터보 파스칼로 작성된

능동적 램상주 프로그램 툴

지난 3월호 이 노트를 통해 게재한 '어셈블리어로 작성된 능동적 램상주 프로그램 툴'에 이어 이번 호에서는 터보 파스칼 4.0과 5.0을 사용한 능동적 램상주 프로그램 작성 방법을 소개하고자 한다.

능동적 램상주 프로그램(active memory-resident program)이란 어떤 프로그램이 실행중일 때라도 예약키(hot key)를 누르기만 하면 당장 그 프로그램의 실행을 중지시키고 자기 자신이 수행되는 프로그램을 말한다.

본 컬럼에서는 88년 12월호부터 89년 3월호까지 4회에 걸쳐서 램상주 프로그램을 만드는 기법과 어셈블리어로 작성된 프로그램 툴을 제공한 바 있다. 처음에는 터보 파스칼과 C언어를 사용한 램상주 프로그램 툴을 계속적으로 소개할 계획이었으나, 강좌의 내용이 너무 어렵다는 독자들이 많았고, 프로그램 로더를 다루어 달라는 편집부의 요청으로 잠시 다른 주제로 본 컬럼을 진행하게 되었다.

이번 호에서는 이미 예고한 바와 같이 터보 파스칼 4.0 및 5.0을 사용하여 능동적 램상주 프로그램을 작성하는 방법을 설명하도록 하겠다. 램상주 프로그램의 작성 기법에 대해서는 지난 호들을 참고하기 바라며, 이 컬럼에서는 터보 파스칼로 램상주 프로그램을 작성할 때 알아 두어야 할 기본적인 사항들에 대해서만 설명하고자 한다. 또한 이러한 기본 지식을 바탕으로 터보 파스칼로 작성된 능동적 램상주 프로그램 툴을 제공한다.

터보 파스칼에 대한 기본 지식

램상주 프로그램은 주로 어셈블리어로 작성되는 것이 보통이지만, 램상주 프로그램에서 사용하는 기법을 잘 이해하기만 한다면 파스칼이나 C언어와 같은 고급 언어로도 램상주 프로그램을 작성할 수 있다. 고급 언어로 램상주 프로그램을 작성하는 것은 어셈블리어를 사용하는 것보다 훨씬 작성하기 쉽고 고치기도 수월하다는 장점이 있다. 하지만 램상주 프로그램이 실행될 때 야기될 수 있는 여러가지 부작용들을 완벽하게 제어할 수 없고 실행 화일의 크기가 커진다는 단점이 있으므로, 프로그래머는 이러한 장단점들을 잘 생각해서 사용할 컴퓨터 언어를 결정해야 할 것이다.

터보 파스칼은 런타임 라이브러리(run-time library)가 재호출(re-entrance)이 가능하도록 작성되었으며 실행중 인터럽트가 걸리는 것을 허용하기 때문에, 별다른 문제없이 램상주 프로그램을 만들 수 있다. 또한 터보 파스칼 4.0 이상에서는 램상주 프로그램의 작성을 용이하게 해주는 interrupt 키워드(keyword) 및 여러가지 프로시져들이 존재하기 때문에, 다른 고수준 언어들에 비해서 훨씬 쉽게 램상주 프로그램을 작성할 수 있다.

interrupt는 터보 파스칼 4.0 이상에서 추가된 키워드로서, 사용 형식은 〈그림 1〉과 같다. 여기서 인자(argument)로 사용되는 레지스터들은 일부 혹은 전부 생략 가능하지만, 중간에 위치한 레지스터 하나만을 생략할 수는 없고 그 전에 정의된 레지스터들을 모두 생략해야만 한다. 즉,

```
procedure IntHandler(DI, ES, BP: word);
```

과 같은 선언은 잘못된 것으로, 다음과 같이 고쳐야 한다.

```
procedure IntHandler(DI, DS, ES, BP: word);
```

interrupt는 사용자가 작성한 프로시져가 인터럽트 처리 루틴(interrupt service routine)임을 나타낸다. 따라서 interrupt로 정의된 프로시져는 프로그램에서 직접 부를 수 없으며, 다음에 설명할 SetIntVec 프로시져를 사용하여 특정 인터럽트에 대한 처리 루틴으로 선언해야만 사용할 수 있다. 또한 interrupt로 선언된 프로시져는 실행될 때의 초기 루틴과 종료 루틴이 일반적인 프로시져들과는 다르게 컴파일된다. 즉, 보통의 프로시져가 수행될 때는 BP, SP, SS, DS 레지스터만을 대피시키지만, interrupt로 선언된 프로시져는 모든 레지스터를 대피시키고 DS 레지

