

Chapter 5

Subprograms and I/O Functions

5.1 Introduction

Main programs and all the subprograms it calls must agree on how data will be passed between them. Parameters can be passed using registers, shared memory, or the stack. A large part of this chapter will deal with the standard C calling conventions that can be used to interface assembly subprograms with C programs. This (and other conventions) often pass the addresses of data (i.e. pointers) to allow the subprogram to access the data in memory. I/O functions are also addressed in this chapter. These include C-functions under *djgpp* and *linux*. DOS and BIOS I/O functions are also discussed.

5.2 Procedure calls and returns

The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy. The *call* instruction *pushes* the address of the next instruction onto the stack and makes an unconditional jump to a sub-procedure address. The *ret* instruction *pops off* an address into the *eip* register and the *cpu* continues the execution of the calling program. When using these instructions, it is very important to manage the stack correctly so that the right address is popped off by the *ret* instruction.

Prior to branching to the first instruction of the called procedure, the *call* instruction pushes the address in the *eip* register onto the stack. The address just

pushed is the return instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the *ret* instruction pops the return instruction pointer from the stack back into the *eip* register. Execution of the calling procedure then resumes. The processor does not keep track of the location of the return instruction pointer. It is up to the programmer to insure that the stack pointer is pointing to the return address on the stack, prior to issuing a *ret* instruction.

When the *call* instruction transfers control to procedures within the current code segment, it is referred to as a *near* call. Near calls provide access to local sub-procedures within the currently running program. The *call* instruction can also transfer control to procedures in a different code segment; these calls are referred to as *far* calls. The following formats are supported by *nasm*:

```
call reg/mem  ;[ss : (e)sp] ← (e)ip;
               ;(e)ip ← reg/mem16,32
call far mem   ;[ss : (e)sp] ← cs : (e)ip;
               ;cs : (e)ip ← seg : offset16,32
```

Near calls: The first form *call reg/mem* is a *near* call instruction. These calls are three bytes long; one byte contains the *opcode* and a 16-bit *displacement* occupies the other two bytes. Recall that this is the format for near unconditional jumps. The displacement is a relative offset specified in the fetched instruction such that: $targetaddress = (e)ip + displacement$. A 16-bit displacement allows a jump in the range of ± 32 Kbytes. Displacements with 32 bits have a jump range of ± 2 Gbytes; in this case after saving the return address in *eip*, control to the procedure is transferred by adding to *eip* the 32-bit displacement specified in the instruction. From the programmer perspective, the processor does the following: (see Fig. 5.1):

1. Pushes the current value of the *eip* register on the stack.
2. Loads the offset of the called procedure in the *eip* register.
3. Begins execution of the called procedure.

Far calls: The *call far mem* form executes a far call by loading the destination address out of memory. This is a 5-byte instruction with the first byte containing the opcode and the remaining four bytes containing a far pointer to the procedure. For 16-bit applications bytes 2 and 3 are loaded into *ip* and bytes 4 and 5 are loaded into *cs*. For 32-bit applications an offset with 32 bits is loaded into *eip*. As shown in Fig. 5.1, from the programmer perspective, the processor performs the following actions:

1. Pushes current value of the *cs* register on the stack.
2. Pushes the current value of the *eip* register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the *cs* register.
4. Loads the offset of the called procedure in the *eip* register.
5. Begins execution of the called procedure.

The *ret* instruction also allows near and far returns to match the near and far *call* instructions. In addition, the *ret* instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument *n* to the *ret* instruction. The following formats are supported by *nasm*:

```
ret      ;(e)ip ← [ss : (e)sp]16,32
ret n    ;(e)ip ← [ss : (e)sp]16,32; (e)sp + n
retf     ;cs : (e)ip ← [ss : (e)sp]16,32
retf n   ;cs : (e)ip ← [ss : (e)sp]16,32; (e)sp + n
```

near returns: *ret* executes a *near* return; it pops only *ip* or *eip* from the stack and transfers control to the new address. The form *ret n* increments the stack pointer by an *n* number of bytes after popping the return address. The *cpu* actions can be summarized as follows:

