

# Laboratório de Arquitetura de Computadores

## Guião 8

### Programação em linguagem C

#### 1 Objetivos

Com este guião pretende-se que o leitor pratique a utilização da linguagem C para programação de aplicações completas com o PEPE-16, incluindo as que envolvem tempo real e múltiplas atividades concorrentes (processos).

É assumido conhecimento prévio da linguagem C. Este guião não constitui um tutorial da linguagem C, concentrando-se nas especificidades da implementação e utilização desta linguagem num processador simples como o PEPE-16, nomeadamente na sua relação com as interrupções e com periféricos, bem como a sua utilização para aplicações de tempo real e/ou com processos.

O compilador de C do PEPE-16 cumpre as características básicas de um compilador standard, mas tem algumas limitações e especificidades, que são descritas na secção 3.

Os aspetos mais relevantes incluem:

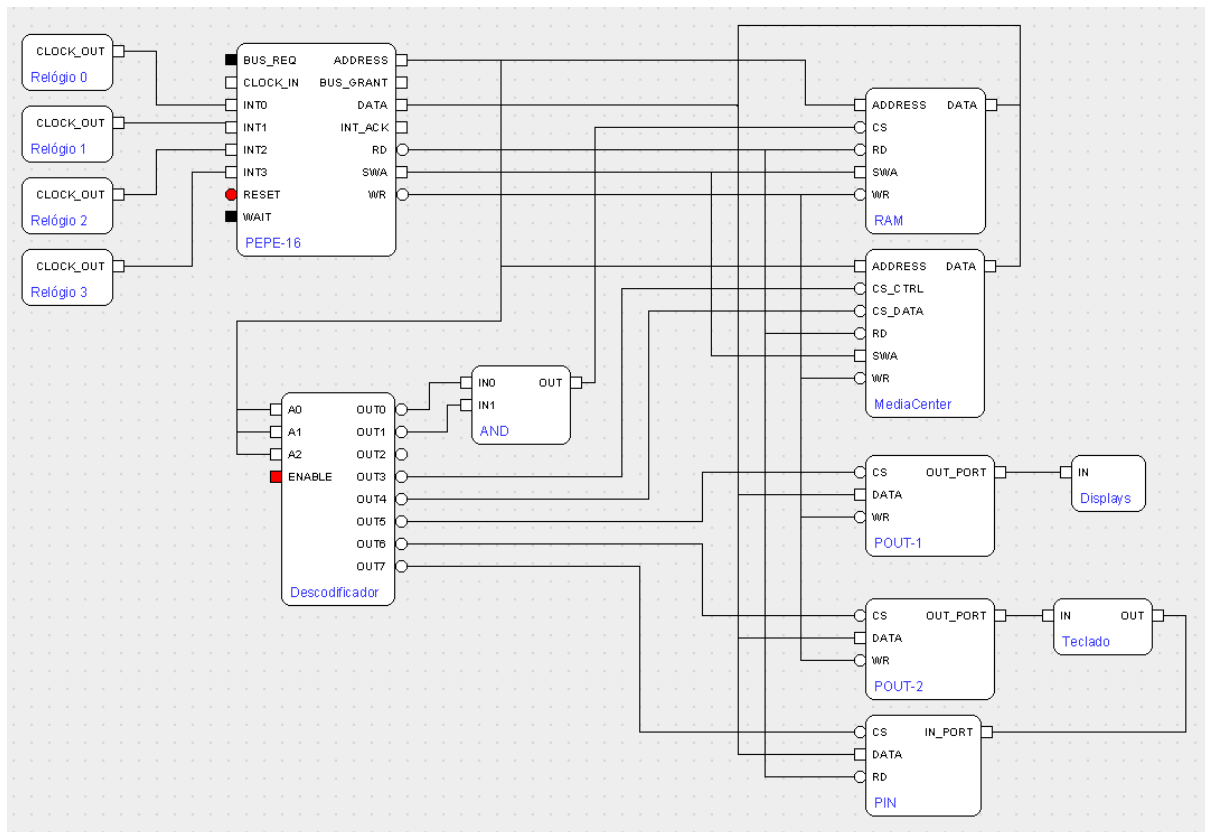
- Acesso a endereços específicos;
- Visualização do código assembly gerado pelo compilador de C;
- Suporte para interrupções;
- Suporte para processos;
- Debugging de programas em C.

Aparte os ficheiros com alguns programas ilustrativos da programação em C no PEPE-16, a maior parte da informação contida neste guião está disponível no próprio Help do simulador.

#### 2 O circuito de simulação

Para executar os programas de exemplo em C, use o circuito contido no ficheiro **lab8.cir**. Este circuito é igual ao que já foi usado nos guiões de laboratório 6 e 7, com um ecrã, quatro relógios, dois displays e um teclado.

O PEPE-16 funciona quer com programas em assembly quer em linguagem C. O programa a carregar no processador precisa de ter a extensão adequada (terminar em “.asm” ou “.c”, respetivamente).



### 3 Características e limitações do compilador de C do PEPE-16

#### 3.1 Limitações do compilador de C

O PEPE-16 é um processador simples e o seu compilador de C implementa apenas os recursos necessários para implementar aplicações básicas. Está ao nível do primeiro padrão da linguagem C (C89), com algumas limitações. Em particular, o compilador PEPE-16 C não suporta:

- Compilação separada. Todo o programa deve estar contido num único ficheiro “.c”;
- Ficheiros de cabeçalho (ficheiros “.h”). Todo o programa deve estar contido num único ficheiro “.c”;
- Bibliotecas (por exemplo, *stdio*). Com este compilador de C, as entradas e saídas devem ser realizadas através de periféricos, localizados em endereços que podem ser especificados com ponteiros (secção 4);
- Tipos de dados para além de **int**, como **long**, **float** e **double**;
- Estruturas, uniões e enumerações;
- As seguintes palavras-chave (e a funcionalidade associada):
  - **auto**
  - **double**
  - **enum**
  - **extern**
  - **float**

- **long**
- **register**
- **struct**
- **typedef**
- **union**
- **unsigned**
- **volatile**

No entanto, suporta características importantes, como aritmética de ponteiros, arrays multidimensionais, conversões de tipos e definições *forward* de funções.

Além disso, inclui diversas funcionalidades destinadas a lidar com o hardware PEPE-16, nomeadamente suporte para:

- Interrupções;
- Colocação de código, dados e áreas de pilha (para começar em endereços específicos);
- Processos (com várias instâncias de processos, variáveis Lock e diretivas Yield e Wait);
- Expansão em linha de instruções em linguagem assembly dentro de uma função C, com a capacidade de aceder a variáveis C a partir dessas instruções).

### 3.2 Tipos de dados

O compilador PEPE-16 C suporta os seguintes tipos de dados:

- **int** (16 bits, com sinal);
- **short** (8 bits, com sinal);
- **char** (8 bits, sem sinal);
- **void** (para funções que não devolvem valor);
- arrays multidimensionais;
- ponteiros para int, short, char, arrays e funções;

A aritmética de ponteiro é suportada.

Os identificadores de array e de funções podem ser utilizados como ponteiros, permitindo assim arrays de ponteiros para funções, por exemplo (útil para invocar uma função escolhida em tempo de execução a partir de uma tabela de ponteiros para funções).

O operador *address* (&) pode ser aplicado a um *Lvalue* (um item com endereço, tal como uma variável global ou estática, ou uma função) para obter o seu endereço, normalmente para inicializar uma variável de tipo ponteiro.

As variáveis podem ser declaradas constantes (com **const**) e as variáveis locais podem ser declaradas estáticas (com **static**).

Como é habitual em C, os ponteiros e os valores para os quais apontam podem ser declarados constantes de forma independente (por exemplo, `const int* const p` – o primeiro **const** torna constante o valor apontado e o segundo torna constante o próprio ponteiro).

A palavra-chave **sizeof** pode ser aplicada a uma expressão para obter o número de bytes (endereçáveis individualmente) ocupados pelo seu tipo de dados de resultado. Isto é particularmente útil com arrays.

A palavra-chave **sizeof** também pode ser aplicada a uma especificação de tipo, que deve ser colocada entre parênteses.

### 3.3 Inicialização de variáveis globais e estáticas

É típico inicializar variáveis diretamente na declaração (por exemplo, `int x = 2;`).

