There are many high-level, structured languages for programming today's PCs. Two popular examples are C++ and Pascal. However, assembly language still has its place in today's programming world. Since it mimics the operations of the CPU at the machine level, assembly language lets you get right to the "heart" of your PC.

In fact, there are some tasks that you can do only by using assembly language. While it's true that the Pascal language is capable enough to handle interrupts, it can't be used to pass keyboard input to DOS, for example. Since Pascal has no native way to do this, you must still insert an assembler module routine to perform the function. Likewise, you can't easily remove a high-level resident program from memory. Once again, you have to write the routine in assembly language to do this.

For many applications, programming code must still be as compact as possible. For example, in programming resident programs, each kilobyte of RAM below the 640K boundary is vital. Programs written in high-level languages usually require a runtime library which may add several additional kilobytes to the size. Assembly language programs don't need these bulky library routines.

However, the most important advantage of assembly language is speed. Although high-level languages can be optimized for speed of execution, even the best optimization cannot replace the experience of a programmer. Here's a simple example. Let's say that you want to initialize two variables in Pascal to a zero value. The compiler will generate the following assembly code:

```
xor ax,ax
mov var1,ax
xor ax,ax
mov var2,ax
```

Here, the Pascal compiler optimized the execution speed by using the XOR instruction to zero the ax register (the fastest way to do this) and storing this value as `var1`. However, due to compiler's limitations, the AX register was again zeroed before the second assignment although this was redundant.

For truly time-critical tasks such as sprite movement and high-speed graphics, the only choice may be to use assembly language.

There are two basic ways to do this:

1. Use an internal assembler such as the one built into Borland Pascal and its asm directive.
2. Use a stand-alone assembler such as Turbo Assembler or Microsoft Assembler.

Each way has its own advantages and disadvantages but using the stand-alone assembler is usually the better choice.
Chapter 1

Assembly Language: The True Language Of Programmers

The stand-alone assembler is designed from the ground up for writing full assembly language programs - not as an add-on to a high-level language. A stand-alone assembler has a complete programming environment with many convenient features. For example, it has directives such as “db 20 dup” that makes programming easier. Only a limited number of directives are available from built-in assemblers. Stand-alone assemblers also offer the advantage of macros which speed up assembly language programming tasks.

We’ve chosen to use a stand-alone assembler in this book wherever possible. Of course there are exceptions such as if the assembly language routine module has to access a procedure's local variables as in Borland’s GetSprite and PutSprite procedures.

Multiplication And Division In Assembly Language

Today’s 486 DX4es and Pentiums are fast. These speed demons can perform a multiplication operation in only six clock cycles. This is a far cry from the 100+ cycles that were required using the ancient 8086 processors or about 20 cycles using yesterday’s 286es.

However, if you really want to impress people with fast multiplication, you can use the shift instructions. The number of bits by which you’re shifting corresponds to the exponent of the multiplicand to base 2; to multiply by 16, you would shift 4 bits since 16 equals 2 to the 4th power. The fastest method of multiplying the AX register by 8 is the instruction SHL AX,3 which shifts each bit to a position eight times higher in value.

Conversely, you can perform division by shifting the contents to the right. For example, SHR AX,3 divides the contents of the AX register by 8.

In the early days of computing, numerical analysts suggested other ways to speed up computations. One common technique was to use factoring. For example, multiplication by 320 can be factored like this:

1. Multiplication of the value by 256 (shift by 8 bits)
2. Multiplication of a copy of the value by 64 (shift by 6 bits)
3. Addition of the two results from above

Mathematicians call this factoring according to the distributive law.

Fixed Point Arithmetic

The preceding examples assume that the values you’re working with are integers. But for many applications, it’s not always appropriate or possible to use integers.

In programming graphics, for example, to draw a line on the screen you need to know the slope of the line. Practically speaking, the slope is seldom an integral number. Normally, in such cases, you would use real (Pascal) or float (C) values which are based on floating point representation. Floating point numbers allow a variable number of decimal places. The decimal point can be placed almost anywhere - which gives rise to the term floating point.
Compared to integers, arithmetic using floating point numbers is very slow. Some PCs have math coprocessors that can perform arithmetic directly. However, if the PC doesn't have a coprocessor then the floating point computations must be performed by software. This accounts for the higher computing times for floating point arithmetic.

Working with floating point number in assembly language isn't very easy. So you can use a high-level language for floating point operations or you can write your own routines. Using high-level language operations is not always easy in Pascal, for example, because the four basic arithmetic operations are not declared as Public. Since both of these alternatives options require a considerable amount of effort, let's look at another alternative.

Many application require only a limited amount of computational precision. In other words, they may not really need eleven significant decimal places. For applications where the values have a narrow range, you may be able to use fixed point numbers.

Fixed point numbers consist of two parts:

1. One part specifies the integer portion of the number
2. The other part specifies the decimal (fraction) part of the number

When using fixed point number, you must first set (or fix) the number of decimal places. Let's see how a fixed point number can change by varying the number of decimal places. The fixed portion and decimal portion of 17 and 1 respectively.

By changing the number of decimal places, the value of the fixed point number is changed:

<table>
<thead>
<tr>
<th>Number of decimal places</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of fixed point number</td>
<td>17.1</td>
<td>17.01</td>
<td>17.001</td>
<td>17.0001</td>
</tr>
</tbody>
</table>

So it's important that there be a clear understanding of how many fixed places the fractional portion will represent.

Now for a quick look at how the mathematical signs are used for fixed point numbers. In fixed point notation, the value -100.3 can be divided into two parts: -100 and -3 (using one decimal place). Adding these two together yields the actual rational number. In this example, adding -100 and -0.3, produces a result of -100.3, which achieves our objective.

The most important advantage of working with these numbers is obvious: They consist of two simple integer numbers which are paired in a very simple way. During addition, any overflow of the fractional portion is added to the integer portion. Using this scheme, even a lowly powered 8086 processor can work efficiently and quickly without a coprocessor.

Realizing that the CPU is not set up to handle fixed point operations automatically, we'll have to program a way to perform the arithmetic operations. We'll see one way to do this in the next section. The method is so flexible that you can even perform more complicated operations, such as root determination by approximation, where you'll really notice the speed advantage of fixed point arithmetic.
The four fundamental arithmetic operations

Because they're so close to integer numbers, developing basic arithmetic operations for the fixed point numbers is no big deal. The math instructions are already built into the processor so the remaining consideration is deciding how to work with the paired numbers.

The program in this chapter shows one way of packaging a math library for fixed point numbers. This program implements the four basic arithmetic operations in Pascal. By rewriting the routines in assembly language, you can make the routines fly even faster, but the Pascal example here demonstrates the method.

Addition

The easiest operation is addition. To add two fixed point numbers, you simply add the integer portions and the fractional portions separately.

Here’s the only complicating factor. If the two fractional portions produce a value greater than one, then you have to handle the "overflow". For example, an overflow occurs for fixed point numbers with two decimal places when the two fractional values sum to a value of 100 or higher. In this case, the overflow is handled by adding one to the integer portion and subtracting 100 from the fractional portion. The reverse is true with negative numbers. In this case, you subtract one from the integer portion and add 100 to the fractional portion.

Subtraction

Subtraction is similar to addition, except the two separate portions are subtracted from one another. Overflow is handled in the same way.

Multiplication

A more elaborate method is used for multiplication. First, each factor is converted to a whole number. Next the two factors are multiplied. Then the product is reconverted back to a fixed point value. During the reconversion, the product is adjusted by dividing by the number of decimals since the factors were increased when they were first converted into whole numbers.

Division

Division is performed by a method that parallels multiplication. As in multiplication, you convert the fixed point dividend and the divisor into whole numbers, thereby temporarily eliminating the decimals. Again after the division, the quotient is adjusted by dividing the number of decimals.

The program BASARITH.PAS, listed below, illustrates this technique:

```pascal
Type Fixed=Record {structure of a fixed point number}
  BeforeDec,
  AfterDec:Integer
End;

Var Var1, {sample variables}
  Var2:Fixed;
```
Const AfterDec_Max=100;              {2 places after decimal point}
AfterDec_Places=2;

Function Strg(FNumber:Fixed):String;
{converts a fixed point number to a string}
Var AfterDec_Str,                    {string for forming the fractional part}
    BeforeDec_Str:String;            {string for forming the integral part}
i:Word;
Begin
    If FNumber.AfterDec < 0 Then       {output fractional part without sign}
        FNumber.AfterDec:=-FNumber.AfterDec;
    Str(FNumber.AfterDec:AfterDec_Places,AfterDec_Str);  
        {generate decimal string}
    For i:=0 to AfterDec_Places do     {and replace spaces with 0s}
        If AfterDec_Str[i] = ' ' Then AfterDec_Str[i]:='0';
    Str(FNumber.BeforeDec,BeforeDec_Str);     {generate integral string}
    Strg:=BeforeDec_Str+','+AfterDec_Str;     {combine strings}
End;

Procedure Convert(RNumber:Real;Var FNumber:Fixed);
{converts Real RNumber to fixed point number FNumber}
Begin
    FNumber.BeforeDec:=Trunc(RNumber);   
        {define integral part}
    FNumber.AfterDec:=Trunc(Round(Frac(RNumber) * AfterDec_Max));
        {define fractional part and store as whole number}
End;

Procedure Adjust(Var FNumber:Fixed);
{puts passed fixed point number back in legal format}
Begin
    If FNumber.AfterDec > AfterDec_Max Then Begin
        Dec(FNumber.AfterDec,AfterDec_Max);  {if fractional part overflows to positive}
        Inc(FNumber.BeforeDec);            {reset and decrement integral part}
    End;
    If FNumber.AfterDec < -AfterDec_Max Then Begin
        Inc(FNumber.AfterDec,AfterDec_Max);  {if fractional part overflows to positive}
        Dec(FNumber.BeforeDec);            {reset and increment integral part}
    End;
End;

Procedure Add(Var Sum:Fixed;FNumber1,FNumber2:Fixed);
{Adds FNumber1 and FNumber2 and places result in sum}
Var Result:Fixed;
Begin
    Result.AfterDec:=FNumber1.AfterDec+FNumber2.AfterDec;  
        {add fractional part}
        {add integral part}
    Adjust(Result);                            
        {Put result back in correct format}
    Sum:=Result;
End;

Procedure Sub(Var Difference:Fixed;FNumber1,FNumber2:Fixed);
{Subtracts FNumber1 from FNumber2 and places result in difference}
Var Result:Fixed;
Begin
    Result.AfterDec:=FNumber1.AfterDec-FNumber2.AfterDec;  
        {subtract fractional part}
        {subtract integral part}
    Adjust(Result);
Difference:=Result;
End;

Procedure Mul(Var Product:Fixed;FNumber1,FNumber2:Fixed);
{multiplies FNumber1 and FNumber2 and places result in product}
Var Result:LongInt;
Begin
  Result:=Var1.BeforeDec*AfterDec_Max + Var1.AfterDec;
  {form first factor}
  Result:=Result * (Var2.BeforeDec*AfterDec_Max + Var2.AfterDec);
  {form second factor}
  Result:=Result div AfterDec_Max;
  Product.BeforeDec:=Result div AfterDec_Max;
  {extract integral and fractional parts}
  Product.AfterDec:=Result mod AfterDec_Max;
End;

Procedure Divi(Var Quotient:Fixed;FNumber1,FNumber2:Fixed);
{divides FNumber1 by FNumber2 and places result in quotient}
Var Result:LongInt;           {intermediate result}
Begin
  Result:=FNumber1.BeforeDec*AfterDec_Max + FNumber1.AfterDec;
  {form counter}
  Result:=Result * AfterDec_Max div (FNumber2.BeforeDec*AfterDec_Max+FNumber2.AfterDec);
  {divide by denominator, provide more places beforehand}
  Quotient.BeforeDec:=Result div AfterDec_Max;
  {extract integral and fractional parts}
  Quotient.AfterDec:=Result mod AfterDec_Max;
End;

Begin
  WriteLn;
  Convert(-10.2,Var1);          {load two demo numbers}
  Convert(25.3,Var2);          {some calculations for demonstration purposes:}
  Write(Strg(Var1),'*',Strg(Var2),'= ');
  Mul(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
  Write(Strg(Var1),'-',Strg(Var2),'= ');
  Sub(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
  Write(Strg(Var1),'/',Strg(Var2),'= ');
  Divi(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
  Write(Strg(Var1),'+',Strg(Var2),'= ');
  Add(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
End.

Addition, subtraction, multiplication and division are implemented in the procedures Add, Sub, Mul and Divi respectively. The main program tests each of the operations.

Procedure Adjust makes the decimal adjustments after addition and subtraction. Procedure Convert converts a floating point number to a fixed point number and Strg generates a string out of this fixed point number so it can be displayed on the screen.
Why fixed point numbers? A sample application

The program above demonstrates the simplicity of fixed point numbers. The following example, however, demonstrates there are also practical applications for fixed point numbers.

In this example, we develop a very fast way to calculate the slope of a line. This method is very fast and rivals the Bresenham algorithm.

