PModeLib Lecture

Peter L. B. Johnson

johnsonp@bilogic.org

23 October 2001

This lecture introduces PModeLib, the protected mode standard library for ECE 291. It covers general use, memory allocation, file I/O, protected mode interrupt handlers, and high-resolution graphics. This lecture is a summary of the material available in Chapter 5 of the Protected Mode Tutorial.

---

1 What is PModeLib?

PModeLib is the standard protected mode library for ECE 291. It provides a much larger set of functions than the real mode library to make programming in protected mode easier. Almost 100 functions cover memory handling, file I/O, graphics files, interrupts and callbacks, text mode, graphics mode, networking, sound, and DMA in addition to some other general purpose functions. A full reference to all of the functions is available online at http://courses.ece.uiuc.edu/ece291/books/pmode-tutorial/pmodelib-ref.html (it's also linked to from the resources page).

---

2 Calling Conventions

The entire library, with the exception of 3 functions, uses the 32-bit C calling convention. Parameters are passed on the stack; the return value is in the accumulator register (`EAX`/`AX`/`AL`, depending on size); `EAX`, `EBX`, `ECX`, and `EDX` may be overwritten by the function; and the function name is prepended with an underscore. As the functions are documented using the C notation, the size of each of the types is as follows:

- short: 16 bit integer (default signed)
- int: 32 bit integer (default signed)
- pointer (of any type): 32-bit
- bool: 32-bit value, 1=true, 0=false

Pointer parameters are indicated by the `*` character. C pointers are just addresses, so pass the variable's address, not its contents.

Because some of the functions can take many parameters, a set of macros which implement the C calling convention has been provided to make it easier to both write and use these functions. We highly suggest using the `invoke` macro to call the library functions.

Example 1. Using the `invoke` macro

```
invoke  _FindGraphicsMode, word 640, word 480, word 32, dword 1
invoke  _SetGraphicsMode, ax
invoke  _CopyToScreen, dword Image, dword 320*4, dword 0, dword 0, dword 320, dword 240, dword 160, dword 120
invoke  _UnsetGraphicsMode
```

---

3 Using PModeLib: A Framework

Any program that uses PModeLib should follow this basic framework. It must also link with `lib291.a` (this will be done by default in the protected mode MP).

```
%include "lib291.inc"

GLOBAL _main

...

SECTION .text

_main
        call    _LibInit         ; You could use invoke here, too
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .initerror

... do stuff using PModeLib functions ...

        call    _LibExit
.initerror:
        ret                      ; Return to DOS
```

---

4 Allocating Memory

Allocating memory is something we've never needed to do before. So why do we need to worry about it now? Primarily because we're going to start working with some *really* big (multi-megabyte) data such as images. Once we go beyond a few kilobytes, it's smart to dynamically allocate memory at run time, and PModeLib provides a function to make this task *much* easier.

_AllocMem takes just a single parameter: *Size*, which specifies the number of bytes to allocate. It returns the starting offset of the newly allocated block, which you can use just like any other offset (such as to a variable). Generally it's a smart idea to store this offset in a variable, which adds a layer of indirection, but makes it easier to allocate and keep track of several memory blocks at once. There is no way to free memory once it's allocated; the memory is freed when the program exits.

Example 2. Allocating Memory

```
%include "lib291.inc"

GLOBAL _main

test1size equ 4*1024*1024       ; 4 MB
test2size equ 1*1024*1024       ; 1 MB

SECTION .bss    ; Uninitialized data

test1off        resd 1   ; stores offset of test1 data
test2off        resd 1   ; stores offset of test2 data

SECTION .text

_main
        push    esi              ; Save registers
        push    edi

        call    _LibInit         ; You could use invoke here, too
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .initerror

        ; Allocate test1 memory block
        invoke  _AllocMem, dword test1size
        cmp     eax, -1          ; Check for error (-1 return value)
        je      near .error
        mov     [test1off], eax  ; Save offset in variable

        ; Allocate test2 memory block
        invoke  _AllocMem, dword test2size
        cmp     eax, -1          ; Check for error (-1 return value)
        je      near .error
        mov     [test2off], eax  ; Save offset in variable

        ; Fill the test1 block with 0's.
        ; We don't need to set es=ds, because it's that way at start.
        xor     eax, eax         ; Fill with 0
        mov     edi, [test1off]  ; Starting address (remember indirection)
```

```
        mov     ecx, test1size/4 ; Filling doublewords (4 bytes at a time)
        rep stosd                ; Fill!

        ; Copy from last meg of test1 to test2
        mov     esi, [test1off]  ; Starting address of source
        add     esi, test1size-1024*1024 ; Move offset to last meg
        mov     edi, [test2off]  ; Destination
        mov     ecx, test2size/4 ; Copying dwords
        rep movsd

.error:
        call    _LibExit
.initerror:
        pop     edi              ; Restore registers
        pop     esi
        ret                      ; Return to DOS
```