스터를 바꾸어주며 프로시져 종료시에는 모든 레지스터의 값들을 복구시켜 준다. 단, 프로시져 정의부에서 선언된 레지스터들 중 프로시져 내에서 사용된 레지스터들은 프로시져가 종료되어도 복구되지 않고, 프로시져 수행시에 바뀐 레지스터의 값을 그대로 넘겨주게 된다. 〈그림 2〉에는 일반적인 프로시져의 초기 루틴과 종료 루틴을 나타내었고, 〈그림 3〉에는 interrupt로 선언된 프로시져의 초기 루틴과 종료 루틴을 표현하였다.

터보 파스칼에서 램상주 프로그램을 작성할 때 사용되는 여러가지 프로시져들은 〈그림 4〉와 같다. 이 프로시져들은 모두 Dos 유니트(unit)에 정의되어 있으므로 이들을 사용하기 위해서는 프로그램의 초기에 uses Dos;라고 선언해 주어야 한다.

GetIntVec 프로시져는 IntNo로 표시되는 인터럽트의 주소를 Vector라는 포인터 변수에 넘겨주는 역할을 한다. SetIntVec 프로시져는 IntNo로 표시되는 인터럽트의 주소를 Vector라는 포인터 변수의 값으로 바꾸어 준다. 이 때 interrupt로 선언된 프로시져의 주소를 나타내기 위해서 프로시져 이름앞에 @ 연산자를 붙인다. 예를 들어서 인터럽트 9번을 가로채는 프로시져의 이름을 ISR09라고 하면, 다음과 같은 명령을 사용해서 인터럽트 처리 루틴의 주소를 바꾸어 줄 수 있다.

```
SetIntVec(9, @ISR09);
```

Keep 프로시져는 21h 인터럽트의 31h번 기능(terminate and stay resident function)과 같은 역할을 하는 것으로, 프로그램, 데이타, 스택 및 힙(heap)을 모두 기억장소에 상주시키는 역할을 한다. 따라서 이 명령을 사용하기 전에 $M 지시어(directive)를 이용하여 스택과 힙의 크기를 적절하게 지정하여야 한다. $M 지시어의 사용 형식은 다음과 같다.

```
{$M StackSize, MinHeap, MaxHeap}
```

여기서 StackSize는 프로그램에서 사용할 스택의 크기를 나타내며, 최소 1024바이트에서 최대 65520바이트까지 지정이 가능하다. MinHeap 및 MaxHeap은 각각 최소 및 최대 힙의 크기를 나타내며, 0에서 65536바이트까지 지정이 가능하다. 단, MaxHeap의 크기는 MinHeap의 크기보다 같거나 커야 한다.

Intr 프로시져는 Regs에서 지정한 값을 가지고 IntNo에서 지정한 인터럽트를 호출하는 역할을 하며, MsDos 프로시져는 Regs에서 지정한 값을 가지고 도스 인터럽트(21h번 인터럽트)를 호출하는 역할을 한다. 여기서 Regs는 Registers 타입의 변수이며, Registers 타입은 Dos 유니트에서 선언된 데이타 타입으로 레지스터의 값을 보관하는 레코드(record) 타입이다. Dos 유니트 내에서 선언

〈그림 1〉 interrupt 키워드의 사용 형식

```
procedure IntHandler(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP: word);
  interrupt;
  begin
    :
    :
  end;
```

〈그림 2〉 일반적인 프로시져의 초기 루틴과 종료 루틴

```
I.  초기 루틴
    push bp                 ; save BP
    mov  bp, sp             ; save SP
    sub  sp, LocalSize      ; allocate local variables

II. 종료 루틴
    mov  sp, bp             ; deallocate local variables
    pop  bp                 ; restore BP
    ret  ParamSize          ; remove parameters and return
```

〈그림 3〉 interrupt로 선언된 프로시져의 초기 루틴과 종료 루틴

```
I.  초기 루틴
    push ax                 ; save all registers
    push bx
    push cx
    push dx
    push si
    push di
    push ds
    push es
    push bp
    mov  bp, sp
    sub  sp, LocalSize      ; allocate local variables
    mov  ax, seg Data       ; set up DS
    mov  ds, ax

II. 종료 루틴
    mov  sp, bp             ; deallocate local variables
    pop  bp                 ; restore all registers
    pop  es
    pop  ds
    pop  di
    pop  si
    pop  dx
    pop  cx
    pop  bx
    pop  ax
    iret                    ; return from int. handler
```