The procedure used here is based on the simple mathematical definition of a straight line: \( y = mx + b \). The slope, called \( m \), is very important. It indicates the steepness by which a straight line ascends on a segment with a length of 1.

However, because this value is seldom a whole number, you can make excellent use of fixed point arithmetic. The sample procedure \texttt{Line} can draw lines with a slope between 0 and 1; for other slopes, you have to add reflections (see Chapter 7).

This program uses a procedure called \texttt{PutPixel}. Although we’ll discuss \texttt{PutPixel} in more detail in Chapter 3, for now we’ll just note that this procedure sets a pixel at the coordinates \((x,y)\) in mode 13h with the color \texttt{Col}.

You’ll find this line algorithm converted to assembly language on the companion CD-ROM. The assembly language version is called \texttt{LINEFCT.PAS} (the routine uses the Pascal built-in assembler).

Uses \texttt{Crt};

Var \( x \):Word;

Procedure \texttt{PutPixel}(x,y,col:word);assembler;
  (sets pixel \((x/y)\) to color \texttt{col} (Mode 13h))
asm
  mov ax,0a000h        \{load segment\}
  mov es,ax
  mov ax,320           \{Offset = Y*320 + X\}
  mul y
  add ax,x
  mov di,ax
  mov al,byte ptr col
  mov es:[di],al       \{and set pixel\}
End;

Procedure \texttt{Line}(x1,y1,x2,y2,col:Word);assembler;
asm
  \{register used:\}
  bx/cx: Fractional/integer portion of of y-coordinate
  si : fractional portion of increase
  mov si,x1           \{load x with initial value\}
  mov x,si
  sub si,x2           \{and form x-difference \( (in\ si)\)\}
  mov ax,y1           \{load y \( (saved in bx)\) with initial value\}
  mov bx,ax
  sub ax,y2           \{and form y-difference \( (in\ ax)\)\}
  mov cx,100          \{expand y-difference for computing accuracy\}
The main program initializes graphics mode 13h through the BIOS and then draws a line from the coordinates (10,10) to (100,50) in color 1. The Line procedure takes advantage of the fact this algorithm is restricted to slopes smaller than one. This is why no integer portion is required and the fractional part of the slope fits completely in a register (SI here). The y-coordinate, which must also be handled as a decimal number, is also placed in registers. The integer portion is placed in BX and the fractional portion is placed in CX.

The main program then loads the x-coordinate with its initial value (x1) and determines the length of the line in x direction (x1-x2), then repeating the process with y. Next the slope is determined by multiplying the y difference by 100 (two decimal places) to determine the fractional portion, then dividing by the x difference and storing this value in SI.

Within the loop: a dot is drawn at the current coordinates and the position of the next dot is determined. To do this, the program increments the fractional portion of the y-coordinate by the fractional portion of the slope.

If an overflow occurs (i.e. if the sum is greater than 100), the integer portion is incremented by one and the fractional portion is de-incremented by 100. Next the x-coordinate is incremented by 1. The procedure is repeated until the x2 value is reached.
Custom Mathematical Functions

If you use floating point numbers, you can use a language such as Pascal with its many built-in functions. These include sine, cosine, root and many others which make it easier, but not faster, to program mathematical problems. In fact, these math functions are among the slowest in a programming language unless you have a math coprocessor.

Integer numbers are sufficient for many practical programming tasks if the range of values is suitable. But a sine from -1 to 1 doesn’t make much sense with integer values. On the other hand, the Pascal internal functions are quite slow. In fact, when an integer number is used, it is first converted into real number and then operated on using the standard, slow Real procedures. The result is that Pascal integer arithmetic is even slower than their floating point equivalents. To overcome this limitation, there’s only one alternative: Write your own functions.

There are two basic methods for programming a function:

1. Pre-build a table with the result values.
2. Determine the result values by approximation.

Tables

You’re probably familiar with tables from your high school math days. You determine the function value by looking up the corresponding argument in the table.

For use in programs, the same principle applies. At the start of the program, you create the desired function table. The table is then available for fast lookup.

The following simple example generates a table for determining the sine function values. We’ll use this same table later. The TOOLS.PAS unit contains a general procedure for calculating tables called (Sin_Gen):

```pascal
procedure sin_gen(var table:Array of word;period,amplitude,offset:word);
{precalculates a sine table the length of one period.
 It is in the array "table". The height is required
 in the variable "amplitude" and the location of the
 initial point is required in variable "offset"}
Var i:Word;
Begin
 for i:=0 to period-1 do
  table[i]:=round(sin(i*2*pi/period)*amplitude)+offset;
End;
```

First, the array name for the table is passed to this procedure. Next the length of the period of the sine functions is passed. The length corresponds to the number of table entries since exactly one period is always calculated. The amplitude specifies the highest value. With an amplitude of 30, for example, the table would contain values from -30 to +30. The last value is the offset, which specifies the shift of the sine function in y-direction. In our example above, an offset of 10 would build a table with values from -20 to 40. Now the program iterates from the first to the last entry of the table and calculates the corresponding values using the regular sine function of Pascal.
To test the sine table, our next program draws circles. We'll use text mode to keep the program simple. The SINTEST.PAS program draws 26 overlapping circles two times. The circles are first drawn using the standard sine and cosine functions. Then the circles are drawn a second time using the tables. The math coprocessor is switched off so we can evaluate the results of the table lookup method. Run the program and you'll notice the difference in speed.

```pascal
{$N-}                           {Coprocessor off}
Uses Crt, Tools;
Var phi,                        {Angle}
x, y: Word;                   {Coordinates}
Character: Byte;             {Used character}
Sine: Array[1..360] of Word; {receives the sine table}
Procedure Sine_Real;            {draws a circle 26 times}
Begin
  For Character := Ord('A') to Ord('Z') do       {26 passes}
    For phi := 1 to 360 do Begin
      x := Trunc(Round(Sin(phi/180*pi)*20+40)); {calculate x-coordinate}
      y := Trunc(Round(Cos(phi/180*pi)*10+12)); {calculate y-coordinate}
      mem[$b800:y*160+x*2] := Character;        {characters on the screen}
    End;
  End;
End;
Procedure Sine_new;             {draws a circle 26 times}
Begin
  For Character := Ord('A') to Ord('Z') do       {26 passes}
    For phi := 1 to 360 do Begin
      x := Sine[phi]+40;          {calculate x-coordinate}
      If phi <= 270 Then          {Cosine as shifted sine}
        y := Sine[phi+90] div 2 + 12
      Else
        y := Sine[phi-270] div 2 + 12;
      End;
      mem[$b800:y*160+x*2] := Character;        {characters on the screen}
    End;
  End;
Begin
  Sin_Gen(Sine, 360, 20, 0);       {prepare sine table}
  CrlScr;                       {clear screen}
  Sine_real;                    {draw circles}
  CrlScr;                       {clear screen}
  Sine_new;                     {clear screen}
End.
```

The main program first builds the sine table. Next, the program calls the two procedures for drawing the circles. The first procedure is **Sine_real**. It calculates the coordinates at the current angle using the built-in sine and cosine functions. Both functions require the angle in radian measure, therefore /180 x pi.

For the radius, the program sets 20 in x-direction and 10 in y-direction. This places the circle in the middle of the image (+40, +12). Finally, the program displays the character using direct video memory access.

The second procedure is **Sine_new**. It takes values for x and y from the table. The cosine is formed by a 90 degree phase displaced sine but has to watch out for the end of the table. This procedure is several times faster, which you'll notice when you start the program.
Approximation

Tables are perfect for functions if the range of values can be predetermined in advance, such as the sine. However, this isn’t always possible for functions like the root, which may take on an infinite range of values. To handle a wider range of values, you can reduce the resolution of the table but this in turn lowers the computing accuracy.

Alternatively, you can compute the value of a function by approximation. A typical math book presents the formula for the root function as follows:

\[ X_{n+1} = \frac{1}{2}(X_n + \frac{a}{X_n}) \]

If you use the number from whose root you want to find as the radicand, and a random initial value for \( X_n \) (for example, 1), you get a value that approximates the desired result. Repeat the process using this number again as \( X_n \) to get an even more precise value. You can continue until you’re satisfied with the accuracy of the result. This is the case for whole numbers when the current result deviates from the previous result by 0 or 1. A difference of 1 is permissible, otherwise the calculation might never end. For example, the calculation will never end when the result always jumps between two adjacent values due to rounding.

This algorithm, by the way, is self-correcting. This is especially important for calculations done “by hand”: If a result is false, and it is used in the next step as the initial value for \( X_n \), the algorithm uses the false value for the approximation. Although this extends the arithmetic operation, you’ll still get the correct solution.

This example is in the ROOT.ASM file. We did not store this procedure in a unit because we’ll need it later as a near procedure. A far procedure, such as a unit would generate, would be too slow to call. The assembly language text contains two procedures: One procedure is `Root` and contains the actual calculation. Although this extends the arithmetic operation, you’ll still get the correct solution.

This file also contains a “frame” function (`Rootfct`). This lets you access the root directly from Pascal when time is not so critical. This “frame” function (`Rootfct`) is passed as a parameter to the radicand and returns the root value as a function result after `Root` is called.

```
.286 ;enable 286 commands at least
e equ db 66h ;operand size prefix (32 bit commands)
w equ word ptr
code segment public
assume cs:code
public root
public rootfct
root proc pascal ;result in ax (function)
e
   xor si,si ;clear intermediate result (in esi)
   db 66h,0fh,0ach,0d3h,10h ;shrd ebx,edx,16d - dx to ebx (upper 16 bits)
   mov bx,ax ;ax to ebx (down) - dx:ax now in ebx
e
   xor dx,dx ;clear edx
e
   mov cx,bx ;store initial value in ecx
```
Notice the "e" listed in several lines of the `root` procedure. At each occurrence of e, the value 66h is inserted in the code. It represents the Operand-Size-Prefix of the 386, which extends the instruction following it to 32 bits. 32 bit instructions result in a large increase in speed because the LongInt results no longer need to be split into two registers. Unfortunately, Pascal compilers still cannot process these instructions directly. This is true even from a stand-alone assembler. So, the only option is to change each instruction to 32 bit "manually" as we've done above.

First, the 386 instruction `shrd` shifts the contents of the register to the upper EBX half and then loads the lower half with AX. ECX serves as storage for the radicand a, also reused later. The loop performs the steps described in the formula: After dividing the radicand by the last approximate value (in EBX) it's added to the quotient. This completes the calculations within the parenthesis. Next, the value is divided by 2 which is compared to the result of the previous one. The iteration ends if the results matches (maximum deviation of 1). Otherwise, the new value Xn is loaded (in the BX register) and the next iteration is performed. Finally, the root in the AX register can be stored by a Pascal function.

We'll use another speed comparison as an example:

```
/rootfct proc pascal a:dword ;translates procedure to Pascal function
    mov ax,word ptr a ;write parameters to register
    mov cx, a+2
    call root ;and extract root
    ret
/rootfct endp
```

You can find `ROOTTEST.PAS` on the companion CD-ROM.
This program, called ROOTTEST.PAS, calculates the root of 87654321. It repeats this 10000 times in two different ways. After startup, even a 486 (with disabled math coprocessor, compiler switch $n-)$ will require a few seconds to compute the results. On the other hand, the second part of the program (custom calculation) is processed in fractions of a second.

High Speed Tuning: Optimizing Comparisons

Next to arithmetic operations, comparisons are the most time consuming tasks that a processor performs. That’s why you should use them only when necessary and then optimize them as much as you can.

OR instead of CMP

The logical operations of the processor offer one simple way to increase speed. For example, the TEST instruction basically uses AND. So, you can use this instruction to check for specific bit combinations. If you’re comparing with 0, you can speed things up even more by using OR.

For example, if register AL contains 0, then the instruction OR AL,A sets the Zero flag, otherwise it returns a cleared Zero flag. You can then use either JZ or JNZ to branch.

Since this instruction sets all the flags to values that correspond to the contents of the register, you can also check a number’s sign, for example: The JS instruction branches to the specified address when the sign is negative.
String comparisons

There are many instances when string comparisons are important. One example is in programming a TSR program that must determine whether it is already resident in memory. To understand the comparison, the effect of the JCXZ instruction is more important than anything else. This instruction jumps to the specified address when the CX register contains 0. Programming a string comparison with the repeat command REPE CMPSB is quite easy:

First, load the pointers to the two strings into the ES:DI and DS:SI register pairs. Next load the length into register CX. Finally, execute the REPE CMPSB instruction, which repeatedly compares the registers until the strings show a difference (when the Zero-Flag is cleared the REPE completes) or until the end of the string is reached (and CX=0). The JCXZ instruction now picks up this small difference. CX is not 0 with variable length strings so the program doesn’t branch. Only with same length strings does CX reach 0 and the program branches.

Variables In Assembly Language

In assembly language, you should always try to keep as many values as possible in the registers since the processor can access these values faster than others. Don’t be afraid to use registers for special tasks (SI, DI, BP) or use them as normal variable storage. However, even with the most clever use of the register set, you still won’t have enough registers. In such cases, you must save these values in memory as you’re forced to rely on normal variables.

