C para microcontroladores

Hans-Jörg Schneebeli

Objetivo

O objetivo é mostrar como se pode programar dispositivos, como microcontroladores, com restrições severas de recursos (memória e velocidade), de uma maneira eficiente usando C. É suposto que o leitor já tenha conhecimentos básicos de C.

Introdução

Devido a restrições de custo, tamanho e consumo, em sistemas embarcados são usados microcontroladores com pouca memória e baixa velocidade. Deve ser notado que o custo de programação é um custo não recorrente, e que o custo de um microcontrolador maior se refletirá em um custo maior por cada unidade fabricada. Além disso, deve ser notado ainda que um tempo maior de desenvolvimento aumenta o custo e diminui o tempo de mercado de um produto.

Por fim, geralmente, existem restrições de tempo de resposta que exigem técnicas eficientes de programação.

Alternativas para programação

Entre as linguagens usadas ou projetadas para uso em sistemas embarcados, destacam-se:

- *Assembly* – possibilita código mais eficiente em termos de velocidade e uso de memória, mas é inerentemente não portável e exige tempo muito grande de desenvolvimento.
- C – permite tempo menor de desenvolvimento, mas exige uma disciplina do programador, similar a exigida em *Assembly*. Não tem suporte direto a concorrência.
- C++ - permite o uso de técnicas de programação orientada a objetos, mas apresenta problemas de tamanho de memória e de velocidade. Para diminuir estes problemas existe uma versão mais enxuta, chamada

Embedded C++, que impede o uso de mecanismos que aumentam demasiadamente o consumo de memória, ou tornam o comportamento temporal dificil de prever. Entre os mecanismos descartados estão herança multipla, exceções e *templates*.

- Java – embora tenha sido inicialmente planejada para sistemas embarcados, a demanda por memória é excessiva para a maior parte das aplicações. O fato de não ser o comportamento no tempo não ser previsível devido ao *garbage collector* impede o seu uso em certas aplicações.
- Ada – foi desenvolvida especialmente para sistemas embarcados, tem suporte embutido a concorrência, existe um padrão para a linguagem, mas é verbosa e o código resultante é maior e mais lento do que o gerado por C.
- Esterel – representa uma alternativa interessante para a programação de sistemas reativos, como é o caso de sistemas embarcados.
- Outras – Existem ainda linguagens como Forth e dialetos de Basic ou Pascal, mas o uso destes é muito raro.

Originalmente, os sistemas embarcados eram desenvolvidos em *Assembly*. Com o aumento de complexidade destes sistemas e a pressão por tempos menores de desenvolvimento, o uso exclusivo de *Assembly* tem diminuido, e hoje alcança apenas cerca de 8% dos projetos. Devido a facilidade de se construir um compilador C, e o fato de que a linguagem implementa um nível de abstração bastante próximo ao hardware, a linguagem C representa um excelente compromisso entre eficiência e facilidade de programação. Atualmente, a absoluta maioria, cerca de 90% dos sistemas, são desenvolvidos em C/C++, as vezes com alguma parte pequena em *Assembly*. O uso do Embedded C++ é crescente mas ainda representa uma parte menor do uso do C. Finalmente o restante é feito basicamente em Java, mas é de se esperar o aumento de seu uso, principalmente em sistemas como celulares e PDAs.

A linguagem C

C foi desenvolvido por Richie e Kernighan na década de 70 para servir como ferramenta de um sistema operacional, que mais tarde passou a ser conhecido como Linux. A ênfase foi em desempenho e portabilidade. O usuário típico seria programador experiente, que necessitasse acessar recursos de hardware, manipular bits e gerenciar memória.

O UNIX foi usado em ambientes universitários na década de 80. Pelo fato de um compilador C fazer parte da maior parte das instalações UNIX, muito software foi escrito usando este compilador já disponível. Além disso, no final da década passou a haver uma versão livre de um compilador C bastante eficiente

(chamando GNU CC, ou mais popularmente gcc). Assim o C acabou sendo usado no desenvolvimento de aplicações fora de suas intenções originais. Isto por exemplo, é o caso de software matemática baseado em manipulação de matrizes, onde o suporte da linguagem é catastrófico.