1. Pops the top-of-stack value (the return instruction pointer) into the *(e)ip* register.
2. If the *ret* instruction has an optional *n* operand then increments the stack pointer by the number of bytes specified by *n* to release parameters from the stack.
3. Resumes execution of the calling procedure.

far returns: *retf* executes a far return: it pops *ip/eip* followed by *cs*. The second form *retf n* will increment the stack pointer a number of bytes given by the argument *n*. When executing a far return, the processor does the following:

1. It pops the top-of-stack value (the return instruction pointer) into the *(e)ip* register.

2. It pops the top-of-stack value (the segment selector for the code segment being returned to) into the *cs* register.
3. If the *ret* instruction has an optional *n* argument then increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. The *cpu* resumes execution of the calling procedure.

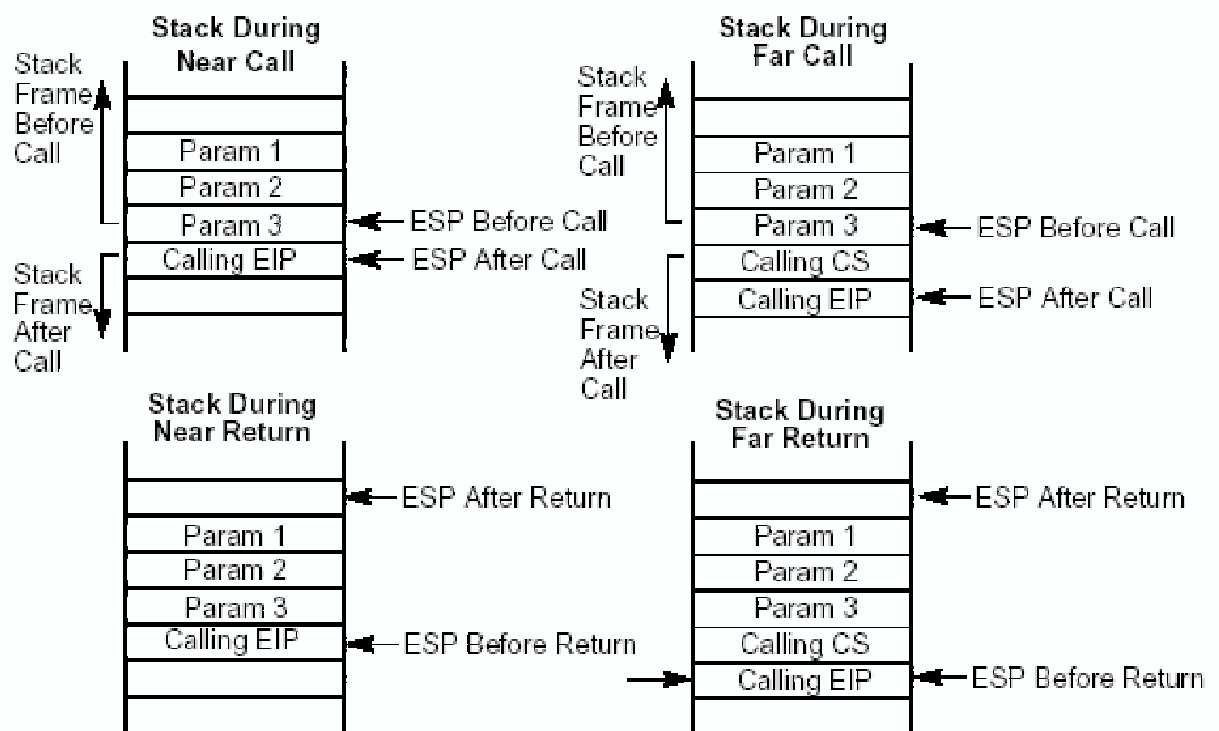


Figure 5.1: Stack operations on Near and Far Calls.