Accessing Pascal variables

It’s easy to access a Pascal variable from assembly language. While you can use more complicated constructs such as mov AX,[offset variable], it’s easier to use mov AX,variable. To perform a type conversion at the same time (e.g., pointer offset to 16 bit register), you have to add a Word ptr or Byte ptr: mov AX,word ptr Pointer + 2.

Accessing arrays and records

Although you can address arrays directly from the assembly language, you have to perform the indexing yourself. Most importantly, determine the size of each elements in the array; individual elements have a length of 2 bytes for Word entries or a length of 4 bytes for Doubleword entries. It’s also possible to have other offsets, for example, with an Array of Record. The 386 can handle these offsets; it can address variables in the form mov AX,[2*ecx]. However, in Pascal (only 286 code!), this is quite difficult to achieve because each such instruction must be stored as a complete sequence of bytes. That’s why it’s better to determine the offset through multiplication using the shl SI,1 instruction.

With most assemblers, you can also specify the offset of the array in the normal form before the index: mov AX,word ptr Arr[SI], the assembler converts this instruction to: mov AX,word ptr [SI+offset Arr]. You no longer need to specify records by using constant offsets. Now, you can access the records directly from Pascal-ASM, as you would from Pascal:

`mov AX,word ptr rec.a.`
TASM and MASM also have a variable similar to records. This variable is called a *structure*. A structure is identical to a Pascal record, allowing you to access it as if from Pascal:

```assembly
data segment

rec_typ struc
  a dw ?
  b db ?
rec_typ ends

extrn rec:rec_typ
```

### Code segment variables

Unfortunately, programmers always seem to run out of registers which is why they’re excited about each additional register they gain. The BP register is available and as accessible as any other register except for one small catch: BP is used to address local variables of the procedure. So, you either have to do without local variables or store them elsewhere when you use the BP register. Global variables are also usually inaccessible, especially in graphic procedures, because the DS register no longer points to the data segment, but instead, points to something else such as sprite data. Your only option in this case is to use code segment variables.

These variables are located in the current code segment with the program code and are addressed at machine language level by the segment override prefix. However, since the assembler takes over at programming level, you probably won’t notice any peculiarities. The drawback is that you can’t access this routine from other procedure or modules. You can simply create the variables in the code segment from TASM and MASM instead of the data segment. The assembler then takes care of correct addressing automatically.

On the other hand, Pascal introduces a complicating factor. Normally, using Pascal you can’t fill the code segment with data from outside of the procedures or functions. However, you can use a little trick: At the beginning of the procedure, you can insert a short routine such as the one below which set the values of the variables. Here’s what that looks like:

```assembly
Procedure Test;assembler;
asm
  jmp @los
  @Var1: dw 0
  @Var2: db 0
@los:
  {Rest of procedure}
End;
```

Remember to add a word ptr or byte ptr to access these variables, because Pascal considers `@Var1` and `@Var2` to be labels and not variables.

### Circular arrays

Arrays aren’t always processed from front to back. They’re often processed in a circular fashion: from front to back and then from the front again. Using the sine table an example again, you may need to find the sine for an angle of 700 degrees. The easiest method for solving this problem is to check the range of arguments
and bring that argument back into the correct range when the end of the table is exceeded. In this example, 360 degrees is subtracted from the original 700 degrees and the resulting 340 is used as new argument.

You can optimize the array form by redesigning the array so the number of entries corresponds to a power of 2, i.e., 32, 64, 128, etc. For these cases, you can determine the index into the array by simple bit masking using the AND instruction. For example, if an array has 64 entries (0-63), each index is ANDed with 63, causing the upper two bits of an eight bit argument to be hidden. Only the lower six bits remain significant. To design such an array, there must be the right number of elements. For example, you can specify a period of 64 when generating the sine.

**Bit mask rotation**

Bit masking is used frequently in system programming. In bit masking, a value is written to a specific register, for example to a VGA card where each bit has a specific task, such as switching on a pixel in the appropriate bit plane. Each plane is selected in order: Planes 0, 1, 2 and 3 and then plane 0 again, by setting bits 0, 1, 2, 3 and then 0 again. In this example, the goal is to process all four bit planes in order and then get back to the original bit plane. How can be get back to bit plane 0 after bit plane 3?

You can select the desired bit plane by using a register. For example, loading a register with the value 01h sets bit 0; this selects bit plane 0. Rotating to the left one bit at a time sets bit 1 to select bit plane 1, bit 2 for bit plane 2, and bit 3 for bit plane 3.

Since you can’t rotate a half-byte (a 4 bit nibble) directly, you can use a little trick. Instead of loading the register for selecting the bit plane with 01h, we use the value 11h which places identical values in both the upper and lower nibbles of the register. Rotate in the same manner but use only the lower nibble for masking and you’ll get the desired effect. After four rotations, the contents of the register is 88h. Rotate left again and you get the original 11h (bit 7 after bit 0 and bit 3 after bit 4) so you’re back where you want to be.

**Masking a specific number of bits**

Sometimes only a specific number of bits need to be selected from a word or byte. We’ll see an example of this in "The GIF Image Format" section of Chapter 3 when we talk about the GIF Loader.

One way of isolating these significant bits is by masking the values. Load a register with 01h, shift this register to the left by the number of bits to be kept and reduce this value by 1. The result is a mask in which the desired bit positions contain 1 and all others contain 0.

Here’s the simple formula:

\[ \text{Mask} := (1 \text{ shl } \text{Number}) - 1. \]

For example, to select bit 6, you would use a mask of 63 (1 SHL 6 -1 = 63), with bits 0-5 set and bits 6 and 7 cleared. Now all you have to do is AND the byte to be masked with this value and you’ve retained the bits you need.

The SHR and SHL instructions on the 386 and above have a curious feature. It’s only possible to shift a maximum of 31 bits, regardless of the register width that is used. For example, to shift AX by 34 bits, (the same as clearing them since AX is only 16 bits wide), you would execute SHL AX,34d, but in reality, there would only be a shift of 2 bits.
This isn't normally important. However, it did frustrate us once for several minutes because we assumed that using a value greater than 31 bits for shifting would clear the register.

**Mysterious Interrupts**

Although interrupts and interrupt programming can provide versatility to your programming, they can also be a mystery for many new users. This may be due to the number of times the system crashes when new programmers start to experiment with interrupts. However, with a little basic knowledge and a few examples which we'll provide, you can quickly learn how to work confidently with interrupts. You may even be comforted to know that crashes happen even to the most experienced programmers.

There are two types of interrupts:

1. **Software interrupts**
   These are triggered by the INT instruction and can be compared to simple subroutines.

2. **Hardware interrupts**
   These are sent to the CPU from external devices through the two interrupt controllers. For example, a keystroke triggers an interrupt that tells the processor to run a program called the interrupt handler which then accepts and processes the character typed at the keyboard.

**Changing vectors**

A programmer can easily add his or her own program to handle these interrupts. For example, you can write your own program to handle the keyboard interrupt that also outputs a "click" from the loudspeaker each time a key is pressed. How do you do this?

First, the background...DOS defines various interrupts. The keyboard interrupt is an example. Each type of interrupt is identified by a number - the interrupt number. And for each type of interrupt, there is a corresponding program routine that runs and handles the processing associated with that interrupt.

The program’s main memory address is called a vector. In low memory, there is a large vector table containing the addresses of all the interrupt handlers.

You can determine the address of an interrupt handler by using DOS functions 35h. Pass the interrupt number in the AL register and the vector is returned in register pair ES:BX.

To change a vector, you can use DOS function 25h. Pass the interrupt number in the AL register and the address of the new vector in the DS:DX register pair.

For example, to determine the vector for interrupt 9 which handles the keyboard interrupt, you would do the following:

```
mov ax,3509h ;Function and interrupt number
int 21h     ;Execute Dos function
```

The vector is returned in es:bx. The following instructions are used to set a new interrupt handler:
If you're changing an interrupt vector to point to one of your own routines, you should save the original vector. You may need to call the original handler after your processing or, in the case of a TSR, when removing it from memory.

**Calling the old handler and exiting**

In the example of the keyboard click, it doesn't make much sense to only click when a key is pressed; you'll probably also want to output a character to the screen. To do this easily, you can call the original handler before or after making the click - that is unless you want to write a custom keyboard driver.

By saving the original interrupt vector, you can then jump to this destination using a far call. But before doing so, you must simulate an interrupt call. The only special requirement of an interrupt call compared to an ordinary far call is saving the processor flag, which is easily duplicated using the pushf instruction.

The following is how the complete call should appear:

```
pushf
   call dword ptr [OldVector]
```

The original vector was saved in the `OldVector` pointer.

Use the IRET instruction to exit an interrupt. However, remember to first restore the original state of the processor register. After all, the interrupt may have been triggered in the middle of a routine that depends on specific registers.

**Disabling interrupts**

The CLI instruction is used to disable interrupts. This instruction can be used to "lock" the processor from further interrupts. When a program has issued the CLI instruction, no further interrupts are accepted by the processor until the STI instruction reenables them.

Sometimes, however, you may want to disable only specific interrupts and leave the others enabled. To do this, you have to reprogram the *interrupt controllers*. These controllers use a different counting method than the vectors: Hardware interrupts are numbered 0-7 (interrupt controller 1) and 8-15 (interrupt controller 2). In this case, we talk about IRQ (interrupt request) 0-15, while the label "Interrupt" refers to the number of vectors.

Controller 1 presents IRQ 0-7 to the CPU as interrupts 8-0fh.

Controller 2 presents IRQ 8-15 to the CPU as interrupts 70h-77h.

The two controllers are linked (cascaded) to each other using IRQ 2, that is, if controller 1 gets this interrupt request, it passes it to controller 2.

The following shows the layout of the controllers:
Because the interrupts are in hierarchical order, IRQs with lower numbers have a higher priority and are
given preference over IRQs with higher numbers. Although you can change this order by reprogramming
the controllers, we recommend leaving the order as is because both the BIOS and DOS depend on this
structure.

The controllers have IMRs (interrupt mask registers) which can be used to hide or mask specific interrupts.
The IMR of the first controller is located at port address 21h, while the IMR of the second controller is located
at port 0a1h. For both ports, a corresponding set bit indicates the interrupt is disabled.

For example, to disable the real time clock, use the following instructions:

```assembly
in al,0a1h                      ;Load IMR 2
or al,01h                       ;Set bit 0
out 0a1h,al                     ;and write back
```

Both controllers have a second port address at 20h or 0a0h, from which the instructions are given. The most
important is the EoI (End of Interrupt) command (numbered 20h). This instruction indicates the end of the
interrupt handler and frees up the corresponding controller for the next interrupt. If you always jump to
the original vector at the conclusion of your custom interrupt handler, the EoI instruction takes care of this
for you. However, if you write a new custom interrupt handler, it’s up to you to see to it that at the end of
the handler, the EoI command (20h) is written to either port 20h or port a0h:

```assembly
mov al,20h
out 20h,al
```

**Reentering DOS**

An interrupt handler can last a few clock cycles (keyboard click) or several seconds (e.g., Print-Screen, Int
5), depending on the application. It’s important, especially in the latter case, to prevent this handler from
processing a second identical interrupt. With Print-Screen, for example, this might result in two copies of
the printout or even a system crash. The cause is the new second interruption accessing variables which the
handler is already using.
The easiest way to prevent this is to use a flag variable to indicate that the handler is already started. When there is renewed activity, you can simply check the flag variable to avoid reentrance.

A more complicated case of reentrance concerns larger TSR for example, which enable a complete program at the press of a key. The problem in this case is with DOS, which doesn't allow you to enable several DOS functions simultaneously. If an interrupt interrupts a DOS function and then calls other DOS functions (e.g., for screen display), the call destroys the DOS stack, so the computer crashes after processing the handler and returning to the interrupted DOS function.

It isn't easy to catch this. First, you have to check the InDos-Flag. You can determine its memory location prior to installation of the handler using the undocumented DOS function 34h, which returns a far pointer in registers ES:BX. The only time the computer can branch to a handler with DOS functions is when this flag contains the value of 0 at the time the handler is called.

You should also install a handler for interrupt 28h, which is constantly being called while COMMAND.COM waits for user input at the command line. In this case, the InDos-flag contains the value 1, because COMMAND.COM itself counts as a DOS function.

Naturally, you can save yourself the trouble of these complicated measures if you don't call any DOS functions in the handler. This is not a problem for most TSRs.

**Intercepting CRTL-C and reset**

Most commercial programs are written to be "bulletproof" - it's supposed to be impossible to exit these programs through a "back door". If you manage to exit the program through a back door, you risk losing data by leaving files still open. After all, it doesn't look very professional to allow a user to abort the program by pressing \( \text{Ctrl} + \text{Break} \) or \( \text{Ctrl} + \text{Alt} + \text{Del} \).

What is the safest way to intercept these BIOS functions?