O resultado é que a linguagem C é uma das mais usadas atualmente e é base para várias outras linguagens como C++, Java e C#. No entanto, é no desenvolvimento de sistemas embarcados que são, por natureza, mais próximos as intenções originais da linguagem que ele é mais usado, representado uma alternativa ao uso de linguagens de montagem (assembly) com resultados muito bons. Embora um bom programador usando assembly possa gerar código menor e mais rápido que o gerado por um compilador C, atualmente a diferença em desempenho e tamanho não é relevante. No pior dos casos, o trecho crítico pode ser programado em Assembly e o restante em C.

A diferença de tamanho de código gerado e de desempenho devido ao uso do C ou do assembly é dependente da arquitetura do microprocessador alvo. Em algumas arquiteturas, com conjunto de instruções mais adequados, a diferença pode ser da ordem de 10%, em outras de até 300%.

O uso do C implica na necessidade de disciplina por parte do programador. Dois erros são muito comuns:

- Usar uma área que já foi liberada (*dangling pointers*)
- Usar além da área alocada (*memory overflow*)

Este são responsáveis por uma grande quantidade de falhas de segurança em códigos para PC. Algumas linguagens como Java e C# possuem um esquema automático de liberação da área e de controle de acesso. Isto, no entanto, representa um custo adicional, em termo de memória e de velocidade de execução além do aceitável para o maior parte dos projetos de sistemas embarcados. Como a linguagem C deixa por conta do programador o controle de acesso, este deve colocar explicitamente os testes de verificação de limites.

As características principais da linguagem são:

- Estrutura simples com poucas palavras chave.
- Operações de entrada e saída são feitas por meio de biblioteca.
- Flexibilidade de implementação.
- Manuseio de memória por meio de apontadores.
- Operadores incremento e decremento.
- Portabilidade.

O modelo de execução é simples, baseado em uma estrutura de pilha. No

momento em que uma rotina é chamada, suas variáveis locais são alocadas. Quando de sua execução, diretamente apenas suas variáveis locais e as globais (definidas fora do contexto de qualquer rotina) são acessíveis. Quando a rotina se encerra, a execução continua na rotina que a chamou, restaurando o acesso a suas variáveis locais.

Figura: A chama B, que chama C

Para garantir a portabilidade, a especificação de tamanho de variáveis em C é extremamente flexível. Ela estabelece apenas o número mínimo de bits de cada tipo, como mostrado na tabela abaixo.

		Tamanhos Tipicos usados em Compiladores				
Tipo	**mínimo**	**8-bits**	**16 bits**	**32 bits**	**64 bits (ILP64)**	**64 bits (LP64)**
char	8	8	8	8	8	8
short	16	16	16	16	16	16
int	16	16	32	32	32	64
long	32	32	32	64	64	64
long long*	64	-	-	64	128	128
ponteiro	16	16	32	64	64	64
float	32	32	32	32	64	64
double*	32	32	32	64	64	64
long double*	128	-	-	128	128	128

* Padrão C9X

Outras áreas que não se pode garantir é o tipo de operação de deslocamento para a direita (se repete ou não o bit de sinal) e a representação de números negativos.

Versões do C

KR Versão original. não havia checagem de parâmetros (exceto por uma ferramenta externa chamada lint).

C89 Versão padronizada pela ANSI. Bibliotecas padronizadas. São introduzidos protótipos de rotinas e a palavra chave void indica ausência de parâmetros ou de valor de retorno.

C99 Tipos padrão de tamanho determinado (stdint.h). Comentários com //.

Uso em Microcontroladores

Em muitas arquiteturas de microprocessadores, as pilhas são implementadas de maneira muito ineficiente e/ou apresentação limitações sérias para o tamanho ou conteúdo. Por exemplo, pilhas no Microchip PIC podem conter em alguns caso, só oito endereços de retorno.