No entanto, é importante compreender que as variáveis globais (definidas fora de qualquer função) e as variáveis estáticas (variáveis locais qualificadas como “static”) geram diretamente declarações de variáveis em linguagem assembly (por exemplo, diretivas `WORD` e `BYTE`).

Estas diretivas incluem um valor inicial, que é inicializado apenas quando o programa é carregado e não quando é reiniciado. Isto pode produzir resultados inesperados quando estas variáveis são alteradas pelo programa, o utilizador interrompe o programa por algum motivo e o reinicia. As variáveis globais e estáticas não serão inicializadas novamente.

Assim, é recomendado simplesmente declarar variáveis globais e estáticas sem as inicializar (por exemplo, `int x;`), e depois realizar a sua inicialização em alguma função (por exemplo, `x = 2;`), uma vez que o código é sempre executado de cada vez que o programa é executado.

As constantes globais e estáticas (por exemplo, `const int x = 2;`) devem ser inicializadas na declaração, mas como não podem ser alteradas, este problema não se coloca.

Os arrays também podem ser declarados e inicializados na declaração (por exemplo, `int y[4] = {3, 5, 7, 9};`), mas se forem variáveis globais ou estáticas não constantes e forem alteradas pelo programa, terão o mesmo problema, pelo que a recomendação é declará-los primeiro e inicializá-los por código. Infelizmente, os arrays não podem ser atribuídos como um todo em C, pelo que deve ser utilizado um ciclo para inicializar um array, um elemento de cada vez.

As variáveis locais (declaradas dentro de alguma função) não marcadas como estáticas são armazenadas na pilha (para suportar recursividade) e são inicializadas por código. Portanto, podem ser inicializados com segurança na declaração.

Se forem inicializadas variáveis globais ou estáticas (constantes ou não) na sua declaração, o inicializador pode ser uma expressão, mas deve ter um valor constante, calculável em tempo de compilação.

## 4 Aceder a endereços de hardware específicos

O compilador de C trata da localização em memória das variáveis e do código. No entanto, é necessário o acesso a endereços de hardware específicos para aceder a periféricos de entrada e saída, uma vez que estes se encontram em endereços definidos pelo hardware de descodificação de endereços do sistema com o PEPE-16.

O acesso a um endereço definido pelo utilizador (seja uma célula de memória ou um porto de um periférico) é feito através da definição de um ponteiro com o valor do endereço necessário, o que pode ser feito de diversas formas, descritas nas secções seguintes.

## 4.1 Conversão de um valor de endereço para um ponteiro

Esta é a forma mais simples. Basta converter um valor (com o endereço para aceder) num ponteiro, o que permite utilizar o acesso por ponteiro em C. No entanto, note que o acesso pode ser em 16 ou 8 bits, para os quais são utilizados ponteiros `int*` ou `char*`, respetivamente, como se mostra no exemplo seguinte.

```
const int ender = 0x6000;           // o endereço da célula ou porto a aceder
int* const pw = (int *) ender;     // pw aponta agora para uma célula ou porto de 16 bits
char* const pb = (char *) (ender + 1); // pb aponta agora para uma célula ou porto de 8 bits
                                     // o que é constante é o ponteiro, não a célula ou porto
int x;                             // variável de 16 bits
char y;                            // variável de 8 bits
void main(){
    *pw = 3;                        // escreve a célula ou porto de 16 bits no endereço 0x6000
    x = *pw;                       // lê a célula ou porto de 16 bits no endereço 0x6000
    *pb = 3;                        // escreve a célula ou porto de 8 bits no endereço 0x6001
    y = *pb;                       // lê a célula ou porto de 8 bits no endereço 0x6001
}
```

**NOTA** – Ponteiros para **int** deve ter um valor par, restrição imposta pelo endereçamento de palavras (16 bits) do PEPE-16. Os ponteiros para **char** não têm esta restrição, pois referem-se a bytes (8 bits), que o PEPE-16 pode endereçar individualmente.

## 4.2 Aritmética de ponteiros

Quando aceder a um intervalo de células ou portos de periféricos (por exemplo, comandos para o módulo MediaCenter), é conveniente definir um endereço base (a partir de um **int** com cast) e depois adicionar-lhe um valor, atuando como um índice no modo de endereçamento indexado do PEPE-16. Isto é ilustrado pelo exemplo seguinte.

```
const int base = 0x6000;           // a base do intervalo de endereços a aceder
int* const pw = (int *) base;     // pw aponta agora para a primeira célula ou porto de 16 bits
char* const pb = (char *) base;   // pb aponta agora para a primeira célula ou porto de 8 bits
                                     // o que é constante é o ponteiro, não a célula ou a porto
const int i = 5;                  // índice
int x;                            // variável de 16 bits
char y;                           // variável de 8 bits
void main(){
    *(pw + i) = 3;                 // escreve a célula ou porto de 16 bits no endereço 0x600A
    x = *(pw + i);                 // lê a célula ou porto de 16 bits no endereço 0x600A
    *(pb + i) = 3;                 // escreve a célula ou porto de 8 bits no endereço 0x6005
    y = *(pb + i);                // lê a célula ou porto de 8 bits no endereço 0x6005
}
```

**NOTA** – A aritmética de ponteiros tem em conta o tamanho do tipo apontado. Uma vez que **ints** ocupam 2 bytes, adicionar 5 a **pw** corresponde na realidade a adicionar 10. Adicionar 5 a **pb** adiciona apenas 5, uma vez que cada **char** ocupa apenas 1 byte.

### 4.3 Ponteiro para um array

Quando se pretender aceder a um intervalo de células ou portas periféricas (por exemplo, comandos para o módulo MediaCenter), uma alternativa à aritmética de ponteiros (o caso anterior) é considerar essas células ou portas como um array e depois definir um ponteiro para esse array (inicializado a partir de um **int** com conversão). Isto é ilustrado pelo exemplo seguinte.

```
const int base = 0x6000;           // a base do intervalo de endereços a aceder
int (* const pw)[] = (int (*)[]) base; // pw é um ponteiro para um array de ints, inicializado
                                     // através de um cast. pw tem o endereço da primeira célula
                                     // ou porto de 16 bits (palavra)
char (* const pb)[] = (char (*)[]) base; // pb é um ponteiro para um array de chars, inicializado
                                     // através de um cast. pb tem o endereço da primeira célula
                                     // ou porto de 8 bits (byte)
const int i = 5;                   // índice
int x;                             // variável de 16 bits
char y;                            // variável de 8 bits
void main(){
    (*pw)[i] = 3;                   // escreve a célula ou porto de 16 bits no endereço 0x600A
    x = (*pw)[i];                   // lê a célula ou porto de 16 bits no endereço 0x600A
    (*pb)[i] = 3;                   // escreve a célula ou porto de 8 bits no endereço 0x6005
    y = (*pb)[i];                   // lê a célula ou porto de 8 bits no endereço 0x6005
}
```

Este exemplo ilustra a equivalência entre a indexação de arrays e a aritmética de ponteiros. Mais uma vez, o número de bytes ocupados por cada elemento do array é tido em conta.

**NOTA** — Os comandos do MediaCenter são todos de 16 bits, pelo que requerem a versão **int**.

## 5 Exemplo simples de um programa em linguagem C no PEPE-16

A listagem seguinte é o programa contido no ficheiro **lab8-xadrez.c** e ilustra a programação em linguagem C no processador PEPE-16, em particular no que respeita ao acesso a endereços específicos, neste caso um periférico (módulo MediaCenter), usando a técnica de ponteiro para um array (secção 4.3).

Este programa simples assume que este módulo está disponível no endereço 0x6000 (para qualquer outro endereço, basta alterar a constante BASE no programa) e produz o padrão xadrez mostrado na figura após a listagem do programa.

Passe para “Simulation” e carregue no PEPE-16 o programa **lab8-xadrez.c**, carregando em **Load source** (📁) ou com *drag & drop*. O processador distingue a linguagem do programa pela extensão do ficheiro.