---

5 File I/O

Just filling memory with constant values isn't very interesting (or useful). It's far more useful to be able to load in data from an external file: graphics being the most obvious example. However, data such as maps, precalculated function tables, and even executable code can be loaded from disk. The library itself loads executable code from disk for the graphics driver.

The library has a set of general file handling functions that make opening, closing, reading, and writing files much easier. The `_OpenFile` function takes a pointer to (the address of) the filename to open, and returns an integer ***handle***, which identifies the file for all of the other file functions. It is therefore possible to have multiple files open at the same time, but be aware that there is a limit on the maximum number of open files, so it's smart to have as few open at the same time as possible: when loading multiple files, open, read, and close one before loading the next.

As the library has a specialized set of functions for loading graphics files, it's wise to use those instead of the generic file functions for loading graphics files. We'll use those when we cover high-resolution graphics using PModeLib.

Example 3. File I/O

```
%include "lib291.inc"

GLOBAL _main

mapsize equ 512*512     ; 512x512 map

SECTION .bss    ; Uninitialized data

mapoff  resd 1  ; Offset of the map data

SECTION .data   ; Initialized data

mapfn   db "mymap.dat",0        ; file to load data from (notice 0-terminated)

SECTION .text

_main
        push    esi              ; Save registers

        call    _LibInit         ; You could use invoke here, too
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .initerror

        ; Allocate memory for map
        invoke  _AllocMem, dword mapsize
        cmp     eax, -1          ; Check for error (-1 return value)
        je      near .error
        mov     [mapoff], eax    ; Save offset

        ; Open file for reading
        invoke  _OpenFile, dword mapfn, word 0
        cmp     eax, -1          ; Check for error (-1 return value)
        je      near .error
        mov     esi, eax         ; EAX will get overwritten by ReadFile so save
```

```
        ; Read mapsize bytes from the file.
        ; Note the indirection for the address of the buffer.
        invoke  _ReadFile, esi, dword [mapoff], dword mapsize
        cmp     eax, mapsize     ; Check to see if we actually read that much
        jne     .error

        ; Close the file
        invoke  _CloseFile, esi

.error:
        call    _LibExit
.initerror:
        pop     esi              ; Restore registers
        ret                      ; Return to DOS
```

---

6 Protected Mode Interrupt Handling

We've previously covered real mode interrupt handling, calling DOS to change the interrupt table to point at our code, chaining to the old interrupt handler for timer interrupts, and other concepts. While the general concepts don't change when we go to protected mode, the implementation does, and there are several functions in PModeLib to make the transition less painful.

The `_Install_Int` and `_Remove_Int` PModeLib functions make it easy to install a standard interrupt handler in protected mode (eg, one for timer or keyboard). The interrupt handler is just a normal subroutine (it should end with a `ret` instruction), and it should return a value in `EAX` to indicate whether the interrupt should be chained to the old handler or not: a zero value indicates the interrupt should just return (real-mode `iret`), a nonzero value indicates the interrupt should chain to the old handler (real-mode `jmp` or `call`).

One thing that is important to remember is to ***lock*** the memory areas an interrupt handler will access; this includes any variables it uses and the interrupt handler code itself. The reason we need to lock these areas is due to paging: any area of the program may be swapped out to disk by the operating system and replaced with another piece of code or data. While it is automatically reloaded when accessed by the program, this can cause unacceptable delay for interrupt handlers, as it may take many milliseconds to load the code or data back from disk. Locking prevents the operating system from paging out that area of memory. So why don't we lock the whole program? It's really unfriendly to do that in a multitasking environment, especially if your program takes up a lot of memory and it's a limited-memory system. Locking is another reason to keep your interrupt handlers short and keep most of the processing in the main loop (which doesn't have to be locked). The PModeLib function `_LockArea` is used to lock memory areas.

