개선된 수동적 램상주 프로그램 툴

글 · 안 철 수(서울의대 생리학교실)

이번 달에는 지난 달에 소개한 수동적 램상주 프로그램의 기본 형식보다 좀 더 개선된 기법을 소개하고자 한다. 이 기법에서는 램상주 프로그램의 존재확인을 위하여 2Fh번 인터럽트를 사용하지 않고 보다 간단하게 11h번 인터럽트를 사용하였고, 램상주 프로그램내에서 그 전의 인터럽트 벡터가 0인지를 검사하는 루틴을 첨가하였다. 또한 램상주 프로그램을 제거하는 루틴도 첨가하여 보다 효율적으로 램상주 프로그램을 작성하는데 도움을 줄 것이다.

이번 달에는 지난 호에 예고한 바와 같이 '능동적 램상주 프로그램 제작기법'에 대해 설명할 예정이었으나 갑자기 컴퓨터 바이러스에 대한 원고를 쓰게 되었기 때문에 미처 필자의 컴퓨터 노트를 손 볼 겨를이 없었다. 처음에는 컴퓨터 바이러스에 대한 기사로 필자의 컴퓨터 노트를 채울 생각도 하였으나 램상주 프로그램에 대한 강좌를 계속하는 것이 기사의 연속성이라는 측면에서 더 좋을 것 같았고, 지난 호에 설명한 수동적 램상주 프로그램의 제작기법에서 개선할 점과 보충할 부분들이 눈에 띄어서 이 달에는 이들에 대한 설명으로 본 컬럼을 이끌어 가기로 했다. 애독자 설문지를 통하여 많은 격려를 해주신 독자 여러분들께 약속을 지키지 못해서 죄송하다는 말을 드리고 싶으며 사과의 뜻으로 다음 달에는 '능동적 램상주 프로그램의 제작기법'과 함께 필자가 최근에 구입한 터보 C 2.0에 대해 소개드릴까 한다.

본고에서는 먼저 지난 달에 소개한 것보다 개선된 수동적 램상주 프로그램의 기본 형식을 소개하고 아울러 램상주 프로그램을 작성할 때 유용한 도스 기능호출(function call) 기능을 몇 가지 알아보고자 한다.

개선된 수동적 램상주 프로그램의 기본 형식

먼저 지난 달의 기본 형식 중 잘못된 곳을 짚고 넘어가기로 한다. ISR2F 프로시져중 'jmp OldISR2F' 명령을 'jmp cs:OldISR2F'로 수정해야 한다. 프로그램 내에서 특별히 지정하지 않으면 세그먼트는 DS 레지스터의 값을 참조하게 되며, 인터럽트 핸들러로 제어가 넘어갈 때는 CS 레지스터와 IP 레지스터의 값만 바뀌어 잘못하면 엉뚱한 곳으로 점프해 버리기 때문이다.

또한 지난 달에 소개한 수동적 램상주 프로그램의 기본 형식은 몇 가지 제한점을 가지고 있다. 첫째, 2Fh번 인터럽트(multiplex interrupt)를 사용하였기 때문에 도스 2.xx에서 PRINT.COM이 기억장소에 상주하고 있을 때는 사용할 수 없으며, 2Fh번 인터럽트를 사용하기 위하여 여러 가지 사항들을 검사해야 하기 때문에 루틴이 길어지게 된다. 능동적 램상주 프로그램에서는 이러한 단점에 비하여 얻을 수 있는 장점이 훨씬 많기 때문에 2Fh번 인터럽트를 사용하는 것이 유리하지만 수동적 램상주 프로그램에서 단순히 램상주 프로그램의 존재확인만을 위해서는 간단한 다른 방법을 사용하는 것이 유리하다. 둘째, 램상주 프로그램은 일반적으로 원래의 인터럽트 수행루틴(interrupt handler)을 불러 정상적인 과정을 실행시킨 다음 램상주 프로그램의 역할을 수행한다. 하지만 다른 프로그램에서 사용되지 않던 인터럽트 벡터는 보통 0으로 채워져 있기 때문에 만약 램상주 프로그램에서 그 전에 사용되지 않던 인터럽트 핸들러를 호출하게 되면 시스템이 정지하게 되는 결과를 초래한다. 따라서 램상주 프로그램내에서 인터럽트 벡터를 검사하여 만약 사용되지 않던(0으로 채워져 있던) 것이면 원래의 인터럽트 수행루틴을 부르지 않게 하는 루틴이 필요하다. 세째, 지난 달의 기본 형식에는 램상주 프로그램을 제거하는 루틴이 없었다.

