수동적 램상주 프로그램 제작기법

글 · 안 철 수(서울의대 생리학교실)

램상주 프로그램 작성 기법은 수동과 능동적 방법이 있다. 이중 수동적 방법은 수행중인 프로그램이 램상주 프로그램에게 제어를 넘길 때 실행되는 프로그램을 말한다. 이 방법은 램상주 프로그램을 부르는 프로그램에서 시스템에 관한 모든 처리를 해주기 때문에 작성하기 쉽다는 장점이 있다. 이에 반해 능동적 방법은 사이드킥과 같은 프로그램을 말하는데 당장 그 프로그램의 수행을 중지시키고 자기 자신이 수행되는 프로그램을 말한다. 복잡한 능동적 방법에 대한 설명은 신년으로 미루고 이번 호에서는 수동적 램상주 프로그램의 제작 기법에 대해서만 다루고자 한다.

램상주 프로그램이란?

램상주 프로그램이란 기억장소내에서 존재하고 있다가 수행중인 프로그램으로부터 제어권을 넘겨받아 실행되는 프로그램을 뜻하며, 일명 TSR(Terminate and Stay Resident) 프로그램이라고도 부른다. 램상주 프로그램 기법은 일반적인 유틸리티를 비롯하여 에디터, 스펠링 체커(spelling checker), 키보드 매크로 프로그램과 LAN 매니저 등의 디바이스 드라이버(device driver)의 작성에 이르기까지 매우 다양한 영역에서 활용되고 있으며, 많은 프로그램들이 공유하는 데이타도 이 기법을 활용하여 기억장소내에 상주시켜 사용할 수 있도록 하고 있다.

하지만 램상주 프로그램은 그 유용성에 비하여 작성하기가 비교적 어렵다. 많은 사람들이 램상주 프로그램을 작성하려고 하다가 수많은 시스템 다운 끝에 포기하는 경우도 램상주 프로그래밍의 어려움 때문으로 본다. 램상주 프로그램 작성의 애로점은 크게 다음의 두 가지로 생각된다. 첫째, 램상주 프로그램을 작성할 때는 정상적인 프로그램의 수행을 가로챔으로써 야기될 수 있는 여러가지 부작용을 고려해야만 하는데, 그 중 한 가지라도 빼먹고 프로그래밍을 한다면 시스템이 정지해 버릴 수 있다는 것이다. 둘째로 램상주 프로그램은 일반적인 매뉴얼에는 나와 있지 않은 잘 밝혀지지 않은 시스템 부분을 사용하기 때문에 이에 관한 정보가 부족하다는 점이다.

사실 램상주 프로그램의 제작기법에 대해서는 이미 87년도에 본지의 특집 기사로 다룬 적이 있다. 바로 87년 2월호에 실린 'IBM PC용 램상주 프로그램의 제작'기사인데, 그 기사에서는 세그먼트와 인터럽트에 관한 기초적인 지식과 그 당시 정보가 부족한 램상주 프로그램의 제작기법에 관한 정보를 제공해 주어서 필자를 비롯한 많은 독자들에게 크게 도움을 준 기사라고 기억된다. 필자가 이미 다루어졌던 주제를 다시 게재하고자 하는 이유는 두 가지이다. 첫째 이유로 'IBM PC용 램상주 프로그램의 제작'은 수동적 램상주 프로그램(passive memory-resident program)의 작성에만 중점을 둔 기사였기 때문에 우리가 일상적으로 많이 사용하는 능동적 램상주 프로그램(active memory-resident program)을 제작하는 데는 미흡한 점이 있었기 때문이다. 수동적 램상주 프로그램이란 수행중인 프로그램이 램상주 프로그램에게 제어를 넘길 때 실행되는 프로그램으로서, 램상주 프로그램을 부르는 프로그램에서 시스템에 관한 모든 처리를 해주기 때문에 작성하기가 비교적 쉽다. 혼자서 동작하는 시계(clock) 프로그램도 이 부류에 속한다. 능동적 램상주 프로그램이란 사이드킥과 같이 어떤 프로그램의 수행도중 예약키(hot key)를 누르기만 하면 당장 그 프로그램의 수행을 중지시키고 자기 자신이 수행되는 것을 말한다. 우리가 일상적으로 사용하는 거의 모든 램상주 프로그램이 이 부류에 속한다. 이 경우 프로그램의 수행을 가로챔으로써 생길 수 있는 여러가지 부작용에 관한

처리를 램상주 프로그램에서 맡게 되기 때문에 수동적 램상주 프로그램에 비하여 작성하기가 훨씬 어렵게 된다.

두번째 이유로 'IBM PC용 램상주 프로그램의 제작'에서는 27h번 인터럽트와 같은 오래된 기법을 사용하였기 때문에 COM 화일 형식에 국한된 램상주 프로그램을 작성할 수밖에 없었다(숫자뒤에 h가 붙은 것은 16진수를 뜻한다). 따라서 EXE 화일 형식을 사용해서 얻을 수 있는 여러가지 장점을 놓치게 된다.

