from Turbo Pascal to Assembly and back

By Michael Bertrand

Turbo Pascal is a supple language. Still, some problems require an assembly language solution. These problems fall into two broad categories: speed (for example, disk I/O) and direct access to the system (for example, get the system time).

Often we want to combine the flexibility of Turbo Pascal with the power of assembly language, which can be done by executing short assembly language routines within a Turbo Pascal program. Information can also be passed between the Pascal and assembly language programs.

Unlike Microsoft languages, Borland's Turbo Pascal does not create OBJ files that can be linked with modules created by other compilers or assemblers. But Turbo Pascal does provide other ways to integrate assembly language code into a Pascal program. Some of these methods are discussed in this article.

MSDOS and INTR procedures

Turbo Pascal's *MSDOS* procedure is used to execute interrupt 21h, a DOS call. Interrupt 21h provides a collection of basic services that can be used to display characters, read the keyboard, open disk files, and access the machine in other ways. Interrupt 21h is the domain of assembly language programmers, but Turbo Pascal's *MSDOS* procedure makes interrupt 21h available to Turbo Pascal programmers.

In implementing the *MSDOS* procedure, a Turbo Pascal record type and variable are used:

```
type reg = record
      ax,bx,cx,dx,bp,si,di,ds,es,flags:
        integer;
  end;

var registers: reg;
```

The ax, bx, etc., are record fields of type ***integer*** that are given the same names as some of the 8088 registers. Assignments are made to these Turbo Pascal fields, and then *MSDOS* is executed. Values returned by the DOS call in registers are recovered by Turbo Pascal in variables of the same name:

```
registers.ax:=$0200;     {AH=2, AL=0}
msdos(registers)         {INT 21h}
```

Recall that the 8088's AX register is 16 bits wide—the same as Turbo Pascal integers. AX can be broken into a high byte (AH) and a low byte (AL). Similarly, BX = (BH, BL), CX = (CH, CL), and DX = (DH, DL). The value in AH always signifies the DOS service being called. In Turbo Pascal, a dollar sign ($) prefix before a numerical constant means the number is in hexadecimal notation (base 16).

For example, we obtain the system time as follows in assembler:

```
mov ah,2Ch          ;get time service
int 21h             ;DOS call
```

The time is returned in the following registers:

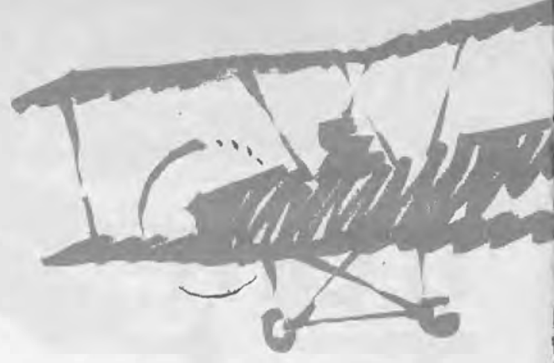
```
CH=hour               (0-23)
CL=minute             (0-59)
DH=second             (0-59)
DL=hundredths/sec     (0-99)
```

There is no built-in way to obtain the system time in Turbo Pascal, but we can get the time by executing the DOS call in Listing 1.

```
procedure get_time(var hour,minute,second,hundredth: byte);
type  regs = record
             ax,bx,cx,dx,bp,di,si,ds,es,flags: integer
           end;
var   registers: regs;

begin
  with registers do
    begin
      ax := $2C00;                    {AH = 2Ch; AL = 00h}
      msdos(registers);               {INT 21h}
      hour :=   hi(cx);               {CH}
      minute := lo(cx);               {CL}
      second := hi(dx);               {DH}
      hundredth := lo(dx)             {DL}
    end {with}
end; {procedure}
```

Listing 1.



The Turbo Pascal *INTR* procedure is used to execute any interrupt—*MSDOS* is a special case for interrupt 21h. The REGISTERS record is used with *INTR*, just as with *MSDOS*, except that the interrupt number must be given:

```
registers.ax :=$0100;    {AH=1,AL=0}
registers.cx :=$0607;    {CH=6,CL=7}
intr($10,registers);     {INT 10h}
```

Any of the BIOS services can be called: video services (INT 10h), disk services (INT 13h), keyboard services (INT 16h), and so on.