이러한 제한점들을 개선한 수동적 램상주 프로그램의 기본 형식을 〈리스트 1〉에 제시하였다. 지난 달의 형식에 비하여 개선된 점은 다음의 세 가지이다.

첫째, 개선된 형식에서는 램상주 프로그램의 존재확인을 위하여 2Fh번 인터럽트를 사용하지 않고 보다 간단

하게 11h번 인터럽트를 사용하였다. 원래 11h번 인터럽트는 BIOS 인터럽트로서 시스템이 가지고 있는 디스크 드라이브, 프린터, RS-232C, 게임 어댑터 등에 대한 정보를 제공해 주는 인터럽트이다(이 기능은 〈표 1〉에 제시하였다). 램상주 프로그램에서 11h번 인터럽트를 사용한 이유는 이것이 BIOS 인터럽트이기 때문에 어떤 도스 버전에서도 문제없이 동작하며, AX 레지스터외에는 다른 레지스터를 건드리지 않기 때문에 쉽게 인터럽트 수행루틴을 작성할 수 있기 때문이다. 램상주 프로그램에서 사용한 11h번 인터럽트 수행루틴을 〈표 2〉에 제시하였다. 이 인터럽트 수행루틴을 기억장소에 상주시킨 후 CX 레지스터에 특정값(HandlerID)을 담아서 11h번 인터럽트를 호출하면 인터럽트 수행루틴은 또 다른 특정값(ExistID)을 담아서 돌려준다. 따라서 램상주 프로그램을 실행시킬 때 11h번 인터럽트를 호출하여 특정값이 돌아오는지를 검사하면 램상주 프로그램이 이미 기억장소내에 존재하고 있는지 쉽게 알 수 있게 된다.

둘째, 램상주 프로그램내에서 그 전의 인터럽트 벡터가 0인지 검사하는 루틴을 첨가하였다.

세째, 램상주 프로그램을 제거하는 루틴을 첨가하였다. 이 형식에서는 편의상 램상주 프로그램을 제거할 때 인자(argument)로 '-'를 사용하도록 하였다. 즉, 프로그램 이름이 TSR이라면 다음과 같은 명령으로 램상주 프로그램을 제거할 수 있다.

```
A>tsr -
```

만약 이 명령이 마음에 들지 않으면 기본 형식을 조금만 고치면 될 것이다. 램상주 프로그램을 제거할 때는 지난 달에 설명한 21h번 인터럽트의 49h번 기능(free allocated memory function)을 사용하였다.

또한 프로그램의 길이를 줄이기 위하여 도스 1.xx의 처리 루틴도 없애버렸다. 구하기도 힘든 도스 1.xx를 구태여 사용하는 사람은 없을 것같기 때문이다. 하지만 만약 상업용 프로그램을 만든다면 이러한 사항도 염두에 두어야 할 것으로 생각된다.