DOS has a somewhat safe, although not always reliable, method available for \( \text{Ctrl} + \text{C} \) or \( \text{Ctrl} + \text{Break} \). When you press one of these combinations, DOS calls interrupt 23h, which then causes a program crash. You can change its vector to your own routine, which simply returns to the caller. The disadvantage of this method is that it doesn't always work, especially with \( \text{Ctrl} + \text{Break} \). Given the right circumstances, it could even lead to a system crash.

Here's a method that is much safer which also intercepts a reset \( \text{Ctrl} + \text{Alt} + \text{Del} \): First, a separate keyboard interrupt handler checks to see whether one of the critical key combinations has been pressed before calling the original handler. If one of these combinations has been pressed, the handler terminates. Acceptable key are passed on to the original handler, which then continues by passing them through to the main program.

This technique is shown in the NO_RST.ASM program. Assemble this into an EXE file with TASM or MASM:

You can find NO_RST.ASM on the companion CD-ROM
Chapter 1

Assembly Language: The True Language Of Programmers

data segment public
start_message: db 'reset no longer possible',0dh,0ah,'$'
buffer: db 40d ; length of input buffer
   db 40 dup (0) ; buffer
old_int9 dd 0 ; old interrupt handler
data ends

code segment public
assume cs:code,ds:data
handler9 proc near ; new interrupt 9 handler
   push ax ; store used register
   push bx
   push ds
   push es
   mov ax,data ; load ds
   mov ds,ax
   in al,60h ; read characters from keyboard in al
   xor bx,bx ; es to segment 0
   mov es,bx
   mov bl,byte ptr es:[417h] ; load keyboard status in bl
   cmp al,83d ; scan code of Del key ?
   jne no_reset ; no, then no reset
   and bl,0ch ; mask Ctrl and Alt
   cmp bl,0ch ; both pressed ?
   jne no_reset ; no, then no reset
   block: ; reset or break, so block
      mov al,20h ; send EoI to interrupt controller
      out 20h,al
      jmp finished ; and exit interrupt
   no_reset: ; no reset, now check Break
      cmp al,224d ; extended key ?
      je poss_Break ; yes -> Break possibly triggered
      cmp al,46d ; 'C' key ?
      jne legal ; no -> legal key
   poss_Break: ; test keyboard status for Ctrl
      test bl,4 ; pressed, then block
      jne block
   legal: ; legal key -> call old handler
      pushf
      call dword ptr [old_int9] ; call original handler
      finished:
      pop es
      pop ds ; get back register
      pop bx
      pop ax
      iret
handler9 endp

start proc near
   mov ax,data ; load ds
   mov ds,ax
   mov dx,offset start_message ; load dx with offset of message
   mov ah,09h ; output message
start proc near
The main program (Start) displays a short message, determines the original vector for keyboard interrupt 9 and sets the new vector to the procedure Handler9. Next, the program calls the DOS character input as a substitute for the program segment that is being protected (e.g., the demo routines). The DOS character input receives a 40 character string. Finally, the original handler is restored and the program ends.

Now when a key is pressed the handler itself is called. It first saves all registers used so the interrupted program doesn’t notice any of the handler’s activities. Then the pressed key’s scan code is determined (placed in AL) from the data port of the keyboard controller and the status of the Ctrl and Alt keys is read out (placed in BL) through the keyboard status variable.

The keyboard status variable is located at address 0:417h. The following table shows its layout:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Bit</th>
<th>Meaning</th>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Ins</td>
<td>6</td>
<td>Caps Lock</td>
<td>5</td>
<td>Num Lock</td>
</tr>
<tr>
<td>4</td>
<td>Scroll Lock</td>
<td>3</td>
<td>Alt</td>
<td>2</td>
<td>Ctrl</td>
</tr>
<tr>
<td>1</td>
<td>Shift left</td>
<td>0</td>
<td>Shift right</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First, the program checks whether the Ctrl key was pressed (pointer to Reset) and then checks whether Ctrl and Alt (Bit 2 & 3 in BL) are set. If the answer to both questions is yes, the program continues at the label.
Block. At this point, the program simply sends an EoI signal to interrupt controller 1 and jumps to the end of the handler.

If there was no reset, the program checks for Ctrl + Break and Ctrl + C starting with the label No_reset. If neither C (scan code 46) nor an enhanced key (scan code 224) has been pressed, we can assume that an acceptable key has been pressed, and the program calls the original handler at the label legal and then terminates. If either Break or C has been pressed, the program checks for the Ctrl key. If this key is set, the program ignores the reset, otherwise, it is an acceptable key.

To use this routine, all you have to do is call your own main procedure instead of the DOS character input.

**Tips On Programming Loops**

There are several ways to optimize machine language programs even in simple areas such as programming loops. This begins with the typical construct of a loop called a loop label. It seems that CPU developers have forgotten this instruction in recent years. For a 386, a construct such as dec CX, jne label is 10% faster, while the same construct on a 486 is about 40% faster. This construct is faster although one additional byte had to be fetched from slow RAM in the last instruction sequence. So the Loop instruction should be used only if decrementing CX doesn’t affect the flags, for example, with complicated string comparisons that cannot be resolved with REP CMPSB.

**The direction flag with string instructions**

A frequent source of errors while using string instructions (lodsb, cmpsb, etc.), which are basically loops, is the direction flag, which specifies the direction in which the string is processed. This flag is usually cleared. However, if you somehow set this flag in your program to process a string from back to front, always remember to clear it again.

**Nesting**

There’s always a trade off between speed and the number of registers used in nested loops. Use as many registers as possible for loop counters before using memory variables. Clever choice of loop limits can increase speed execution. For example, by counting backwards, you can determine the end of a loop by checking the zero flag when a register reaches zero.

**16/32 bit accesses**

To minimize the number of memory accesses, use 16-bit or 32-bit instructions. Starting with the 386, even a one byte access by the CPU is executed as a double word. You’ll benefit since it takes even take longer to move a single byte than it does to move a single double word.

Some tasks will still require 8-bit instructions. VGA cards, for example, don’t like it when you access video memory wider than 8 bits in plane-based mode (such as mode X, which we’ll explain later), because the internal plane registers (latches) are only 8 bits wide.
Chapter 1
Assembly Language: The True Language Of Programmers

Practical 386 Instructions

In addition to the 386's basic features (Virtual Mode, Paging, 32-Bit-Register), several other very useful instructions are available. Since some of these instructions combine several 286 instructions, they can increase processing speed tremendously in critical areas. Take advantage of these instructions, even in Real Mode.

The MOVSX and MOVZX instructions

First, are the MOVSX and MOVZX instructions. Both can move an 8 bit register directly to a word register and a 16-bit register to a 32-bit register, which usually requires two instructions to accomplish. The letters "S" and "Z" in these instructions represent "signed" and "zero" and apply to the upper half of the destination register. All bits in the destination register are filled with either 0 or 1 with MOVSX, depending on the signs of the source register, so the original signs are preserved. MOVZX, on the other hand, clears the upper half of the destination register.

For example, if BL contains -1 (ffh), the two instructions will produce the following results:

```assembly
movzx ax,bl  ; ax now contains 255 (00ffh)
movsx ax,bl  ; ax now contains -1  (ffffh)
```

Different SET commands

It's also possible to optimize comparisons on a 386. The 386 can handle the 30 SETxx instructions, which are a combination of CMP, conditional jump and MOV. Each conditional jump has a counterpart in a set instruction (SETz, SETnz, SETs, etc.). If the condition applies, the associated byte operand is set to 1, otherwise it's set to 0:

```assembly
dec cx                          ;Decrease (loop) counter
sete al                         ;use al as flag
```

In this example, which could have been taken from a loop, AL is normally (CX > 0) set to 0. AL isn’t set to 1 until the end, when CX becomes 0. In this way, even Pascal Boolean variables can be set directly in accordance with an assembly condition (SETxx byte ptr Variable).

Fast multiplication and division: SHRD and SHRL instructions

The 386 can perform arithmetic operations directly in 32 bit registers (in particular, multiplication and division operations), which is much faster than the conventional method using DX:AX. How do you get numbers in DX:AX format into an extended register (e.g., EAX)?

Unfortunately, you cannot directly address the upper halves of these register. Once again, however, the 386 has specific instructions for this purpose which do more than load registers: SHLD and SHRL, the enhanced Shift instructions.

In addition to the number of bits to be shifted, these instructions expect two operands instead of one. First, the instruction shifts the first (destination) operand by the corresponding number of bits; but instead of filling the vacated bits at the low order (shift left) or the high order (shift right) with 0, they are filled from the rotated second (source) operand. However, this operand itself is not changed.
For example, if AX contains 3 (0000 0000 0000 0011b) and BX contains 23 (0000 0000 0001 0111b), the instruction SHRD BX,AX,3 first rotates BX to the right by three (=2). However, at the same time the high order is filled with the bits from AX, so the result in the destination operand amounts to (BX) 0110 0000 0000 0010b = 6002h = 24578.

As we said, these instructions are used most frequently for loading 32 bit registers (EBX here) from two 16 bit registers (DX:AX in our example). This is done first when SHRD loads the upper half: SHRD EBX,EDX,16d. This instruction moves DX “from the top” into the EBX register. Then, the lower half is loaded with the desired value, while the upper half, which has already been set, remains unchanged: MOV BX,AX. By the way, this method is also used in the Root procedure we described in this chapter.

Enhanced multiplication with IMUL

You can also use the new multiplication instructions. Starting with the 386, you can multiply practically any register by any value: IMUL DX,3 which multiplies DX by 3. You can also use IMUL AX,DX,3 to multiply DX by 3 and place the result in AX. Unlike the earlier forms of IMUL, you can save a lot of extra coding by using these new instructions.

Using 386 instructions in Pascal programs

All 386 instructions have one common problem: Borland Pascal is currently unable to process them either in an internal assembler or through linked external programs (if an object code is linked by the $L-Directive, the processor specification used there must match the one set in Pascal).

So your only option is to trick the compiler by linking the inline assembler. You do this by calling the Turbo debugger and entering the desired command there in its final form. The debugger then shows the hex code for this instruction. Write down this code and insert it in the program after a db directive, for example:

```pascal
db 66h,0fh,0ach,0d3h,10h ;shrd ebx,edx,16d
```

However, changes such as this are no longer easy. You either have to check the inner structure (for instance, in this example, converting the 16d (10h) operand into 8 by overwriting the last instruction byte with 8 wouldn’t be a problem), or you have to reassemble the appropriate instruction by hand using Turbo Debugger.

Perhaps the best alternative is to wait and hope Borland soon realizes the 386 has become a standard, and as such, deserves to be supported.
In this chapter we’ll describe a practical application of assembly language programming. We’ll use examples to show you what you can do with your PC by programming in assembly language. Here we’ll show you how to program the parallel and serial interfaces, your PC speakers (for samples too!) as well as TSRs.

The Parallel Port

For most users, only the screen and the keyboard are more important than the parallel port. The reason the parallel port is so important is that it connects your PC to your printer. Since the BIOS provides excellent support for the parallel port, you usually don’t have to rewrite the BIOS routines.

However, the parallel port can do much more for you. For example, you can use it to transfer data to other computers, or by using a few electronic components, you can even use it as a sound card. The parallel port has a 25 pin sub-D connector. The table on the left shows its pin layout.

Programming the parallel interface

Every parallel port has three registers which are located at adjacent addresses. The base address determines the port addresses. The base address is usually 378h for the first parallel port and 278h for the second parallel port. However, these values can vary between computers. For example, a Hercules card with an integrated parallel port fits in as the first interface at port 3BCh. If you don’t know which port to use, the word entries for LPT1, LPT2, etc., which specify the base address, are found starting at the address of 0:0408h.

The data register is located at the base address. The data to be output to the port is written to this write-only register. For each 1 bit written to this port, the corresponding data line is set to High and for each 0, the data line is set to Low.

The status register is located at the next address. You determine the printer’s status from this register. The corresponding control lines of the printer cable are written to this read-only register (only BUSY appears inverted in the register).
Chapter 2

Assembly Language In Practice

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Busy (0 = Printer cannot currently accept data)</td>
</tr>
<tr>
<td>6</td>
<td>Ack (0 = Printer has read characters)</td>
</tr>
<tr>
<td>5</td>
<td>PE, Paper empty (1 = no more paper)</td>
</tr>
<tr>
<td>4</td>
<td>SLCT (0 = Printer is off-line)</td>
</tr>
<tr>
<td>3</td>
<td>Error (0 = an error occurred)</td>
</tr>
<tr>
<td>2-0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Shift left</td>
</tr>
</tbody>
</table>

A Busy flag indicates the printer is busy and cannot accept any more data. For example, the printer buffer may be full. Acknowledge (ACK) is always set to 0 when the printer has accepted the character from the data lines and the next character can be written. PE and Error represent error states to be passed on to the user, so they can be handled. SLCT reflects the current status of the on-line switch of the printer. The printer cannot accept any data if it is off-line.