MISRA
Cyclone

A maior parte do compiladores para microcontroladores atuais usa ainda o padrão C89. Os itens do padrão C9X estão sendo paulatinamente incorporados. Por exemplo, quase todos já aceitam comentários do tipo //.

Tipos de Dados

Os tipos padrão de dados são inadequado para o uso em sistemas embarcados, se houver necessidade de definição de tamanho. Existem então algumas alternativas para definição de variáveis com tamanho conhecido:

- Padrão (somente C9X)
- Uso de preprocessador
- Uso de definição de novo tipo

O padrão C9X estabelece um cabeçalho que define um conjunto de tipos padrão com tamanho pré-estabelecido. A implementação pode ser feita pelo compilador de diversas maneiras, inclusive usando simplesmente uma definicão de novo tipo. O arquivo cabeçalho é o stdint.h e define, entre outros,

inteiros com sinal	int8_t	int16_t	int32_t	int64_t
inteiros sem sinal	uint8_t	uint16_t	uint32_t	uint64_t

Quando não houver este arquivo, uma maneira ainda comum é o uso de pré-processador para definir os tipos abaixo. Isto deve ser ajustado de acordo com o compilador e o processador.

BYTE	Byte	8 bits	unsigned char
WORD	Word	16 bits	unsigned short
INT	Int	16/32 bits	
LONGWORD	Long	32 bits	

BYTE	Byte	8 bits	unsigned char
POINTER	Pointer	32 bits	

Usam-se todos os caracteres em maiúsculo ou apenas o inicial, para evitar confusão com palavras chave, que sempre são em minusculas.

Existem duas maneiras de definir estes tipos:
Usando o pré-processador.

```
#define BYTE unsigned char
#define WORD unsigned short
#define LONG unsigned long
```

ou usando typedef

```
typedef unsigned char BYTE;         /* 8 bits */
typedef unsigned short WORD;        /* 16 bits */
typedef unsigned long LONG;         /* 32 bits */
```

Uma vantagem do segundo esquema é que os tipos definidos são conhecidos pelo compilador. No esquema usando o pré-processador, todas as referencias aos tipos são substituídas pelos equivalentes antes que o compilador processe o código fonte.

Para se descobrir as características importantes do processador e do compilador, ver o apêndice A.

Entrada e Saida

O acesso a registradores de CPU, como apontador de pilha ou contador de programa, em geral só é necessário, na inicialização. Durante a execução, o modelo de execução de C controla estes registradores.

Em sistemas embutidos, o mais importante é o acesso a registradores que controlam o hardware. O modo deste acesso é dependente da arquitetura do processador e existem basicamente duas maneiras:

- acesso por instruções especiais
- acesso como posições de memória

Embora arquiteturas mais modernas não usem mais instruções especiais para acesso a hardware, a arquitetura do Intel 8086 e seus derivados usa esta forma. Existem então instruções especiais (no caso, IN e OUT) e um espaço de

endereçamento separado para os registradores de hardware. Externamente, deve haver um pino que sinalize qual a forma de endereçamento usado. A desvantagem é que este pino poderia ser usado para duplicar a quantidade de memória endereçavel.

Para o acesso em C, usam-se rotinas de bibliotecas para o acesso a estes registradores. Tipicamente para acesso a registradores de 8 bits, há as rotinas inportb e outportb, que são usadas como mostrado abaixo.

```
WORD e = 0x243;          // endereco
...
BYTE b = inportb(e);     // leitura
...
outportb(e,b);           // escrita
```

