ROZDZIAŁ DWUDZIESTY TRZECI: **MONITOR EKRANOWY PC**

Monitor ekranowy jest bardzo złożonym systemem. Po pierwsze nie jest pojedynczym urządzeniem jakie istnieją dla portów równoległego i szeregowego., lub nawet kilkoma urządzeniami (jak system klawiaturowy PC). Są dosłownie tuziny różnych kart rozszerzeń monitora dostępnych dla PC. Ponadto, każde rozszerzenie wspiera kila różnych trybów wyświetlania . Mając daną dużą liczbę trybów wyświetlania i używanych rozszerzeń wyświetlania, łatwo byłoby napisać grubą książkę o samym monitorze PC. Jednakże to nie jest ten tekst. Ta książka byłaby lekko niekompletna bez przynajmniej wzmianki o monitorze PC, ale nie ma dosyć miejsca aby wgłębiać się w ten temat. Dlatego też, rozdział ten będzie omawiał tryb wyświetlania tekstowego 80x25 , który wspierają prawie wszystkie rozszerzenia wyświetlania.

23.1 ODWZOROWANIE PAMIĘCI VIDEO

Wiele urządzeń peryferyjnych w PC używa wejścia / wyjścia odwzorowanego. Program komunikujący się z odwzorowanymi I/O używa instrukcji 80x86 in, out, ins i outs uzyskując dostęp od urządzeń w przestrzeni adresowej I/O PC. Podczas gdy chip kontrolera video, który pojawia się na karcie monitora PC również odwzorowywuje rejestry PC w przestrzenia I/O, karty te również stosują drugą postać I/O wejście/ wyjście odwzorowywane w pamięci.. W szczególności, tryb tekstowy 80x25 jest niczym więcej niż dwu wymiarową tablicą słów, z czego każde słowo w tablicy odpowiada znakowi na ekranie. Tablica ta pojawia się powyżej punktu 649 K w przestrzeni adresowej PC. Jeśli przechowujemy dane w tej tablicy używając standardowych instrukcji adresowania (np. mov), będziemy wpływali na znaki pojawiające się na ekranie.

W rzeczywistości są dwie róże tablice o które musimy się martwić. System monochromatyczny (pamiętacie?) lokuje swój start pod lokacją B000:0000 w pamięci. System kolorowy lokuje pod adresem B800:0000 w pamięci. Te lokacje są adresami bazowymi kolumnowej organizacji elementów tablicy jak zadeklarowano poniżej :

```
Display: array [0..79, 0..24] of word;
```

Jeśli wolimy działać z rzędowym pozycjonowaniem elementów tablicy, żaden problem, monitor jest równy poniższej definicji tablicy :

```
Display: array [0..24, 0..79] of word;
```

Zauważmy, że lokacja (0,0) jest to lewy górny róg a lokacja (79, 24) jest to prawy dolny róg monitora (wartości w nawiasach to współrzędne x i y, ze współrzędna poziomą x pojawiająca się najpierw)

Mniej znaczący bajt każdego słowa zawiera kod PC/ASCII dla znaku jaki chcemy wyświetlić. Bardziej znaczący bajt każdego słowa jest atrybutem bajtu. Wrócimy do atrybutu bajtu w kolejnej sekcji.