Abra a interface de simulação do MediaCenter, fazendo clique no módulo. Execute o programa, carregando no botão **Start** (▶) do PEPE-16, cujo resultado é ilustrado pela figura seguinte.

```

/*
Ficheiro: lab8-xadrez.c

Descrição: Este programa ilustra o acesso a endereços específicos, desenhando um
          padrão de xadrez usando comandos para um periférico MediaCenter.
*/

const int BASE = 0x6000;           // endereço base dos comandos do MediaCenter
int (* const comandos)[] = (int(*)[])BASE; // ponteiro para um array de comandos

const int DEFINE_LINHA    = 5;      // número do comando para definir a linha
const int DEFINE_COLUNA   = 6;      // número do comando para definir a coluna
const int DEFINE_COR_PIXEL = 9;     // número do comando para desenhar um pixel
const int APAGA_AVISO     = 0x20;   // número do comando para limpar o fundo
const int APAGA_ECRÃ     = 1;       // número do comando para limpar ecrã

const int N_LINHAS        = 32;     // número de linhas do ecrã (altura)
const int N_COLUNAS       = 64;     // número de colunas do ecrã (largura)

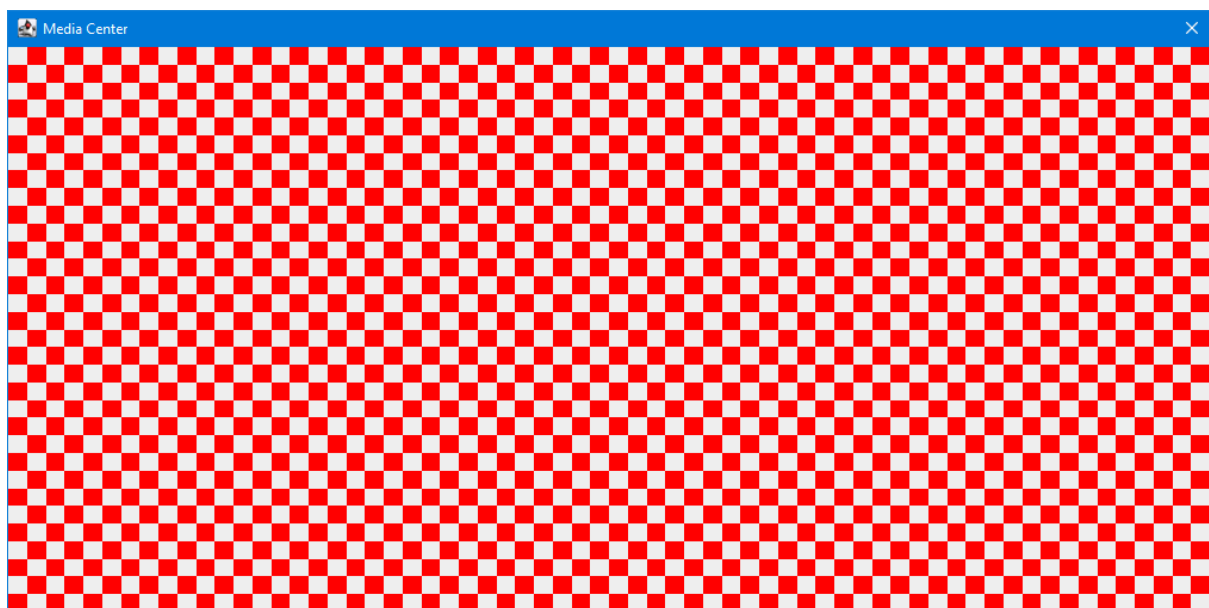
const int VERMELHO = 0xFF00;        // cor do pixel

int linha = 0;
int coluna = 0;
int cor = 0;

void main() {
    (*comandos)[APAGA_AVISO] = 0;    // limpa o aviso "no background image"
    (*comandos)[APAGA_ECRÃ] = 0;     // limpa todos os pixéis do ecrã

    for (linha = 0; linha < N_LINHAS; linha++) {
        for (coluna = 0; coluna < N_COLUNAS; coluna++) {
            (*comandos)[DEFINE_LINHA] = linha;    // define a linha
            (*comandos)[DEFINE_COLUNA] = coluna;   // define a coluna
            (*comandos)[DEFINE_COR_PIXEL] = cor;   // define a cor do pixel
            cor = cor == 0 ? VERMELHO: 0;         // alterna a cor em cada pixel
        }
        cor = cor == 0 ? VERMELHO: 0;             // alterna a cor do primeiro pixel
                                                    // em cada nova linha
    }
}

```



A funcionalidade deste programa é equivalente à do programa do ficheiro **lab5-comandos-xadrez.asm**, que está incluído no guião 5 e que também está disponível neste guião, para facilidade de referência. Compare os dois programas (em C e em assembly), para verificar as diferenças de programação em alto e baixo nível).

**NOTA** – A execução deste programa em C (**lab8-xadrez.c**) demora mais do dobro do tempo que a execução do programa em assembly (**lab5-comandos-xadrez.asm**), embora o resultado seja o mesmo. Este é o preço da programação em linguagem de alto nível. A programação assembly é de nível inferior e mais difícil, mas é muito mais eficiente.

## 6 Visualização do código assembly gerado pelo compilador de C

O compilador PEPE-16 C analisa o ficheiro fonte C, verifica a sua correção (sintaxe e semântica) e, se não forem detetados erros, gera instruções e diretivas em linguagem assembly PEPE-16, de forma a implementar a funcionalidade especificada pelas declarações fonte C e orientações. O compilador da linguagem assembly (assembler) será então invocado para gerar o código máquina, que o PEPE-16 executa de facto.

É útil poder inspecionar as instruções assembly geradas pelo compilador. Isto não só é elucidativo para se ter uma ideia de como funciona um compilador, como também permite verificar o que é realmente gerado e em que endereços, o que pode ser útil para detetar erros em tempo de execução.

Em particular, é possível fazer o debug de um programa de C não só ao nível da fonte, mas também ao nível da instrução assembly, incluindo a verificação dos valores dos registos (ver secção 10).

A figura seguinte mostra o programa **lab8-xadrez.c** na interface de simulação do PEPE-16, utilizando a vista “Source with code”, em que cada linha do programa em C é seguida pelo código assembly que gera. Por uma questão de brevidade, apenas são mostradas as declarações das variáveis e parte das instruções da função principal, em duas colunas (obtidas na interface do PEPE-16 através de *scroll* da janela do programa).

Note o código inicial inserido automaticamente pelo compilador, antes da primeira linha do programa fonte, em C. Inicializa o registo **SP** e chama a função principal (utilizando um registo no **CALL** para evitar as limitações de intervalo do *label* do **CALL**).

O programa principal termina com a leitura de uma variável **LOCK**, de modo a suspender a execução (colocando o PEPE-16 em estado de espera) em vez de utilizar a espera ativa por um ciclo infinito. Como esta variável não é escrita em lado nenhum, o processador fica em estado de espera (Waiting) caso a função **main** retorne.

Existe também a declaração da área do stack.

O resto mostra a declaração de variáveis e as instruções assembly geradas por cada instrução C, incluindo os endereços de cada variável e instrução.