Existem variações para acesso a registradores do tipo WORD com 16 bits (inportw e outportw). Além disso, em muitos casos, por motivos de eficiencia pode-se inserir diretamente o código assembly na posição do chamado da rotina, evitando-se o curso de passagem de parametros e chamado e retorno da rotina. Se o compilador aceitar uma extensão não padrão de inserção de código assembly no meio de código C, a rotina acima pode ser definida (preferencialmente em um arquivo cabeçalho) através das instruções de pré-processador abaixo.

```
#define inportb(e)       asm("     IN   AL,e;");
#define outport(e,b)     asm("     MOV  AL,e"; OUT e,AL;");
```

A tendência atual é o uso de endereços de regiões de memória para acesso ao registradores de hardware. A vantagem é que podem ser usadas as instruções normais de acesso a memória e a desvantagem, é a diminuição da área de memória realmente utilizável.

O mecanismo básico é o uso de apontadores, como mostrado abaixo.

```
BYTE p = (BYTE *) 0x876; // inteiro convertido para ponteiro
BYTE b;
...
     b = *p;             // leitura
     *p = '\x12';             // escrita
...
```

Neste caso, um apontador é inicializado com o valor inteiro. O cast (BYTE *) só é usado para indicar o compilador, que o padrão de bits a seguir (um inteiro) seja interpretado como um apontador. A seguir ele é usado para um acesso de leitura e outro de escrita. Como observação, '\x11' poderia ser escrito também 0x11, mas

alguns compiladores, usariam uma instrução para manipulação de inteiros para manipula-lo e usaria somente os bits menos significativos para a operação de escrita.

Isto tudo pode ser feito em uma unica instrução de leitura e outra de escrita.

```
BYTE b;
...
     b                 = *((BYTE *) 0x876);  // leitura
     *((BYTE *) 0x876) = '\x12';             // escrita
...
```

Para facilitar ainda mais, pode ser usado o pré-processador definindo um simbolo como no exemplo abaixo, que sem duvida é bem mais legivel.

```
#define PORTA *((BYTE *) 0x876)
...
BYTE b;
...
     b = PORTA;
     PORTA = '\x12';
...
```

O conjunto de definições deste tipo pode ser colocado em um arquivo cabeçalho, especifico para um modelo de uma arquitetura contendo as definições de todos os registradores de hardware do dispositivo. Assim, tem-se código com leitura mais facil e ao mesmo tempo, um nome padrão para se acessar estes registradores.

Deve ser levado em conta que o compilador no esforço de otimizar o código pode prejudicar o funcionamento deste tipo de acesso. Isto acontece, por que o compilador assume, que o programa é o único responsável pela modificação dos valores em memória e procura manter o valor em um registrador de acesso mais rápido. Assim uma estrutura do tipo abaixo, que espera que o valor da porta seja zero

```
     while ( PORTA != 0 ) {}
```

resulte em um código do tipo

```
          MOVE 0x876,R0  ; move valor da memória para R0
LOOP:     BNZ  R0,LOOP   ; testa se valor é diferente de zero
                         ; e repete o teste sem buscar
                         ; o novo valor na memória
```

A solução é indicar ao compilador que uma determinada posição de memória

pode ter o valor modificado independente da ação do programa. Isto é feito com a palavra chave volatile. Assim, definindo-se PORTA através da instrução abaixo, resolve-se este problema.

```
#define PORTA *( (volatile BYTE *) 0x876)
```

Manipulação de Bits

Na programação de sistemas embarcados é muito importante e freqüente a necessidade de se setar um bit (fazê-lo igual a 1), resetar um bit (fazê-lo igual a 0), complementar um bit (trocá-lo de 1 para 0 ou vice-versa) e finalmente testar o valor.

O C padrão não tem suporte para variáveis tipo tipo booleano (embora C9X especifique variáveis tipo bool_t, que podem assumir os valores false e true, através de um arquivo cabeçalho stdbool.h).

Também não tem suporte para a definição de constantes inteiras em formato binário. É comum, no entanto, que compiladores implementem como extensão, constantes do tipo 0b0101 ou 0B0111, valendo 5 e 7 respectivamente. O correto é usar a representação decimal (5 ou 7) para valores de contadores e a representação hexadecimal (0x05 ou 0x07 ou ainda, em caso de bytes, '\x05' ou '\x07'), para a manipulação de bits.