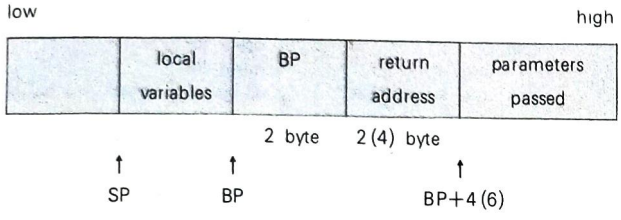
〈그림 4〉 램상주 프로그램에서 사용되는 프로시져들

```
GetIntVec(IntNo: byte; var Vector: pointer);

SetIntVec(IntNo: byte; Vector: pointer);

Keep(ExitCode: word);

Intr(IntNo: byte; var Regs: Registers);

MsDos(var Regs: Registers);
```

〈그림 5〉 Registers 타입의 선언 형식

```
Registers = record
              case integer of
                0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: word);
                1: (AL,AH,BL,BH,CL,CH,DL,DH: byte);
              end;
```

〈그림 6〉 프로시져 내에서 사용하는 스택의 구조



된 Registers 타입은 〈그림 5〉와 같다.

터보 파스칼로 램상주 프로그램을 작성하기 위해서는 프로시져내에서 스택을 사용하는 방법에 대해서도 알아야 한다. 프로시져 내에서 사용하는 스택의 구조는 〈그림 6〉과 같다. 주프로그램에서 프로시져를 호출하면 파라미터들을 선언된 순서대로 스택에 저장한 후, 원래의 프로그램의 주소를 스택에 보관하고 프로시져로 제어를 넘긴다. 아 때 한 세그먼트(64KB) 내에서 호출하는 경우에도(near call) 보관되는 주소가 16비트의 크기를 가지지만, 다른 세그먼트에서 호출하는 경우에는(far call) 32비트의 크기를 가지게 된다. 프로시져 내에서는 〈그림 2〉의 초기 루틴을 실행시킨다. 즉, 먼저 스택에 BP 레지스터의 값을 저장한 다음, SP 레지스터의 값을 BP 레지스터로 옮긴다. 그 후 프로시져 내의 지역 변수(local variable)가 사용되는 영역을 스택에서 확보한다. 따라서 프로시져 내에서 파라미터의 위치는 near call의 경우에는 BP+4, far call의 경우에는 BP+6이 된다

램상주 프로그램 툴 사용 방법

램상주 프로그램의 제작 기법과 터보 파스칼에 대한 기본 지식을 바탕으로 작성된 램상주 프로그램 툴은 〈리스트 1〉과 같다.

지금까지 설명한 내용만 모두 소화한다면 리스트를 이해하는데 별 어려움이 없을 것으로 생각된다. 하지만 리스트를 완전히 이해하지 못하더라도 터보 파스칼로 프로그래밍할 수 있는 사람이라면 누구나 쉽게 램상주 프로그램을 작성할 수 있을 것이다.

램상주 프로그램 툴을 사용하여 램상주 프로그램을 작성하려면, 프로그램의 처음 부분에서 나오는 상수들을 정의해 주어야 한다. 제일 먼저 정할 것은 프로그램에서 사용