Program					Program				
Line	Addr	Label	Mnem...	Operands	Line	Addr	Label	Mnem...	Operands
0000 H		\$_main:	MOV	SP, \$_SP_main	26	022A H	void main() {		
0004 H			MOV	R0, main	022C H		main:	PUSH	R11
0008 H			CALL	R0	022E H			MOV	R11, SP
000A H			MOV	R0, [_\$_main_lock]	022F H			SUB	SP, 4
000E H		\$_main_lock:	LOCK	0	0230 H			PUSH	R1
0010 H		\$_\$STK_main:	STACK	100H	0232 H			PUSH	R2
		\$_\$SP_main:			27		(*comandos)[APAGA_AVISO] = 0;		// limpa o aviso "n...
1		/*			0234 H			MOV	R1, 0
2		Ficheiro: lab8-xadrez.c			0236 H			MOV	[6040H], R1
3					28		(*comandos)[APAGA_ECRÃ] = 0;		// limpa todos os pi...
4		Descrição: Este programa ilustra o acesso a endereços ...			023A H			MOV	R1, 0
5		padrão de xadrez usando comandos para um perif...			023C H			MOV	[6002H], R1
6		*/			29				
7					30		for (linha = 0; linha < N_LINHAS; linha++) {		
8		const int BASE = 0x6000; // endereço base dos com...			0240 H			MOV	R1, 0
0210 H		BASE:	WORD	6000H	0242 H			MOV	[linha], R1
9		int (* const comandos)[] = (int(*)[])BASE; //ponteiro para u...			0246 H		cond_L5:	MOV	R1, [linha]
0212 H		comandos:	WORD	6000H	024A H			MOV	R2, 20H
10					024C H			CMP	R1, R2
11		const int DEFINE_LINHA = 5; // número do comand...			024E H			JGE	_L9
0214 H		DEFINE_LINHA:	WORD	5	0250 H			MOV	R1, 1
12		const int DEFINE_COLUNA = 6; // número do coma...			0252 H			JMP	_L8
0216 H		DEFINE_COLU...	WORD	6	0254 H		_L9:	MOV	R1, 0
13		const int DEFINE_COR_PIXEL = 9; // número do coma...			0256 H		_L8:	CMP	R1, 0
0218 H		DEFINE_COR...	WORD	9	0258 H			JZ	_L7
14		const int APAGA_AVISO = 0x20; // número do coman...			31		for (coluna = 0; coluna < N_COLUNAS; coluna++) {		
021A H		APAGA_AVISO:	WORD	20H	025A H			MOV	R1, 0
15		const int APAGA_ECRÃ = 1; // número do comando...			025C H			MOV	[coluna], R1
021C H		APAGA_ECR...	WORD	1	0260 H		cond_L10:	MOV	R1, [coluna]
16					0264 H			MOV	R2, 40H
17		const int N_LINHAS = 32; // número de linhas do e...			0266 H			CMP	R1, R2
021E H		N_LINHAS:	WORD	20H	0268 H			JGE	_L14
18		const int N_COLUNAS = 64; // número de colunas ...			026A H			MOV	R1, 1
0220 H		N_COLUNAS:	WORD	40H	026C H			JMP	_L13
19					026E H		_L14:	MOV	R1, 0
20		const int VERMELHO = 0xFF00; // cor do pixel			0270 H		_L13:	CMP	R1, 0
0222 H		VERMELHO:	WORD	0FF00H	0272 H			JZ	_L12
21					32		(*comandos)[DEFINE_LINHA] = linha; // define a l...		
22		int linha = 0;			0274 H			MOV	R1, [linha]
0224 H		linha:	WORD	32	0278 H			MOV	[600AH], R1
23		int coluna = 0;			33		(*comandos)[DEFINE_COLUNA] = coluna; // define ..		
0226 H		coluna:	WORD	64	027C H			MOV	R1, [coluna]
24		int cor = 0;			0280 H			MOV	[600CH], R1
0228 H		cor:	WORD	0	34		(*comandos)[DEFINE_COR_PIXEL] = cor; // define ..		
25					0284 H			MOV	R1, [cor]

## 7 Utilização da linguagem assembly no programa fonte C

O compilador de C do PEPE-16 oferece a possibilidade de incorporar instruções de assembly PEPE-16 no ficheiro fonte em linguagem C. Estas instruções serão inseridas no ponto em que aparecem e serão tratadas como se tivessem sido geradas pelo compilador de C.

O bloco de instruções assembly pode aparecer onde quer que uma instrução C seja permitida:

```
asm {
    instruções de assembly
}
```

As instruções dentro de um bloco **asm** seguem a sintaxe normal das instruções de assembly, incluindo comentários, mas têm algumas especificidades e restrições que devem ser tidas em conta:

- Os comentários consistem em “;” (e não “//”) até ao fim da linha;
- As constantes hexadecimais utilizam a notação assembly (por exemplo, 1000H, não 0x1000);
- **EQU** é a única diretiva permitida (todas as outras são proibidas dentro de um bloco **asm**);
- Não é permitida a utilização dos registos **R11**, **SP**, **RE**, **BTE** e **TEMP**;
- Saltar para um label fora do bloco **asm** gera um erro. Saltar através de um registo não é verificado;

- A chamada de uma função C é permitida (mesmo fora do bloco **asm**), mas os argumentos não são passados (o compilador passa os argumentos da função pela pilha), pelo que se recomenda invocar apenas funções sem argumentos;
- Alguns erros semânticos específicos, como uma constante fora do intervalo (por exemplo, `ADD R1, 25`), são detetados apenas após a geração de todas as instruções assembly do programa, o que significa que a linha do erro reportada não corresponderá ao seu número de linha no ficheiro com o programa em C.

É possível aceder às variáveis de C a partir de dentro do bloco **asm**, com as seguintes regras:

- Apenas os identificadores das variáveis podem ser utilizados (sem expressões);
- Um identificador precedido de “\$” acede ao seu conteúdo (leitura e escrita). Inclui todas as variáveis (globais, estáticas, locais e argumentos);
- Um identificador precedido de “&” acede ao seu endereço (apenas leitura). Inclui todas as variáveis (globais, estáticas, locais e argumentos) e funções;
- Os tipos das variáveis devem ser **int** ou ponteiro (os identificadores de array e de função são considerados ponteiros com o seu endereço inicial). Os tipos **char** e **short** não são permitidos;
- Apenas as instruções **MOV** (e não **MOVB**) podem ser utilizadas para aceder às variáveis desta forma;
- Os labels correspondentes a identificadores de variáveis ou funções globais ou estáticas podem ser utilizados normalmente pelas instruções assembly. Neste caso, o acesso ao conteúdo da variável (equivalente a “\$”) requer parênteses retos.

O exemplo seguinte ilustra algumas destas características:

```
int x = 5;
int * p =(int *)0x2000;
int w[5];

void f(){
}

void main(){
    int y = 3;                                ; variável local, alocada na pilha

    asm{
        endereço    EQU 1000H
        MOV    R1, endereço
        MOV    R3, $p                        ; equivalente a MOV R3, [p]
        MOV    $p, R1                        ; equivalente a MOV [p], R1
        CALL   f                             ; também se poderia usar CALL &f
        MOV    R2, $x                        ; equivalente a MOV R2, [x]
        MOV    R2, &x                        ; equivalente a MOV R2, x
        MOV    R2, x                         ; igual ao anterior
        MOV    R2, &w                        ; equivalente a MOV R2, w
        MOV    R2, &y                        ; não equivalente a MOV R2, y
                                                ; y é local e reside na pilha
                                                ; gera código para obter o seu endereço
    }
}
```

A utilização de instruções assembly dentro de uma função C deve ser reservada para casos especiais de nível muito baixo, e não como solução geral. Na maioria dos casos, usar a linguagem C apenas é suficiente para construir programas completos.

## 8 Interrupções

### 8.1 Suporte para interrupções

Para permitir uma determinada interrupção no PEPE-16 (no intervalo 0 .. 3), deve ser declarada uma função como sendo o gestor dessa interrupção, ou seja, a função a invocar quando essa interrupção ocorre. Isto é feito em C com uma diretiva `#pragma INTERRUPT`, precedendo a declaração da função e especificando o número da interrupção (0 .. 3) a que está associada, como ilustrado no exemplo seguinte. O nome das funções pode ser um qualquer, desde que seja único no programa.

```
#pragma INTERRUPT 0
void int_0(){
    // trata da interrupção 0
}
... // declaração de funções para as interrupções 1 e 2

#pragma INTERRUPT 3
void int_3(){
    // trata da interrupção 3
}
```

Podem ser associadas a interrupções até 4 funções, utilizando esta diretiva. Apenas aquelas que sejam usadas precisam de ser especificadas e podem aparecer em qualquer ordem. O compilador gera automaticamente as instruções para permitir as respetivas interrupções, a tabela de exceções e a inicialização do BTE.

O nome de cada uma destas funções pode ser qualquer identificador válido, desde que seja único no programa. Os anteriores são apenas um exemplo.

Se não for especificada nenhuma função de interrupção, nenhuma interrupção será permitida.


### 8.2 Exemplo de uso de interrupções em linguagem C

O programa contido no ficheiro **lab8-cooperativas-teclado-OK.c** ilustra o funcionamento das interrupções em linguagem C no PEPE-16.


Este programa é equivalente ao contido no ficheiro **lab7-cooperativas-teclado-OK.asm**, mas agora em linguagem C. Este programa em assembly faz parte do guião 7, mas está também aqui incluído para facilidade de referência.

Estes programas ilustram a técnica de implementar várias atividades de tempo real concorrentes, usando funções cooperativas (em que nenhuma das funções é bloqueante). Pode editar os dois ficheiros e verificar as diferenças de programação em assembly e em linguagem C.

O comportamento dos dois programas é igual (4 barras a descer, displays a contar, com o sentido de contagem normalmente ascendente, mas descendente caso haja uma tecla premida na 4ª linha do teclado). Veja no guião 7 a descrição detalhada deste exemplo, em linguagem assembly.

Para verificar o funcionamento, carregue o programa **lab8-cooperativas-teclado-OK.c**, fazendo clique no botão **Load source**  ou por *drag & drop*.