램상주 프로그램 작성시의 주의점

지금부터 설명하고자 하는 내용은 수동적 램상주 프로그램 뿐만 아니라 능동적 램상주 프로그램에도 적용된다. 램상주 프로그램을 처음 작성하려고 했을 때 저지르기 쉬운 실수 중의 하나는 램상주 프로그램이 인터럽트를 통하여 호출되었을 때 모든 제어가 램상주 프로그램으로 넘어간다고 생각하는 것이다. 하지만 인터럽트가 걸릴 때 바뀌는 것은 CS 레지스터와 IP 레지스터 값뿐이며 다른 모든 것은 수동적 램상주 프로그램의 경우에는 호출하는 프로그램, 능동적 램상주 프로그램의 경우는 인터럽트가 걸리기 직전에 수행되던 프로그램이 가지고 있던 값을 그대로 가지게 된다. 즉, 다른 레지스터의 값과 스택의 위치 뿐만 아니라 PSP(program segment prefix), DTA(disk transfer area), 사용자 브레이크 처리 루틴 및 여러 가지 예외 처리 루틴의 위치까지도 그 전에 수행되던 프로그램에서 사용하던 값을 그대로 가지게 된다. 따라서 램상주 프로그램내에서 이들을 사용하려면 그 전의 위치를 지정해 놓은 다음 램상주 프로그램내에서 사용할 위치를 새로 지정해 주어야 한다.

먼저, 램상주 프로그램에서 데이타를 사용하려면 반드시 DS 레지스터의 값을 램상주 프로그램이 위치하는 세그먼트의 값으로 바꾸어 주어야 한다. 앞에서도 언급했지만 그렇게 하지 않으면 DS 레지스터의 값은 그 전의 프로그램이 가지고 있던 값을 가지고 있기 때문에 엉뚱한 값이 데이타로 사용된다. 그 전에 PUSH 명령으로 DS 레지스터의 값을 저장하여야 함은 물론이다.

DTA(disk transfer area)는 디스크 입출력시에 데이타를 주고 받는데 사용되는 영역으로, 보통 때는 PSP(program segment prefix)의 80h~FFh에 위치하며 도스내에서 내부변수(internal variable)를 사용하여 그 위치를 저장하고 있다. 만약 램상주 프로그램내에서 21h 인터럽트의 여러 기능중 디스크 입출력과 관계되는 기능들-예를 들어 11h, 12h, 14h, 15h, 21h, 22h, 27h, 28h, 4Eh, 4Fh 등-을 사용한다면 DTA의 위치를 램상주 프

〈표 1〉 11h번 인터럽트의 사용법

```
Int 11h (Equipment determination)

Call with: nothing
Returns  : bit of AX
           15,14   = number of printers
           13      = (not used)
           12      = game adapter
           11,10,9 = number of RS232 cards
           8       = (not used)
           7,6     = number of diskette drives
                     (00 = 1, 01 = 2, 10 = 3, 11 = 4)
           5,4     = video mode
                     (01 = 40X25 color, 10 = 80X25 color,
                      11 = 80X25 monochrome)
           3,2     = motherboard RAM
                     (00 = 16K, 01 = 32K, 10 = 48K, 11 = 64K)
           1       = (not used)
           0       = 1 if there are diskette drives attached
```

〈표 2〉 램상주 프로그램에서 사용하는 11h번 인터럽트 수행 루틴

```
ISR11    PROC  NEAR                ; INT 11h Handler
                                   ; Call with: CX = HandlerID
                                   ; Return   : CX = ExistID
                                   ;            AX = status

         cmp   cx, HandlerID
         jne   NoMatch             ; jump if not match
         mov   cx, ExistID

NoMatch: jmp   cs:OldISR11

ISR11    ENDP
```

로그램내로 바꾸어 주는 것이 좋다. 그 이유는 램상주 프로그램에서 아무런 조치도 취하지 않고 디스크 입출력을 수행한다면 프로그램내에서 사용중이던 데이타를 없애버릴 위험이 있기 때문이다. 다행히도 사용중인 DTA의 위치를 읽어들이고 새로 지정할 수 있는 기능이 마련되어 있다. 21h번 인터럽트의 2Fh번 기능(get DTA address)은 DTA의 위치를 알려주며, 21h번 인터럽트의 1Ah번 기능(set DTA address)은 DTA의 위치를 새로 지정해 줄 수 있게 한다. 이들 기능들의 사용법과 활용예를 〈표 3〉에 요약하였다.