〈리스트 1〉 터보 파스칼로 작성된 능동적 램상주 프로그램 툴

```
0001 program TSR_Tool;
0002
0003 {$M 1024,0,0}
0004 {$V-,R-,S-,B-}
0005
0006 uses Crt, Dos;
0007
0008 const
0009     HandlerID : word = $5154;
0010     ExistID   : word = $4F4B;
0011
0012     { activation key ; ALT & Left-SHIFT }
0013     Activate  : word = $000A;
0014
0015 var
0016     Regs : Registers;
0017
0018     { old addresses of interrupt service routines }
0019     OldISR09 : pointer;
0020     OldISR11 : pointer;
0021     OldISR28 : pointer;
0022
0023     OldSS  : word;
0024     TSR_SS : word;
0025     TSR_SP : word;
0026
0027     InDosFlag : ^word;
0028
0029     KbdStatus : word absolute $0000:$0417;
0030     ShiftStatus : word;
0031
0032     EquipList : word absolute $0000:$0410;
0033
0034     InTSR : boolean;
0035
0036
0037 procedure CallInt(p : pointer);
0038 begin
0039      inline($9C/             { PUSHF             }
0040             $FF/$5E/$04);  { CALL FAR [BP+4] }
0041 end;
0042
0043 procedure DeInstall;
0044 begin
0045     inline (
0046         $8C/$16/OldSS/      { MOV  OldSS, SS }
0047         $8E/$16/TSR_SS/     { MOV  SS, TSR_SS }
0048         $8B/$26/TSR_SP      { MOV  SP, TSR_SP }
0049         );
0050
0051     SetIntVec($09,OldISR09); { restore INT 9  }
0052     SetIntVec($11,OldISR11); { restore INT 11 }
0053     SetIntVec($28,OldISR28); { restore INT 28 }
0054
0055     { deallocate this program's memory block }
0056     Regs.AX := $4900;
0057     Regs.ES := PrefixSeg;
0058     Intr($21,Regs);
0059
0060     { restore the stack of the interrupted program }
0061     inline (
0062         $8E/$16/OldSS       { MOV  SS, OldSS }
0063         );
0064
0065     Halt(0);                { exit to DOS }
0066
0067 end;
0068
0069 procedure TSRprogram;
0070 begin
0071     { ****************************************
0072       *                                      *
0073       *          MAIN PROGRAM OF TSR         *
0074       *                                      *
0075       **************************************** }
0076 end;
0077
0078 procedure ISR09;
0079 interrupt;
0080 begin
0081     inline($FA);        { CLI }
0082     CallInt(OldISR09); { call the original keyboard interrupt }
0083
0084     ShiftStatus := KbdStatus and $000F;
0085
0086     if (InTSR) and (ShiftStatus = Activate) then
0087         DeInstall;
0088
0089     if (not InTSR) and (ShiftStatus = Activate) and (InDosFlag^ = 0) then
0090       begin
0091         InTSR := true;
0092         { save the interrupted program's stack segment and pointer, and }
0093         { restore TSR's stack segment and pointer }
0094         inline (
0095             $8C/$16/OldSS/      { MOV  OldSS, SS }
0096             $8E/$16/TSR_SS/     { MOV  SS, TSR_SS }
0097             $8B/$26/TSR_SP      { MOV  SP, TSR_SP }
0098             );
0099
0100         TSRprogram;
0101
0102         { restore the stack of the interrupted program }
0103         inline (
```

```
0104              $8E/$16/OldSS      { MOV  SS, OldSS }
0105              );
0106          InTSR := false;
0107        end;
0108      inline($FB); { STI }
0109  end;
0110
0111  procedure ISR11(AX,BX,CX,DX,SI,DI,DS,ES,BP: word);
0112  interrupt;
0113  begin
0114      CallInt(OldISR11);
0115      if (CX = HandlerID) then
0116          CX := ExistID;
0117      inline($FB); { STI }
0118      AX := EquipList;
0119  end;
0120
0121  procedure ISR28;
0122  interrupt;
0123  begin
0124      inline($FA);        { CLI }
0125      CallInt(OldISR28); { call the original interrupt hand
                                                        ler }
0126
0127      if (InTSR) and (ShiftStatus = Activate) then
0128          DeInstall;
0129
0130      if (not InTSR) and (ShiftStatus = Activate) then
0131        begin
0132          InTSR := true;
0133          { save the interrupted program's stack segment an
                                                      d pointer }
0134          { restore TSR's stack segment and pointer }
0135          inline (
0136              $8C/$16/OldSS/      { MOV  OldSS, SS }
0137              $8E/$16/TSR_SS/     { MOV  SS, TSR_SS }
0138              $8B/$26/TSR_SP      { MOV  SP, TSR_SP }
0139              );
0140
0141          TSRprogram;
0142
0143          { restore the stack of the interrupted program }
0144          inline (
0145              $8E/$16/OldSS      { MOV  SS, OldSS }
0146              );
0147          InTSR := false;
0148        end;
0149      inline($FB); { STI }
0150  end;
0151
0152  function Install : boolean;
0153  var
0154      EnvSegPtr : ^word;
0155  begin
0156      { check to see if TSR is already installed }
0157      Regs.CX := HandlerID;
0158      Intr($11,Regs);
0159      if (Regs.CX = ExistID) then
0160          begin
0161            Install := false;
0162            exit;
0163          end;
0164
0165      { capture INT 9 (BIOS keyboard)}
0166      GetIntVec($09,OldISR09);
0167      SetIntVec($09,@ISR09);
0168
0169     { capture INT 11 (BIOS equipment)}
0170      GetIntVec($11,OldISR11);
0171      SetIntVec($11,@ISR11);
0172
0173      { capture INT 28 (DOS idle)}
0174      GetIntVec($28,OldISR28);
0175      SetIntVec($28,@ISR28);
0176
0177      { locate the InDOS flag }
0178      Regs.AX := $3400;
0179      Intr($21,Regs);
0180      InDosFlag := Ptr(Regs.ES,Regs.BX);
0181
0182      { save the location and status of TSR's stack }
0183      TSR_SS := SSeg;
0184      TSR_SP := SPtr;
0185
0186      { make a pointer to the Environment Segment stored in
                                                      the PSP }
0187      EnvSegPtr := Ptr(PrefixSeg,44);
0188
0189      { deallocate the local environment block }
0190      Regs.AX := $4900;
0191      Regs.ES := EnvSegPtr^;
0192      Intr($21,Regs);
0193
0194      InTSR := false;
0195
0196      Install := true;
0197  end;
0198
0199  begin
0200
0201      if Install then
0202        begin
0203          Writeln('Program installed');
0204          Keep(0);
0205        end
0206      else
0207        Writeln('Already installed.');
0208
0209  end.
```