5.3 Passing Parameters

Unlike HLL call statements, call instructions in assembly do not provide a list of arguments in the instruction itself to pass along to the subroutine called. However, as passing parameters in HLL relies on the stack, in addition to other mechanisms, the stack-based mechanism can easily be implemented in assembly. There are mainly three ways by which a caller procedure can pass parameters to a callee procedure:

1. *Use of registers.* Storing values or addresses into registers before the call is made.
2. *Use of global variables.* The caller must declare as *global* all variables used to pass values or addresses.
3. *Use of the stack.* This mechanism requires the execution by the caller procedure of one push instruction for each parameter passed as illustrated in Fig. 5.1. A common strategy to access parameters in the stack is to capture the contents of the ESP register into *ebp* register and access any location in the stack relative to the contents of *ebp*. However, since the caller program is also using the *ebp* register, its contents must be secured in the stack before overwriting it. Therefore, the first set of instructions to be performed by the callee program are the following:

```
subprocedure:
    push ebp
    mov ebp, esp
    ...
```

5.3.1 Examples

Example 1: Display \$-terminated strings using a sub-procedure. The only parameter needed is the offset of the message to display; since the offset is needed in register *dx* it is stored there before the call is made:

```
; Assemble using the 16-bit nasm assembler:
;                               nasm16 -f obj test_calls.asm -o test_calls.obj
; this will produce: test_calls.obj
; to link do: alink test_calls
; it will produce: test_calls.exe
```

```
USE16
```

```
SEGMENT mystack stack
    resb 100h
stacktop:
```

```
SEGMENT data
    msg1 db "hello world", 13, 10, '$'
    msg2 db " ", 13,10, '$'
```

```
msg3 db "this is the next line", 13, 10, '$'
```

```
SEGMENT code
```

```
..start:
```

```
    mov ax, data           ;initialize ds to point to
    mov ds, ax             ;the data segment
    mov ax, mystack
    mov ss, ax             ;initialize ss:sp to access the
    mov sp, stacktop       ;stack segment
```

```
    mov dx, msg1
    call display
    mov dx, msg2
    call display
    mov dx, msg3
    call display
```

```
    mov ax, 04C00H         ; select a DOS function code
    int 21H                ; to return to DOS
```

```
display:
```

```
    mov ah, 09h            ; select a DOS function code
    int 21h                ; call DOS service
    ret                   ; to display the string
```

Example 2: Display null-terminated strings in page 0 in video memory. The parameters required are a pointer to the message (*ds:si*) and a pointer to video memory (*es:di*). The corresponding registers are initialized used to pass these pointers.

```
; this code uses the segmented memory mode
; to assemble do: nasm16 -f obj vmcall.asm -o vmcall.obj
; this will produce: vmcall.obj
; to link do: alink vmcall
; It will produce: vmcall.exe
```

```
USE16                               ;equivalent to [BITS 16]
```

```
video_seg      equ 0B800h           ;video memory
attribute      equ 47h              ;color attribute
```

```
SEGMENT mystack stack
    resb 100h
```

```
stacktop:
```

```
SEGMENT data
```

```
    msg1    DB      'Hello, World !           ',00h
    msg2    DB      '                        ',00h
    msg3    DB      'This is the next line    ',00h
```

```
SEGMENT code PUBLIC
```

```
..start:
```

```
    mov ax, data
    mov ds, ax
    mov ax, mystack
    mov ss, ax
    mov sp, stacktop
```

```
;display messages
```

```
    mov si, msg1
    mov di, 10*160 + 2*25           ;offset of first message
    call vmshow
    mov si, msg2
    mov di, 11*160 + 2*25           ;offset of 2nd message
    call vmshow
    mov si, msg3
    mov di, 12*160 + 2*25           ;offset of 3rd. message
    call vmshow
```

```
;wait for a character
```

```
    mov ah, 8
    int 21h
```

```
;return to dos
```

```
    mov ax, 4c00h
    int 21h
```

```
vmshow:
```

```
    mov ax, video_seg
    mov es, ax
    mov ah, attribute
    mov cx, 80
```

```
next_char:
```

```
    lodsb                ;get the input string byte
    cmp al,00h           ;check for a null character
    je null_ch           ;if null, then quit
    stosw                ;store ax to video memory
    loop next_char       ;loop back to do it again
```

```

null_ch:
    ret