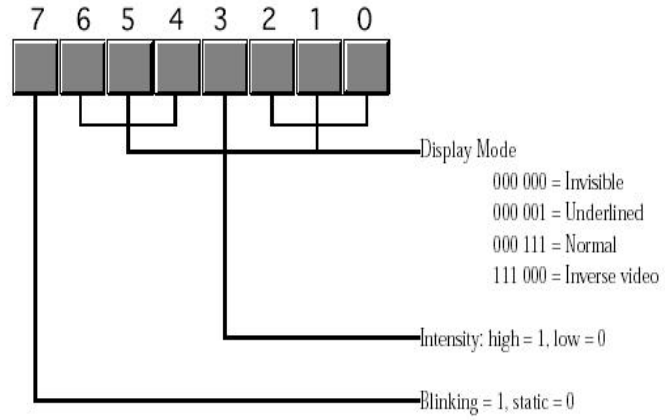
Wyświetlenie stron zajmuje odrobinę mniej niż 4Kilobajty w odwzorowaniu pamięci. Karata wyświetlacza kolorowego w rzeczywistości dostarcza 32K dla trybu tekstowego i pozwala wybrać jeden z ośmiu różnych wyświetleń Każde takie wyświetlanie zaczyna się od granicy 4K pod adresami B800:0, B800:1000, B800:2000,...B800:7000. Zauważmy, że większość nowoczesnych kolorowych monitorów w rzeczywistości dostarcza pamięci spod adresu A000:0 do B000:FFFF (i więcej), ale tryb tekstowy używa tylko 32K od B800:0 do B800:7FFF. W rozdziale tym skoncentrujemy się tylko na pierwszej stronie kolorowego monitora pod adresem B800:0. Jednakże wszystkie omówienia w tym rozdziale odnoszą się również do innych stronic wyświetlania.

Karta monochromatyczna dostarcza tylko pojedynczej strony wyświetlania . Istotnie, najwcześniejsze monitory monochromatyczne miały tylko 4K wbudowanej pamięci (kontrastuje to z kolorowymi monitorami o wysokiej rozdzielczości , które mają do czterech megabajtów wbudowanej pamięci.

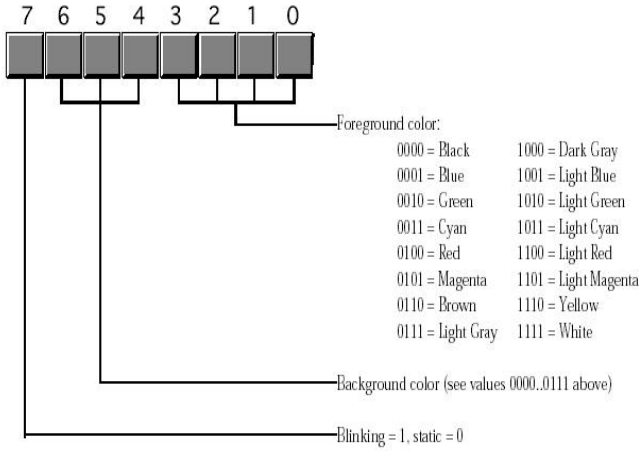
Możemy zaadresować pamięć monitora jak zwykły RAM. Możemy przechować nawet zmienne programowe lub nawet kod w tej pamięci. Jednakże nigdy takie cos nie jest dobrym pomysłem. Przede wszystkim zapisujemy , w dowolnym czasie , na ekran, więc zniszczymy zmienne przechowywane w aktywnej pamięci monitora. Nawet jeśli przechowujemy taki kod lub zmienne w nieaktywnej stronie wyświetlacza (np. strony od jeden do siedem na kolorowym monitorze) używanie pamięci w ten sposób nie jest dobrym pomysłem ponieważ dostęp do karty jest bardzo wolny. Pamięć główna działa od dwóch do dziesięciu razy szybciej (w zależności od maszyny)

23.2 ATRYBUT BAJTU VIDEO

Atrybut video związany jest z każdym znakiem na ekranie sterujący podkreśleniem, intensywnością, odwróceniem video i migotaniem obrazu na ekranie monochromatycznym. Steruje migotaniem i kolorem znaku pierwszoplanowym/ tła na kolorowym wyświetlaczu. Poniższy obrazek pokazuje możliwe wartości atrybutów:



Monochrome Display Adapter Attribute Byte Format



Chcąc uzyskać odwrotność video, po prostu zmieniamy kolory pierwszoplanowy i tła. Zauważmy, że kolor pierwszoplanowy zero z kolorem tła siedem tworząc czarny znak na białym tle, standardowy kolor odwróconego video i takiej samej wartości atrybutu używanego na monitorze monochromatycznym.