Abra a interface de simulação do ecrã, dos displays, do teclado e de todos os relógios.

Execute o programa, carregando no botão **Start**  do PEPE-16.

Verifique que as barras descem, cada uma ao ritmo da interrupção gerada pelo relógio respetivo, e que os displays aumentam ou diminuem o seu valor, consoante não haja ou haja, respetivamente, uma tecla da última linha carregada.

Termine o programa, carregando no botão **Stop**  do PEPE-16 e depois no botão **Reset** , o que faz o PEPE-16 voltar ao estado inicial.

Selecione a vista “Source with code” para visualizar o código gerado pelo compilador de C, tal como ilustrado pela figura seguinte:

- As primeiras instruções (a partir do endereço 0000H) geradas pelo compilador de C. Note a inicialização do registo **BTE**, as instruções de permissão das interrupções e a tabela de rotinas de interrupção (**\$\_BTE\_TABLE**), com os endereços das rotinas de interrupção usadas;
- As instruções assembly geradas pelas funções das interrupções 0 e 1. A diretiva **#pragma INTERRUPT** permite não só saber que a rotina que se segue deverá estar na tabela de rotinas de interrupção (e em que posição), mas também que esta rotina deve terminar com **RFE** (Return From Exception) e não **RET**.

Program

Line	Addr	Label	Mnem...	Operands
0000 H		<b>\$_main:</b>	MOV	SP, \$_SP_main
0004 H			MOV	BTE, \$_BTE_TABLE
0006 H			EI0	
0008 H			EI1	
000A H			EI2	
000C H			EI3	
000E H			EI	
0010 H			MOV	R0, main
0014 H			CALL	R0
0016 H			MOV	R0, [\$_main_lock]
001A H		<b>\$_main_lock:</b>	LOCK	0
001C H		<b>\$_BTE_TABLE:</b>	WORD	rot_int_0
001E H			WORD	rot_int_1
0020 H			WORD	rot_int_2
0022 H			WORD	rot_int_3
0024 H		<b>\$_STK_main:</b>	STACK	100H
		<b>\$_SP_main:</b>		
1			/*	
2			Ficheiro: lab8-cooperativas-teclado-OK.c	
3				
4			Descrição: Este programa ilustra o funcionamento das in...	
5			de várias rotinas cooperativas, em que nenhuma d...	
6			Equivalente ao lab7-cooperativas-teclado-OK.asm, ...	
7			*/	
8				
9			const int BASE = 0x6000; // endereço base dos com...	
0224 H		<b>BASE:</b>	WORD	6000H
10			int (* const comandos)[] = (int(*)[])BASE; // ponteiro para u...	
0226 H		<b>comandos:</b>	WORD	6000H
11				
12			const int DEFINE_LINHA = 5; // número do comand...	
0228 H		<b>DEFINE_LINHA:</b>	WORD	5
13			const int DEFINE_COLUNA = 6; // número do coma...	
022A H		<b>DEFINE_COLU...</b>	WORD	6
14			const int ESCRIVE_8_PIXELS = 0xE; // número do com...	
022C H		<b>ESCRIVE_8_PIXELS:</b>	WORD	0EH

Program

108			/*	
109			ROT_INT_0 - Rotina de atendimento da interrupção 0	
110			Assinala o evento na componente 0 da variável eve...	
111			*/	
112			#pragma INTERRUPT 0	
113			void rot_int_0(){	
0440 H		<b>rot_int_0:</b>	PUSH	R11
0442 H			MOV	R11, SP
0444 H			PUSH	R1
114			evento_int[0] = 1; // assinala que houve uma interrupç...	
0446 H			MOV	R1, 1
0448 H			MOV	[(evento_int+0)], R1
115			}	
044C H		<b>rot_int_0\$ret:</b>	POP	R1
044E H			MOV	SP, R11
0450 H			POP	R11
0452 H			RFE	
116			/*	
117			ROT_INT_1 - Rotina de atendimento da interrupção 1	
118			Assinala o evento na componente 1 da variável eve...	
119			*/	
120				
121			#pragma INTERRUPT 1	
122			void rot_int_1(){	
0454 H		<b>rot_int_1:</b>	PUSH	R11
0456 H			MOV	R11, SP
0458 H			PUSH	R1
123			evento_int[1] = 1; // assinala que houve uma interrupç...	
045A H			MOV	R1, 1
045C H			MOV	[(evento_int+2)], R1
124			}	
0460 H		<b>rot_int_1\$ret:</b>	POP	R1
0462 H			MOV	SP, R11
0464 H			POP	R11
0466 H			RFE	

## 9 Processos

### 9.1 Suporte para processos

O suporte no compilador de C para os processos PEPE-16 é conseguido com diretivas `#pragma` adicionais. Estas diretivas correspondem a diretivas em linguagem assembly. O guião 7 fornece mais detalhes.

#### 9.1.1 `#pragma PROCESS`


Para declarar um processo, deve ser declarada uma função imediatamente precedida por uma diretiva `#pragma PROCESS`, utilizando a seguinte sintaxe:

```
#pragma PROCESS número_de_instâncias @ tamanho_da_pilha
void nome_função (argumentos){
    while (1){
        // instruções do processo
    }
}
```

As seguintes observações devem ser tidas em conta:

- As funções marcadas como processo (precedidas pela diretiva `#pragma PROCESS`) devem obrigatoriamente retornar **void**;
- Invocar o nome da função com os seus argumentos, se existirem, cria um processo que se torna executável, em vez de executar o seu código imediatamente, como seria o caso de uma invocação de uma função normal (não marcada como processo);
- A mesma função de processo pode ser invocada várias vezes, criando de cada vez uma nova instância de processo, com o código da função, mas com uma área de pilha separada;
- *número\_de\_instâncias* deve ser substituído por uma expressão constante que especifique o número máximo de instâncias de processo permitidas, ou seja, o número de vezes que a função pode ser invocada. Isto é necessário para reservar espaço na pilha para todas as instâncias do processo. Se *número\_de\_instâncias* for omitido, o valor 1 será assumido por omissão. Se especificado e for maior que 1, então é obrigatório que a função tenha pelo menos um argumento e o primeiro argumento deve especificar o número da instância ( $0..número\_de\_instâncias - 1$ ) quando a função é invocada. O programa comportar-se-á mal se o valor do primeiro argumento estiver fora deste intervalo quando a função for invocada;
- *tamanho\_da\_pilha* deve ser substituído por uma expressão constante que especifique o número de palavras de 16 bits a reservar para cada instância de processo. Será multiplicado pelo número de instâncias permitidas para reservar espaço suficiente para todas as instâncias deste processo. Se o tamanho da pilha (e o “@”) for omitido, será assumido o valor predefinido do tamanho da pilha. Este tamanho padrão pode ser alterado com a diretiva `STACK_SIZE` (secção **Error! Reference source not found.**);
- O corpo da função do processo inclui tipicamente um ciclo infinito, uma vez que um processo implementa normalmente alguma atividade de longa duração. Se a função do

processo chegar ao fim ou executar uma instrução de retorno, a instância do processo será terminada.

- NOTA** – Embora a função “main” não possa ser precedida pela diretiva `#pragma PROCESS`, é na realidade o corpo do processo principal, que existe mesmo que não seja criado qualquer outro processo. Isto pode ser observado na janela de simulação do processador PEPE-16 (separador Processos do painel adicional, que pode ser aberto carregando no botão );
- Este processo principal tem apenas uma instância e o seu tamanho de pilha é o tamanho padrão da pilha do processo (secção **Error! Reference source not found.**);
  - Além disso, as diretivas `YIELD` e `WAIT` podem ser utilizadas dentro da função “main” e podem utilizar variáveis `LOCK` como qualquer outro processo;
  - Se a função “main” regressar, o processo principal bloqueia-se a si próprio lendo um `LOCK` que nunca é escrito por nenhum processo. Os processos que criou continuarão até terminarem ou serem bloqueados num *deadlock* (aguardando indefinidamente numa variável `LOCK` na qual nenhum processo escreve).

### 9.1.2 `#pragma YIELD` e `#pragma WAIT`

Estas diretivas especificam pontos do código do processo em que o processador pode mudar para outro processo e indicam ao compilador de C que deve gerar as diretivas `YIELD` e `WAIT`, respetivamente, em linguagem assembly. Isto permite que os processos deem a outros a oportunidade de executar quando chegar a sua vez.