For example, we can print the screen from within an assembly language program by executing an interrupt 5h (this interrupt is invoked when <SHIFT> <PrtSc> is pressed at the keyboard):

```
int 5h    ;print the screen
```

The same thing is done within a Turbo Pascal program by:

```
intr($05,registers);
```

In the next example, we invoke one of the video services at interrupt 10h to change the cursor appearance. This is done in assembler as follows:

```
mov ah,01h             ;set cursor service
mov ch,hi_scan_line    ;scan lines: 0-7
                       ;if color/graphics
mov cl,lo_scan_line    ;scan lines: 0-12
                       ;if monochrome
int 10h                ;video services
```

Different shaped cursors result from assigning different values to CH and CL. Some possible assignments for the eight scan lines on a color monitor are shown in Figure 1.

If we wish to turn the cursor off altogether, we assign a greater value to CH than to CL:

```
TURN
CURSOR          hi_scan_line (CH) = 5
OFF:            lo_scan_line (CL) = 3
```

With *INTR*, we can also execute this video BIOS call in Turbo Pascal, as shown in Listing 2.

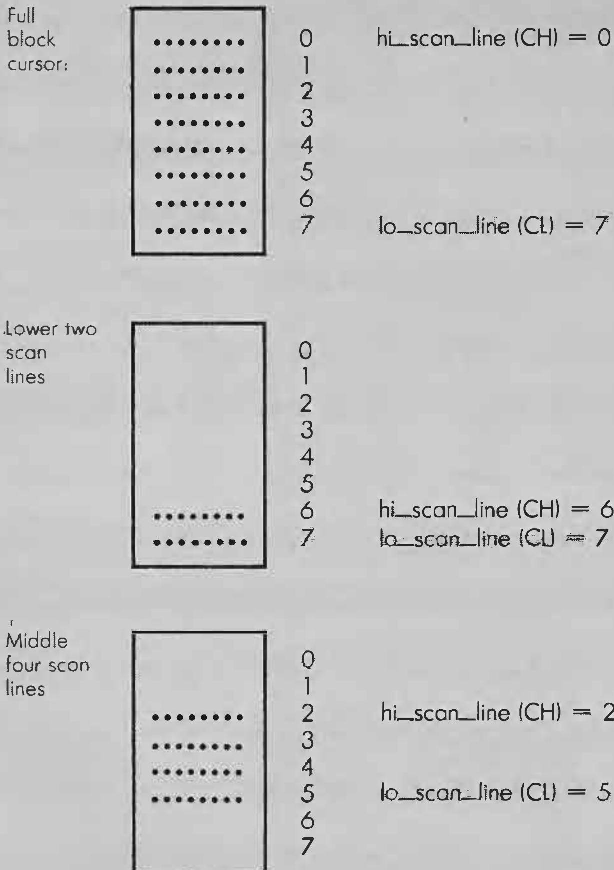


Figure 1.

INLINE statement

Turbo Pascal's *INLINE* statement provides a way to insert machine code directly into a Turbo Pascal program:

```
inline($B4/$02/        {mov ah,2}
       $CD/$21);       {int 21h }
```

$B4/$02 are the two bytes of machine language representing the 8088 instruction *mov ah,2,* and *$CD/$21* is the machine language for *int 21h.* The four bytes are actually inserted into the compiled Pascal program at this point. Turbo Pascal syntax requires that the bytes be separated by a slash mark (/) and that the sequence of bytes be parenthesized.

We programmers think in terms of *mov ah,2,* but *INLINE* requires us to determine the corresponding machine language representation. This can be done with the simple assembler in *DEBUG.* For more complicated assembly language routines, especially those involving labels, LST files can be used. These files, generated by the macro assembler, contain the assembly language source code side-by-side with the corresponding machine language.

A disk file can be read into memory much more quickly in assembler than with Turbo Pascal's *READ*. Listing 3 shows an example of *INLINE* code to open a disk file, read a specified number of bytes into a Turbo Pascal array, and then close the file.