Musimy być ostrożni ,kiedy wybieramy kolory pierwszoplanowy i tła dla tekstu na kolorowym monitorze. Pewne kombinacje są niemożliwe do odczytu (np. białe znaki na białym tle). Inne kolory razem

mogą spowodować, że trudno jest je odczytać, jeśli nie jest to niemożliwe (jak jasno zielone litery na zielonym tle?) Musimy być ostrożni wybierając kolory!

Znaki migające są doskonałe do przykuwania uwagi do jakiegoś ważnego tekstu na ekranie (jak ostrzeżenie). Jednakże łatwo jest przesadzić z migającym tekstem na ekranie. Nigdy nie powinniśmy mieć więcej niż jednego słowa lub zdania migającego na ekranie w danym czasie. Co więcej, nigdy nie powinniśmy zostawiać migających znaków na monitorze dłuższy czas. Po kilku sekundach, zamieńmy znaki migające na normalne aby uniknąć irytacji użytkownika naszego programu.

Zapamiętajmy, że możemy łatwo zmienić atrybuty różnych znaków na ekranie bez wpływania na bieżący tekst. Pamiętajmy, atrybut bajtu pojawia się pod adresem nieparzystym w przestrzeni pamięci dla wyświetlacza. Możemy łatwo wejść i zmienić te bajty pozostawiając same dane kodu znaku.

23.3 OPROGRAMOWANIE TRYBU TEKSTOWEGO

Możemy zapytać dlaczego ktoś chce zawracać sobie głowę pracowaniem bezpośrednio z wyświetlaczem odwzorowywanym w pamięci PC. Przecież DOS, BIOS i biblioteka Standardowa dostarczają dużo bardziej dogodnego sposobu wyświetlania tekstu na monitorze. Obsłużenie nowej lini (powrót karetki i przesuniecie o jedną linię) na końcu lini, lub , jeszcze gorzej, przesuwanie ekranu kiedy wyświetlacz jest pełny, to dużo racy. Pracy, którą automatycznie zajmują się wyżej wspomniane podprogramy. Pracy, którą musimy wykonać sami jeśli chcemy uzyskać dostęp bezpośredni do pamięci ekranu. Więc czemu zawracać sobie głowę?

Są dwa powody: wydajność i elastyczność. Podprogramy ekranowe BIOS są straszliwie wolne Możemy łatwo uzyskać zwiększenie wydajności od 10 do100razy poprzez zapisanie bezpośrednio do pamięci monitora. Dla typowego projekty informatycznego może nie być to ważne, zwłaszcza jeśli pracujemy na szybkiej maszynie, takiej jak Pentium 150 MHz. Z drugiej strony, jeśli projektujemy program, który wyświetla i usuwa kilka okien lub menu pop-up na ekranie, podprogramy BIOS nie zrobią tego.

Chociaż funkcja BIOS int 10h dostarcza dużego zbioru podprogramów video I/O, będzie mnóstwo funkcji jakich nie będziemy chcieli aby BIOS dostarczył. W takim przypadku przejście bezpośrednio do pamięci ekranowej będzie najlepszym rozwiązaniem tego problemu.

Inna trudnością z podprogramem BIOS jest to, że nie jest współużywalny. Nie możemy wywołać funkcji wyświetlacza BIOS z podprogramu obsługi przerwania, ani nie możemy swobodnie wywołać BIOS z jednocześnie wykonywanych procesów. Jednakże poprzez napisanie własnego podprogramu obsługi video, możemy łatwo stworzyć okno dla każdego współbieżnego wątku jaki wykonuje aplikacja. Wtedy każdy wątek może wywołać nasz podprogram wyświetlając swoje dane wyjściowe niezależnie od innych wątków wykonywanych w systemie.

Program AMAZE.ASM jest dobrym przykładem programu, który bezpośrednio uzyskuje dostęp do trybu tekstowego poprzez bezpośrednie przechowywanie danych w odwzorowywanej w pamięci monitora tablicy wyświetlania. Program ten uzyskuje dostęp bezpośredni do pamięci ponieważ jest bardziej dogodnie to zrobić (tablica wyświetlacza ekranowego odwzorowuje całkiem dobrze wewnętrzną tablicę labiryntu) .Proste gry video również dobrze odwzorowywują pamięć monitora.