```

Example 3: This example is the same as example 2 but illustrates parameter passing using the stack:

```

USE16                                ;equivalent to [BITS 16]

video_seg      equ 0B800h            ;video memory
attribute      equ 47h               ;color attribute

SEGMENT mystack stack
    resb 100h
stacktop:

SEGMENT data
    msg1      DB      'Hello, World !'      ',00h
    msg2      DB      '                    ' ',00h
    msg3      DB      'This is the next line' ',00h

SEGMENT code PUBLIC

..start:
    mov ax, data
    mov ds, ax
    mov ax, mystack
    mov ss, ax
    mov sp, stacktop

;display messages
    push word msg1                    ;store offset of string
    push word 10*160+2*25             ;store offset of video memory
    call vmshow                       ;display
    push word msg2
    push word 11*160+2*25
    call vmshow
    push word msg3
    push word 12*160+2*25
    call vmshow
;wait for a character
    mov ah, 8
    int 21h

```



```

;return to dos
    mov ax, 4c00h
    int 21h

vmshow:
    push bp
    mov bp, sp
    mov di, [bp+4]           ;load offset of video memory      ;
    mov si, [bp+6]           ;load offset of string
    mov ax, video_seg
    mov es, ax               ;initialize video segment
    mov ah, attribute
    mov cx, 80

next_char:
    lodsb                   ;get the input string byte
    cmp al, 00h             ;check for a null character
    je null_ch              ;if null, then quit
    stosw                   ;store ax to video memory
    loop next_char          ;loop back to do it again

null_ch:
    pop bp
    ret

```

Example 4: This example illustrates the use of C-functions in a 32-bit protected mode environment to display messages (null-terminated strings) using the system call *printf*. The only parameter passed using the stack is the offset of the message to display. The mechanism to interface with C programs is covered in the next section.

```

; This program illustrates the use of system calls.
; Assemble using the 32-bit nasm assembler
;               nasm32 -f coff wprintf.asm
; this will produce: wprintf.o
; to link under djgpp do: gcc -o wprintf wprintf.o

SEGMENT .text
    global _main
    extern _printf

_main:
    push dword msg1         ;store pointer to message
    call _printf            ;display it
    add esp, 4              ;restore stack pointer

```

```
    push dword msg2
    call _printf
    add esp,4

    push dword msg3
    call _printf
    add esp,4
    ret

SEGMENT .data
    msg1:  db      "Hello, World !"           ", 10, 0"
    msg2:  db      "                        "   ", 10, 0"
    msg3:  db      "This is the next line"     ", 10, 0"
```

As shown in example 4 some system calls such as *printf* and *scanf* allow a varying number of arguments. As the function is called from assembly parameter values are pushed into the stack in the inverse order in which they would be listed. After the parameters are used by the function, the top of the stack is restored by adding to the *esp* register the space in bytes required by the arguments passed.

5.4 Interfacing C and Assembly Language

As shown in the previous section an alternative to passing parameter values is the use of the stack. This is what C functions do. The use of global variables to pass parameter values involve the allocation of memory to be used from the beginning until the end of the program. In contrast, the use of the stack allows the use of memory only while the procedure called is active. Another advantage of using the stack is the flexibility of storing the status of the calling program to preserve its integrity. As a result of such flexibility procedures are *reentrant*, i.e., they can be called any time at any place even *recursively*.

The section of the stack assigned to each call is referred to as the *stack frame*. It is a fixed block of memory within the stack used for parameters, return address, local variables and register storage. Any time a procedure is called, its stack frame is pushed into the stack; when finished, the stack frame is popped off the stack. Since a stack frame is assigned to each call and not to the procedure itself, *recursive* (*nested*) calls are possible.

5.4.1 Interfacing with 16-bit programs

In the following description, the words *caller* and *callee* are used to denote the programs doing the calling (a C-program) and the program which gets called (an assembly program), respectively. Also, C compilers require that all global variables and procedures have a leading underscore appended to the global symbol. The *elf* compiling specification does not require this *renaming* of global symbols. The caller pushes the function parameters onto the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last). As this the mechanism used to store parameters into the stack by C calls, it is referred as the *C-convention* to parameter passing and it is illustrated in Fig. 5.2a.