Usando structs

Uma das maneiras mais fáceis para acessar bits é usando o fato pouco conhecido de campo de bits em estruturas. Por exemplo, a declaração abaixo, especifica os bits de um tipo byte.

```
typedef struct {
        unsigned        bit7 : 1;
        unsigned        bit6 : 1;
        unsigned        bit5 : 1;
        unsigned        bit4 : 1;
        unsigned        bit3 : 1;
        unsigned        bit2 : 1;
        unsigned        bit1 : 1;
        unsigned        bit0 : 1;
} Port;
```

Uma variável do tipo Byte pode ser definida através de

```
Port x;
```

e os bits individuais através de

```
    x.bit4 = 1;
```

```
    x.bit2 = 0;
    x.bit1 = ~x.bit1;
    if( x.bit3 ) ....
```

Deve ser chamada a atenção para o fato que o tamanho da tipo de dado, sizeof(Port), acima não precisa ser exatamente um byte. Embora isto seja comum em compiladores para microcontroladores, Em compiladores para PC é usual que o tamanho da estrutura seja igual ao de um inteiro, isto é, sizeof(byte) igual a sizeof(int).

Um compilador de boa qualidade usará as instruções de manipulação de bits do processador, se este as tiver. Em outros casos, as instruções usarão as operações bit-a-bit que são descritas a seguir.

Um dos problemas é que não se pode acessar todos os bits em bloco, o que pode resultar em código ineficiente. Para isto, usa-se uma construção obscura do C que define dois tipos de dados ocupando a mesma região de memória.

```
typedef union  {
    BYTE val;
    struct {
        unsigned        bit7 : 1;
        unsigned        bit6 : 1;
        unsigned        bit5 : 1;
        unsigned        bit4 : 1;
        unsigned        bit3 : 1;
        unsigned        bit2 : 1;
        unsigned        bit1 : 1;
        unsigned        bit0 : 1;
    };
} Port;
```

Assim é possível se acessar os bits de forma individual ou todo o registrador como mostrado abaixo.

```
Port PORTA;
...
PORTA.bit7 = 1;
if ( PORTA.bit5 ) {
    ....
}
...
PORTA.val = 0;
...
```

Usando operações bit-a-bit

Para a manipulação de bit em variáveis inteiras podem ser usadas as operações

Operação	Operador C
E	&
OU	|
OU-Exclusivo	^
Negação	~

Estes não devem ser confundidos com os operadores lógicos E (&&),OU (||) e de negação (!), que retornam o valor 0 ou 1.

Para se setar um bit em uma variável deve-se usar uma mascara, que tem todos os bits zero, exceto o bit na posição desejada. Por exemplo, para se setar o bit 2 deve-se usar a mascara 000000100, que em C, é representada por 0x04 em formato hexadecimal, 4 (formato decimal) ou 04 (formato octal). A operação

```
x = x | 0x4;
```

ou melhor

```
x |= 0x4;
```

seta o bit 2 do inteiro do tipo char, short, int ou long.

Problemas com acesso compartilhado

Do mesmo modo que os valores presentes em uma porta de um microcontrolador podem variar, valores de uma variável podem variar em função de outros processos ou de interrupção. A linguagem C não tem suporte direto a concorrência. Isto exige que o programador tenha claro os perigos envolvidos.

Processos concorrentes são implementados em C através de bibliotecas.

Interrupções

Não existe uma maneira padrão de se definir rotinas de interrupção. Embora uma rotina de interrupção seja semelhante a um rotina normal que não retorne nenhum valor e também não receba nenhum parâmetro (com o protótipo void intproc(void)), ela apresenta características especificas:

- O comando de retorno de uma rotina de interrupção não é idêntico ao do retorno de uma rotina comum.
- A rotina de interrupção deve restaurar todos os registradores, mas as rotinas comuns podem, em muitos sistemas, modificar alguns registradores (chamados registrados de rascunho (*scratch registers*), sem que seja necessário restaurá-los.

