x < 5) {
   func();
   ++x;
}

unsigned char ascii2dec(unsigned char letra){
   unsigned char digito;
   switch (letra) {
      case 'A':
      case 'B':
      case 'C':
      case 'D':
      case 'E':
      case 'F':
         digito=letra+10-'A';
         break;
```

```
      case 'a':
      case 'b':
      case 'c':
      case 'd':
      case 'e':
      case 'f':
         digito=letra+10-'a';
         break;
      default:
          digito=letra-'0';
   }
   return digito;
}

for(J=100;J<1000;J++) {
    process();
}


I=100;
do {
   process();
   I--;
} while (I>0);
```

7 Vetores

Os vetores são conjuntos de valores de um determinado tipo, que podem ser acessados pelo nome do vetor, e pela especificação da posição dentro do vetor.

Pode-se ter vetores de qualquer um dos tipos aceitos pela linguagem. Um vetor para inteiros, com capacidade para 10 números é definido da seguinte forma:

```
int vet[10];
```

A inicialização de um vetor pode ser feita da seguinte forma:

```
int vet[5] = {1, 2, 3, 4, 5};
```

E o acesso é da forma:

```
int a;
int vet[5];

a = vet[2];
vet[4] = 3;
```

Assim pegamos o valor da posição 2 do vetor e colocamos na variável *a*, e colocamos o valor 3 na posição 4 do vetor.

É importante destacar que os índices do vetor vão de *0* até *n-1*, onde *n* é o tamanho do vetor. Portanto o vetor acima, que tem tamanho 5, tem seus índices de 0 a 4, como índice 0 representando a primeira posição, 1 a segunda,... e 4 a última posição do vetor.

8 Ponteiros

Uma das qualidades do C é que ele provê meios de se trabalhar com a memória diretamente. Isso permite que o programador realize muitas operações como se estivesse utilizando assembly, mas com a facilidade de uma linguagem de alto nível. Infelizmente esse potencial pode ser também uma fonte de erros muito difíceis de ser detectados.

Os ponteiros são variáveis que contém endereços. Assim pode-se realizar operações tanto com o conteúdo das posições de memória apontadas por eles, quanto com os próprios ponteiros. Como todas as outra variáveis, os ponteros têm um tipo, que indica ao compilador o numero de bytes que aquele ponteiro estará sendo realmente apontado por aquele ponteiro. Por exemplo, no caso do ICC11, os inteiros têm 2 bytes, assim quando se fizer uma operação com o conteúdo da posição apontada por aquele ponteiro, será uma operação com 16 bits.

Em C pode-se obter o endereço de uma variável facilmente, para isto basta colocar um & (e comercial) antes do nome da variável, ou seja, se temos uma variável *var*, &*var* é o endereço de *var* (mas não um ponteiro).

O endereço de funções também são acessíveis, e são dados simplesmente pelo nome da função, sem os parênteses logo após. Portanto para pegar o endereço de uma função basta:

```
void myFun() {
 printf('Não faz nada');
}

void main() {
   int ender;
   ender = (int) myFun;
}
```

8.1 Declaração de ponteiros

Para se declarar um ponteiro utiliza-se a seguinte sintaxe:

```
int  *pt1;    /* declara um ponteiro para um inteiro (16bits) */
char *pt2;    /*  declara um ponteiro para um caractere (8bits) */
```

```
unsigned char *pt3;  /* declara um ponteiro para um caractere,
                        considerando-o de 0 a 255 (8bits)*/
```

Além da declaração deseja-se que o ponteiro aponte para uma determinada posição de memória, portanto é preciso setar o seu valor. Para isso basta que se conheça o endereço a ser apontado, e então fazer um *cast* para o tipo de dado do ponteiro, da seguinte forma:

```
pt1 = (int *) 0x1000;
pt3 = (unsigned char *) 0x100A;
```

Agora tem-se dois ponteiros inicializados, o primeiro, *pt1*, um apontador para inteiro, apontando para o endereço 0x1000. E o segundo um apontador para caracteres (8 bits) sem sinal (0 a 255), apontando para o endereço 0x100A.

8.2 Referência a ponteiros

Os ponteiros podem ser utilizados tanto para escrita, quanto leitura na memória. Essas duas operações são classificadas como *referência a ponteiros*.

O uso dos ponteiros é bastante semelhante ao uso de variáveis comuns, exceto pelo asterisco (*) colocado antes do nome do ponteiro para indicar o conteúdo da posição de memória apontada pelo ponteiro. Assim o uso seria:

```
void main() {
   unsigned char *PORTE = (unsigned char *) 0x100A;
   char *disp = (char *) 0xBF01;
   unsigned char a;

   a = *PORTE; /* Lê da Porta E (DIP-SWITCH e Push-buttom) */
   *disp = 0xFF; /* Escreve $FF em $BF01, ou seja apaga displays */

}
```

8.3 Aritmética de ponteiros

O ICC11 trata ponteiros como se fossem vetores, assim caso se queira acessar a terceira posição de memória apartir de onde está o ponteiro, pode-se tanto utilizar ***(pt+3)**, ou simplesmente **pt[3]**.