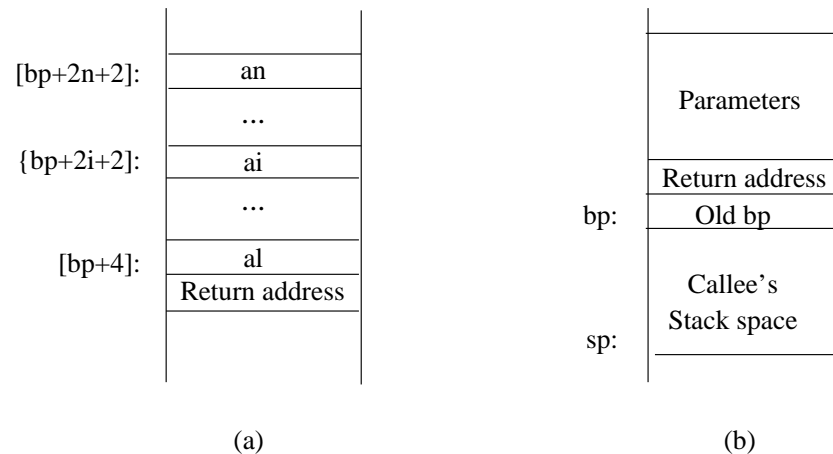


Figure 5.2: (a) C Parameter convention, (b) Callee's stack frame

After placing parameters into the stack, the caller program then executes a *call* instruction to pass control to the callee program. The callee receives control, and saves the contents of $(e)sp$ into $(e)bp$ to locate parameters in the stack frame relative to the contents of $(e)bp$. Hence the callee, must push the previous value of $(e)bp$ as shown in Fig. 5.2b. The following template illustrates the entry code and the exit code of a 16-bit sub-procedure *myfunc*:

```
global myfunc

myfunc:
    push bp
    mov bp, sp
    ...
    code for myfunc
    ...
```

```

mov sp, bp           ;restore the value of sp
pop bp
ret

```

After pushing *bp* into the stack and copying the value of *sp* into *bp*, *myfunc* can then access parameters relative to *bp*. The word at *bp* holds the previous value of *bp* as it was pushed last; the next word, at *bp* + 2, holds the offset part of the return address, pushed implicitly by the *call* instruction. For a near *call* the parameters start at [*bp* + 4]; in a far *call* the return address requires the segment part stored at [*bp* + 4], and the parameters begin at [*bp* + 6].

A callee program can use the stack by pushing local variables and decrease *sp* further; thus, saved local variables and registers are accessible at negative offsets from *bp*. To return a value to the caller, the callee should leave the value in *al*, *ax* or *dx:ax* depending on the size of the value.

Once the callee program has finished processing, it restores *sp* from *bp*; this step is necessary if local variables have been pushed into the stack. The callee then pops the previous value of *bp*, and returns via *ret* or *retf* depending on the memory model used. When the caller regains control, the function parameters are still on the stack, so it typically adds an immediate constant to *sp* to remove them (instead of executing a number of slow POP instructions).

Example 5: To illustrate the C-passing passing parameter convention, consider the stack frame of a 16-bit video display function callable from a C program.

```
video(string ptr, row, column, attribute, length, page)
```

The function *video* transfers a null-terminated string into video memory for display at the coordinates given by the *row* and *column* parameters. The string will be displayed with the selected *attribute*. Furthermore, the *row*, *column* and *page* will be used to calculate the corresponding offset in the video buffer in text mode. The set of parameters must be available on the stack in the order shown in Fig. 5.3.

Note that a far call is assumed as both the *segment* and the *offset* part of a physical address where the string buffer is located are passed. Likewise the reference to the next instruction is passed using both the segment and offset part.

