CS221
Irvine Link Library, Procedures

Using the Irvine Link Library

The Irvine link library contains several useful routines to input data, output data, and perform several tasks that one would normally have to use many operating system calls to complete. In actuality, the Irvine library is simply an interface to these OS calls (e.g., it invokes either DOS calls or Windows 32 library routines).

This library is called irvine32.lib (for 32 bit protected mode) and irvine16.lib (for 16 bit real mode) and should have been installed when you installed the CD-ROM from the Irvine book. Chapter 5 contains a full list of the library routines. We will only cover a few basic ones here.

Most of what you will use in the Irvine link library are various procedures. To invoke a procedure use the format:

```
call procedureName
```

The call will push the IP onto the stack, and when the procedure returns, it will be popped off the stack and continue executing where we left off, just like a normal procedure in C++ or Java. The procedures should handle saving and restoring any registers that might be used. It is important to keep in mind that these are all high-level procedures – they were written by Irvine. That is, an x86 machine does not come standard with the procedures available. Consequently, if you ever wish to use these routines in other settings, you might need to write your own library routines.

Parameters are passed to Irvine's procedures through registers. Here are some of the procedures available:

Clrscr	Clears the screen, moves the cursor to the upper-left corner
Crlf	Writes a carriage return / linefeed to the display
Gotoxy	Locates the cursor at the specified X/Y coordinates on the screen. DH = row (0-24), DL = column (0-79)
Writechar	Writes a single character to the current cursor location AL contains the ASCII character
DumpRegs	Display the contents of the registers
ReadChar	Waits for a keypress. AH = key scan code AL = ASCII code of key pressed

Let's see these in action:

```
Include Irvine32.inc
.code
main proc
      call Clrscr
      mov dh, 24
      mov dl, 79  ; bottom-right corner
      call Gotoxy ; Move cursor there
      mov al, '*'
      call WriteChar    ; Write '*' in bottom right

      call ReadChar     ; Character entered by user is in AL
      mov dh, 10
      mov dl, 10
      call Gotoxy
      call WriteChar    ; Output the character entered at 10,10
      call CrLf   ; Carriage return to line 11

      call DumpRegs     ; Output registers

      ; output a row of '&'s to the screen, minus first column
      mov al, '&'
      mov cx, 79
      mov dh, 5   ; row 5
L1:   mov dl, cl
      call Gotoxy
      call WriteChar
      loop L1

      call CrLf
      exit
main endp
end main
```

Here are some more:

Randomize	Initialize random number seed
Random32	Generate a 32 bit random integer and return it in eax
RandomRange	Generate random integer from 0 to eax-1
Readint	Waits for/reads a ASCII string and interprets as a a 32 bit value. Stored in EAX.
Readstring	Waits for/reads a ASCII string. Input: EDX contains the offset to store the string ECX contains max character count Output: EAX contains number of chars input

Writeint	Outputs EAX as a signed integer
Writestring	Write a null-terminated string. Input: EDX points to the strings offset

Here is another code sample:

```
Include Irvine32.inc

.data
myInt DWORD ?
myChar BYTE ?
myStr BYTE 30 dup(0)
myPrompt BYTE "Enter a string:",0
myPrompt2 BYTE "Enter a number:",0

.code
main proc
      ; Output 2 random numbers
      call Randomize          ; Only call randomize once
      call Random32
      call WriteInt           ; output EAX as int
      call Crlf
      move ax, 1000
      call RandomRange
      call WriteInt           ; output EAX as int, will be 0-999
      call Crlf

      ; Get and display a string
      mov edx, offset myprompt
      call Writestring        ; Display prompt
      mov ecx, 30             ; Max length of 30
      mov edx, offset myStr
      call Readstring
      call Writestring        ; Output what was typed
      Call Crlf

      ; Get a number and display it
      mov edx, offset myprompt2
      call Writestring        ; Display prompt
      call ReadInt                  ; Int stored in EAX
      call Crlf
      call WriteInt
      call Crlf

      exit
main endp
end main
```

There are other procedures for displaying memory, command line arguments, hex, text colors, and dealing with binary values. See Chapter 5 of the textbook for details.