할 스택의 크기이다. 여기서는 $M 지시어를 사용하여 스택의 크기를 1024바이트로 설정하였는데, 프로그램의 크기가 크다면 스택의 크기도 증가시켜야 한다.

HandlerID와 ExistID는 램상주 프로그램이 기억장소에 이미 존재하고 있는지를 검사할 때 사용하는 상수들이다. Hanldler ID와 ExistID의 값은 0~FFFFh 이내의 어떤 값을 사용해도 되지만, 서로 다른 값으로 정의해 주어야 한다. 만약 같은 값을 사용하게 되면 램상주 프로그램이 기억장소에 존재하지 않을 경우에도 존재하는 것으로 오인하게 되어 램상주 프로그램을 기억장소에 상주시킬 수 없게 된다.

Activate는 램상주 프로그램에서 사용할 예약키(hot key)를 지정해주는 상수이다. 여기서는 간단하게 쉬프트형 키(〈Shift〉, 〈Ctrl〉, 〈Alt〉)의 조합으로 예약키를 구성할 수 있도록 하였다. 사용할 조합에 따른 값은 〈표 1〉과 같다. 예를 들어서 왼쪽 〈Shift〉키와 〈Alt〉키를 동시에 누를 때 램상주 프로그램이 실행되도록 하려면 Activate의 값을 Ah로 정하면 된다.

상수들은 정의해 준 후에는 램상주시키고자 하는 프로그램을 TSRprogram 프로시져 내에 위치시키면 램상주 프로그램이 완성된다. 단, 화면에 출력시키는 램상주 프로그램을 작성할 경우에는 TSRprogram 프로시져 내에서 그 전의 화면 모드, 내용 및 커서의 위치를 대피시키는 루틴이 필요하다. 완성된 프로그램은 터보 파스칼 4.0 및 5.0으로 컴파일하여 실행 프로그램을 만든다.

이렇게 해서 작성된 램상주 프로그램은 보통의 프로그램들처럼 도스 프롬프트하에서 프로그램의 이름을 입력함으로써 기억장소 내에 상주시킬 수 있으며, 이때는 'Program installed'이라는 메시지를 출력한다. 만약 프로그램이 이미 기억장소 내에 상주하고 있다면 'Already inatlled' 라는 메시지를 출력하고 수행을 중지하게 된다. 이미 상주하고 있는 프로그램을 기억장소에서 제거하고자 할 때는 램상주 프로그램이 수행되고 있을 때 한 번 더 같은 예약키를 누르면 된다.

〈표 1〉 Activate의 값 (*, pressed)

〈Alt〉	* * * * * * * *
〈Ctrl〉	* * * * * * * *
left 〈Shift〉	* * * * * * * *
right 〈Shift〉	* * * * * * * *
Activate	0 1 2 3 4 5 6 7 8 9 A B C D E F

램상주 프로그램 툴의 동작 원리

먼저, 프로그램은 바이트지 88년 7월호의 기사를 참고하여 제작하였음을 밝혀둔다.