The parallel port has one additional register called the control register which further affects the printer's operation. You can read and write to this register. The following table shows the layout of the control register:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>IRQ enable (1 = IRQ active)</td>
</tr>
<tr>
<td>3</td>
<td>SLCT (0 = Switch printer off-line)</td>
</tr>
<tr>
<td>2</td>
<td>Reset (0 = Reset printer)</td>
</tr>
<tr>
<td>1</td>
<td>Auto LF (1 = Printer completes a line feed after CR)</td>
</tr>
<tr>
<td>0</td>
<td>Strobe (0 = Data present/on line)</td>
</tr>
</tbody>
</table>

Theoretically, by setting bit 4 it would be possible to enable IRQ 5 or 7, which is triggered when the acknowledge signal is raised by the printer. However, to avoid complications with sound cards, this feature is usually disabled. The parallel interface usually operates using the Polling method (the processor waits for a flag to change).

With the SLCT line you can switch some printers to off-line status from the computer. Use the Auto-LF line to control the automatic line feed character (once again, not with all models). All printers use the Reset and Strobe lines. The Strobe line lets the receiver know that a byte is on the data lines.
Printer control

Outputting a character to the parallel port is simple: Wait until the busy bit is set (busy line inactive), write the character to the data port, give a strobe signal (set and immediately clear line) and wait for an acknowledge signal. To demonstrate this, the following program does precisely this with all the characters of a sample string. This program is called PAR_TEST.PAS:

```
Const Base=$378;                {base address of parallel port}

Procedure PutChar_Par(z:Char);
   {outputs a character to parallel port (base address in "Base")}
Begin
   While Port[Base+1] and 128 = 0 Do;  {wait for end of Busy}
      Port[Base]:=Ord(z);            {place character on port}
   Port[Base+2]:=Port[Base+2] or 1;  {send strobe}
   Port[Base+2]:=Port[Base+2] and not 1;
   While Port[Base+1] and 64 = 1 do;  {wait for Ack}
End;

Procedure PutString_Par(s:String);
   {outputs string to parallel port, uses PutChar_Par}
Var i:Integer;                  {character counter}
Begin
   For i:=1 to Length(s) do       {each character}
      PutChar_Par(s[i]);          {send to parallel port}
End;

Begin
   PutString_Par('Hello, Abacus Printer Test'#13#10);
   PutString_Par('abcdefghijklmnopqrstuvwxyz0123456789'#13#10);
End.
```

Other Applications

Although the main reason you use the parallel port is to connect a printer to your computer, there’s a very good reason it’s not called a “printer port”. The port has 8 output lines, 5 input lines and 4 I/O channels (in the control register). The I/O channels can serve as either inputs or outputs, depending on the command.

For example, one frequent application is networking two computers using a parallel null modem cable and the INTERLNK.EXE driver of DOS 6.x. This driver uses a 4 bit transmission protocol. The protocol uses data registers D0 to D3 for output, and uses the ERROR, SLCT, PE and ACK lines as input. The function of the Strobe line (“announcing” data) takes on D4 as the output and Busy as the input. The following table shows how the cable is connected.
Other programs such as LapLink can communicate with this cable. While you can chose to write your own programs for data transfer, there are many shareware and commercial software programs available.

Another option, although no longer popular, is to use the parallel port as a sound card replacement. After all, the parallel port has an 8 bit digital output. The data just needs to be converted to analog signals.

You can build a simple digital-to-analog conversion, but it's easier to buy a sound card. If you have enough courage, you can build a sound card for much less money with a pair of resistors:

![DA Converter](image)

The resistors guarantee that D7, the maximum value data line, contributes the most to the analog output signal and that the voltage is reduced by the resistors with declining bit significance (D7-D0).

Plug this adapter (similar to the Covox device of the mid-1980s) into the parallel port and connect it to an amplifier. It can be operated by almost any mod player: Sound data uses a specific number of bytes per second (up to 44,000), which duplicates the frequency of the analog vibration. Fortunately, the PC uses an unsigned format so the data can be output directly to the parallel port (data port, for LPT1 usually at port address 378h). You don’t need to worry about the other two registers, since they only control communication with printers and the like. The data lines reproduce the exact contents of the data port.

If you pay attention to the sampling rate and output the data at this speed (e.g., sampling rate 22 KHz or 22,000 bytes per second to the port), you’ll hear the beep at the output. This simply indicates the processor is extremely busy, for port accesses last “forever”. So we don’t recommend extensive graphic operations at the same time you play back audio data. If you need to do this, it’s better to buy a sound card.
In this chapter we'll talk about the most important technical terms in PC graphics programming. Although we cannot explain all the terms in this chapter, the information will provide you with a general overview of the terms used in PC graphics programming.

**Terms You Need To Know**

**Pixel**

Pixel is an abbreviation for picture element. It represents the smallest element on a video display screen. Your monitor screen is divided into thousands of tiny dots. A pixel is one or more dots that are treated as a unit.

A pixel can be one dot on a monochrome screen, three dots (red, green and blue) on color screens, or clusters of these dots.

You can change the color of each pixel individually in graphics mode.

**Palette**

All VGA 256-color modes use a palette. This is unlike TrueColor (16.7 million colors) and HiColor (65,536 colors), in which each pixel is assigned specific values of the three primary colors: Red, green and blue.

A pixel is represented by a pointer to a palette entry in palette modes. The palette then provides the three primary color values for each of the 256 colors. All pixels of the same color have identical values.

The advantage of this method is that it saves significant memory. Instead of 18 bit color codes, only 8 bits are used per pixel in video memory. This advantage is lost, however, in graphic modes with higher color depth. The palette would become so large 16 or even 24 bits per pixel would be required.

A further advantage of palette-based modes is their usefulness for certain graphic effects. All colors containing Color 1 for example can instantly be converted to a different hue, just by changing its palette entry (3 bytes). This provides a simple method for fading images in and out - you don't need to raise or lower the brightness of each pixel. Instead, you simply increment the 256 palette entries from 0 to their maximum value (or vice versa). We'll talk more about this and other palette-based effects in Chapter 5.

**Sprite**

A sprite is basically a small image which can be positioned freely on the screen, but can also be made transparent so it can move across a background. Some home computers have a special chip for this purpose which relieves the CPU of the tasks for producing these effects.
Sprites are used commonly in video games where they move freely across the screen and pass by or through or even collide with each other.

**Cathode rays**

Luminous pixels of specific color and brightness are produced by a ray of accelerated electrons striking the rear of the screen. The image is constructed line by line from left to right (as seen from the front) and from top to bottom.

**Retrace**

Movement of the cathode ray as the screen image is being constructed. There are both vertical and horizontal retraces. A horizontal retrace occurs following construction of a screen line, and denotes the rapid movement of the ray to the beginning of the next line. A vertical retrace occurs when the ray has reached the bottom of the screen and returns to the top (first) line.

Basically, modifications to screen content, including changes to specific VGA registers, should occur only during a retrace. By doing this, the changes won’t conflict with image construction, which would lead to flickering in the affected area.

It makes no difference whether you wait for a horizontal or a vertical retrace. However, when performing extensive changes or register manipulations, you may want to make the changes during a vertical retrace since it lasts much longer (approximately 200 times longer).

To wait for a vertical retrace, use the procedure `WaitRetrace` (in the ModeXLib.asm module, although it is generally valid for all graphic and text modes). When a vertical retrace is in progress, bit 3 of Input Status Register 1 (port address 3dah) is set. This procedure relies on this signal.

It’s not enough to see if a vertical retrace is in progress. Instead, use the procedure `WaitRetrace`. In this case, `WaitRetrace` would end immediately and the screen modification would be performed, but there might not be enough time. Therefore, `WaitRetrace` waits for the beginning of a vertical retrace. A loop called `@wait1` first waits for any retrace in progress to finish, i.e., for the cathode ray to reappear on the screen, before using the second loop `@wait2` to wait for the next retrace.

```assembly
WaitRetrace proc pascal far
    mov dx,3dah ;Input Status Register 1
    @wait1:
        in al,dx    ;Bit 3 = 0 if ray is constructing image
        test al,8h
        jnz @wait1
    @wait2:
        in al,dx    ;Bit 3 = 1 if retracing
        test al,8h
        jz @wait2
        ret         ;Ray is now at the very bottom of the screen
Endp
```

**Double-scan**

Halving the y-resolution from 400 to 200 by double-displaying each line. This is used for emulating the 200-line modes not supported by VGA.

Speed is the essential element of graphic programming. Graphic programmers have always had to face the problems of insufficient speed. Fortunately, processors and graphic cards are becoming faster so today you
never have a speed problem with CGA graphics, as was common only a few years ago. However, today's standard of graphic quality requires several programming tricks.

One important trick is programming graphic chips directly which provides a great speed advantage over BIOS routines. This works for many reasons, one of which is removing many "validity checks."

However, by doing this it's possible to set a point at coordinates (5000,7000), with, of course, some rather bizarre effects.

If validity checks must be included for any reason, for example with interactive users (with a mouse), the checks should be performed outside of the display procedures (PutPixel, etc.), to avoid invoking them with "valid" pixels as well.

### Basis In BIOS Mode 13h

In developing VGA, IBM invented a practical method of addressing video memory in 256-color mode: Chaining bitplanes in a linear address space. Soon after, BIOS programmers used this technique for video mode 13h.

#### Memory organization

Video memory begins at segment a000. The organization of video memory is very simple: Each pixel is assigned one byte which contains a color, or more accurately, a pointer to an entry in the color palette. Addressing follows the path of the cathode ray during image construction - a pixel at coordinates (0,0) is located at offset 0; pixel (0,1) is located at offset 320, etc., until pixel (319,199) is reached at offset 63999.

Thus the address of a pixel at coordinates (x,y) is determined by the following formula:

$$\text{Offset} = Y \times 320 + X$$

We're now ready to program a simple star-scroller which sets pixels according to a certain pattern and then erases them:

```
Uses Crt;
Var Stars:Array[0..500] of Record
   x,y,Plane:Integer;
End;

st_no:Word;

Procedure PutPixel(x,y,col:word);assembler;
{sets pixel (x/y) to color col (Mode 13h)}
as
   mov ax,0a000h       {load segment}
   mov es,ax
   mov ax,320         {Offset = Y \times 320 + X}
   mul y
   add ax,x
   mov di,ax         {load offset}
   mov al,byte ptr col {load color}
   mov es:[di],al     {and set pixel}
End;
```
Graphic Know-how From Underground

Chapter 3

Begin
Randomize;               \{initialize random numbers\}
asm mov ax,13h; int 10h End;  \{set Mode 13h\}
Repeat                      \{executed once per display\}
  For St_no:=0 to 500 do Begin  \{calculate new position for each star\}
    With stars[st_no] do Begin
      PutPixel(x,y,0);        \{clear old pixel\}
      Dec(x,Plane shr 5 + 1); \{continue moving\}
      if x <= 0 Then Begin    \{left ?\}
        x:=319;               \{then reinitialize\}
        y:=Random(200);       \{left ?\}
        Plane:=Random(256);  \{then reinitialize\}
      End;                  \{left ?\}
      PutPixel(x,y,Plane shr 4 + 16);  \{set new pixel\}
    End;                   \{set new pixel\}
  End;                   \{set new pixel\}
Until KeyPressed;            \{run until key pressed\}
TextMode(3);                \{run until key pressed\}
End.

The inner loop is most significant in this example. In the inner loop, the previous star is first erased from
the screen. Then the star is moved according to its speed (calculated from \texttt{Plane}). When the star moves past
the left edge (x <= 0), it's repositioned to the right edge with new random values for its y-coordinate and
speed.

Finally, the pixel is set at the new position, also calculated from \texttt{Plane}, i.e., the slower stars are further
in the background and appear darker. The program uses the standard default palette used by all VGA cards
at startup. It contains a series of gray values between 16 (black) and 31 (white).

\section*{Internal structure of Mode 13h}

The linear memory structure which makes programming Mode 13h so easy is actually simulated for the
CPU. VGA converts the linear address internally back to a planar address. The two lower address lines (Bits
0 and 1 of the offset) are used to select the read/write plane. When bits 0 and 1 have been set to 0, the
remaining six bits (2-7) are used as physical addresses within the plane.

A similar process (odd/even addressing) is also used by all text modes. In this process, the lowest address
line is used for selecting between plane 0 and 1, so from the CPU's point of view, the character and attribute
bytes are directly next to each other. Internally, however, the character is stored in Plane 0 and the attribute
in plane 1. Planes 2 and 3 are for character set storage.

\section*{The GIF Image Format}

GIF is the most widely used format for graphic images. GIF was developed in 1987 by CompuServe for fast,
economical exchange of images between computers.

GIF has several important advantages compared to other formats such as PCX. GIF, unlike other graphic
formats, is not tied to a particular graphic mode because its data format is usable by all graphic systems.
GIF supports image resolutions to 16,000 x 16,000 pixels with a palette of 256 colors out of 16.7 million. Also,
any number of images with the same global or local color palette can be stored in one file (an option which
is seldom used).
GIF has an even more important advantage, however. GIF allows excellent compression of images coupled with high decompression speeds. It uses the modified LZW compression process which is also the basis for other compression programs.