이번 호에서는 먼저 'IBM PC용 램상주 프로그램의 제작'에 게재된 방법에 비하여 좀 더 개선된 수동적 램상주 프로그램의 작성기법에 대하여 생각해 보고자 하며, 신년 호에서는 능동적 램상주 프로그램을 작성할 때 알아 두어야 할 여러가지 사항에 대해서 설명하고자 한다. 또한 어셈블리어를 몰라서 안타까워 할 독자들을 위하여 터보C, 터보 파스칼 3.0과 4.0을 사용하여 램상주 프로그램을 작성하는 기법을 계속 소개하고자 하니, 어셈블리어를 모르더라도 꾹꾹 참고(?) 읽어주시면 계속되는 기사를 이해하는데 큰 도움이 될 것으로 생각한다.

수동적 램상주 프로그램 제작기법 개요

수동적 램상주 프로그램(passive memory-resident program)은 수행중인 프로그램이 제어를 넘겨줄 때 비로소 실행되는 프로그램이다. 이 램상주를 부르는 프로그램은 수행도중 갑자기 인터럽트가 걸리는 것이 아니기 때문에 램상주 프로그램을 부르기 전에 하드웨어와 BIOS, 그리고 도스가 안전한 상태에 있다는 것을 확인해야 할 책임이 있다. 왜냐하면 하드웨어 인터럽트나 특정 BIOS, 그리고 도스 인터럽트는 수행되는 도중에 같은 인터럽트를 사용할 수 없기 때문이다(이것을 'non-reentrant'하다고 말한다). 따라서 수동적 램상주 프로그램은 시스템의 상태에 신경쓸 필요가 전혀 없고, 비교적 간단하게 프로그래밍을 할 수 있으며 프로그램내에서 자유로이 BIOS 및 도스 기능호출(function call)을 이용할 수 있다.

수동적 램상주 프로그램의 예로 런타임 라이브러리(run-time library)를 들 수 있다. 런타임 라이브러리란 프로그램의 수행중 사용되는 루틴들을 모아 놓은 것이다. 넓은 의미의 런타임 라이브러리는 링크시 프로그램 자체에 포함되는 링크 라이브러리(link library)를 포함하는 개념이지만 좁은 의미로는 프로그램내에 포함되지 않고 독립적으로 존재하면서 프로그램 수행시 기억장소에 로드되어 그 역할을 수행하는 것을 말한다. 런타임 라이브러리는 주 프로그램에서 이것을 로드하고 부르는 루틴을 따로 제작해야 하는 단점이 있는 반면, 많은 프로그램에서 사용하는 공통된 루틴이 있을 경우에 유용하며 라이브러리를 바꿀 때 다시 프로그램과 링크시키지 않아도 되는 장점이 있다. 또한 저장 영역의 크기도 줄어들며, 프로그램을 로드하는 시간을 절약할 수 있다. 퀵베이직 4.0의 BRUN40.EXE가 이 런타임 라이브러리의 좋은 예이다.

〈그림 1〉 램상주 프로그램의 기본 형식(87년 2월호의 기사)

```
INTERRUPTS SEGMENT AT 0H
           ORG (INTERRUPT #) X 4
           OLD_INT_VECTOR LABEL DWORD
INTERRUPTS ENDS

CODE_SEG   SEGMENT
           ASSUME CS:CODE_SEG
           ORG  100H
 ENTRY:    JMP  INSTALL
           .
           .
           OLD_INTERRUPT DD ?
           ; 데이타들을 이곳에 둔다.
           .
           .
RESIDENT   PROC  NEAR
           .
           .
           PUSHF
           CALL OLD_INTERRUPT
           .
           .
           ; 램상주 프로그램을 이곳에 둔다.
           .
           .
           IRET
RESIDENT   ENDP

INSTALL    PROC  NEAR
           .
           .
           ; 초기화 루틴을 이곳에 둔다.
           .
           .
           ASSUME DS:INTERRUPTS
           MOV  AX, INTERRUPTS
           MOV  DS, AX
           MOV  AX, OLD_INT_VECTOR
           MOV  OLD_INTERRUPT, AX
           MOV  AX, OLD_INT_VECTOR[2]
           MOV  OLD_INTERRUPT[2], AX
           .
           .
           MOV  OLD_INT_VECTOR, OFFSET RESIDENT
           MOV  OLD_INT_VECTOR[2], CS
           .
           .
           MOV  DX, OFFSET INSTALL
           INT  27H
INSTALL    ENDP

CODE_SEG   ENDS
           END  ENTRY
```

인터럽트의 개념 및 수동적 램상주 프로그램의 형식은 'IBM PC용 램상주 프로그램의 제작' 기사에서 잘 설명된 바 있다. 그 기사에서 설명된 램상주 프로그램의 기본 형식은 〈그림 1〉과 같다. 〈그림 1〉을 잘 살펴보면 수동적 램상주 프로그램이 일반 프로그램과 다른 점을 금방 알아 차릴 수 있다. 처음 프로그램을 수행시키면 INSTALL 프로시져가 수행된다. 여기서는 특정한 인터럽트 벡터가 RESIDENT 프로시져를 가리키게 바꿔준 후 27h번 인터럽트를 사용하여 기억장소내에 상주하는 것으로 프로그램의 수행을 끝마치게 된다.