The file is opened with the "open a file" service at interrupt 21h (AH = 3D). This service requires that the address of the file name first be loaded into the DX register:

```
lea dx,file_name
```

Upon return from interrupt 21h, the AX register contains the file handle, a 16-bit quantity that is identified with this file in subsequent operations. This handle must be specified to read or close the file, for example.

Turbo Pascal global variables are declared to hold the file name, handle, and other relevant information (*BUFFER*, the array into which the file will be read; *FILE_SIZE;* and *OK*, which tells whether the file was successfully opened).

The *INLINE* code finds the Turbo Pascal variables by direct reference:

```
$8D/$16/file_name/{lea dx,file_name}
```

An LST file would contain the two-byte *$8D/$16* opposite the *lea* instruction, together with a two-byte address. As shown previously, we simply fill in a Turbo Pascal global variable name (*file_name*).

In effect, the Pascal program sends information to the *INLINE* routines (file name, file handle, and buffer address) and receives information from the *INLINE* routines (file handle, number of bytes actually read, and error status). This two-way transfer of information is possible because the *INLINE* code can locate the addresses at which Turbo Pascal variables are stored.

To make this transfer work, you need some knowledge of how Turbo Pascal stores variables. The name of the file to be opened is read into the Pascal string variable *file_name* with *readln*. As already discussed, the *INLINE* code finds the address of this variable with:

```
$8D/$16/file_name/{lea dx,file_name}
```

However, Turbo Pascal uses the first byte of a string variable to store the number of characters currently assigned to the string variable. The string itself—the file name in this case—is stored starting at the second byte. This is the reason for the next instruction:

```
inc dx
```

Turbo Pascal takes an entire byte to store a Boolean variable, with the information residing in the least significant bit: 1 = TRUE, 0 = FALSE. The disk-related DOS calls signal an error by setting the carry flag. If we find the carry flag set in the *INLINE* routine, we put a 0 in Boolean variable *OK:*

```
mov ok,0
```

OK will then register as FALSE when we return to Turbo Pascal.

Somewhat different methods are needed to find variables other than global variables. The instructions in the *INLINE* example depend on the fact that Turbo Pascal maintains its global variables in the data segment. Different kinds of variables are stored differently:

Kind of variable	Where variable stored
Global	Data segment
Local	Stack segment
Variable parameter	Address in stack segment
Typed constant	Code segment

Turbo Pascal externals

Externals are COM files inserted directly into a Turbo Pascal program. Parameters are passed on the stack. Turbo Pascal views the COM file as a procedure. The external declaration must be followed by the name of the COM file,

```
procedure set_cursor (hi_scan_line, low_scan_line: byte);
type  regs = record
             ax,bx,cx,dx,bp,di,si,ds,es,flags: integer
           end;
var   registers: regs;
      ch,cl:     byte;

begin
  with registers do
    begin
      ax := $0100;            {AH = 01h; AL = 00h}
      ch := hi_scan_line;
      cl := low_scan_line;
      cx := 256*ch + cl;      {CH is high byte of CX}
                              {CL is low byte of CX}
      intr($10,registers)     {INT 10h}
    end {with}
end; {procedure}
```

Listing 2.

```
program read_file_into_buffer;
const max_size = 1000;
var   file_name:               string[50];
      buffer:                  array[1..max_size] of byte;
      file_size, handle:       integer;
      ok:                      boolean;
(*-----------------------------------------------------------------------*)
procedure get_name_open_file;
begin
  clrscr;
  write('ENTER PATH AND NAME OF FILE TO READ: ');
  readln(file_name);
  file_name := file_name + chr(0);    {ASCIIZ format}
  ok := true;                         {TRUE = 01h in memory}

  inline
 ($B0/$00/                   {mov  al,0           ;read only access        }
  $B4/$3D/                   {mov  ah,3Dh         ;open a file service     }
  $8D/$16/file_name/         {lea  dx,file_name   ;DX <-- addr of Turbo var }
  $42/                       {inc  dx             ;go to string data       }
  $CD/$21/                   {int  21h            ;DOS call                }
                             {                    :NOTE:  AX <-- handle    }
  $A3/handle/                {mov  handle,ax                               }
  $73/$05/                   {jnc  instr after next                        }
  $C6/$06/ok/$00)            {mov  ok,0           ;ok = FALSE(00h)         }
end; {procedure}
(*-----------------------------------------------------------------------*)

procedure read_file;
begin
  inline
 ($8B/$1E/handle/            {mov  bx,handle      ;BX <-- handle           }
  $B4/$3F/                   {mov  ah,3Fh         ;read a file service     }
  $8D/$16/buffer/            {lea  dx,buffer      ;DX <-- addr of Turbo var }
  $8B/$0E/file_size/         {mov  cx,file_size   ;# bytes to read         }
  $CD/$21/                   {int  21h            ;DOS call                }
                             {                    ;AX = #bytes actually read}
  $A3/file_size/             {mov  file_size,ax                            }
  $B4/$3E/                   {mov  ah,3E          ;close file service      }
  $CD/$21)                   {int  21h            ;DOS call                }
end; {procedure}
(*-----------------------------------------------------------------------*)
begin  {main}
  get_name_open_file;   {name in file_name, error status in ok}
  if ok then
    begin
      file_size := max_size;
      read_file              {NOTE: file_size has been adjusted}
    end
  else
    write('ERROR')
end.
```