PSP(program segment prefix) 역시 도스내의 내부 변수로 그 위치가 저장되어 있다. COM 화일의 경우에는 모든 세그먼트 레지스터가 PSP를 가리키고 있기 때문에 문제가 되지 않지만, EXE 화일의 경우에 PSP 내의 정보를 활용하고자 하는 경우에 PSP의 위치를 아는 것이 필수적이다. 21h번 인터럽트의 51h번 기능(get PSP address)은 PSP의 위치를 알려주며, 21h번 인터럽트의 50h번 기능(set PSP address)은 PSP의 위치를 새로 지정할 수 있게 하는 기능이다. 이들 인터럽트는 도스 매뉴얼에는 나와 있지 않은 기능들로, 'Reserved' 혹은 'Internal to DOS'라고 되어 있다. 하지만 많은 사람들이 도스를 분석하여 책에는 나와 있지 않은 이러한 기능들을 찾아내었다. 단, 도스 매뉴얼에는 언급이 안되어 있으므로 새로운 도소 버전에서의 호환성이 결여될 우려가 있으므로 조심해서 사용해야 한다. 하지만 필자의 생각으로는 이 기능을 사용하는 램상주 프로그램들이 많으므로 마이크로소프트사에서도 이를 무시할 수는 없으리라고 생각된다. 이들 기능들의 사용법과 활용예를 〈표 4〉에 요약하였다.

〈표 3〉 21h번 인터럽트의 2Fh번 기능과 1Ah번 기능

```
I. Int 21h Function 2Fh (get DTA address)

   Call with: AH    = 2Fh
   Returns  : ES:BX = segment:offset of DTA

   (예)
                mov  ah, 2fh                ; function number
                int  21h                    ; transfer to DOS
                mov  word ptr cur_dta, bx   ; offset
                mov  word ptr cur_dta[2], es ; segment
                .
                .
       cur_dta  dd   0                      ; double word variable
                                            ; to hold current DTA
                                            ;          address

II. Int 21h Function 1Ah (set DTA address)

   Call with: AH    = 1Ah
              DS:DX = segment:offset of DTA
   Returns  : nothing

   (예)
                mov  ah, 1ah            ; function number
                mov  dx, seg buffer     ; address of DTA
                mov  ds, dx
                mov  dx, offset buffer
                int  21h                ; transfer to DOS
                .
                .
       buffer   db   128 dup (?)
```

〈표 4〉 21h번 인터럽트의 51h번 기능과 50h번 기능

```
I. Int 21h Function 51h (get PSP address)

   Call with: AH = 51h
   Returns  : BX = PSP segment

   (예)
                mov  ah, 51h      ; function number
                int  21h          ; transfer to DOS
                mov  OldPSP, bx   ; save PSP address
                .
                .
       OldPSP   dw   0

II. Int 21h Function 50h (set PSP address)

   Call with: AH = 50h
              BX = PSP segment
   Returns  : nothing

   (예)
                mov  ah, 50h      ; function number
                mov  bx, es       ; PSP segment
                int  21h          ; transfer to DOS
```

마지막으로 능동적 램상주 프로그램에서는 고려하지 않아도 되지만 수동적 램상주 프로그램에서는 중요한 것이 있다. 바로 호출 프로그램과 램상주 프로그램 간의 데이타 교환이다. 이것은 특히 런타임 라이브러리에서 중요한 것으로, 어떤 데이타를 런타임 라이브러리에게 넘겨주고 런타임 라이브러리에서 이 데이타를 처리한 후 다시 호출 프로그램에 처리한 결과를 되돌려 주는 형식이 된다. 간단한 데이타의 경우는 레지스터를 이용하여 데이타를 교환하는 형식을 사용하는 것이 간편하고 속도는 빠르지만, 복잡한 형식의 데이타나 많은 양의 데이타를 교환하지 못한다. 따라서 런타임 라이브러리에서는 스택을 이용하여 데이타를 교환하는 방식이 일반적으로 사용되고 있다.