5.4.2 Interfacing with 32-bit programs

Saving and loading parameters from the stack for 32-bit applications, is basically the same mechanism described for 16-bit applications. The *i*th entry within the

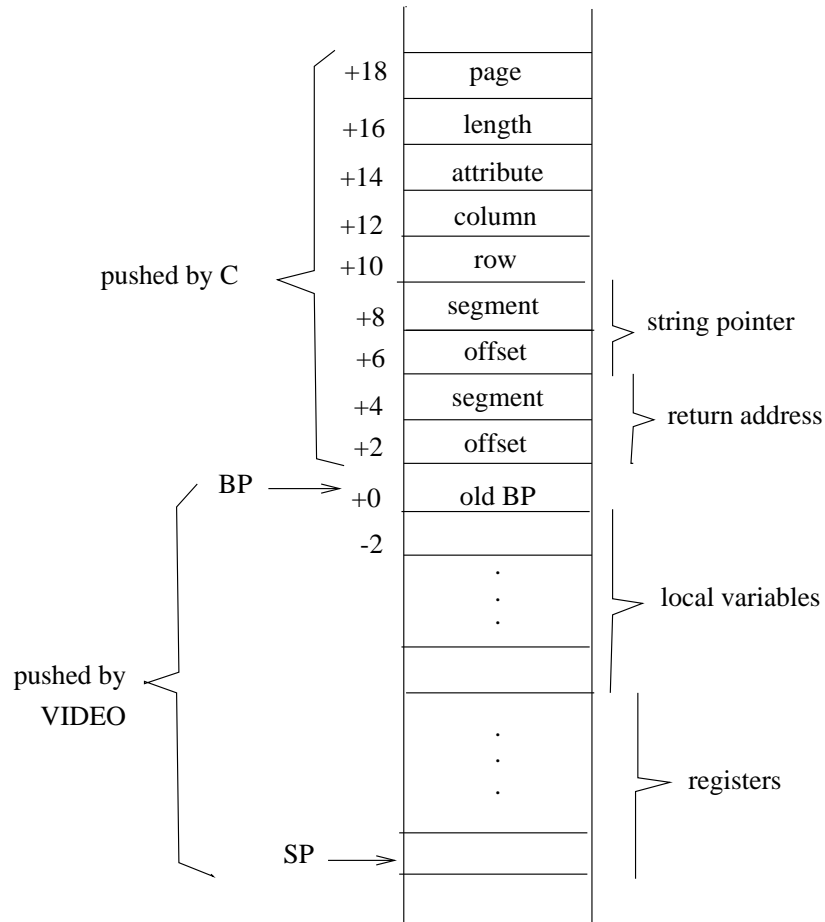


Figure 5.3: Stack frame for the call to VIDEO in real mode

parameter stack space changes to $[ebp + 4i + 4]$ to take into account that values now take 4 bytes. A callee procedure must preserve the values of *ebx*, *esp*, *ebp*, *esi* and *edi* as well as the contents of the segment registers *cs*, *ds*, *es* and *ss*; the values in these registers should be the same before and after the *call* is executed because under protected sub-procedures should not alter the contents of segment registers. Values are returned in *eax* if they are 32-bit or smaller in size. Values are returned in *edx:eax* if they require a 64-bit representation. Strings, structures, and other items above 32 bits in size are returned by reference, i.e., a pointer is returned in *eax*. As shown in example 4, C-library functions can be called directly from assembly procedures. C-libraries, however, may alter the state of the caller assembly program; therefore, to preserve the values in registers in use they must be saved into the stack before the library function is called. Fig. 5.4 shows the stack frame associated to a call under a 32-bit flat model programming environment. Notice that there is no need to pass the segment component of pointers.

ebp+28	page
ebp+24	length
ebp+20	attribute
ebp+16	column
ebp+12	row
ebp+8	string ptr.
ebp+4	eip
ebp+0	old ebp