Portanto, o compilador deve ter a informação se uma rotina é para tratamento de interrupção para gerar o código correto. A maneira tradicional é o uso da palavra chave interrupt (em alguns casos, __interrupt, pois simbolos iniciados com sublinhado são de uso exclusivo da implementação). Assim uma rotina de interrupção seria descrita como

```
interrupt void intproc(void) {
     ...
}
```

A maneira mais moderna, compativel com o padrão, é o uso do comando pragma para passar esta informação. Assim a rotina acima seria descrita como

```
#pragma interrupt
void intproc(void) {
     ...
}
```

Isto, no entanto, em sistemas com vetores de interrupção, só resolve parte do problema, pois ainda é necessário que o processador saiba associar uma interrupção a uma rotina de interrupção. Quando o vetor de interrupção estiver em memória RAM, este deve ser inicializado quando do inicio da execução do programa.

Por exemplo, em uma máquina de 16 bits (endereços também com 16 bits) com o vetor de interrupção para 8 níveis nas 16 primeiras posições de memória, o endereço da rotina de interrupção de nível n é obtido com o código

```
     WORD endereco = *( (WORD *)( n*2 ));
```

Em geral, existe uma rotina do tipo setintproc, que modifica uma posição do vetor de interrupção para que aponte para a rotina especificada. Ela retorna um apontador para a rotina que estava encarregada da interrupção, Existem duas maneiras de se definir esta rotina, uma usando typedef, que é a forma mais legível, como mostrado abaixo.

```
typedef void (*pfunc)(void);
```

```
pfunc setintproc(int level,pfunc f) {
     ...
}
```

A outra maneira define a mesma função de forma direta, sem auxilio de um tipo de dados, que faz com que fique menos legível.

```
void (*setintproc(int level, void (*f)(void) ))(void) {
          ...
}
```

A forma de usar é idêntica, só havendo diferença na definição do ponteiro da função.

```
void func(void) {
     ...
}

int main(void) {
void (*oldfunc)(void); // poderia ser pfunc oldfunc;
     ...
     oldfunc = setintproc(10,func);
     ...
}
```

Deve ser observado que a notação `pfunc oldfunc` pode ser usada com a definição direta e vice-versa. Ainda deve ser observado que pode ser necessário, embora não seja comum, que a especificação da rotina inclua a palavra chave interrupt.

Mapeamento de memória

Programas em C, uma vez compilados, possuem diversos tipos de segmentos para os vários tipos de informação necessários em um código objeto.

- Código
- Constantes
- Dados inicializados
- Dados não inicializados
- Pilha

Em principio, o código objeto é relocável, isto é, pode ser adaptado para ser

executado em qualquer endereço de memória. A tarefa do ligador é juntar estes códigos objeto, colocá-los nas posições adequadas de memória e acertar as referências. O acertar de referências mencionado é ajustar uma instrução de chamado de rotina para que tenha o endereço correto do ponto de entrada. Isto só pode ser feito, depois do mapeamento dos diversos segmentos dos códigos objeto nas regiões de memória adequadas. O procedimento básico é concatenar os segmentos de diversos tipos separadamente. Por exemplo, todos os segmentos código são concatenados gerando o segmento código do módulo executável.

Em termos de hardware, os microcontroladores tem uma região de memória não volatil (ROM) e uma de memória volátil (RAM). Em alguns casos, estas não são contíguas.

Figura

O ligador deve então mapear os segmentos de código e constantes para uma região de ROM e os segmentos de dados (inicializados ou não) em uma área de RAM. Além disso, deve manter uma cópia da área de dados inicializados em ROM para que seja copiada para a região adequada na RAM durante a inicialização. Isto é uma duplicação desnecessária e que desperdiça um recurso valioso.