Procedures and Interrupts

As programs become larger and written by many programmers, it quickly becomes difficult to manage writing code in one big procedure. It becomes much more useful to break a problem up into many modules, where each module is typically a function. This is the idea behind modular programming that you should have seen in CS201 and CS101.

When a procedure is invoked, the current instruction pointer is pushed onto the stack and then the IP is loaded with the address of the procedure. When the procedure exits, the old instruction pointer is popped off the stack and copied into the instruction pointer. Consequently, we then continue operation from the next instruction after the procedure invocation.

In this fashion, the stack is serving as a temporary storage area for the instruction pointer. Although the IP is pushed and popped off the stack automatically, you can also use the stack yourself to save your own variables.

The instruction to push something on the stack is PUSH:

```
        PUSH register
        PUSH memval
```

Two registers, CS and EIP, cannot be used as operands (why?)

To get values off the stack, use POP:

```
        POP register                ;  top of stack goes into register
        POP memval                  ; top of stack goes into memory location
```

For example:

```
        PUSH eax
        POP ebx
```

Essentially copies AX into BX by way of the stack.

A common purpose of the stack is to save a temporary value. For example, let’s say that we want to make a nested loop, where the outer loop goes 10 times and the inner loop goes 5 times:

```
        MOV ecx, 10
L1:     …
        …                   ; stuff for outer loop
        MOV ecx, 5          ; Setup inner loop
L2:     …
        …                   ; stuff for inner loop
        LOOP L2
        …
        LOOP L1
```

The obvious problem here is that we are wiping out the counter ECX for the outer loop in the inner loop. An easy solution is to save the value in ECX before we execute the inner loop, and then restore it when we finish the inner loop:

```
        MOV ecx, 10
L1:     …
        …                   ; stuff for outer loop
        PUSH ecx            ; Save ECX value in outer loop
        MOV ecx, 5          ; Setup inner loop
L2:     …
        …                   ; stuff for inner loop
        LOOP L2
        POP ECX             ; Restore ECX value in outer loop
        …
        LOOP L1
```

Another common place where values are pushed on the stack temporarily is when invoking a function call that wipes out register values we want. For example, the Irvine Readstring function expects EDX to point to the offset of the string in memory. The number of characters input by the user is returned in AX. If these registers contained values we wanted to save, we could push them onto the stack and restore them later after the Readstring operation is finished.

Most high-level languages pass parameters to function by pushing them on the stack. The function then accesses the parameters as offsets from the stack pointer. This has the advantage that an arbitrary (up to the size of free space on the stack) number of parameters can be passed to the function. In contrast, the Irvine Link Library and DOS interrupt routines pass parameters through registers. This has the disadvantage that a limited number of values can be passed, and we might also need to save the registers if the function changes them in some way. However, it is faster to pass parameters in registers than to pass them on the stack.

One final PUSH and POP instruction is quite useful:

PUSHA	: Push ALL 16 bit registers on the stack, except for Flags Code Segment EIP, and Data Segment
POPA	: Pops ALL 16 bit registers off and restores them
PUSHAD	: Pushes all extended registers except above
POPAD	: Pops all extended registers

If we want to save the flags registers, there is a special instruction for it:

PUSHF	: Push Flags
POPF	: Pop Flags

Writing Procedures

You have already been defining your own procedures – the main procedure works just like any other procedure.

The format to define a procedure is:

```
<Procedure-Name> proc
        …
        …       ; code for procedure
        …
        ret     ; Return from the procedure
<Procedure-Name> endp
```

The keyword proc indicates the beginning of a procedure, and the keyword endp signals the end of the procedure. Your procedure must use the RET instruction when the procedure is finished. This causes the procedure to return by popping the instruction pointer off the stack.