Poniższy program dostarcza doskonałego przykładu aplikacji, która musi uzyskać dostęp bezpośredni do pamięci video. Program ten to TSR przechwytywania ekranu. Kiedy naciśniemy lewy shift a potem prawy, program ten skopiuje zawartość bieżącego ekranu do wewnętrznego bufora. Kiedy naciśniemy prawy shift po którym nastąpi lewy shift, program skopiuje bufor wewnętrzny na ekran .Pierwotnie program ten został napisany do przechwytywania ekranów CodeView dla celów laboratoryjnych towarzyszących tej książce. Istnieją komercyjne programy do przechwytywania ekranów (np. HiJak), które dobrze pracują ale są niekompatybilne z CodeView. Ten krótki TSR pozwala przechwycić ekran w CodeView, wyjść z CodeView, wprowadzić z powrotem ekran CodeView na ekran i stosować program taki jak HiJak do przechwytywania wyjścia.

```
; GRABSCRN.ASM
;
; Krótki TSR przechwytujący bieżący stan ekranu i wyświetlający go później.
;
; Zauważmy, ze ten kod nie aktualizuje int 2Fh (przerwanie równoczesnych procesów) ani nie można usunąć
; tego kodu z pamięci, z wyjątkiem przeładowania. Jeśli chcemy móc zrobić te dwie rzeczy (jak również
; sprawdzenie poprzedniej instalacji) zobacz rozdział o programach rezydentnych.
;
; cseg i EndResident muszą pojawić się przed segmentem biblioteki standardowej !

cseg                segemnt para public ‘code’
```

```
OldInt9         dword   ?
ScreenSave      byte    4096   dup  (?)
ceg             ends

; Oznaczamy segment znajdując koniec sekcji rezydentnej

EndResident     segment para public 'Resident'
EndResident     ends

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                ;list

RShiftScan      equ     36h
LShiftScan      equ     2ah

; Bity  dla klawiszy modyfikujących shift

RShfBit         equ     1
LShfBit         equ     2

KbdFlags        equ     < byte ptr ds.:[17h]>

byp             equ     <byte ptr>

; Adres segmentowy ekranu. Wartość ta jest tylko dla ekranu kolorowego. Zmień na B000h jeśli chcesz użyć
; tego programu z monitorem monochromatycznym

ScreenSeg       equ     0B800h

cseg            segment para public 'code'

; MyInt9-       ISR INT 9. Podprogram ten  odczytuje port klawiatury aby sprawdzić czy nadszedł kod
;               klawiaturowy klawisza shift. Jeśli jest ustawiony bit prawego klawisza w KbdFlags, nadchodzi
;               kod klawiaturowy lewego shift'a, chcemy skopiować dane z naszego wewnętrznego bufora do
;               pamięci ekranowej. Jeśli jest ustawiony bit lewego shift'a i naschodzi kod klawiaturowy
;               prawego shift'a   chcemy skopiować pamięć ekranową do naszej lokalnej tablicy. W innym
;               przypadku (żaden z nich nie nadszedł) zawsze przekazujemy sterowanie do oryginalnego
;               programu obsługi INT 9
;

MyInt9          proc    far
                push    ds.
                push    ax
                mov     ax, 40h
                mov     ds., ax
                in      al., 60h                        ;odczyt portu klawiatury
                cmp     al., RshiftScan                 ;wciœniêto prawy shift?
                je      DoRight
                cmp     al, LShiftScan                  ;lewy shift?
                Jne     QuitMyInt9

;jeśli jest to kod klawiaturowy lewego shift'a, zobaczmy czy prawy jest już wciśnięty

                test    KbdFlags, RShfBit
                je      QuitMyInt9                      ;skok jeśli nie

;Okay, prawy shift wciśnięty i wdzieliśmy, że lewy też, kopiujemy nasze lokalne dane do pamięci ekranowej:
```

```
                pushf
                push    es
                push    cx
                push    di
                push    si
                mov     cx, 2048
                mov     si, cs
                mov     ds., si
                lea     si, ScreenSave
                mov     di, ScreenSeg
                mov     es, di
                xor     di, di
                jmp     DoMove

;Okay, mamy już widzieliśmy kod klawiaturowy prawego shift'a, zobaczmy czy lewy shift jest wciśnięty. Jeśli
;tak, zapisujemy bieżącą daną ekranową do naszej lokalnej tablicy

DoRight:        test    KbdFlags, LShfBit
                je      QuitMyInt9

                pushf
                push    es
                push    cx
                push    di
                push    si
                mov     cx, 2048
                mov     ax, cs
                mov     es, ax
                lea     di, ScreenSave
                mov     si, ScreenSeg
                mov     ds, si
                xor     si, si

DoMove:         cld
            rep movsw
                pop     si
                pop     di
                pop     cx
                pop     es
                popf

QuitMuInt9:
                pop     ax
                pop     ds
                jmp     OldInt9
MyInt9          endp

Main            proc
                assume  ds:cseg

                mov     ax, cseg
                mov     ds, ax

                print
                byte    "Screen capture TSR", cr, lf
                byte    "Pressing left shift, then right shift, captures"
                byte    "the current screen.", cr, lf
                byte    "Pressing right shift , then left shift, dsiplays "
```

```
                byte    “the last aaptured scren.”, cr, lf
                byte    0

;Aktualizujemy wektor przerwań INT 9. Zauważmy ,że  powyższe instrukcje uczyniły  cseg
; bieżącym segmentem danych  więc możemy przechować starą wartość INT 9 bezpośrednio w
; zmiennej OldInt9

                cli                             ;wyłączamy przerwania
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[9+4]
                mov     word ptr OldInt9, ax
                mov     ax, es:[9*4+2]
                mov     word  ptr OldInt9+2, ax
                mov     es:[9*4], offset  MyInt9
                mov     es:[9*4+2], cs
                sti                             ; włązamyprzrwania

; jedyne co musimy zrobić to zakończyć i pozostawić pamięci

                print
                byte    „Installed”, cr, lf,0

                mov     ah, 62h                 ;pobranie wartości PSP programu
                int     21h

                mov     dx, EndResident         ;obliczenie rozmiaru programu
                sub     dx, bx
                mov     ax, 3100h
                int     21h
Main            endp
cseg            ends

sseg            segemnt para stack ‘stack’
stk             db      1024   dup {“stack”)
sseg            ends

zzzzzzseg       segment para public ‘zzzzzz’
LastBytes       db      16   dup (?)
zzzzzzseg       ends
                end     Main
```