```
void main() {
int *pt = (int *) 0x7000;  /* pt: ponteiro para um inteiro */
int i;

for (i=0; i<10; ++i)
   *(pt + i) = 0;
}
```

Este programinha coloca *0* em 20 (inteiro ocupa 2 bytes) posições de memória consecutivas apartir da posição 0x7000.

Este mesmo programa poderia ser feito utilizando a seguinte sintaxe:

```
void main() {
int *pt = (int *) 0x7000;  /* pt: ponteiro para um inteiro */
int i;

for (i=0; i<10; ++i)
   pt[i] = 0;
}
```

9 Funções pré-definidas no ICC11

O ICC11 é um compilador C não completamente ANSI, e não oferece muitas das funções padrão definidas com a linguagem C. O ICC11 oferece duas funções básicas de escrita, e que podem ser facilmente utilizadas nos programas que deverão ser feitos.

Algumas outras funções o próprio BUFALLO oferece (por exemplo INPUT em $FFAC), e para utilizá-las basta fazer uma simples interface entre seu programa em C e uma chamada à sub-rotina assembly do BUFALLO.

9.1 putchar

Uma das funções que o ICC11 oferece é a *putchar*, que simplesmente ecoa na tela o caractere passado a ela como parâmetro. Note que esta função faz a mesma coisa que a rotina OUTPUT do BUFALLO.

Para imprimir o caractere ***M*** na tela basta:

```
putchar('M');
```

9.2 printf

A outra função de saída que o ICC11 oferece é o *printf*, e pode ser utilizada para imprimir somente uma string, da forma: `printf("Meu texto");`. Ou pode ser utilizada para imprimir o valor de variáveis formatando o valor de acordo com determinadas strings de controle, da forma:

```
int a=7, b=65;
printf("Eu gosto do número %d",a); /* Imprime o número na forma decimal */
printf("Eu esqolhi o número %x também",b); /* Imprime o número 65 na forma hexadecimal 0x41 */
```

A string de controle é indicada pelo símbolo "%", e no C ANSI pode ter os seguintes valores:

%c	ASCII(caractere)
%s	String
%d	decimal com sinal
%u	decimal sem sinal
%o	octal sem sinal
%x	hexadecimal sem sinal
%f	ponto flutuante
%e	notação científica (exponencial)
%g	formato geral, escolhe a representação mais curta entre %f e %e

Note que o C ANSI trabalha com números de ponto flutuante, mas o ICC11 não, portanto os últimos três formatos não são aceitos. Em C padrão há também algumas variações das strings de controle, podendo ser passado o número de dígitos com o qual o número será representado, número de casas decimais, mas o ICC11 também não provê esta funcionalidade.

Além dos caracteres normais do teclado, às vezes é necessário imprimir na tela alguns caracteres "não exatamente gráficos", como mudança de linha, tabulações, e alguns outros. Em C utiliza-se a barra invertida para indicar este tipo de sequência. Os caracteres deste tipo mais usados são:

sequência	nome	valor
`\n`	newline, linefeed	$0A = 10
`\t`	tab	$09 = 9
`\b`	backspace	$08 = 8
`\a`	Beep	$07 = 7
`\r`	return	$0D = 13
`\"`	ASCII quote	$22 = 34
`\\`	ASCII back slash	$5C = 92
`\'`	ASCII single quote	$27 = 39

É bom prestar atenção ao fato que aspas, e aspas simples são utilizadas como delimitador de strings e caracteres respectivamente, assim quando se quer imprimir na tela uma aspas (") não se pode simplesmente colocar uma aspas pra ser impressa, é preciso colocar a sequência correspondente, que é `\"`. Para aspas simples e barra invertida ocorre da mesma forma.

No C padrão há mais algumas destas sequências, mas que não serão citadas aqui, fica a cargo de quem tiver interesse, pegar um livro de C e dar uma olhada.

Por exemplo:

10 Assembly in-line

A linguagem C é bastante flexível, permite uma certa proximidade com a máquina, apesar de "ser" uma linguagem de alto nível, mas às vezes não é possível fazer tudo somente com ela, aí é preciso recorrer novamente ao assembly, aí vem a dúvida: como integrar os dois?

Todos os compiladores permitem que código assembly seja colocado diretamente dentro do programa, aí cada compilador cria suas conveções de como fazer isso. No ICC11 utiliza-se a função asm(). Assim para executar um LDAA $7000 dentro do seu programa basta:

```
void main() {

   asm(" LDAA $7000");

}
```

É bom deixar bem claro, o espaço entre a primeira aspa, e a instrução LDAA é NECESSÁRIO. NUNCA se esqueçam disso.

10.1 Utilizando variáveis do C no assembly

O compilador ICC11 permite que variáveis criadas no seu programa em C seja utilizadas no assembly inline. Para isso basta colocar o símbolo % antes do nome da variável. Como mostrado abaixo:

```
void main() {
   char a;

   asm(" STAA %a"); /* Salva valor do ACCA na variável a */

}
```

11 Conclusão

Durante o curso vocês verão que o uso do C realmente facilita a programação das tarefas propostas. Algumas delas bem trabalhosas, mas com um objetivo muito bom, conhecer o funcionamento do hardware, e controlá-lo através de um software.

A grande proximidade com a máquina não termina agora, muito menos o uso dos manuais do HC11. Na verdade agora é que começa realmente as intermináveis leituras de manuais, pois agora não é só ver o que determinada instrução faz, mas sim ver como funciona determinada interface de I/O, determinada porta, e coisas assim.

O acesso direto à memória também continua, agora não mais através de instruções como LDAA, mas com a utilização dos ponteiros.

Não pensem que agora vai ficar fácil demais, mas também não pensem que vai ficar mais difícil porque os programas serão mais complexos, simplesmente será diferente, uma nova experiência na programação e conhecimento do funcionamento da máquina.

Espero que todos "se divirtam". Nem tanto, mas que todos possam aproveitar ao máximo esta oportunidade (que para muitos será única).