Note that all other registers are not automatically pushed on the stack. Therefore, any procedures you write must be careful not to overwrite anything it shouldn't. You may want to push the registers that are used just in case, e.g.:

```
MyProc PROC
        Push EAX            ; If we use EAX, push it to save its value
        Push EBX
        …
        …                   ; Use EAX
        …
        POP EBX             ; Restore original value in EBX
        POP EAX             ; Restore original value in EAX
MyProc ENDP
```

To invoke a procedure, use call:

call procedure-name

Here is an example of a program that uses a procedure to compute EAX raised to the EBX power (assuming EBX is a relatively small positive integer). In this example we save all registers affected or used by the procedure, so it is a self-contained module without unknown side-effects to an outside calling program:

```
Include Irvine32.inc

.data

.code
main proc
      mov eax, 3
      mov ebx, 9
      call Power  ; Compute 3^9
      call WriteInt

      exit
main endp

power proc
      push ecx
      push edx    ; MUL changes EDX as a side effect
      push esi
      mov esi, eax
      mov ecx, ebx
      mov eax, 1
L1:   mul esi        ; EDX:EAX = EAX * ESI.
      loop L1
      pop esi
      pop edx
      pop ecx
      ret
power endp

end main
```

Note that we can also make recursive calls, just like we can in high-level languages. However, if we do so, we must push parameters on the stack so that there are separate copies of the variables for each invocation of the procedure. We can access these variables as offsets from the Stack Pointer; typically the Base Pointer is used for this purpose.

Software Interrupts

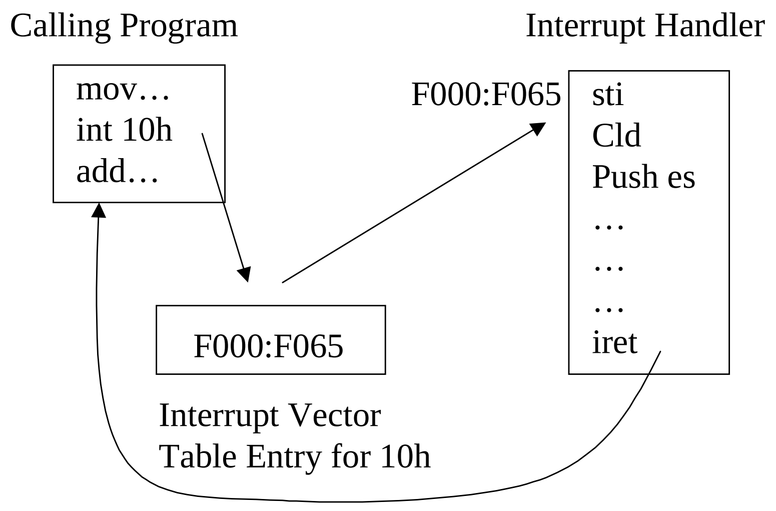
Technically, a software interrupt is not really a "true" interrupt at all. It is just a software routine that is invoked like a procedure when some hardware interrupt occurs. However, we can use software interrupts to perform useful tasks, typically those provided by the operating system or BIOS.

Here, we will look at software interrupts provided by MS-DOS. A similar process exists for invoking Microsoft Windows routines (see Chapter 11). Since we will be using MS-DOS, our programs must be constructed in real mode.

Software interrupts in DOS are invoked with the INT instruction. The format is:

```
INT <number>
```

Number indicates which entry we want out of the interrupt vector table. For example:



These interrupt handlers typically expect parameters to be passed in certain registers. For example, DOS interrupt 21h with AH=2 indicates that we want to invoke the code to print a character.

Here are some commonly used interrupt services

INT 10h	- Video Services
INT 16h	- Keyboard services
INT 1Ah	- Time of day
INT 1Ch	- User timer, executed 18.2 times per second
INT 21h	- DOS services

See chapter 13, chapter 15, appendix C, and the web page linked from the CS221 home page for more information about all of these interrupts, particularly the DOS interrupt services.

BIOS-Level Video Control (INT 10h) – Chapter 15.4-15.5

Let's say a little bit more about using the video interrupt. If we are in real mode then Int 10h allows us to select different video modes. Early computers supported only monochrome, but later versions allowed for CGA, EGA, and VGA resolutions. With the different modes we have different ways to display text (in various colors, for example), and different resolutions for graphics.