Só podem ser especificadas dentro de uma função de processo, precedendo a instrução na qual a troca de processo pode ser feita. A sua sintaxe é muito simples, como no exemplo seguinte:

```
#pragma PROCESS
void nome_função (argumentos){
    while (1){
        #pragma YIELD          // ou #pragma WAIT (pode mudar para outra instância aqui)
        // instruções do processo
    }
}
```

### 9.1.3 `#pragma LOCK`

A diretiva `#pragma LOCK` permite especificar variáveis que podem ser utilizadas para a comunicação e sincronização entre instâncias de processos. Uma instância de processo que lê uma variável qualificada com `LOCK` é automaticamente bloqueada até que alguma outra instância faça uma atribuição a essa variável com algum valor. Quando isto ocorre, todas as instâncias de processo bloqueadas nessa variável são desbloqueadas (tornando-se executáveis) e leem o valor escrito na variável.

Esta diretiva em linguagem C gera uma declaração `LOCK` em linguagem assembly (em vez de `WORD`).

A diretiva `LOCK` deve preceder imediatamente a declaração de uma variável. Por exemplo:

```
#pragma LOCK int x;
```

A diretiva LOCK só pode ser aplicada a variáveis do tipo **int**, ponteiro ou array de **ints** ou de ponteiros.

As variáveis LOCK devem ser globais (declaradas fora de qualquer função).

**IMPORTANTE** – Cuidado com teste múltiplos ao valor lido de uma variável LOCK! Isto é típico de **ifs** encadeados. Cada teste é uma leitura e o processo bloqueia em cada leitura, o que naturalmente não é o pretendido. Neste caso, o que se deve fazer é ler a variável LOCK apenas uma vez, colocando o valor lido numa variável auxiliar, e depois fazer os testes múltiplos com essa variável auxiliar. Os programas que ilustram o suporte para processos (secção 9.2) exemplificam esta situação, na função **main**.

## 9.2 Exemplos de uso de processos em linguagem C


### 9.2.1 Exemplo com quatro bonecos

O programa contido no ficheiro **lab8-processos-quatro-bonecos-displays-teclado.c** ilustra o suporte para processos em linguagem C no PEPE-16.


Este programa é equivalente ao contido no ficheiro **lab7-processos-quatro-bonecos-displays-teclado.asm**, mas agora em linguagem C. Este programa em assembly faz parte do guião 7, mas está também aqui incluído para facilidade de referência.

Estes programas ilustram a técnica de implementar várias atividades de tempo real concorrentes, usando o suporte para processos dado pelo PEPE-16, em que as funções marcadas como processo podem ser bloqueantes (ter ciclos potencialmente infinitos), desde que tenham pontos em que o processador pode mudar para outro processo (diretivas YIELD ou WAIT, ou leituras de variáveis LOCK). Pode editar os dois ficheiros e verificar as diferenças de programação em assembly e em linguagem C.

O comportamento dos dois programas é igual (4 bonecos em movimento alternado, displays a contar, com o sentido de contagem, ascendente ou descendente, controlado pelas teclas da 4ª linha do teclado). Veja no guião 7 a descrição detalhada deste exemplo, em linguagem assembly.

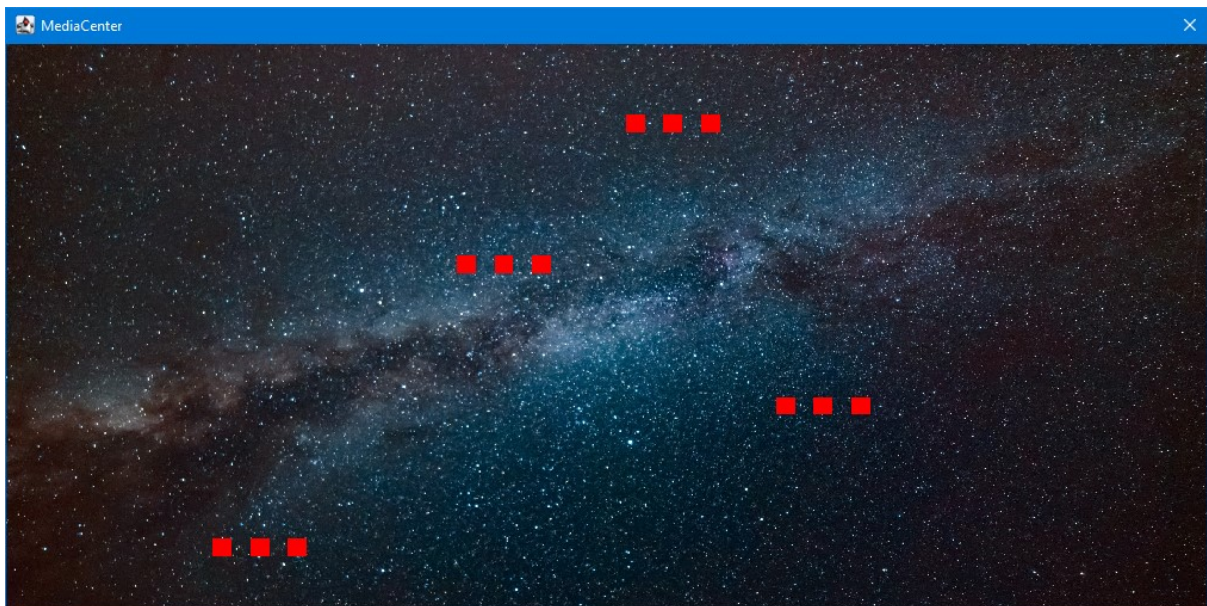
Para verificar o funcionamento, carregue o programa **lab8-processos-quatro-bonecos-displays-teclado.c**, fazendo clique no botão **Load source**  ou por *drag & drop*.

Abra a interface de simulação do ecrã, dos displays e do teclado.

Execute o programa, carregando no botão **Start**  do PEPE-16.

Verifique que os 4 bonecos se movem de um lado para o outro, cada uma ao ritmo da interrupção gerada pelo relógio respetivo, tal como ilustrado na figura seguinte.





Verifique também que os displays aumentam ou diminuem o seu valor, consoante não haja ou haja, respetivamente, uma tecla da última linha carregada.

Termine o programa, carregando no botão **Stop** (■) do PEPE-16.

Aspetos mais relevantes a notar neste programa:

- Diretivas `STACK_INIT` e `STACK_SIZE` (linhas 18 e 19), indicando onde deve começar a zona do stack e qual o tamanho em palavras do stack de cada processo por omissão, caso não seja indicado outro valor. Estas declarações não são obrigatórias, e estão aqui apenas por exemplo. Se `STACK_INIT` não for usado, a zona do stack é declarada após tudo o resto. Se `STACK_SIZE` não for usada, o tamanho por omissão é 0x100 palavras;
- Inicialização das variáveis que mantêm o estado dos bonecos (linha, coluna e sentido de movimento), nas linhas 62 a 66. Poder-se-iam inicializar diretamente na declaração, mas tal traduz-se em assembly em variáveis que só são inicializadas quando o programa é carregado, e não quando se recomeça o programa após o terminar, sem recarregar o programa do ficheiro. Isto faz com que nesse caso o programa recomece com os valores com que as variáveis ficaram quando se terminou o programa. Para evitar isso, a inicialização destas variáveis é feita no início da execução de cada instância do processo boneco, o que acontece sempre que o programa é (re)executado;
- As linhas 69 a 72 ilustram a declaração das variáveis `LOCK`;
- As linhas 152 e 161 ilustram a utilização da diretiva `YIELD`, que permite os ciclos bloqueantes ao providenciarem um ponto de comutação para um outro processo, caso haja algum que possa executar. Note o ciclo `while (1)`, que nunca termina;
- A linha 250 ilustra o cuidado a ter com variáveis `LOCK`, tal como referido na secção 9.1.3. Se se quiser fazer testes múltiplos ao valor lido, deve ler-se para uma variável auxiliar e depois fazer os testes com essa variável auxiliar. Se se testar a variável `LOCK` diretamente, em cada teste o processo bloqueia-se até à próxima escrita na variável `LOCK`;
- Note ainda que a linha 250 atua também como uma diretiva `YIELD`, permitindo o ciclo infinito (`while (1)`) do processo. A leitura da variável `LOCK` bloqueia o processo e passa ao processo executável seguinte, pelo que o ciclo infinito não bloqueia o programa.