그 후 특정한 인터럽트가 호출되면 이 램상주 프로그램이 원래의 인터럽트 수행루틴대신 실행된다. 하지만 대부분의 램상주 프로그램은 먼저 원래의 인터럽트 수행루틴을 불러 정상적인 과정을 실행시킨 다음 램상주 프로그램의 역할을 수행한다.

본고에서는 먼저 이 기본 형식에서 사용된 인터럽트 벡터를 바꾸는 방법과 27h 인터럽트에 대한 개선점을 설명한 후, 램상주 프로그램이 이미 기억장소에 존재하고 있는지를 알아보는 방법과 기억장소내에서 램상주 프로그램을 제거하는 방법에 대하여 설명하기로 한다. 그 후 개선된 수동적 램상주 프로그램의 형식을 제시해 보고자 한다.

인터럽트 벡터를 바꾸는 법

인터럽트 벡터를 바꾸는 방법은 크게 두 가지가 있다. 첫번째 방법은 〈그림 1〉에서와 같이 직접 그 인터럽트 벡터가 있는 번지를 읽어서 바꾸는 것이며, 두번째 방법은 도스의 21h번 인터럽트를 사용하는 방법이다. 21h번 인터럽트의 35h번 기능(Get Interrupt Vector function)은 인터럽트 벡터의 값을 읽어들이며 25h번 기능(Set Interrupt Vector function)은 인터럽트 벡터를 원하는 주소로 바꿔주는 기능이다. 이들 기능의 사용법 및 사용예는 〈표 1〉에 요약하였다.

인터럽트 벡터를 바꿀 때 도스의 기능호출을 사용하는 것은 인터럽트 벡터를 직접 바꾸는 방법에 비하여 속도가 느리다는 단점이 있는 반면, 사용하기 편리하다는 장점이 있으며

〈표 1〉 21h번 인터럽트의 25h와 35h번 기능

```
I. Int 21h Function 25h (Set Interrupt Vector)

   Call with: AH    = 25h
              AL    = interrupt number
              DS:DX = segment:offset of interrupt handler
   Returns  : nothing

   (예)
              mov  ah, 25h       ; function number
              mov  al, 0         ; interrupt number
              mov  dx, seg ISR   ; address of interrupt
              mov  ds, dx
              mov  dx, offset ISR
              int 21h
              .
              .
        ISR:  .
              .

II. Int 21h Function 35h (Get Interrupt Vector)

    Call with: AH    = 35h
               AL    = interrupt number
    Returns  : ES:BX = segment:offset of interrupt handler

    (예)
               mov  ah, 35h       ; function number
               mov  al, 0         ; interrupt type
               int  21h           ; transfer to DOS
               mov  INT0SEG, es   ; segment of handler
               mov  INT0OFF, bx   ; offset of handler
               .
               .
      INT0SEG  dw   0
      INT0OFF  dw   0
```

마이크로소프트사에서도 장래에 나올 MS-DOS에서의 호환성을 약속하고 있다. 따라서 많은 램상주 프로그램들이 도스의 기능호출을 사용하고 있으며 필자 또한 이 방법의 사용을 적극 권장하고 싶다.

램상주 프로그램의 종료방법

프로그램 종료시 프로그램을 램상주로 만드는 방법도 크게 두 가지가 있다. 첫번째는 〈그림 1〉에서 처럼 27h번 인터럽트(Terminate and Stay Resident interrupt)를 사용하는 방법이다. 또 다른 방법으로는 21h번 인터럽트 중 31h번 기능(Terminate and Stay Resident function)을 사용하는 것이다. 이들 사용법 및 사용예를 요약해 보면 〈표 2〉와 같다.

일반적으로 27h번 인터럽트를 사용하는 것은 유연성이 적은 좋지 않은 방법이다. 그 이유는 27h번 인터럽트는 상주할 기억장소의 크기를 바이트로 표시하기 때문에 최대 크기가