Os compiladores C determinam qme que segmento uma variável será armazenada (se constante ou dado inicializado) pelo contexto ou, acordo com instruções fornecidas pelo programador.

Nas versões antigas de C havia um problemint x[10];

int x[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

char *p = “texto”;

O problema de alinhamento

Alocação dinâmica de memória

Referencias

Kernighan, Richie. *The C Programming Language*.
Jack Ganssle. *Microcontroller C Compilers*. Electronic Engineering Times, November 1990.
Jack Ganssle. *My love-hate relationship with C*. Embedded.com. 10/16/03.
Wikipedia. Linguagem de programação C.
James A C Joyce. *Why C Is Not My Favourite Programming Language*. http://www.kuro5hin.org/print/2004/2/7/144019/8872
Harsha S. Adiga. *Porting Linux applications to 64-bit systems*.http://www-128.ibm.com/developerworks/linux/library/l-port64.html. 12 Apr 2006.

MISRA. *Guidelines for the use of the C language in critical systems*. http://www.misra-c2.com/

The Open Group. *64-Bit Programming Models: Why LP64?*. http://www.unix.org/version2/whatsnew/lp64_wp.html.

Apêndice A

O programa abaixo pode ser usado para descobrir as caracteristicas do processador e do compilador. Para isto, ele deve ser compilado no computador hospedeiro e executado no processador alvo.

```
/*
 * imprime tamanhos de tipos de variaveis em C
 *
 * Hans 2006
 */

#include <stdio.h>


int main (void) {
unsigned char b,c;
unsigned short s = 0x1234;
unsigned int i = 0x1234;

  b = 1;
  c = 0;
  while( b ) {
      b <<= 1;
      c++;
  }
  printf("Caracteristicas do processador\n");
  printf ("Tamanho de um Byte .. %3d bits\n\n",c);
  if( *((unsigned char *) &s) == 0x12 ) {
     printf("Big Endian (Byte MAIS significativo primeiro)\n");
  } else if( *((unsigned char *) &s) == 0x34 ) {
     printf("Little    Endian    (Byte    MENOS    significativo
primeiro)\n");
  } else {
     printf("Endianness indeterminado\n");
  }

  printf ("Tamanhos tipos em Bytes\n");
  printf ("char ................ %3d\n", sizeof (char));
  printf ("unsigned char ....... %3d\n", sizeof (unsigned char));
  printf ("signed char ......... %3d\n", sizeof (signed char));
  printf ("int ................. %3d\n", sizeof (int));
  printf ("unsigned int......... %3d\n", sizeof (unsigned int));
  printf ("signed int...,,...... %3d\n", sizeof (signed int));
  printf ("short int ........... %3d\n", sizeof (short int));
   printf ("unsigned short int .. %3d\n", sizeof (unsigned short
int));
   printf ("signed short int .... %3d\n", sizeof (signed short
```

```
int));
  printf ("long int ..... %3d\n", sizeof (long int));
   printf ("signed long int ..... %3d\n", sizeof (signed long
int));
   printf ("unsigned long int ... %3d\n", sizeof (unsigned long
int));
  printf ("float ..... %3d\n", sizeof (float));
  printf ("double ..... %3d\n", sizeof (double));
  printf ("long double ..... %3d\n", sizeof (long double));

#ifdef OUTRAS
   printf( "long char ..... %3d\n", sizeof(long char) );
    printf( "unsigned long char . %3d\n", sizeof(unsigned long
char) );
   printf( "short char ..... %3d\n", sizeof(short char) );
    printf( "unsigned short char. %3d\n", sizeof(unsigned short
char) );
   printf( "long float ..... %3d\n", sizeof(long float) );
   printf( "short float ..... %3d\n", sizeof(short float) );
   printf( "short double ..... %3d\n", sizeof(short double) );
#endif

  return 0;
}
```

Se o processador alvo, não tiver um dispositivo de saída ou não dispuser da biblioteca padrão, deve-se usar o esquema