Example 4. Hooking the timer interrupt

```
%include "lib291.inc"

GLOBAL _main

SECTION .bss

timercount resd 1        ; Number of ticks received

SECTION .text

; Timer interrupt handler
TimerDriver
        inc     dword [timercount]

        ; No PIC acknowledge (out 20h, 20h) required because we're chaining.
        mov     eax, 1           ; Chain to the previous handler
        ret                      ; Note it's ret, not iret!
TimerDriver_end

_main
        call    _LibInit         ; You could use invoke here, too
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .initerror

        ; Lock up memory the interrupt will access
        invoke  _LockArea, ds, dword timercount, dword 4
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .error
```

```
        ; Lock the interrupt handler itself.
        ; Note that we use the TimerDriver_end label to calculate the length
        ;  of the code.
        invoke  _LockArea, cs, dword TimerDriver, dword TimerDriver_end-TimerDriver
        test    eax, eax        ; Check for error (nonzero return value)
        jnz     near .error

        ; Install the timer handler
        invoke  _Install_Int, dword 8, dword TimerDriver
        test    eax, eax        ; Check for error (nonzero return value)
        jnz     near .error

        ; Loop until we get a keypress, using int 16h
.loop:
        mov     ah, 1           ; BIOS check key pressed function
        int     16h
        jz      .loop           ; Loop while no keypress

        xor     eax, eax        ; BIOS get key pressed
        int     16h

        ; Uninstall the timer handler (don't forget this!)
        invoke  _Remove_Int, dword 8

.error:
        call    _LibExit
.initerror:
        ret                     ; Return to DOS
```

See the `examples` directory in `V:/ece291/pmodelib` for more examples.

---

7 High-Resolution Graphics

Now for the fun stuff! High-resolution graphics is where protected mode really shows off its full capabilities. Most of what we'll do in this section is nearly impossible in real mode due to 64k segment limitations. We're going to make a very short program which will load a 640x480 graphics file from disk and display it on the screen.

7.1 Graphics Files

We won't go into all the details of the various graphics file formats here, but let's briefly list the formats supported by PModeLib:

- BMP - Windows Bitmap format. Uncompressed, no alpha support. This format is only provided for completeness, and for the ability to save images. PNG and JPG provide compression and alpha channel support. The PModeLib functions `_LoadBMP` and `_SaveBMP` support loading and saving of 8-bit and 24-bit images.
- PNG - Portable Network Graphics format. Non-lossy compression, alpha channel support, and many bit depths. It's good for sprites and non-photographic images. The PModeLib function `_LoadPNG` can load any PNG image.
- JPG - JPEG image format. Lossy compression, no alpha support, only 24-bit images. It's excellent for photographic images, as very high compression rates can be achieved with little quality loss. The PModeLib function `_LoadJPG` can load any JPG image.

All of these functions assume an internal pixel format of uncompressed 32-bit RGBA (the loaded A channel is 0 for formats that don't support it). This may or may not be the same format as the video mode selected, so it may be necessary to write conversion functions to convert between the pixel format used by these functions and the pixel format of the display.

7.2 Video Graphics

In real mode, we used BIOS Interrupt 10h to set graphics modes and segment B800h or A000h to address the graphics memory. In protected mode, we'll use PModeLib functions to both set the graphics mode and copy image data to the screen. In fact, there

isn't a way to directly access the graphics memory, so it's necessary to do double-buffering (although it's possible to double-buffer just regions of the screen rather than the entire display area).

There's another issue with the graphics drivers we use: they require that the keyboard be remapped to a different IRQ and I/O port than the normal keyboard interface (which is at IRQ 1, I/O port 60h). The `_InitGraphics` function returns the remapped values in the variables whose addresses are passed to it. This will be clearer when we look at the code.