Figure 5.4: Stack frame for the call to VIDEO in protected mode

Example 6: The sub-procedure *vmshow* previously coded to access video memory in real mode, is re-designed to be callable from a C-program in protected mode environment. Both the C-code and the assembly code are listed:

```
/* to compile and link: gcc -o dvtest dvtest.c vmshow.o */

#include <stdio.h>
#include <conio.h>
#include <dpmi.h>          /* we need this for the DPMI wrapper functions */
#include <string.h>

extern void vmshow(char *, int);

/* This will hold the selector we need to access the text mode
   video memory buffer. */

int vm_buffer = -1;

int main(void){
int row;
int col;

/* Set up our frame buffer descriptor, and store the selector to use */

vm_buffer = __dpmi_segment_to_descriptor(0xb800);
if(vm_buffer < 0) return 1;
```

```

    clrscr();
    row = 10; col = 25;
    vmshow("Hello world", 160*row+2*col);
    row = 11;
    vmshow(" ", 160*row+2*col);
    row = 12;
    vmshow("This a third line", 160*row+2*col);

    getchar();          /*type any character to terminate the program*/
    clrscr();

/* The descriptor must be released because it takes up resources */

    __dpmi_free_ldt_descriptor(vm_buffer);

    return 0;
}

void clrscr(void){
    int row;
    int col;
        for(row=0; row<25; row++)
            for(col=0; col<80; col++) vmshow(" ", 160*row+2*col);
}

; vmshow.asm
; to assemble do: nasm32 -f coff vmshow.asm -o vmshow.o

attribute          equ 47h          ;color attribute

global _vmshow
extern _vm_buffer

SEGMENT .text
_vmshow:
    push ebp
    mov ebp, esp
    push edi
    push esi
    push eax
    push es
    push ecx

```

```

    mov esi, [ebp+8]          ;offset message
    mov es, [_vm_buffer]
    mov edi, [ebp+12]
    mov ah, attribute
    mov ecx, 80
next_char:
    lodsb                    ;get the input string byte
    cmp al, 00h              ;check for a null character
    je null_ch               ;if null, then quit
    stosw                    ;store ax to video memory
    loop next_char           ;loop back to do it again
null_ch:
    pop ecx
    pop es
    pop eax
    pop esi
    pop edi
    pop ebp
    ret

```

As in example 3 *vmshow* will display a message at the coordinates *row*, *col* at page 0. A stack frame is setup to access at $[ebp + 8]$ the offset where the message is stored, and at $[ebp+12]$ a constant for the offset within page 0. Note that in protected mode there is no need to initialize the *ds* register needed to access the message in real mode. However, this is not the case for the video text buffer which is located at *B800h* which can not be accessed directly in protected mode. To ensure access to video memory *dpmi* services are invoked. From the C-program the segment pointer is converted to a descriptor-based format compatible with flat-model applications.

The Dos Protected Mode Interface (DPMI) is a set of library calls defined to allow *dos* programs to access the extended memory of IA computers while maintaining system protection. DPMI defines a specific subset of DOS and BIOS calls that can be made by protected mode DOS programs. It also defines a new interface via software interrupt 31h that protected mode programs use to allocate memory, modify descriptors, call real mode software, etc. Any operating system that currently supports virtual DOS sessions should be capable of supporting DPMI without affecting system security.

Some DPMI implementations can execute multiple protected mode programs in independent virtual machines. Thus, DPMI applications can behave exactly like any other standard DOS program and can, for example, run in the background or in a window (if the environment supports these features). Programs that run in protected mode also gain all the benefits of virtual memory and can run in a 32-bit

flat model if desired. Note that DPMI services are only available to protected mode programs.

Usually, memory-mapped devices or absolute addresses below the 1MB mark can not be accessed directly. This is because the combination *seg:offset* does not apply in protected mode. A special selector is required to gain access to a device or an absolute address. To create such a selector in example 5, the *djgpp* library function *__dpmi_segment_to_descriptor* with the *0xB800* argument is used. An *int* type variable *vm_buffer* is used to store the selector which is used in *vmshow* to initialize *es* with an offset already popped from the stack frame at *[ebp + 12]*. The remaining code is similar to the sub-procedure shown in example 3. An additional change is that the stack frame is used to store and restore the registers used by the sub-procedure and thus maintain the integrity of the calling program.