〈표 2〉 27h번 인터럽트와 21h번 인터럽트의 31h 기능

```
I. Int 27h (Terminate and Stay Resident interrupt)

   Call with: DX = offset of last byte plus 1 of program
              CS = segment of PSP(program segment prefix)
   Returns  : nothing

   (예)
    START:    .
              .
              mov  dx, offset POINT
              int  27h
              .
              .
    POINT     LABEL  BYTE
              END    START

II. Int 21h Function 31h (Terminate and Stay Resident)

    Call with: AH = 31h
               AL = return code
               DX = memory size to reserve (in paragraphs)
    Returns  : nothing

    (예 1) COM type
       Program SEGMENT
               ORG  0
       SegOrg  EQU  $
               ORG  100h
       Start:  .
               .
               mov  dx, offset LastByte
               sub  dx, offset SegOrg
               add  dx, 15
               mov  cl, 4
               shr  dx, cl
               mov  ah, 31h          ; keep process
               int  21h
               .
               .
       LastByte:
       Program ENDS
               END  Start

    (예 2) EXE type
               .
               .
               mov  ax, es           ; get PSP address
               mov  dx, seg EndAddr  ; get last segment address
               sub  dx, ax           ; difference is program size
               mov  ah, 31h          ; keep process
               int  21h
               .
               .
       Program ENDS
       EndAddr SEGMENT
       EndAddr ENDS

               END  Start
```

64k 바이트로 제한되며, 기억장소의 시작 세그먼트 주소(segment address)가 CS레지스터에 있어야만 하기 때문에 사실상 EXE 형식의 화일에는 사용할 수 없다는 것이다. 또한 31h번 기능과는 달리 프로그램의 종료시 넘겨주는 반환값(return code)을 사용할 수 없는 단점이 있다. 27h번 인터럽트를 사용할 때 장점이라고 할 수 있는 것으로는 도스 1.xx와의 호환성이 유지된다는 점과 기억장소의 크기를 바이트로 표시하므로 별다른 계산을 거치지 않고서도 쉽게 그 크기를 구할 수 있다는 것 뿐이다. 반면에 31h번 기능은 프로그램의 크기를 패러그래프(paragraph, 1 패러그래프=16바이트)로 표시하므로 필요한 만큼 프로그램의 크기를 증가시킬 수 있으며 반환값을 사용할 수 있는 등의 장점이 있다. 도스 1.xx 사용자가 거의 없으리라는 점을 생각해 볼 때, 필자는 31h번 기능의 사용을 권하고 싶으며, 마이크로소프트사에서도 새로운 램상주 프로그램을 작성할 때 뿐만 아니라 기존의 램상주 프로그램을 개정할 때도 21h번 인터럽트의 31h번 기능의 사용을 권장하고 있다.

여기서 한 가지 더 언급하고 넘어갈 것은 어떤 기능을 수행하느냐에 관계없이 기억장소에 상주하는 것은 프로그램 자체만이 아니라는 점이다. 이 기능들을 수행하면 상주할 프로그램과 프로그램이 가지는 도스 정보영역(DOS environment) 및 프로그램을 사용하기 위해 할당된 기억장소의 영역이 같이 상주하게 된다. 따라서 나중에 램상주 프로그램을 기억장소내에서 제거하기 위해서는 이러한 영역들을 모두 제거해야만 한다.

CALL명령과 INT명령

어셈블리어를 아는 독자중에 CALL명령과 INT명령의 차이점을 모르는 사람은 거의 없으리라 생각된다. CALL명령이 실행되면 특정루틴의 시작번지로 바로 제어가 넘어가서 그 루틴이 실행된 후 RET 명령에 의해서 주 프로그램으로 돌아와 CALL 명령의 바로 다음 명령이 수행된다. 이에 반해, INT 명령을 실행시키면 기억장소의 최상위에 있는 (0000 : 0000~0000 : 0400) 인터럽트 벡터 테이블에서 그 주소를 찾아 그 루틴(인터럽트 핸들러라고 한다)을 수행한 후 IRET 명령에 의하여 주 프로그램으로 돌아온다.

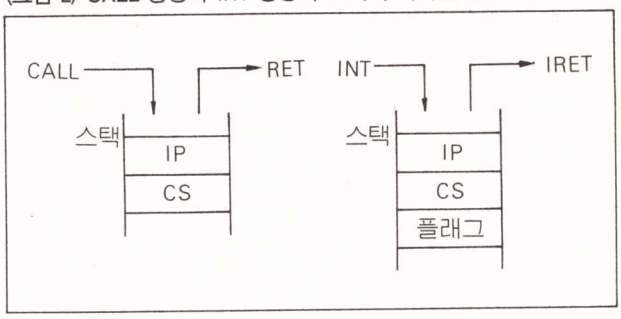
하지만 그 명령들이 스택에 미치는 작용에 대해서는 모르는 사람이 의외로 많다. 스택에 대한 이들 명령어의 효과를 〈그림 2〉에 요약하였다. CALL 명령은 현재의 CS 레지스터와 IP 레지스터의 값을 순서대로 스택에 저장시킨 후 호출한 프로그램으로 제어를 옮긴다. RET 명령은 스택에서 IP 레지스터와 CS 레지스터의 값을 가져와 원래의 프로그램으로 복귀하게 된다. 반면에, INT 명령은 먼저 플래그(flag)를 스택에 저장시킨 후 CS 레지스터와 IP 레지스터의 값을 저장한다. 또한 IRET 명령은 스택에서 IP 레지스터와 CS 레지스터의 값을 가져온 후, 플래그의 값도 가져온다. 따라서 인터럽트를 가로 챈 램상주 프로그램에서 원래의 인터럽트를 부를 때는 먼저 PUSHF 명령을 사용하여 현재의 플래그 값을 스택에 저장시킨 후 CALL 명령으로 원래의 인터럽트 루틴을 부르게 된다. 만약 그렇게 하지 않으면 원래의 인터럽트 루틴은 IRET 명령으로 끝나기 때문에 스택에 존재하는 엉뚱한 값이 플래그로 들어가게 되며, 운이 나쁘면 시스템이 정지해 버리는 결과를 초래할 수 있다.