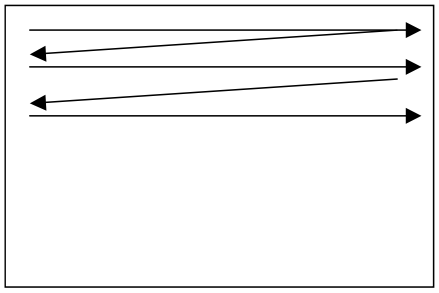
For example:

```
mov ah, 0                    ; 0 in AH means to set video mode
mov al, 6                    ; 640 x 200 graphics mode
int 10h

mov ah, 0
mov al, 3                    ; 80x25 color text
int 10h

mov ah, 0
mov al, 13h                  ; linear mode 320x200x256 color graphics
int 10h
```

Looking at the last mode, this turns the screen to graphics mode with a whopping 320x200 resolution and 256 colors. This means that each color is represented by one byte of data. There is a color palette stored on the video card that maps each number from 0-255 onto some color (e.g., 0=black, 1=dark grey, etc.). We can set these colors using the OUT instruction, but for our purposes we will just use the default palette.

Video memory begins at segment A0000:0000. This memory location is mapped to the graphics video screen, just like we saw that memory at B800:0000 was mapped to the text video screen.

The byte located at A000:0000 indicates the color of the pixel in the upper left hand corner (coordinate x=0,y=0). If we move over one byte to A000:0001, this indicates the color of the pixel at coordinate (x=1, y=0). Since this graphics mode gives us a total of 320 horizontal pixels, the very last pixel in the upper right corner is at memory address A000:013F, where 13F = 319 in decimal. The coordinate is (x=319, y=0). The next memory address corresponds to the next row; A000:0140 is coordinate (x=0, y=1). The memory map fills the rows from left to right and columns from top to bottom:



Not only can we access pixels on the screen by referring to the memory address, but by storing data into those memory locations, we draw pixels on the screen. Can you figure out what the following program does?

```
Include Irvine16.inc
.data
mynum BYTE 3

.code
main proc
      mov ah, 0               ; Setup 320x200x256 graphics mode
      mov al, 13h
      int 10h

      mov ax, 0A000h          ; Move DS to graphics video buffer
      mov ds, ax

      mov ax, 128
      mov cx, 320
      mov bx, 0
L1:   mov [bx], al            ; Stores AL into DS:BX
      inc bx
      loop L1

      mov cx, 320
      mov ax, 120
      mov bx, 320*199
L2:   mov [bx], ax
      inc bx
      loop L2

      call Readchar

      mov ax, @data     ; Restore DS to our data segment
      mov ds, ax        ; Necessary if we want
                        ; to access any variables
                        ; since we changed DS to A000

      mov al, mynum           ; Stores DS:MyNum into AL
      mov ah, 0         ; Restore text video mode
      int 10h

      exit
main endp

end main
```

Notice how we must restore DS to @data (the location of our data segment) if we ever want to access variables defined in our segment. This is because we changed DS to A000 to access video memory, and unless we change it back then we cannot access variables in our data segment.

An alternate technique is to use the Extra Segment register. We can set ES to the segment we want, and then reference addresses relative to ES instead of the default of DS.

Here is an example which also shows what colors are available by default:

```
Include Irvine16.inc
.data
.code
main proc
      mov ax, @data           ; set up DS register in case we
      mov ds, ax              ; want to access any variables declared
                              ; in .data.

      mov ah, 0               ; Setup 320x200x256 graphics mode
      mov al, 13h
      int 10h

      mov ax, 0A000h          ; Move ES to graphics video buffer
      mov es, ax

      mov cx, 255             ; Only loop up to 255 times
      mov bx, 0
L1:   mov es:[bx], bl         ; Move BL into [BX] instead of 128
      inc bx                  ; Note use of ES to override default DS
      loop L1

      call ReadChar

      mov ah, 0               ; Restore text video mode
      mov al, 3
      int 10h

      exit
main endp

end main
```