This is the reason why we use GIF graphic images. The demos which we've included on the companion CD-ROM will load your images quickly without excessive memory requirements.

**The standard format**

GIF uses a block structure not as elaborate as TIFF format but which allows for easy handling of information about resolution and number of colors. The following table describes the basic structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Format ID “GIF”</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Version ID, currently “87a” or “89a”</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>Logical Screen Descriptor Block</td>
</tr>
<tr>
<td>0dh</td>
<td>n</td>
<td>Global Color Map (optional), in 256-color modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGA-compatible palette (n=768 bytes)</td>
</tr>
<tr>
<td>30dh</td>
<td>n</td>
<td>Extension Block (optional)</td>
</tr>
<tr>
<td>317h</td>
<td>n</td>
<td>Local Color Map (optional), in 256-color modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGA-compatible palette (n=768 bytes)</td>
</tr>
<tr>
<td>317h</td>
<td>n</td>
<td>Raster Data Block (graphic data LZW-compressed)</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>Terminator ID (89h)</td>
</tr>
</tbody>
</table>

The offset values assume that a Global Color Map exists but that Extension Blocks and Local Color Maps do not. Instead of the Terminator ID, you can add any number of Image Descriptor Blocks with the accompanying palette and raster data. End-of-file occurs only with the terminator. It’s not often that multiple images are stored in a single file. To make our example less complicated, we won’t attempt to design a full-featured GIF viewer but simply a quick load image routine that works with a single image.

Following the 6-byte long Format ID GIF87a or GIF89a is the Logical Screen Descriptor Block (LSDB). It defines the logical screen and therefore the global resolution and color data. The following table shows the structure of this block:
Chapter 3  
Graphic Know-how From Underground

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>Screen width</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Screen height</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Resolution flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7 1 = Global Color Map exists</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 6-4 : Color-depth in bits (minus one)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 : Reserved (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 2-0 : Number of bits per pixel (minus one)</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Background color (color-number in palette)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>Pixel Aspect Ratio:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7 : Global palette sort sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 6-0 : Pixel Aspect Ratio</td>
</tr>
</tbody>
</table>

This block includes the global palette (Global Color Map), which for each of the 256 colors contains a three-byte entry with values for the primary colors red, green and blue. Note that 8 bits are available for each color component for a total of $2^8$ (16.7 million) possible colors. For a VGA display each individual value must first be shifted two bits to the right before being sent to the VGA-DAC, since VGA uses only the lower 6 bits of each component.

Another unusual characteristic, which generally makes no difference on a PC, is the palette sort sequence. Both global and local palettes can be stored in VGA sequence (this is the normal situation). This means color 0 is first defined by red, blue and green, then color 1, etc. Another option, which is seldom used, is to sort the palette according to color frequency. Therefore, the red components of colors 0 to 255 are stored first followed by all green components and ending with blue.

To this point, all data is considered global and applies to all the images within the GIF file. Next are the image specific data for each individual image.

First is the Extension Block which can contain any type of data. Often, a paint program or a scan program will insert copyright information in the Extension Block. For our sample GIF loader, we'll ignore this data and just jump over it. An extension block starts with the character "!" (ASCII 21h) and ends with a null value (ASCII 0H).

Next the local image is described in the Image Descriptor Block. It has the following structure:
The Flag byte is important here. It indicates whether the Image Descriptor Block also has a local palette which takes precedence over the global one. Separate entries can also exist for sort sequence and number of pixels. Bit 6 indicates whether the image is interlaced, as in an interlaced monitor. First, all even image lines; (0, 2, 4 etc.) are written. These are followed by the odd lines. This feature was designed to provide a rough overview of image content when loading at slow transmission speeds (such as downloading from a CompuServe mailbox or loading from a diskette).

Directly following this block is the local palette (if one exists). The entries are the same as for the global palette. The Raster Data Blocks follow the local palette. They contain the actual image data in LZW format. The data itself, like the image description, is also stored in block format, whereby the length byte consists of only 8 bits, thus limiting block size to a maximum of 256 bytes (including the length byte).

**LZW compression process**

The number of bits which represent a pixel once again appears in the first Raster Block which directly precedes the length byte. This value is used by the LZW process to compress the images.

The most important advantage of using LZW is the file size of the image is reduced.

Unlike other compression schemes such as RLE (Run Length Encoding) used in the PCX format, LZW can handle both adjacent identical bytes and sequences of bytes that are not adjacent. To do this, an “extended alphabet” is used. Instead of the usual 8 bits, this alphabet uses additional encoding bits. For example, by using 9 bits per character, a file can contain codes 256-511 in addition to the usual codes 0-255. These are used to represent character strings.

The meaning of the extended alphabet is constructed dynamically during compression and decompression.
Chapter 3

Graphic Know-how From Underground

Here's how it works. Characters are read from the source file or video memory until a character string is encountered which is no longer found in the alphabet. This happens in the beginning after only two characters: The first character will be in the alphabet (an ordinary character from 0 to 255), while the string formed from the first two characters does not yet exist.

When the compressor reaches this point it writes this character string into the alphabet, so the next time the string occurs it can compress it by replacing it with this code. The code of the longest character string still contained in the alphabet is then written to the destination file and the character string initialized with the last character read. Characters are again added until the string is no longer found in the alphabet.

Now the question is how many bits you should use for the alphabet. If you use too few the alphabet will soon overflow and reinitializing it greatly reduces the compression rate. However, taking too many bits immediately wastes space because the upper bits will never be needed. The solution is the modified LZW process, which uses a variable byte width. You begin with nine bits, which allows an alphabet with 512 entries. If this limit is exceeded another bit is simply added on.

On the other hand it makes no sense to keep extending indefinitely. This wastes bits unnecessarily, although the majority of character strings already in the alphabet will never be used again. Therefore, when reaching a width greater than 12 bits, a clear-code is eventually sent. This clear-code completely clears the alphabet and resets the width back to nine bits. Compression then basically starts from the beginning.

The effectiveness of this algorithm depends strongly on the length of the file to be compressed. You'll need large amounts of data to access the alphabet repeatedly and be able to store long character strings in a small number of bits. This algorithm is therefore best suited for images of several kilobytes in size.

What is even more important for our purposes is the decompression algorithm. It takes the compressed data and transforms them back into recognizable images. Before you can understand the packing process, however, you must first understand the compressor.

Interestingly, the LZW process does not require storing the alphabet. The alphabet is regenerated from the packed data during decompression. The program uses the fact that only character strings coded in the compressed data are those that already occurred and therefore exist in the alphabet.

The following describes how the decompressor proceeds. Each compressed character read is first checked to see whether it's actually a real, uncompressed byte. This would be indicated by a value less than 256. These characters can be written directly to the destination file (or video memory). When encountering an extended code, however, the corresponding character string is retrieved from the alphabet and then written. Of course, the alphabet must be constructed at the same time by combining the last decompressed character string (or uncompressed character) with the first character of the just decoded character string and entering it into the alphabet.

This corresponds exactly to the compression process but "in reverse". So, the alphabets formed during compression and decompression correspond exactly at any point in time. An exception to the "any point" is when a character occurs whose code is not yet in the alphabet. When compressing a character string of the form AbcAbcA, if the character string Abc already exists in the alphabet, the compressor writes this entry's code to the destination file and forms the new alphabet entry AbcA, which appears again immediately afterward and is therefore also used by writing it to the destination file.

The decompressor at this point still does not recognize the character string however; how would it know the next character will be an A, since it is not writing to the destination file. You should be able to notice when
this situation occurs because it arises only with character strings as described above. If a code appears that is not yet in the alphabet, the last decoded character string plus its first character is simply written to the video memory and the new character string recorded in the alphabet.

An alphabet could require a great amount of memory, which was a problem in the past. The algorithm was therefore again improved. Although it may seem more complicated, it actually simplifies your work even more. As we have seen in both compressing and decompressing each new alphabet entry is formed from an already existing character string plus a new character. What is being stored is simply the code of the old character string and the code of the new character. We, therefore, need just two more entries: Prefix and Tail.

**GIF-loader optimized to 320 x 200**

GIF is clearly a universal format. It supports multiple resolutions, color-depths and is completely system-independent. A good GIF-viewer must recognize and support all variations of this format. Unfortunately, this feature doesn't work very well at high speeds. Since many shareware GIF-viewers are available, writing another one is not the goal of this chapter. Instead, we'll develop a loader optimized to specific formats, which will compensate for its lack of versatility with a high degree of speed.

The most popular format for demos and entertainment scenes is 320 x 200 with 256 colors. This graphic mode is our main objective. As we'll see later, expanding to other 256-color modes is no great problem. However, 16 colors is not included because the unpacker would then need to be rewritten completely (4-bit color-depth instead of 8-bit).

As an example we will use the routine LoadGIF from unit GIF. You'll see this routine often in other programs which we discuss in this book:

```pascal
unit gif;                       {Header for gif.asm}
Interface
uses modexlib;                  {because of SetPal}
var
  vram_pos,                     {current position in VGA-RAM}
  rest, errorno:word;          {remaining bytes in RAM and error}
  gifname:String;              {Name, including #0}
Procedure LoadGif(GName:String); {Loads Gif file "GName.gif" into vscreen}
Procedure LoadGif_Pos(GName:String;Posit:Word); {Loads Gif file at screen offset Posit}
Implementation
Procedure ReadGif;external;   {custom Gif loader, complete in Assembler}
{$l gif}
Procedure LoadGif;
{Loads Gif file "GName.gif" into vscreen}
Begin
  If pos('.',gname) = 0 then   {add ".gif" extension if necessary}
    gname:=gname+'.gif';
  Gifname:=GName+#0;;          {generate ASCIIZ string}
  vram_pos:=0;                {start in VGA-Ram at Offset 0}
  ReadGif;                    {and load image}
  If Errorno <> 0 Then        {terminate if error}
    Halt(Errorno);
```

You can find GIF.PAS on the companion CD-ROM.
The two procedures in this unit are `LoadGIF` and `LoadGIF_Pos`. They set the framework for calling the next assembly language portion and thereby simplify the load process. `LoadGIF` is the normal procedure. It simply loads an image into the virtual screen (vscreen) and pages any overflow (images larger than 320 x 200) into video memory starting with Offset 0. You can also pass the offset in `LoadGIF_Pos` where paging begins.

Both procedures add the extension .GIF to the filename if necessary. They also add the terminating 0 to the string required by DOS. A simple error procedure is also implemented (it prevents a system crash in this form) by stopping the entire program if `ReadGIF` passes an error number other than 0. Unlike a full-featured GIF-viewer, this demo assumes the files reside in the current directory. Finally, `SetPal` is called, which sets the loaded image palette.

You can find the actual load procedure `ReadGIF` in the file GIF.ASM. True assembly language code lets you achieve maximum speed.
width1  dw 0         ;number of bits in a byte
max     dw 0         ;maximum alphabet length for current width
stackp  dw 0         ;pointer within alphabet stack
restbits dw 0        ;number of bits remaining to be read
restbyte dw 0        ;number of bytes still in buffer
specialcase dw 0     ;buffer for special case
cur_code dw 0        ;code just processed
old_code dw 0         ;previous code
readbyte dw 0         ;byte just read
lbyte   dw 0         ;last physical byte read
data ends
extrn p13_2_modex:far ;needed for overflow

code segment public
assume cs:code,ds:data

public readgif
GifRead proc pascal n:word
;reads n physical bytes from file
    mov ax,03f00h      ;function 3fh of Interrupt 21h: read
    mov bx,handle     ;load handle
    mov cx,n          ;load number of bytes to read
    lea dx,buf        ;pointer to destination buffer
    int 21h            ;execute interrupt
ret
GifRead endp

GifOpen proc pascal
;opens the Gif file for read access
    mov ax,03d00h      ;function 3dh: open
    lea dx,gifname + 1 ;pointer to name (skip length byte)
    int 21h            ;execute
    mov handle,ax      ;save handle
ret
GifOpen endp

GifClose proc pascal
;closes Gif file
    mov ax,03e00h      ;function 3eh: close
    mov bx,handle     ;load handle
    int 21h             ;execute
ret
GifClose endp

GifSeek proc pascal Ofs:dword
;positioning within the file
    mov ax,04200h      ;function 42h, subfunction 0: seek relative to start of file
    mov bx,w handle   ;load handle
    mov cx,word ptr Ofs + 2 ;load offset
    int 21h             ;execute
ret
Endp

ShiftPal proc pascal
;aligns 24-bit palette format to 18-bit VGA format
    mov ax,ds          ;source and destination arrays in the data segment
    mov es,ax
    mov si,offset buf  ;read from data buffer
    lea di,palette     ;write to palette
    mov cx,768d        ;copy 768 bytes
@ll:
Chapter 3

Graphic Know-how From Underground

lodsb ; get bytes
shr al,2 ; convert
stosb ; and write
loop @l1
ret
Endp

FillBuf proc pascal
; reads a block from the file in buf
   call gifread pascal,1 ; read one byte
   mov al, b buf[0] ; load length after al
   xor ah, ah
   mov w restbyte, ax ; and store in restbyte
   call gifread pascal, ax ; read bytes
ret
Endp