램상주 프로그램의 존재확인 방법

램상주 과정에서 만약 기억장소내에 램상주 프로그램이 존재하는지를 확인하지 않는다면 여러가지 문제점이 발생하게 된다. 그 문제점이란 여러 개의 같은 프로그램이 기억장소내에 상주하게 되면 사용가능한 기억장소의 영역이 줄어 들게 되며 시스템의 수행속도 또한 저하될 가능성이 있기 때문이다. 또한 운이 나쁘면 HGKEY.COM의 경우와 같이 시스템이 정지해 버릴 수도 있다. 따라서 램상주 프로그램을 작성하는데 있어서 다른 프로그램이 기억장소내에 이미 존재하고 있는지를 확인하는 루틴은 필수적이다.

램상주 프로그램이 기억장소내에 이미 존재하는지 알아보기 위한 방법은 여러가지가 있을 수 있으나 그 중 대표적인 방법은 세 가지 정도이다. 첫번째 방법은 바꿀 인터럽트 벡터를 읽어서 인터럽트 핸들러와 램상주 프로그램을 MOVS 명령으로 비교해 보는 것이다. 만약 인터럽트 핸들러와 램상주 프로그램이 동일하다면 이미 프로그램이 기억장소에 상주한다는 사실을 알 수 있다. 이 방법은 프로그래밍은 간단하지만 속도가 느리며 특히 램상주 프로그램들의 초기 루틴들이 같

〈그림 2〉 CALL 명령과 INT 명령이 스택에 미치는 효과



〈그림 3〉 램상주 프로그램의 존재확인을 위한 프로그램의 구성

```
ENTRY:   JMP  START
         DB   '<routine name>'     ; Data area
         .
         .
START:   ; Beginning of the Code
         .
         .
```

을 경우에는 효과가 없을 가능성이 크다. 하지만 모든 램상주 프로그램들을 〈그림 3〉과 같이 프로그램의 초기에 프로그램 고유의 이름 또는 특정 순서의 문자를 위치시킨다면 이 방법을 사용하는 것이 쉽고 빠를 것이다.

두번째 방법은 램상주 프로그램이 인스톨될 때 특정 인터럽트 벡터에 인터럽트 핸들러의 주소대신 특정한 코드를 위치시키는 방법이다. 따라서 이 곳을 검사하면 램상주 프로그램의 존재여부를 쉽게 알 수 있게 된다. 하지만 만약 다른 램상주 프로그램에서 이 인터럽트 벡터를 사용한다면 원하는대로 작동하지 않을 가능성이 있다.

세번째 방법은 램상주 프로그램이 인스톨되는 과정에서 특정한 인터럽트 핸들러를 변경하여 이 인터럽트가 호출될 때

〈그림 4〉 램상주 프로그램에서 사용할 2Fh번 인터럽트 핸들러

```
ISR2F   PROC  NEAR              ; INT 2Fh Handler
                                ;     Call with: AH = handler ID
                                ;                AL = function No
                                ;     Return for function 0 :
                                ;                AL = 0FFh
        CMP   AH, MultiplexID
        JE    ID_OK             ; ID가 맞으면 계속 검사
        JMP   OldISR2F          ; ID가 틀리면 그 전의 핸들러로
ID_OK:  TEST  AL, AL            ; function이 0인가 ?
        JNZ   Exit
        MOV   AL, 0FFh          ; 0이면 AL에 FFh를 돌려줌
Exit:   IRET
ISR2F   ENDP
```

〈표 3〉 2Fh번 인터럽트의 기능

```
Int 2Fh (Multiplex interrupt)

  Call with: AH = identification byte
             AL = function number
  Returns  : AL = signal byte

  (예)  function = 0 일 때 FFh를 돌려준다면
             mov  ah, MultiplexID    ; ID byte
             mov  al, 0              ; function number
             int  2Fh
             cmp  al, 0FFh           ; signal byte
             je   AlreadyInstalled
```