수동적 램상주 프로그램의 활용

수동적 램상주 프로그램의 사용 영역은 크게 세 가지 분야로 나눌 수 있다. 첫째는 지난 달에 언급한 바 있는 런타임 라이브러리이다. 런타임 라이브러리는 호출하는 프로그램에서 시스템에 대한 모든 책임을 지기 때문에 수동적 램상주 프로그램의 형식을 사용해도 아무런 문제가 없다. 두번째 영역은 기존의 도스 혹은 BIOS의 인터럽트 처리루틴을 개선하는 경우이다. 이 경우 역시 정상적인 인터럽트 호출과정을 밟기 때문에 수동적 램상주 프로그램의 형태로 많이 사용된다. 프린터로 가는 인터럽트를 가로채서 여러 가지 역할을 수행하는 유틸리티들도 여기에 속한다. 세번째 영역은 주변장치 구동 프로그램(device driver)이다. 주변장치 구동 프로그램이란 도스에게 특정 주변장치를 어떻게 제어하는가를 알려주는 프로그램이다. 도스 디스켓내의 ANSI.SYS는 키보드와

모니터를 도스에 의해 제공되는 기본적인 제어보다도 더 세밀하게 제어할 수 있는 능력을 부여하는 대표적인 주변장치 구동 프로그램의 예이다. 주변장치 구동 프로그램에 관해서는 램상주 프로그램에 대한 강좌가 마무리되는 대로 다룰 예정이다.

어셈블리어 도사들을 위한 문제

램상주 프로그램은 원리를 잘 이해하고 주의깊게 설계한다면 다른 프로그램에 미치는 영향을 최소한으로 줄이면서 하고 싶은 일을 성공적으로 수행시킬 수 있다. 원래 램상주 프로그램은 마이크로소프트사에서 공식적으로 지원하지 않았던 것으로, 여러 프로그래머들의 연구에 의하여 비공식적으로 정립된 기법이다. 하지만 점차로 램상주 프로그램이 많아짐에 따라 마이크로소프트사에서도 이를 외면할 수 없는 처지에 이르렀다. 공식적으로는 새로운 버전의 도스에서의 호환성을 책임질 수 없다고 말하고 있으나, 만약 'SideKick'과 같은 유명 프로그램이 작동하지 않는 도스를 만든다면 아무도 이를 사용하려 하지 않을 것이다. 비공식적인 기법을 사용하는 유명 프로그램들을 작동시키기 위해 OS/2의 개발이 지연되었던 것을 생각하면 표준이란 특정 회사에서 정하는 것이 아니라 그에 따르지 않더라도 널리 보급된 것이 표준을 이끌어 나가는 것 같다.

마지막으로 지금까지의 강좌가 너무 쉽다고 생각할지도 모를 어셈블리어 도사들(?)에게 간단한 문제를 하나 제시하겠다. 문제의 프로그램은 필자가 본지 88년 9월호의 'MS-DOS 3.3의 활용'에서 소개했던 프로그램으로, 〈NumLock〉 키를 자동으로 오프 상태로 만들어 주는 15바이트 크기의 다음과 같은 간단한 프로그램이다.

```
xor ax, ax
mov ds, ax
mov al, [417h]
and al, 0DFh
mov [417h], al
int 20h
```

문제는 '이 프로그램을 최소한으로 줄이면 몇 바이트까지 줄일 수 있을까요?'이다. 해답은 다음 달에 '능동적 램상주 프로그램의 제작기법'에서 소개하도록 하겠다.