Listing 3.

between quotes, as it appears in the disk directory.

Listing 4 is a Turbo Pascal program to fill the color/graphics screen with a prescribed attribute and character. Both of these parameters are passed to a COM external that does the filling (fast).

FILL.COM must be on the default drive for program *fill_screen* to compile. FILL.ASM, the source code for FILL .COM, is shown in Listing 5. Recall that FILL.ASM must be assembled with the macro assembler, linked, and converted to a COM file with *EXE2BIN* to produce FILL.COM.

The first two and last two instructions of the COM file must be:

```
push bp
mov bp,sp
.........
.........
pop bp
ret 4
```

The operand of *ret* will vary depending on how many bytes are passed on the stack. Here, two integers, or four bytes, are passed, hence *ret 4*.

Procedure *FILL(ATTR, CH)* has two parameters in the Pascal program—*ATTR* is the first parameter and CH is the second. The parameters are found in the COM file relative to the base pointer

```
program fill_screen;
var  attr, ch:  integer;
procedure fill(attr,ch: integer); external 'fill.com';

begin
  write('ENTER ATTRIBUTE, THEN CHARACTER: ');
  read(attr, ch);
  fill(attr, ch)       {NOTE: attr is FIRST parameter}
end.                   {      ch is SECOND parameter }
```

Listing 4.

```
;FILL.ASM -- For use as an assembler external with Turbo Pascal.

code    segment             ;this must be converted to FILL.COM
        assume cs:code      ;TAKES 2 INTEGER PARAMETERS

fill    proc near
        push bp             ;these two statements are necessary
        mov  bp,sp          ;the passed parameters are on the stack
                            ;and BP is used to point to them
                            ;BP, CS, DS, and SS must always be preserved
;-----------------------------------------------------------------------
        push ds             ;old DS must be saved -- it's changed here
        mov  ax,0B800h      ;we are going to poke directly into screen RAM
                            ;assumes color card -- would be 0B000h on mono
        mov  ds,ax          ;DS <-- 0B800h
        mov  al,[bp]+4      ;the 2nd parameter is at [BP]+4 (the character)
        mov  ah,[bp]+6      ;the 1st parameter is at [BP]+6 (the attribute)
        mov  cx,1920        ;number of character/attributes to write
        mov  bx,0           ;BX will point to screen offset
again:  mov  [bx],ax        ;loop to fill the top 24 lines
        inc  bx
        inc  bx             ;increment twice because the mov writes
                                two bytes
        loop again
        pop  ds             ;restore Turbo Pascal's data segment
;-----------------------------------------------------------------------

        pop  bp             ;these two statements are necessary
        ret  4              ;The 4 is required because four bytes were passed on
                            ;the stack as parameters. The 'ret' increments
                            ;the stack pointer (SP) by 2, but we need to inc-
                            ;rement SP by another 4 to get back to Turbo's SP.
fill    endp
code    ends
end
```

Listing 5.

register (BP), which is used to locate information in the stack segment.

BP locates the parameters in the order opposite to which they were passed. The second, or last, parameter (CH) is found at [BP]+4; the first, or next-to-last, parameter (*ATTR*) is at [BP]+6.