〈표 4〉 21h번 인터럽트의 49h기능

```
Int 21h Function 49h (Free Allocated Memory)

  Call with: AH = 49h
             ES = segment of block to be released
  Returns  : if function successful
                  Carry flag = clear
             if function failed
                  Carry flag = set
                  AX         = error code
                               7 (if memory control blocks
                                                  destroyed)
                               9 (if incorrect segment in ES)
  (예)
             mov  ah, 49h        ; function number
             mov  es, BUFF       ; segment of memory block
             int  21h
             jc   FAIL           ; jump, release failed
             .
             .
    FAIL:    .
             .
    BUFF     dw  0               ; contains segment of previously
                                                  allocated block
```

특정한 값을 레지스터에 담아서 넘겨주게 하는 것이다. 따라서 이 인터럽트를 호출했을 때 특정 레지스터에 특정한 값이 넘어오는지를 검사하게 되면 이 램상주 프로그램이 기억장소 내에 존재하는지의 여부를 알 수 있게 된다. 이 방법은 필자가 생각하기에 가장 안전한 방법이며 추천하고 싶은 기법이다. 사용 가능한 특정 인터럽트 핸들러는 여러 개가 있지만 가장 보편적으로 사용되는 것이 2Fh번 인터럽트(multiplex interrupt)이다. 이 인터럽트는 보통 도스 3.0 이상에서 프린터 스풀러(Printer Spooler)에 사용되는 인터럽트지만 램상주 프로그램의 존재여부 확인 및 램상주 프로그램과 데이타를 주고 받을 때 많이 사용된다. 하지만 2Fh번 인터럽트도 도스 3.0이상에서는 아무런 문제가 없지만 도스 2.xx에서는 PRINT.COM이 기억장소에 상주하고 있을 때 사용할 수 없다는 단점이 있다. 그 이유는 도스 3.0 이상에서는 응용 프로그램에서 프린터 스풀러 기능을 사용할 수 있도록 기능호출 형식으로 정의해 놓았지만 도스 2.xx에서는 그러한 기능호출이 정의되어 있지 않고 PRINT.COM이 독자적으로 이 인터럽트를 이용하여 그 기능을 수행하는 형식으로 되어 있기 때문이다.

변형된 2Fh번 인터럽트 핸들러를 〈그림 4〉에 표시하였다. 이 핸들러는 AH 레지스터에 프로그램에서 미리 정한 MultiplexID와 같은 값이 넘어오고, AL 레지스터에 미리 정한 기능값(function number : 보기에서는 0)이 넘어오면 AL 레지스터에 정해진 값(여기서는 FFh)을 돌려줘서 램상주 프로그램의 존재여부를 알린다.

또한 도스'2.xx에서 PRINT.COM이 기억장소에 존재하지 않거나, 도스 3.0이상의 경우에 〈그림 4〉의 인터럽트 핸들러가 기억장소에 상주한 상태에서 2Fh번 인터럽트의 사용법 및 사용예를 〈표 3〉에 보였다. 도스 버전의 검사와 PRINT.COM의 존재여부 검사 방법은 수동적 램상주 프로그램의 기본 형식을 제시할 때 함께 설명할 것이다.

램상주 프로그램의 제거 방법

램상주 프로그램이 더 필요하지 않을 경우에 그대로 둔다는 것은 기억장소의 낭비를 초래하는 일이다. 또한 많은 기억장소가 요구되는 응용 프로그램을 수행시킬 필요가 있을 때 램상주 프로그램을 제거할 수 있는 방법이 있다면 시스템을 끈 후 다시 켜야 하는 불편함이 따를 것이다.

램상주 프로그램의 제거 과정은 크게 두 가지로 나눌 수 있다. 첫번째 과정은 인터럽트 벡터를 원래대로 복귀시키는 것이다. 이 과정에서 유의할 점은 인터럽트 벡터가 다른 프로그램에 의해서 변형되었을 가능성을 염두에 두어야 한다. 만약 이것을 검사하지 않고 인터럽트 벡터를 복구시킨다면 다른 프로그램이 동작하지 않을 가능성이 있다.

두번째 과정은 할당된 기억장소를 제거하는 것이다. 이 때는 21h번 인터럽트의 49h번 기능(Free Allocated Memory fu-

nction)을 사용한다. 이 인터럽트의 사용법과 사용예는 〈표 4〉에 제시하였다. 할당된 기억장소를 제거할 때는 앞에서 말한 바와 같이 프로그램뿐만 아니라 프로그램의 정보영역 및 할당해놓은 기억장소까지 제거해야 한다. 하지만 정보영역은 보통의 램상주 프로그램에서 사용하지 않으므로 인스톨 과정 중에 제거하는 것이 일반적인 방법이다.

수동적 램상주 프로그램의 기본 형식

지금까지 살펴 본 여러가지 사항을 응용하여 만든 수동적 램상주 프로그램의 기본 형식은 〈리스트 1〉과 같다. 〈리스트 1〉에서 보는 바와 같이 램상주 프로그램은 크게 두 부분으로 나뉜다. 즉, INSTALL 프로시져와 RESIDENT 프로시져이다.