〈리스트 1〉 개선된 수동적 램상주 프로그램의 기본 형식

```
CR          EQU  0Dh
LF          EQU  0Ah

IntNo       EQU  ?          ; 프로그래머가 정해야함
HandlerID   EQU  ?          ;       "
ExistID     EQU  ?          ;       "

TSR     SEGMENT

        ASSUME cs: TSR, ds: TSR

;-------------------------------------------------------

RESIDENT PROC NEAR

        ORG  100h
 Entry: jmp  INSTALL

        OldISR11 DD  ?          ; original INT 11h handler
        OldInt   DD  ?          ; original INT handler

        ;
        ; 데이타를 이 곳에 둔다.
        ;

 Main:  ; TSR 내에서 사용되는 register를 PUSH

        mov  ax, cs             ; set up data segment
        mov  ds, ax

        cmp  word ptr OldInt, 0 ; check original address
        jnz  Skip
        cmp  word ptr OldInt[2], 0
        jnz  Skip

        pushf                   ; call original INT handler
        call OldInt

 Skip:
        ;
        ; 램상주 프로그램을 이 곳에 둔다.
        ;

        ; register를 POP 명령으로 복구시킨다.

        iret

RESIDENT ENDP

;-------------------------------------------------------

ISR11   PROC  NEAR              ; INT 11h Handler
                                ; Call with: CX = HandlerID
                                ; Return   : CX = ExistID
                                ;            AX = status









        cmp  cx, HandlerID
        jne  NoMatch            ; jump if not match
        mov  cx, ExistID

 NoMatch:jmp  cs:OldISR11

ISR11   ENDP

;-------------------------------------------------------

INSTALL  PROC  NEAR

        ; Program already installed ?

        mov  cx, HandlerID      ; CX = Handler ID value
        int  11h                ; equip status interrupt
        cmp  cx, ExistID
        je   Exist              ; already installed

        ; Set Interrupt Vector

        mov  ah, 35h            ; get 11h interrupt veotor
        mov  al, 11h
        int  21h
        mov  word ptr OldISR11, bx
        mov  word ptr OldISR11[2], es

        mov  ah, 25h            ; set 11h interrupt veotor
        mov  al, 11h
        mov  dx, offset ISR11
        int  21h

        mov  ah, 35h            ; get interrpt veotor
        mov  al, IntNo
        int  21h
        mov  word ptr OldInt, bx
        mov  word ptr OldInt[2], es

        mov  ah, 25h            ; set interrupt vector
        mov  al, IntNo
        mov  dx, offset Main
        int  21h

        ; Free the environment

        mov  es, ds:[2Ch]       ; ES = segment of environment
        mov  ah, 49h            ; free memory block
        int  21h

        ; Terminate and Stay Resident

        mov  dx, offset INSTALL
        add  dx, 15
        mov  cl, 4
        shr  dx, cl
        mov  ah, 31h
        int  21h

        ; Argument check




Exist:  mov  al, ds:[80h]       ; argument exist ?
        cmp  al, 0
        je   Exit
        mov  al, ds:[82h]       ; argument is '-' ?
        cmp  al, '-'
        jne  Exit

        ; Get the segment of TSR

        mov  ah, 35h            ; segment of TSR -> ES
        mov  al, 11h
        int  21h

        ; Restore the changed INT vectors

        push ds                 ; save DS

        mov  ah, 25h            ; reset 11h interrupt vector
        mov  al, 11h
        mov  dx, es:word ptr OldISR11
        mov  ds, es:word ptr OldISR11[2]
        int  21h

        mov  ah, 25h            ; reset interrupt vector
        mov  al, IntNo
        mov  dx, es:word ptr OldInt
        mov  ds, es:word ptr OldInt[2]
        int  21h

        ; Free the allocated memory

        mov  ah, 49h            ; free memory block of TSR
        int  21h

        ; Print messages

        pop  ds                 ; restore DS
        mov  dx, offset Mess1   ; print 'Deinstalled'
        mov  ah, 9
        int  21h

        int  20h                ; exit

 Exit:  mov  dx, offset Mess2   ; print 'Already installed'
        mov  ah, 9
        int  21h

        mov  al, 1              ; exit with return code
        mov  ah, 4Ch
        int  21h

 Mess1  DB   CR, LF, 'Deinstalled', CR, LF, '$'
 Mess2  DB   CR, LF, 'Already Installed', CR, LF, '$'

INSTALL  ENDP

;-------------------------------------------------------

TSR     ENDS
        END  Entry
```