5.5 DOS and BIOS Function Calls

Hardware components can be accessed and controlled through the use of DOS and BIOS interrupts. These are software-directed interrupts that require specific registers to be preloaded with the required parameters. DOS and BIOS calls can perform the function selected by an 8-bit code function preloaded in *ah*. An interrupt service routine is activated by executing the instruction *int n*, where *n* is an 8-bit integer used to locate an entry in the interrupt vector table. This entry contains the address where the interrupt service routine is located. Common *int* instructions include the following:

int 10h: to service video functions

int 16h: to provide keyboard services

int 17h: printer services

int 1Ah: get/set number of ticks from the timer

int 21h: DOS services (I/O, file handler, memory management, etc.)

int 33h: mouse operations

5.5.1 BIOS Calls

BIOS calls are normally associated with display functions. One way to display text on the screen quickly is to use the BIOS interrupt 10h functions. A brief list of the more useful functions include the following:

Code function 0: Set video mode

Code function 2: Set cursor position

Code function 3: Read current cursor position

Code function 5: Change active page

Code function 6: Scroll active page up

Code function 9: Write attribute/character at the current cursor position

set video mode. This interrupt sets the video mode of the display. The register *ah* is preloaded with the code function *00h*. An 8-bit number associated with the desired video mode is loaded in *al*. For example the following set of instructions:

```
mov ah, 0
mov al, 3
int 10h
```

will set the screen to display 25 lines with 80 characters per line. After the interrupt is issued, the video mode is updated, the screen is cleared, and the cursor is placed in the upper left corner of the video screen.

Set cursor position. This interrupt places the cursor at any desired position on the text screen. Register *ah* must be loaded with *02h*, the function code. The coordinates, row and column, for the cursor position must be preloaded in *dh* and *dl*, respectively. The page selected for display is preloaded in register *bh*. For example to place the cursor at row 10 and column 25 in page 0 the following set of instructions are executed:

```
mov ah, 2
mov dh, 10
mov dl, 25
mov bh, 0
int 10h
```

Read current cursor position. This interrupt will return the row and column position of the current text cursor. Register *ah* must be preloaded with the function code *03h*, and the register *bl* must contain the page number. The output, row and column are returned in *dh* and *dl*, respectively.

Change active page. This interrupt changes the active page. After the execution of this interrupt the page displayed changes to the one specified in the call. The

function code is 05h that must be preloaded into *ah*. The desired new page is preloaded into register *al*.

Scroll active page up. This interrupt is called within a procedure to scroll the current active video page up. The function code is preloaded in *ah*. The following set of registers must also be preloaded with the information specified:

al = Number of rows to scroll up

bh = Attribute used

ch = Row number at top of region

cl = Column number at top-left of region

dh = Row number at bottom of region

dl = Column number at bottom-right of region

The attribute byte preloaded into *bh* controls what foreground and background colors will be used in the scroll region when scrolling is completed. Also if *al* is preloaded with 0, it specifies all the rows in the entire region. The following code clears the entire video screen:

```
mov ah, 6
mov al, 0
mov bh, 7
mov ch, 0
mov cl, 0
mov dh, 24
mov dl, 79
int 10h
```

Write attribute/character at the current cursor position. This interrupt can be used to display not only a character but the *attribute* byte with background and foreground color display information. As usual the register *ah* must contain the function code 09h. In addition to the function code the following input values are required in the indicated registers:

al = ascii code of character

bh = display page

bl = character's attribute

cx = Number of characters to write

The following code segment displays the character A with a foreground in a magenta background:

```
mov ah, 09h
mov al, 'A'
mov bh, 0
mov bl, 47h
mov cx, 1
int 10h
```

5.5.2 DOS Calls

Examples:

01: Get a character from keyboard (with echo)

input: ah = 01h

output: al = character

example:

```
mov ah, 01h
int 21h
mov byte [char], al
```

02h: Display a character

input: ah = 02h

input: dl = character

example:

```
mov ah, 02h
mov dl, 'A'
int 21h
```

09h: Display a \$-terminated string

input: ah = 09h

input: dx = offset of string

example:

```
mov ah, 09h
mov dx, msg
int 21h
```