INSTALL 프로시져에서는 먼저 도스 버전을 검사한다. 여기서 만약 도스 1.xx라면 수행을 중지시킨다. 또한 도스 2.xx라면 PRINT.COM이 기억장소에 상주하는지의 여부를 검사하여 상주한다면 수행을 중지시킨다. 두번째 과정으로 2Fh번 인터럽트를 호출하여 램상주 프로그램이 이미 존재하는지의 여부를 확인한다. 프로그램이 존재하지 않는다면 2Fh번 인터럽트 벡터 및 프로그램이 사용할 인터럽트 벡터를 바꾸어 준다. 만약 프로그램이 사용할 인터럽트가 이미 다른 인터럽트 핸들러에 의해서 사용되던 것이라면 그 전 인터럽트 벡터를 프로그램내에서 보관한다. 세번째 과정으로 램상주 프로그램의 정보영역을 제거한다. 마지막으로 만약 모든 과정이 성공적으로 수행되었다면 RESIDENT 프로시져와 2Fh번 인터럽트 핸들러를 램상주로 만들면서 수행을 종료하게 된다.

RESIDENT 프로시져는 비교적 간단하다. 먼저 레지스터들을 대피시킨 후 DS 레지스터를 램상주 프로그램의 것으로 바꾼다. 그 후, 만약 이미 사용되는 인터럽트를 가로챘다면 그 인터럽트 핸들러를 먼저 수행시킨 후 자신의 루틴을 실행하며, 만약 쓰이지 않던 인터럽트를 사용했을 경우라면 자신의 루틴만을 수행하면 된다. 여기서는 기존의 인터럽트 핸들러를 가로챈 경우를 예로 들었다. 마지막으로 레지스터들을 복구시킨 후, IRET 명령으로 수행을 마친다.

여기서는 편의상 램상주 프로그램의 제거과정은 제외하였다. 그 이유는 기억장소상에서 프로그램을 제거하는 방법이 프로그래머마다 매우 다양하기 때문이다. 어떤 사람은 한 번 인스톨된 후에 다시 프로그램을 실행시키면 프로그램이 기억장소에서 제거되는 식으로 프로그래밍하기를 좋아하는 반면, 어떤 사람은 특정한 명령어행 인자(command-line argument)나 특정키를 누를 때 프로그램이 제거되도록 프로그래밍하는 경우도 있다. 사이드킥과 같이 특정키를 눌렀을 때 프로그램이 제거되도록 하려면 능동적 램상주 프로그래밍 기법이 필요하다. 이러한 방법은 신년 호에서 다루도록 하겠다.

88년을 보내며

88년을 보내며 독자 여러분에게 자그마한 정보를 주고 싶다. 그 정보란 어셈블리어를 공부는 했는데 프로그램을 작성하지 못하는 사람들에게 필자가 권하고 싶은 책들이다. 처음 어셈블리어를 공부하고자 하는 사람들에게 로버트 래퍼(Robert Lafore)의 'Assembly Language Primer for the IBM PC & XT'를 권하고 싶다. 이 책의 가장 큰 장점은 명령어 중심의 나열이 아닌 실제로 사용하면서 하나씩 익혀 나가는 학습방법이다. 또한 초보자들에게 공포감(?)을 주는 매크로 어셈블러보다는 DEBUG.COM을 먼저 사용하게 함으로써 쉽게 어셈블리어에 접근하도록 한다. 이 책이나 그외 다른 책으로 어셈블리어의 기초를 익힌 사람들에게는 스티븐 호너(Steven Holzner)의 'Advanced Assembly Langauge on the IBM PC'를 권하고 싶다. 이 책은 어셈블리어의 기초를 마친 사람들을 대상으로 기초적인 사항들을 정리해 주고 실제로 프로그램을 작성할 수 있는 기초를 배양해 준다. 또한 부록에 있는 BIOS 및 도스 인터럽트의 요약이 잘 정리되어 있어 찾아보기 쉽다. 하지만 램상주 프로그램에 관해서는 틀린 말들과 오래된 테크닉들이 많이 게재되어 있으므로 그 곳은 조심해서 읽으라고 권하고 싶다. 또 한 가지 빼놓을 수 없는 책은 레이 던캔(Ray Duncan)의 'Advanced MS-DOS'이다. 이 책은 필자가 가장 좋아하는 책으로서 도스에 관한 거의 모든 사항이 실려 있으며, 특히 이 책의 절반 정도를 차지하는 인터럽트를 정리하는 내용을 살펴보면 이만큼 잘 정리된 책을 찾아볼 수 없다는 것을 느낀다. 특히 레이 던캔이 의대를 나온 소아과 전문의라는 점에서 의사인 필자로 하여금 이 책을 편애하게 만드는 또 하나의 이유인지도 모른다. 그 외에 피터 노턴(Peter Norton)의 'Inside the IBM PC'도 IBM PC의 기본 구조를 이해하는데 없어서는 안될 책이며 어셈블리어를 공부하기 전에 가장 먼저 읽어야 할 책이다. 또한 공부하고 난 후에도 참조해야 할 귀중한 보배이다.