상수들의 의미는 사용 방법에서 다루었으므로 여기서는 변수들에 대해서만 먼저 설명하겠다. OldISR09, OldISR11, OldISR28은 각각 원래의 인터럽트 9, 11h, 28h번 처리 루틴의 주소를 보관하는 변수이다. OldSS는 원래의 프로그램이 가지고 있던 SS 레지스터의 값을 보관하며, TSR_SS와 TSR_SP는 램상주 프로그램의 SS 레지스터와 SP 레지스터의 값을 보관한다. InDoFlag는 지금 도스 인터럽트가 사용중인지를 나타내는 도스 수행 표시기(InDOS flag)의 값을 보관한다. KbdStatus는 BIOS 데이타 내에 있는 키보드 상태를 나타내며, ShiftStatus는 〈NumLock〉이나 〈CapsLock〉 등의 키와 무관하게 쉬프트형 키의 상태를 저장하는 데 사용된다. EquipList는 BIOS 데이타 영역 내의 주변기기 상태를 나타내며, 마지막으로 InTSR은 지금 램상주 프로그램이 수행되고 있는 중인지를 나타낸다.

램상주 프로그램 툴은 프로그램을 기억장소에 상주시키는 부분, 예약키를 인지하여 램상주 프로그램을 호출하는 부분, 램상주 프로그램 고유의 역할을 수행하는 부분과 프로그램을 기억장소에서 제거하는 부분으로 크게 나눌 수 있다.

프로그램을 기억장소에 상주시키는 부분(Install 함수)에서는 11h번 인터럽트를 이용하여 램상주 프로그램이 기억장소 내에 존재하는지를 먼저 확인하는 작업을 수행한다. 만약 프로그램이 존재하지 않으면 9, 11h , 28h번 인터럽트 벡터를 바꾸어 주고, 도스 수행 표시기(InDOS flag)의 위치 및 SS, SP 레지스터의 값을 보관하고, 프로그램의 정보영역(environment)을 제거한 후 참값(true)을 반환한다. 참값이 반환되면 주프로그램에서는 프로그램을 상주시키게 된다. 만약 램상주 프로그램이 이미 존재한다면 Install 함수에서는 거짓(false)을 반환하게 되며, 주프로그램에서는 에러 메시지를 출력시킨 후 실행을 중지하게 된다.

예약키는 9번 인터럽트(keyboard-action interrupt)를 가로챈 ISR09 프로시져에서 검사하게 되며, 예약키가 눌러졌을 때 램상주 프로그램이 실행중이 아니고 도스 인터럽트가 사용중이 아니라면 램상주 프로그램을 호출한다. 만약 예약키가 눌러졌을 때 램상주 프로그램이 실행중이라면 램상주 프로그램을 기억장소에서 제거하게 된다. 또한 ISR28 프로시져에서도 예약키를 검사해서 예약키가 눌러졌을 때 램상주 프로그램이 실행 중이 아니라면 램상주 프로그램을 실행시키고, 램상주 프로그램이 실행 중이라면 램상주 프로그램을 기억장소에서 제거하게 된다. ISR11 프로시져에서는 HandlerID를 검사한 후, 같은 값이 사용되었다면 ExistID를 반환하여 자신의 존재 여부를 알려주게 된다. DeInstall 프로시져에서는 인터럽트 벡터를 원래대로 복구시킨 후, 기억장소에서 램상주 프로그램을 제거하게 된다.

어셈블리어와 다른 점

파스칼로 작성된 램상주 프로그램 툴과 어셈블리어로 작성된 램상주 프로그램 툴(본지 89년 3월호에 게재)의 차이점은 크게 두 가지이다. 첫째, 어셈블리어용에서는 9번 인터럽트에서 예약키를 인지하고 8번 인터럽트에서 수시로 시스템을 검사하여 안전한 상태에 있을 때 램상주 프로그램을 실행시키기 때문에 예약키를 한번만 누르면 램상주 프로그램을 실행시킬 수 있지만, 파스칼용에서는 9번 인터럽트에서 예약키 인지와 시스템 검사를 함께 처리하기 때문에 시스템이 안전한 상태에 있지 않으면 한 번 예약키를 눌러서 램상주 프로그램을 실행시킬 수 없다. 둘째, 파스칼용에서는 예외처리 루틴을 대피시키지 않았기 때문에 에러가 발생했을 때는 원래 프로그램의 예외처리 루틴으로 제어가 넘어가게 된다.

필자가 이러한 루틴들을 포함시키지 않은 이유는 지금 상태로도 실행화일의 크기가 어셈블리어에 비하여 크고, 보통의 상황에서는 문제가 되지 않는 기능들이기 때문이다. 하지만 본격적인 유틸리티를 작성하고자 한다면 8번 인터럽트 처리 루틴과 예외 처리 루틴을 따로 작성하면 될 것이다.