Under Windows 2000, we need to load a special graphics driver called `EX291` before running any program that uses PModeLib graphics. Just enter **`EX291`** at the command prompt before running the program to load the driver. Currently, PModeLib graphics *require* Windows 2000, they cannot work on Windows 98 or Windows ME.

7.3 Displaying an Image on the Screen

Okay, here's the complete code to a simple program that loads a 640x480 image named `image.jpg` and displays it on the 640x480 display. Since this program uses PModeLib graphics, we'll need to load the `EX291` driver before running it on Windows 2000.

This program combines all of the concepts earlier in this lecture, except we're using a specialized image loading function instead of general file I/O to load the image.

Example 5. Displaying an Image

```
%include "lib291.inc"

GLOBAL _main

imagesize equ 640*480*4          ; 640x480 image, 32 bits per pixel

SECTION .bss    ; Uninitialized data

imageoff        resd 1  ; Offset of the image data

doneflag        resb 1  ; =1 when we're ready to exit (set by KeyboardHandler)

kbINT   resb 1  ; keyboard interrupt number (standard = 9)
kbIRQ   resb 1  ; keyboard IRQ (standard = 1)
kbPort  resw 1  ; keyboard port (standard = 60h)

SECTION .data   ; Initialized data

imagefn db "image.jpg",0         ; image file to read (notice 0-terminated)

SECTION .text

KeyboardHandler
        ; Indicate that we're finished on any keypress
        ; If we wanted to check the key, we'd need to use [kbPort], not 60h.
        mov     byte [doneflag], 1

        ; Acknowledge interrupt to PIC.
        ; As the IRQ might be >=8 (a high IRQ), we may need to
        ;  out A0h, 20h, in addition to the normal out 20h, 20h.
        mov     al, 20h
        cmp     byte [kbIRQ], 8
        jb      .lowirq
        out     0A0h, al
.lowirq:
        out     20h, al

        xor     eax, eax         ; Don't chain to old handler
        ret
KeyboardHandler_end

_main
        push    esi              ; Save registers

        call    _LibInit         ; You could use invoke here, too
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .initerror
```

```
        ; Allocate memory for image
        invoke  _AllocMem, dword imagesize
        cmp     eax, -1          ; Check for error (-1 return value)
        je      near .error
        mov     [imageoff], eax  ; Save offset

        ; Load image
        invoke  _LoadJPG, dword imagefn, dword [imageoff], dword 0, dword 0
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .error

        ; Initialize graphics (and find remapped keyboard info)
        invoke  _InitGraphics, dword kbINT, dword kbIRQ, dword kbPort
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .error

        ; Lock up memory the handler will access
        invoke  _LockArea, ds, dword doneflag, dword 1
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .exitgraphics

        invoke  _LockArea, ds, dword kbIRQ, dword 1
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .exitgraphics

        ; Lock the interrupt handler itself.
        invoke  _LockArea, cs, dword KeyboardHandler, dword KeyboardHandler_end-KeyboardHandler
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .exitgraphics

        ; Install the keyboard handler
        movzx   eax, byte [kbINT]
        invoke  _Install_Int, dword eax, dword TimerDriver
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .exitgraphics

        ; Find 640x480x32 graphics mode, allowing driver-emulated modes
        invoke  _FindGraphicsMode, word 640, word 480, word 32, dword 1
        cmp     ax, -1           ; Did we find a mode?  If not, exit.
        jne     near .uninstallkb

        ; Go into graphics mode (finally :)
        invoke  _SetGraphicsMode, ax
        test    eax, eax         ; Check for error (nonzero return value)
        jnz     near .uninstallkb

        ; Copy the image to the screen
        invoke  _CopyToScreen, dword [imageoff], dword 640*4, dword 0, dword 0, dword 640, dword 480, dword 0, dword 0

        ; Wait for a keypress
.loop:
        cmp     byte [doneflag], 0
        jz      .loop

        ; Get out of graphics mode
        invoke  _UnsetGraphicsMode

.uninstallkb:
        ; Uninstall the keyboard handler
        movzx   eax, byte [kbINT]
        invoke  _Remove_Int, dword eax

.exitgraphics:
        ; Shut down graphics driver
        invoke  _ExitGraphics

.error:
        call    _LibExit
.initerror:
        pop     esi              ; Restore registers
        ret                      ; Return to DOS
```