사실 수동적 램상주 프로그램은 그 활용도가 극히 제한되어 있다. 하지만 이 방법에 대한 기초를 든든히 하지 않으면 능동적 램상주 프로그램을 작성하기가 힘들다. 올해는 수동적 램상주 프로그램으로 가볍게 마무리 짓기로 하고, 골치아픈 능동적 램상주 프로그램은 내년으로 넘기기로 하자. 독자 여러분! Marry Christmas and Happy New Year!

〈리스트 1〉 수동적 램상주 프로그램의 기본 형식

```
        CR          EQU  0Dh
        LF          EQU  0Ah
        IntNo       EQU  ?      ; 사용자가 정함
        MultiplexID EQU  ?      ; 사용자가 정함

TSR     SEGMENT

        ASSUME  cs: TSR, ds: TSR

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

RESIDENT PROC NEAR

        ORG  100h
 Entry: jmp  INSTALL
 OldInt   DD ?
 OldISR2F DD ?

        ; 데이타들을 이곳에 둔다.

 Main:  ; 이곳에 레지스터들을  대피시킨다.

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

        pushf                   ; call original interrupt handler
        call OldInt

        ; 램상주 프로그램을 이곳에 둔다.

        iret

RESIDENT ENDP

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

ISR2F   PROC  NEAR              ; INT 2Fh Handler
                                ;     Call with: AH = handler ID
                                ;                AL = function No
                                ;     Return for function 0 :
                                ;                AL = 0FFh
        cmp  ah, MultiplexID
        je   ID_OK              ; jump if requested
        jmp  OldISR2F           ; chain to previous handler
 ID_OK: test al, al
        jnz  Exit               ; jump if reserved or undefined
        mov  al, 0FFh           ; AL = 0FFh (installed)

 Exit:  iret                    ; return from interrupt

ISR2F   ENDP

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

INSTALL PROC  NEAR

        ; get DOS version

        mov  ah, 30h            ; INT 21h function 30h
        int  21h                ; (get MS-DOS version)
        xchg ah, al             ; AH = major version
                                ; AL = minor version
        cmp  ah, 2
        jb   ERR1
        ja   DOS3

        ; DOS 2.xx

        mov  ah, 35h            ; get interrupt vector
        mov  al, 2Fh            ; multiplex interrupt
        int  21h

        mov  ax, es
        or   ax, bx
        jz   PrevSet            ; if not modified, jump

        mov  ah, 0FFh           ; look for PRINT.COM
        xor  al, al
        int  2Fh                ; if resident, AH = queue
                                ; otherwise, AH is unchanged
        cmp  ah, 0FFh
        jne  ERR2               ; PRINT.COM already installed

        ; Program already installed ?

DOS3:   mov  ah, MultiplexID    ; AH = multiplex int ID value
        xor  al, al             ; AL = 0
        int  2Fh                ; multiplex interrupt
        test al, al
        jz   SetVector          ; jump if OK to install
        cmp  al, 0FFh
        jne  ERR3               ; can't install
        jmp  ERR4               ; already installed

        ; Set Interrupt Vector
```

```
Prevset :
        mov  ah, 25h
        mov  al, 2Fh
        mov  dx, offset Exit
        int  21h

SetVector:
        mov  ah, 35h            ; get 2Fh interrupt vector
        mov  al, 2Fh
        int  21h
        mov  word ptr OldISR2F, bx
        mov  word ptr OldISR2F[2], es

        mov  ah, 25h            ; set 2Fh interrupt vector
        mov  al, 2Fh
        mov  dx, offset ISR2F
        int  21h
        mov  ah, 35h            ; get interrpt vector
        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

        ; Error handling routine

ERR1:   mov  al, 1
        jmp  ErrHandler

ERR2:   mov  al, 2
        jmp  ErrHandler
ERR3:   mov  al, 3
        jmp  ErrHandler
ERR4:   mov  al, 4

ErrHandler:
        push ax
        mov  bx, offset MessageTable
        xor  ah, ah
        shl  ax, 1
        add  bx, ax
        mov  dx, [bx-2]
        mov  ah, 9
        int  21h

        pop  ax                 ; DOS 1.xx ?
        cmp  al, 1
        je   DOS1

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

DOS1:   push es                 ; push PSP:0000
        xor  ax, ax
        push ax
        ret                     ; far return (jump to PSP:0000)

MessageTable DW  Message1
             DW  Message2
             DW  Message3
             DW  Message4

Message1     DB  CR, LF, 'MS-DOS 2.0 or later version required'
             DB  CR, LF, '$'
Message2     DB  CR, LF, 'Can''t install: PRINT.COM active'
             DB  CR, LF, '$'
Message3     DB  CR, LF, 'Can''t install this TSR'
             DB  CR, LF, '$'
Message4     DB  CR, LF, 'Already Installed', CR, LF, '$'

INSTALL  ENDP

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

TSR      ENDS
         END  Entry
```