GetPhysByte proc pascal
; gets a physical byte from the buffer
   push bx ; caller needs bx
   cmp w restbyte, 0 ; no more data in buffer ?
   ja @restthere
   pusha ; then refill buffer
   call fillbuf
   popa
   mov w bufInd, 0 ; and reset pointer
@restthere: ; data in buffer
   mov bx, w bufInd
   mov al, b buf[bx] ; get byte
   inc w bufInd ; move pointer
pop bx ; finished
ret
Endp

GetLogByte proc pascal
; gets a logical byte from the buffer, uses GetPhysByte
   push si ; caller needs si
   mov ax, w width1 ; get byte width1
   mov si, ax ; and store
   mov dx, w restbits ; shift lbyte 8 remaining bits to the right
   mov cx, 8
   sub cx, dx ; obtain difference
   mov ax, w lByte
   shr ax, cl ; and shift
   mov w cur_code, ax ; store code
   sub si, dx ; remaining bits already obtained -> subtract
   nextbyte:
   call getphysbyte ; get new byte
   xor ah, ah
   mov w lbyte, ax ; store in lbyte for next logical byte
   dec w restbyte ; mark byte as obtained
   mov bx, 1 ; mask remaining bits in obtained byte
   mov cx, si ; add number of bits
   shl bx, cl ; shift 1 by number
   dec bx ; and decrement
   and ax, bx ; mask byte
   mov cx, dx ; shift to correct position
   shl ax, cl ; (by remaining bytes to the left)
   add w cur_code, ax ; and add to result
   sbb dx, w width1 ; decrement remaining bits
add dx,8                      ;by amount exceeding 8 bits
jns @positive
add dx,8
@positive:
sub si,8                      ;up to 8 bits obtained -> subtract
jle @finished                ;<= 0 -> finished, end
add dx,w width               ;otherwise increment remaining bits by missing bits
sub dx,8
jmp @nextbyte                 ;and continue
@finished:
mov w restbits,dx             ;store remaining bits for next call
mov ax,w cur_code             ;and load ax
pop si
ret
Endp

ReadGif proc pascal
;loads a Gif image called gifname in vscreen, overflow is paged out to screen
push ds                       ;store ds
call GifOpen                  ;open file
jnc ok                        ;error ?
pop ds
ret
ok:
call gifseek pascal, 0,13d    ;skip first 13 bytes
push 768d                     ;load 768 bytes of palette
call gifread
call shiftpal                 ;and convert to *palette*
call gifread pascal,1        ;skip one byte
@extloop:                      ;skip extension blocks
cmp w buf[0],21h              ;another Extension-Block existing ?
jne @noext                    ;no, then continue
    call gifread pascal,2     ;read first two bytes
    mov al,b buf[1]           ;data block length
    inc al
    xor ah,ah
    call gifread pascal, ax   ;and skip
    jmp @extloop
@noext:
call gifread pascal, 10d      ;read remainder of IDB
test b buf[8],128             ;local palette ?
ej @noloc                     ;no, then continue
    push 768
    call gifread
    call shiftpal
    jmp @extloop
@noloc:
    les di,dword ptr vscreen   ;load destination address
    mov w lbyte,0              ;last read byte 0
    mov w free,258             ;first free entry 258
    mov w width1,9             ;byte width1 9 bits
    mov w max,511               ;so maximum entry is 511
    mov w stackp,0              ;stack pointer to beginning
    mov w restbits,0            ;no remaining bits
    mov w restbyte,0            ;for remaining bytes to obtain
@mainloop:
call getlogbyte               ;run for each logical byte
ret
Endp
Chapter 3

Graphic Know-how From Underground

```
cmp ax, eof                   ; end of file ID
    jne @no_abort
    jmp @abort
@no_abort:
    cmp ax, clr                   ; clr-code ?
    jne @no_clear
    jmp @clear                    ; yes, then clear alphabet
@no_clear:
    mov w readbyt, ax             ; store current byte
    cmp ax, w free                 ; code already in alphabet (<free)
        jb @code_in_ab            ; no, then special case, i.e., give last string
    mov w cur_code, ax             ; for processing
    mov bx, w stackp
    mov cx, w specialcase          ; and add first character (always concrete)
    mov w abstack[ bx], cx         ; enter onto stack
    inc w stackp
@code_in_ab:                  ; code exists in alphabet:
    cmp ax, clr                   ; < clr code ?
        jb @concrete              ; then concrete character
    mov bx, w cur_code             ; current code as pointer in alphabet
        shl bx, 1                  ; word-array (!)
        push bx
        mov ax, w ab_tail[ bx]       ; get tail (concrete)
        mov bx, w stackp
        shl bx, 1                  ; likewise word-array
        mov w abstack[ bx], ax      ; enter
        inc w stackp
    pop bx
    mov ax, w ab_prfx[ bx]         ; get prefix
    mov w cur_code, ax             ; give as current code for decoding
    cmp ax, clr                   ; > clr-Code
        ja @fillstack_loop         ; then continue decoding
@concrete:                     ; now just the concrete values onto the stack
    mov bx, w stackp
        ; push last code onto stack
        shl bx, 1                  ; word-array
    mov w abstack[ bx], ax
    mov w specialcase, ax          ; also keep for special case
    inc w stackp
    mov bx, w stackp
        ; prepare to read stack
    dec bx
        ; move pointer backward and
    shl bx, 1
    @readstack_loop:               ; process stack
        mov ax, w abstack[ bx]       ; get character from stack
        stosb                      ; and write to destination memory
    cmp di, 0                      ; segment overflow ?
        jne @noovl1
    call p13_2_modex pascal,vram_pos,16384d
    add vram_pos, 16384d           ; the page section to video RAM
    les di, dword ptr vscreen     ; next position in VGA-RAM and reset destination pointer
@noovl1:
        dec bx
            ; stack position to next element
        dec bx
        jns @readstack_loop          ; finished processing ? no, then continue
    mov w stackp, 0                ; set stack pointer variable to 0
    mov bx, w free                 ; now enter in alphabet
        shl bx, 1                  ; go to position "free"
    mov ax, w old_code             ; write last code in prefix
    mov w ab_prfx[ bx], ax
```
Chapter 3

Graphic Know-how From Underground

```
mov ax, w cur_code             ;current code in tail
mov w ab_tail[ bx ], ax       ;store byte read as last code
mov ax, w old_code, ax         ;to next position in alphabet
mov ax, w free
mov ax, w free
cmp ax, w max                  ;maximum reached ?
ja @no_mainloop
jmp @mainloop
@no_mainloop:
  cmp b width1, 12             ;has width1 reached 12 bits ?
  jb @no_mainloop2
  jmp @mainloop
@no_mainloop2:
  inc w width1                  ;otherwise increment
  mov cl, b width1               ;calculate new maximum value
  mov ax, 1                      ;shift 1 by new width1 to the left
  shl ax, cl
  dec ax                        ;and decrement
  mov w max, ax                  ;enter
  jmp @mainloop
@clear:                         ;reset alphabet:
  mov w width1, 9                ;width1 back to original value
  mov w max, 511                 ;reset maximum to 511
  mov w free, 258                ;first free position at 258
  call getlogbyte
  mov w specialcase, ax          ;record as special case
  mov w old_code, ax             ;and also as last byte read
  stosb                         ;this value directly to memory, because
                              ;concrete

  cmp di, 0                      ;segment overflow ?
  jne @noovl2
  call p13_2_modex pascal, vram_pos, 16384d
  add vram_pos, 16384d           ;then page to video RAM
  les di, dword ptr vscreen      ;move VGA-RAM pointer forward and reset start
  address
@noovl2:                         ;go back to main loop
  jmp @mainloop
@abort:                           ;terminate through eof-code
  call gifclose
  mov rest, di                   ;store number of bytes still to be copied
  pop ds                         ;end
ret
Endp
```

code ends
end

`GIFOpen, GIFRead, GIFSeek` and `GIFClose` in this module serve as file handling routines. They open the file, read data (n Bytes), set the file pointer to position `offs` and close the file.

One procedure not directly involved with loading is `ShiftPal`. It reads the palette from the data buffer where it was loaded previously by `GIFRead`. It then transfers it to the array "Palette" so it can be set directly. This is also where the format alignment occurs by using right-shifting to generate the 6-bit values of VGA format from the 8-bit values of GIF format.
The next level of the GIF-loader consists of the procedure \texttt{GetPhysByte}. The task of this procedure is to obtain exactly one byte from the file. Of course, you could also accomplish this directly through \texttt{GIFRead}, but even with caches, individual disk accesses always take longer than intermediate buffering through a program.

\texttt{GetPhysByte} first checks whether additional data exists in the buffer, whose fill status is indicated by \texttt{RestByte}. The program either reads the next byte or the buffer must be refilled with new data by \texttt{FillBuf}. \texttt{FillBuf} reads the current block length followed by the corresponding number of bytes from the file.

There is one problem you might encounter with this compression method. The data obtained must be processed again - a single byte is no longer eight bits in width. Depending on the current status of the alphabet, a byte consists of a minimum of nine bits. Therefore, the program must take the continuous bit-stream (transmitted by \texttt{GetPhysByte} in 8-bit blocks, i.e., physical bytes) and filter out the current number of bits. This task is carried out by \texttt{GetLogByte}.

The variables \texttt{LByte} and \texttt{Restbits} are responsible for the stream of data. The variable \texttt{LByte} contains the last physical byte transmitted by \texttt{GetPhysByte}. This byte must still be converted to \texttt{Restbits} bits. To do this, \texttt{LByte} is shifted by eight \texttt{Restbits} to the right and this value stored in \texttt{cur_code}.

The next step is obtaining a new byte, whose required (lower) bits are now masked out and added to. For bytes consisting of ten or more bits another physical byte may be needed; in this case the program returns to label \texttt{@nextbyte} and processes this byte as well.

The most important procedure in this module (and the only one declared as public) is \texttt{ReadGIF}. The following are passed globally to this procedure:

- \texttt{GIFname} contains the complete filename with terminal 0.
- \texttt{vscreen} contains a pointer to the virtual screen which must be previously allocated (done in Unit-Start).

After opening the file (with error checking), the program skips the first 13 bytes. The data contained there (ID and Logical Screen Descriptor) are not relevant here since we’re assuming a 320 x 200 format. Other formats are not checked (due to speed considerations) and will produce garbage on the screen. Our main priority here is speed.

Another assumption is the existence of a global palette which is then loaded and aligned through \texttt{ShiftPal}. Next the ID of the next block is read, skipping over any Extension Blocks. The Image Descriptor Block (IDB) follows the Extension Blocks. The program reads 10 bytes - 9 bytes from the IDB plus the first byte of the Raster Data Block. Next if a local palette exists (Bit 7 of the IDB Flag Register), it’s now loaded and set.

The label \texttt{@nolok} initiates the actual decoding of raster data. The procedure first initiates several variables. The variable \texttt{free} contains the first unused position in the alphabet, \texttt{width} contains the current byte width, \texttt{max} the highest used position in the alphabet. The main loop \texttt{@mainloop} then begins, which is executed for each byte.
Each byte read is first compared to the variables `eof` (end-of-file) and `clr` (clear-alphabet). The character `eof` indicates the file is completely loaded. The character `clr` indicates that an alphabet overflow occurred during compression. This overflow requires clearing and re-initialization of the alphabet. To synchronize the packing and unpacking, at this point you also need to reset the alphabet when decompressing.

Next the program checks to see if this code already exists in the alphabet. If not, we have a special case as described above, which is handled as follows. The last string (compressed in `Old_Code`) is combined with the previously stored second-to-last character and pushed onto the stack (`abstack`). This stack is later processed in the reverse order and written to main memory.

After dealing with this special case (or if no special case occurred) the program continues at `code_in_ab`. This routine checks whether the value is less than `clr`, i.e., within the "normal" alphabet range (0..255). If so, it's a real code and is pushed onto the stack at the label `concrete`. Otherwise it must first be decoded.

This is where we use the stack. The decompressor uses the compressed alphabet, which has a recursive structure. The postfix (tail) is always real. A new character is added here during compression. The prefix, on the other hand, is often compressed and must be decompressed through an alphabet entry. This alphabet entry itself consists of a compressed and an uncompressed portion. This routine continues until an alphabet entry has two real portions. Until then, all postfixes (tails) are pushed onto the stack since they have not yet been used. They can now be written to the destination.

The above procedure is executed in the loop labelled `@fillstack_loop`. Push the tail of the current code onto the stack and (if compressed) decompress the prefix until it too is actual, then exit the loop.

At `concrete` the last prefix is pushed onto the stack, the stack is saved for a possible special case and finally emptied in the loop `readstack_loop`.

At this point (after stosb) the program deals with images larger than 320 x 200. The only problem with these images is their size - they no longer fit into the virtual screen segment, i.e., they're paged out to video memory. The pointer is moved here for further paging, if necessary, so the remainder of the image can be written to the correct position.