### 9.2.2 Exemplo com N bonecos

O exemplo anterior suporta até 4 bonecos, ligando cada boneco a uma dada interrupção, de forma rígida. Para permitir mais bonecos em movimento, tem de se fazer algumas alterações, nomeadamente distribuir os bonecos pelas 4 interrupções disponíveis. Por exemplo, isto significa que, com 8 bonecos, 2 bonecos devem avançar quando uma dada interrupção ocorre.

O programa contido no ficheiro **lab8-processos-N-bonecos-displays-teclado.c** estende o exemplo anterior para permitir mais bonecos.

Os processos são os mesmos, mas cada instância do processo boneco tem de inicializar a linha do seu boneco, bem como decidir qual a interrupção a que dá atenção, por meio de fórmulas, com base no número da instância. Pode verificar estas diferenças nas linhas 180 e 186, respetivamente, do ficheiro **lab8-processos-N-bonecos-displays-teclado.c**.

Outro aspeto relevante é a declaração do processo boneco, na linha 174:

```
#pragma PROCESS N_BONECOS @ 0x600 / N_BONECOS
```

Esta declaração indica dois valores, separados por “@”:

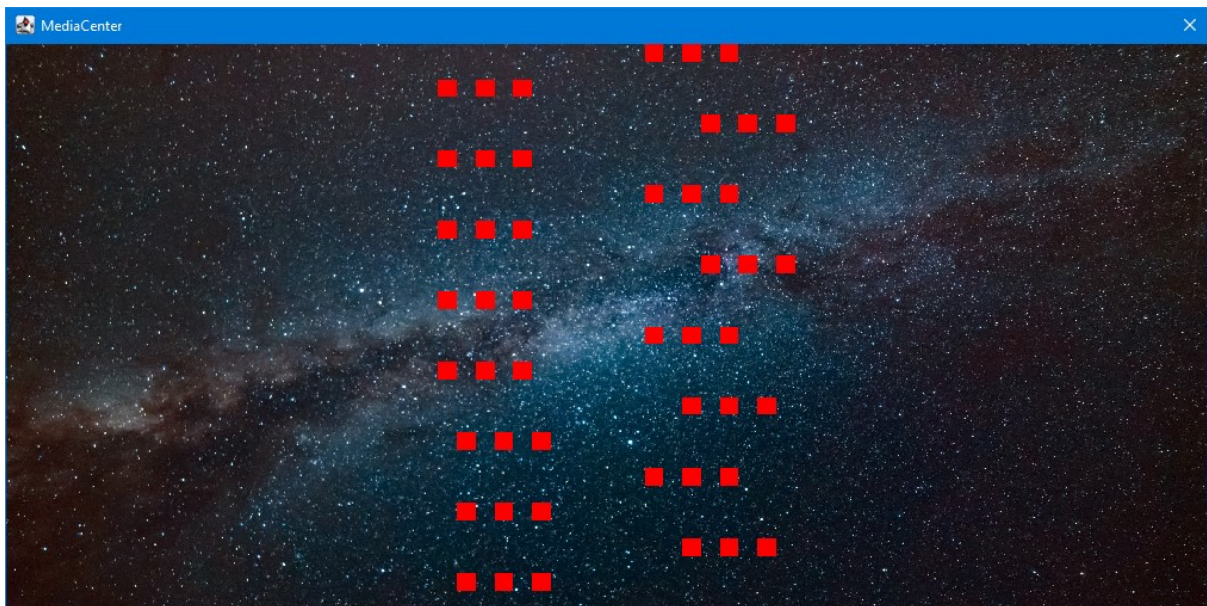
- o número de instâncias que podem ser criadas, que pode ir até N\_BONECOS (constante definida no programa);
- o tamanho do stack por cada instância, que neste caso é 0x600 a dividir pelo número máximo de instâncias. Isto quer dizer que, mesmo que se varie o valor a constante N\_BONECOS, o valor total do stack reservado para todas as instâncias do processo boneco é sempre 0x600. Não tinha de ser assim, podendo ser um valor fixo por instância, mas nesse caso mais instâncias implica mais área de stack. Em qualquer dos casos, convém verificar que nem a área total do stack ultrapassa a memória disponível nem a área de stack reservada para cada instância é demasiado pequena.

Para verificar o funcionamento, carregue o programa **lab8-processos-N-bonecos-displays-teclado.c**, fazendo clique no botão **Load source** (📁) ou por *drag & drop*.

Abra a interface de simulação do ecrã, dos displays e do teclado.


Execute o programa, carregando no botão **Start** (▶) do PEPE-16.

A figura seguinte ilustra o caso de 16 bonecos, mas convém ter em conta que o programa em linguagem C é significativamente mais lento que o correspondente em linguagem assembly, pelo que dar a volta a todos os processos pode demorar mais tempo do que o período das interrupções, e algumas perdem-se (não são atendidas a tempo), pelo que os bonecos não se distanciam tanto como os períodos das respetivas interrupções permitiria. Quantos mais bonecos, maior o problema.



## 10 Depuração (debugging) de programas em linguagem C

A depuração (debugging) de programas em linguagem C é quase idêntica à depuração de programas em linguagem assembly, como descrito nos guiões 2 e 5. Isto envolve três mecanismos:

- **Refrescamento** periódico da interface do PEPE-16, carregando no botão  da barra de ferramentas (toolbar) da interface do PEPE-16. Embora periódico e não contínuo (seria demasiado rápido), dá para ter uma ideia das instruções pelas quais o PEPE-16 vai passando (a barra azul indica qual a instrução que será executada a seguir). Isto é particularmente útil quando o programa fica bloqueado algures;
- **Execução passo a passo** (step), seja manual ou de avanço automático (autostep), seja de uma instrução só (single step) ou de uma invocação completa de uma função (step over). Permite avançar a execução do programa de uma forma muito controlada (embora drasticamente mais lenta do que em execução normal) e verificar o conteúdo dos registos e das variáveis globais após cada passo de execução;
- **Pontos de paragem** (breakpoints), em que o programa procede em execução normal até chegar a um ponto em que a paragem foi definida, e o programa fica em pausa. É então possível ver o conteúdo dos registos ou variáveis globais, executar passo a passo ou recomeçar a execução (resume).

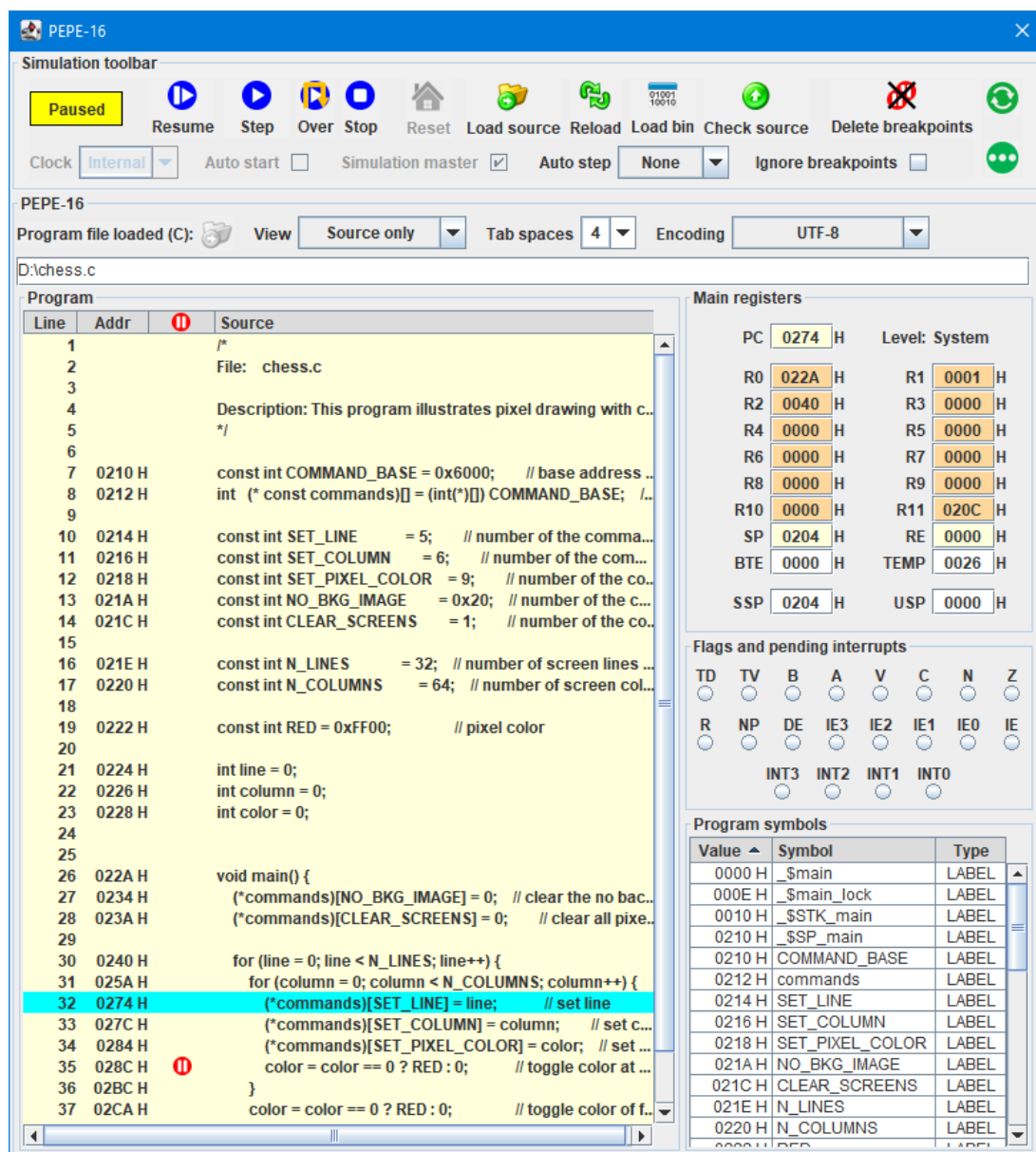
No entanto, cada instrução em linguagem C corresponde normalmente a várias instruções em linguagem assembly. Isto significa que é possível depurar um programa diretamente em C (vista “Source only”) ou ao nível da linguagem assembly (todas as outras vistas). A secção 6 ilustra a vista “Source with code”.