The last parameter is always at [BP]+4, but the location of the next-to-last parameter depends on the size of the last parameter. If the last parameter were a *string[3]*, which is four bytes long, for example, then the next-to-last parameter would be at [BP]+8. The second-to-last parameter would be at some still higher address, and so on. The stack of FILL.COM is shown in Figure 2.

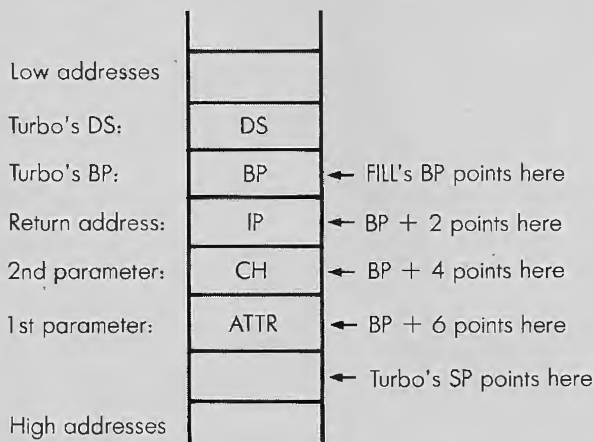
Turbo Pascal allows a program to know the address of any of its variables. If *VAR_NAME* is a Turbo Pascal variable, then *OFS(VAR_NAME)* is the variable's offset address and *SEG (VAR_NAME)* is the variable's segment address.

These addresses are integers and can be passed as integer parameters to an external COM file. The COM file then knows where the Turbo Pascal variable is and can access it freely. As with *INLINE* code, information can be sent in both directions through Turbo Pascal variables.

DEBUG breakpoints

Interrupt 3 generates a *DEBUG* breakpoint. This means that when a program running under *DEBUG* control encounters an INT 3, the program stops execution, the 8088 registers and flags are displayed, and control reverts to the *DEBUG* command line. From here you can single step through a compiled program and examine memory, whether Turbo Pascal variables or stack.

This process is easiest if you first compile the Turbo Pascal program as a COM file and then execute the COM file under *DEBUG* control. To single step through a breakpoint, reset the instruction pointer (IP) to the byte after the INT 3, then resume single stepping (otherwise you never get past the INT 3).



Turbo's BP and the instruction pointer (IP) of the return address are always as shown. Passed parameters are at higher addresses, with the last parameter passed listed first, and so on.

Turbo's data segment (DS) was saved on the stack only because FILL.COM alters it. This value is restored into DS (pop ds) before returning to Turbo.

Figure 2.

INT 3 is coded as a single byte (*$CC*), so this *INLINE* statement will generate a breakpoint in a Turbo Pascal program:

```
inline($CC)
```

This technique is invaluable for debugging. For example, in the COM file external, we can insert a breakpoint just before the call to the external:

```
read(attr, ch);
inline($cc);
fill(attr, ch)
```

If we then run the Turbo Pascal program as a COM file under *DEBUG* control, the Pascal *read* will function as usual and wait for our input. But after pressing *Enter*, control reverts to *DEBUG* and we can then begin single stepping through the program. After a few instructions inserted by the compiler, we come to the external:

```
push bp
mov bp,sp
push ds
```

and so on. The stack can be viewed to resolve any question about the location of parameters. ■

References

Bradley, David J. *Assembly Language Programming for the IBM Personal Computer*. Prentice-Hall, 1984. [A thorough introduction to 8088 assembly language. Includes a number of topics on the IBM PC.]

Lafore, Robert. *Assembly Language Primer for the IBM PC & XT*. Plume/Waite, 1984. [A fine book for beginners. Doesn't assume you already know the subject.]

Norton, Peter. *Programmer's Guide to the IBM PC*. Microsoft, 1985. [A good reference for the INT 21h services, the BIOS services, and other software aspects of the machine.]

Swan, Tom. *Mastering Turbo Pascal*. Hayden, 1986. [A solid survey of Turbo Pascal, including its nonstandard elements. Treats the subject of this article and much more.]

Michael Bertrand teaches mathematics and programming at Madison Area Technical College, Madison, Wis.

Artwork: Steve Campbell