23.4 PODSUMOWANIE

System video PC używa tablicy odwzorowania pamięci dla danych ekranowych. Jest to 80x25 kolumnowa organizacja tablicy słów. Każde słowo w tablicy odpowiada pojedynczemu znakowi na ekranie. Tablica zaczyna się pod adresem B000:0 dla wyświetlacza monochromatycznego i B800:0 dla wyświetlacza kolorowego.

*”Odwzorowywanie pamięci video”

Najmniej znaczący bajt jest kodem znaku PC/ASCII dla tej szczególnej pozycji ekranu, bardziej znaczący bajt zawiera atrybut dla tego . Atrybut wybiera migotanie, intensywność i kolor pierwszoplanowy / tła (na wyświetlaczu kolorowym)

*”Atrybut bajtu video”

Jest kilka powodów dla których możemy chcieć trudzić się uzyskaniem bezpośredniego dostępu do pamięci monitora. Szybkość i elastyczność są dwoma podstawowymi powodami dal których ludzie idą

bezpośrednio do pamięci ekranowej. Możemy stworzyć własną funkcję, której nie wspiera BIOS i robi to raz lub dwa razy szybciej niż BIOS poprzez zapisanie bezpośrednio do pamięci ekranowej

*”Oprogramowanie trybu tekstowego”