Na vista “Source only”:

- Apenas as instruções do programa fonte em C são visíveis;
- Quando o programa é carregado mas não executado (ficando pronto para iniciar), a barra azul aparece no cabeçalho da função “main”, pois a execução começa aí;

- Cada passo de execução (single step) executa uma instrução de C, incluindo todas as instruções assembly que esta gera;
- Uma instrução **asm** conta como uma única instrução em C, mesmo que muitas instruções em linguagem assembly estejam visíveis no seu interior;
- Um step over executa uma invocação completa de uma função, incluindo a execução do seu corpo completo, mesmo que invoque outras funções;
- Os pontos de paragem (breakpoints) podem ser colocados em qualquer instrução em C em que a execução possa ser colocada em pausa, incluindo os cabeçalhos das funções (a execução pausa após a invocação, mas antes de executar o corpo da função);
- Não é suportado inspecionar o valor das variáveis nesta vista, mas os pontos de paragem de dados podem ser definidos nas variáveis.

A figura seguinte ilustra a depuração do programa de xadrez da secção 5 na vista “Source only”, com a barra azul a indicar a instrução fonte C a executar a seguir, na linha fonte 32. Pode também ser visto um ponto de paragem, definido na linha fonte 35.



Em todas as outras vistas serão mostradas as instruções de assembly geradas por cada instrução em C. A vista mais utilizada é a fonte com visualização de código (“Source with code”), em que as instruções de assembly aparecem após a instrução de C que as gerou, segundo a ordem das linhas no programa fonte em C.

Nas vistas que expõem instruções ao nível de assembly:

- Tanto as instruções de C como as instruções assembly são visíveis (exceto na vista apenas de código – “Code only”);
- Quando o programa é carregado mas não executado (está pronto para iniciar), a barra azul aparece no endereço 0000H, onde o processador inicia a execução. A função “main” é invocada após algumas inicializações, que podem ser executadas passo a passo;
- Um ponto de paragem já não pode ser definido numa instrução de C, mas sim na primeira instrução de nível assembly que esta gera. Isto é apenas mais detalhe do que um ponto de paragem na vista “Source only”;
- No entanto, é agora possível definir pontos de paragem em instruções assembly subsequentes geradas pela mesma instrução de C, proporcionando mais controlo de depuração de execução;
- Estes pontos de paragem intermédios aparecem a azul (II) em vez de vermelho (II), precisamente para explicitar que estes não correspondem ao início de uma instrução em C.

A figura seguinte ilustra a depuração do mesmo programa de xadrez (secção 5), agora em código-fonte com visualização de código (vista “Source with code”), com instruções de C e as instruções de assembly que cada uma destas gera.

A barra azul indica ainda a instrução de C a executar a seguir, na linha de origem 32. O ponto de paragem da linha 35 do programa fonte ainda pode ser visto (na primeira instrução assembly gerada pela instrução de C na linha 35), mas dois outros pontos de paragem foram definidos (representados a azul) dentro do conjunto de instruções de assembly gerado pela instrução de C na linha 35.

Note a quantidade de instruções assembly geradas por uma só instrução em C, pelo que definir pontos de paragem ao nível do assembly permite a depuração com mais detalhe (se necessário).

PEPE-16

Simulation toolbar

Paused Resume Step Over Stop Reset Load source Reload Load bin Check source Delete breakpoints

Clock Internal Auto start Simulation master Auto step None Ignore breakpoints

PEPE-16

Program file loaded (C): View Source with code Tab spaces 4 Encoding UTF-8

D:\chess.c

Program

Line	Addr	Label	Mnem...	Operands
31			for (column = 0; column < N_COLUMNS; column++) {	
	025A H		MOV	R1, 0
	025C H		MOV	[column], R1
	0260 H	cond_L10:	MOV	R1, [column]
	0264 H		MOV	R2, 40H
	0266 H		CMP	R1, R2
	0268 H		JGE	_L14
	026A H		MOV	R1, 1
	026C H		JMP	_L13
	026E H	_L14:	MOV	R1, 0
	0270 H	_L13:	CMP	R1, 0
	0272 H		JZ	_L12
32			(*commands)[SET_LINE] = line; // set line	
	0274 H		MOV	R1, [line]
	0278 H		MOV	[600AH], R1
33			(*commands)[SET_COLUMN] = column; // set c...	
	027C H		MOV	R1, [column]
	0280 H		MOV	[600CH], R1
34			(*commands)[SET_PIXEL_COLOR] = color; // set ...	
	0284 H		MOV	R1, [color]
	0288 H		MOV	[6012H], R1
35			color = color == 0 ? RED : 0; // toggle color at ...	
	028C H		MOV	R1, [color]
	0290 H		CMP	R1, 0
	0292 H		JNZ	_L22
	0294 H		MOV	R1, 1
	0296 H		JMP	_L21
	0298 H	_L22:	MOV	R1, 0
	029A H	_L21:	CMP	R1, 0
	029C H		JZ	_L23
	029E H		MOV	R1, R11
	02A0 H		ADD	R1, 0FFFEH
	02A2 H		MOV	R2, [RED]
	02A6 H		MOV	[R1], R2
	02A8 H		JMP	_L24
	02AA H	_L23:	MOV	R1, R11
	02AC H		ADD	R1, 0FFFEH

Main registers

PC 0274 H Level: System

R0	022A H	R1	0001 H
R2	0040 H	R3	0000 H
R4	0000 H	R5	0000 H
R6	0000 H	R7	0000 H
R8	0000 H	R9	0000 H
R10	0000 H	R11	020C H
SP	0204 H	RE	0000 H
BTE	0000 H	TEMP	0026 H
SSP	0204 H	USP	0000 H

Flags and pending interrupts

TD	TV	B	A	V	C	N	Z
R	NP	DE	IE3	IE2	IE1	IE0	IE
INT3	INT2	INT1	INT0				

Program symbols

Value	Symbol	Type
0000 H	\$_main	LABEL
000E H	\$_main_lock	LABEL
0010 H	\$_STK_main	LABEL
0210 H	\$_SP_main	LABEL
0210 H	COMMAND_BASE	LABEL
0212 H	commands	LABEL
0214 H	SET_LINE	LABEL
0216 H	SET_COLUMN	LABEL
0218 H	SET_PIXEL_COLOR	LABEL
021A H	NO_BKG_IMAGE	LABEL
021C H	CLEAR_SCREEN	LABEL
021E H	N_LINES	LABEL
0220 H	N_COLUMNS	LABEL