The program begins the actual decompression process with the label `@noovl1`. The loop continues until the entire stack has been emptied, after which the alphabet must still be updated. This is done by entering the last value as the prefix and the current code as the tail. Next, the alphabet pointer (`free`) is updated. If the pointer goes past the alphabet limit, the alphabet is expanded and a new maximum value (`max`) is set. The procedure then continues as usual.

On the other hand, no further expansion occurs if the width has already reached twelve bits. Instead, the routine returns to the main loop. The packer should return `clr` which is processed at `@Clear`:

The alphabet variables are reset to their initial values and the next character (guaranteed to be uncompressed) is written directly. This is also where any overflow must be handled through paging.

Finally, the procedure must close the file and store the "fill status" of the virtual screen. The calling program now only needs to copy the remainder from the virtual screen to the screen:

```c
p13_2_ModeX (vram_pos, rest div 4);
```

For the special case 320 x 200 you can also just insert constants:

```c
p13_2_ModeX (0, 16000);
```
Chapter 3

Graphic Know-how From Underground

In Mode 13h, on the other hand, the Pascal Move command is used due to the simple memory structure:

\[
\text{Move (VScreen^, Ptr(^a000, 0)^, 64000);}\]

This is exactly what procedure Show_Pic13 of the ModeXLib unit does.

PCX Is Quick And Simple

In addition to GIF, there are many other formats with a variety of capabilities and compression methods. The PCX format was originally developed by ZSoft for its paint program and in the meantime has attained widespread use.

The PCX format doesn’t feature either high compression rates or convertibility between different computer systems. The most important feature of PCX is its simplicity. Frame data is very simply laid out, and its compression algorithm (RLE) is equally uncomplicated.

Since its invention PCX has gone through several versions, each with quite different capabilities. Here we will be using Version 3.0 only, which fully supports 256-color images. Regardless of version, all PCX files have a 128-byte header which contains image data such as size and position. This header in Version 3.0 requires 70 bytes. The rest is disregarded and is usually filled with zeros.

Structure of PCX files

The structure of the header is very simple:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Format ID 0ah</td>
<td>14</td>
<td>2</td>
<td>Vertical resolution in dpi (usually image height)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Version (5 for Version 3.0)</td>
<td>16</td>
<td>48</td>
<td>Palette (16-color, normally disregarded at 256 colors)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>RLE compression used (1) or not (0)</td>
<td>64</td>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Bits per pixel</td>
<td>65</td>
<td>1</td>
<td>Number of planes (1 plane at 256 colors)</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Coordinates of top left corner (x,y as words)</td>
<td>66</td>
<td>2</td>
<td>Bytes per image line, rounded up to even number</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Coordinates of bottom right corner (x,y)</td>
<td>68</td>
<td>2</td>
<td>Palette type (1=color or b/w, 2=grayscale)</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>Horizontal resolution in dpi (usually image width)</td>
<td>70</td>
<td>58</td>
<td>Reserved (usually 0)</td>
</tr>
</tbody>
</table>

A compression flag follows the Format ID and version number. A 0 means that no compression was used, which in some cases actually produces a smaller file (more on this later). A 1 indicates that RLE compression was used.
The image section coordinates and the resolution follow the color-depth (8 bits at 256 colors). We're using the term "resolution" loosely since some programs use this area to store (as well as request) image width and height. We can trace the palette back to the days of 16-color images and is irrelevant at 256 colors. However, to be safe, you should store the first 16 colors of the palette here.

The number of bit-planes (1 at 256 colors) appears at offset 65. This is followed by the number of bytes per image line (must be rounded up to an even number). The word at offset 68 indicates whether the palette consists of color- or gray values (1 is for both color and black-and-white). The fill-bytes are next.

The actual palette data for 256-color images are located at the end of the file and are identified by 0ch. So, there are 769 bytes added on after the graphic data.

The graphic data directly following the header are either uncompressed or compressed by the RLE process. RLE compression combines identical adjacent bytes into 2-byte combinations. Single bytes are written directly to the file, while repetitions are identified by setting the upper two bits of the byte. Here we have a length byte directly followed by a data byte. The data byte when decompressing is written to video memory according to the number of times indicated by the length byte.

Of course, now there is no way to process single bytes whose upper two bits have been set. These bytes must therefore be stored as byte-repetitions of length 1. Since two bytes are then occupied in the compressed file, it's quite possible for the amount of data to expand rather than contract. So if an image has few single-color regions, which is usually the case with 256-color images, the compressed file may easily become larger than the uncompressed file.

The main advantage of this compression method is its high processing speed. Decoding of simple byte-repetitions is significantly faster and easier than using the LZW codes of the GIF format, which must always be searched out from the alphabet first.

Our objective in this chapter is to create a universal screen-grabber (copies screen contents to a file) for both 320 x 200 x 256 formats, which can also process such features as split-screen. This time we are emphasizing clarity over maximum speed - the compressor for example is written in Pascal. We'll illustrate the principles of how to process (i.e., store) PCX images.

One application of our program for example, is taking a demo you have programmed and generating screen shots from certain sections, which can then be shown as a sequence with the name of the programmer, graphic designer and musician.

You'll find the complete source code of this program in the file GRABBER.PAS:

```pascal
{$G+}{Sm 1024,0,0}                   {requires little stack and no heap}
Uses ModeXLib,Crt,Dos;
Var OldInt9:Pointer;              {pointer to old keyboard handler}
active:Boolean;             {set, if hard copy already in motion}
no:Word;                    {Number of picture, for assigning filenames}
installed:Boolean;          {already installed ?}
Mode,                        {current VGA-Mode: 13h, ffh (Mode X)}
| or 0 {neither of the two}
| Split_at,                   {Split-Line (graphic line}
| You can find GRABBER.PAS on the companion CD-ROM
```

49
Graphic Know-how From Underground

Chapter 3

LSA, Skip:Word;  {Linear Starting Address}  {Number of bytes to skip}

Procedure GetMode;
{sets current graphic mode 13h or Mode X (No. 255)}
{and frame data (Split-Line, Start address)}
Begin
Mode:=$13;  {Mode 13h Standard}
asm  {set Bios-Mode}
mov ax,0f00h  {Function: Video-Info}
int 10h
cmp al,13h  {Bios-Mode 13h set ?}
je @bios_ok
mov mode,0  {if no -> neither Mode 13h nor X active}
@bios_ok:
End;
If Mode=0 Then Exit;  {wrong mode -> abort}

Port[$3c4]:=4;  {read out TS-Register 4 (Memory Mode)}
If Port[$3c5] and 8 = 0 Then  {Chain 4 (Bit 3) inactive ?}
Mode:=$ff;  {then Mode X}
Port[$3d4]:=0d;  {Linear Starting Address Low (CRTC 0dh)}
LSA:=Port[$3d5];  {read out}
Port[$3d4]:=0c;  {Linear Starting Address High (CRTC 0ch)}
LSA:=LSA or Port[$3d5] shl 8;  {read out and enter}
Port[$3d4]:=18;  {Line Compare CRTC 18h}
Split_at:=Port[$3d5];  {read out}
Port[$3d4]:=7;  {Overflow Low}
Split_at:=Split_at or (Port[$3d5] and 16) shl 4;  {mask out Bit 4 and move to Bit 8}
Port[$3d4]:=9;  {Maximum Row Address}
Split_at:=Split_at or (Port[$3d5] and 64) shl 3;  {mask out Bit 6 and move to Bit 9}
Split_at:=Split_at shr 1;  {convert to screen lines}

Procedure PCXShift;assembler;
{prepares current palette for PCX (shift 2 to the left)}
asm
mov si,offset palette  {pointer to palette in ds:si}
mov cx,768
lp:
lodsb  {get value}
shl al,2  {shift}
mov ds:[si-1],al
loop @lp  {and complete loop}
End;

Var pcx:File;  {PCX file to disk}

Procedure Hardcopy(Startaddr,splt, Skip:Word;s : string);
{copies graphic 320x200 (Mode 13 o. X) as PCX to file s}
{current screen start (Linear Starting Address) in Startaddr}
{Split line in splt}
Var Buf:Array[0..57] of Byte;  {receives data before saving}
Aux_Ofs:Word;

const
Header1:Array[0..15] of Byte = ($0a,5,1,8, 0,0, 0,0, $3f,1, 199,0,540,1,200,0);
Header2:Array[0..5] of Byte = ($0,1,$40,1,0,0);
plane:Byte=0;                         {current plane no.}
var count:Byte;                       {number of equivalent characters}
value,                      {value just fetched}
lastbyt:Byte;               {previous value}
i:word;                     {byte counter}

begin
asm                             {read out palette}
xor al,al                     {start with color 0}
mov dx,3c7h                   {use Pixel Read Address }
out dx,al                     {to inform DAC of this}
push ds                       {pointer es:di to palette}
pop es
mov di,offset palette
mov cx,768                    {read out 768 bytes}
mov dx,3c9h                   {Pixel Color Value}
rep insb                      {and read}
cmp mode,13h                  {Mode X ?}
je @Linear                    {then:}
mov dx,03ceh                  {set write and read mode to 0}
mov ax,4005h                  {using GDC-Register 5 (GDC Mode)}
@Linear:
End;
Assign(pcx,s);                {open file for writing}
Rewrite(pcx,1);
BlockWrite(pcx,Header1,16);   {write Header part 1}
PCXShift;                     {prepare palette}
BlockWrite(pcx,palette,48);   {enter first 16 colors}
BlockWrite(pcx,Header2,6);    {write Header part 2}
FillChar(buf,58,0);           {write 58 nulls (fill header)}
BlockWrite(pcx,buf,58);
plane:=0;                     {start with Plane 0}
count:=1;                     {initialize number with 1}
If split<200 Then
  If Mode = $ff Then
    aux_ofs:=(i div 320) * skip  {set auxiliary offset taking}
    Else
    aux_ofs:=(i shr 2 - split) div 320) * skip;  {with splitting reference to VGA-Start}
asm                             {read out pixel}
mov ax,0a000h
mov es,ax
Chapter 3

Graphic Know-how From Underground

mov si,i  (load offset)
cmp mode,13h  (Mode 13h ?)
je @Linear1
shr si,2  (no, then calculate offset)

@Linear1:
cmp si,splt  (Split-Line reached ?)
jb @continue  (no, then continue)
sub si,splt  (otherwise, apply everything else)
sub si,startaddr  (to screen start)

@continue:
add si,startaddr  (add start address)
add si,aux_ofs  (add auxiliary offset)
cmp mode,13h  (Mode 13h ?)
je @Linear2  (no, then Mode X read method)
mov dx,03ceh  (using GDC-Register 4 (Read Plane Select))
mov ah,plane  (select current plane)
inc plane  (and continue shifting)
mov al,4
and ah,03h
out dx,ax

@Linear2:
mov al,es:[si]  (read out byte)
mov value,al  (and save in value variable)
End;
If i<>0 Then Begin  (no compression with first byte)
If (Value = lastbyt) Then Begin(same bytes ?)
If (Count=64) or  (counter too high already ?)
(i mod 320 =0) Then Begin  (or beginning of line ?)
buf[0]:=$c0 or (count-1);  (then buffer)
buf[1]:=lastbyt;  (write counter status and value)
count:=1;
BlockWrite(pcx,buf,2);  (and to disk)
End;
End Else  (different bytes :)
If (Count > 1) or  (several of the same ?)
(lastbyt and $c0 <> 0) Then  (value too large for direct writing ?)
Begin
buf[0]:=$c0 or count;
buf[1]:=lastbyt;
lastbyt:=Value;
Count:=1;
BlockWrite(pcx,buf,2);
End Else Begin  (single, legal byte:)
buf[0]:=lastbyt;
(lastbyt:=Value;
BlockWrite(pcx,buf,1);
End;
End Else lastbyt:=value;  (with first byte save only)
End;
buf[0]:=$0c;  (insert ID palette)
BlockWrite(pcx,buf[0],1);  (and write)
BlockWrite(pcx,palette,256*3);  (and add palette)
Close(pcx);  (close file)
End;

Procedure Action;
{called upon activation of the hot-key}
Var nrs:String;  (string for assigning name)
Begin if not active Then Begin
active:=true; str(no,nrs); Inc(no); GetMode;
If Mode <> 0 Then HardCopy(LSA,SPLIT_AT,'hard'+nrs+'.PCX');
active:=false;
End;
End;

Procedure Handler9; interrupt; assembler;
{new interrupt handler for keyboard IRQ}
asm
pushf
cli
in al,60h
cmp al,34d
jne @finished
xor ax,ax
mov es,ax
mov al,es:[417h]
test al,8
je @finished
call action
@finished:
sti
End;

Procedure identification; assembler;
{Dummy-Procedure, contains Copyright message for installation ID}
{NOT EXECUTABLE CODE !}
asm
db 'Screen-Grabber, (c) Data Becker 1995/Abacus 1995';
End;

Procedure Check_Inst; assembler;
{Checks whether Grabber is already installed}
asm
mov installed,1
push ds
les di,oldint9
mov di,offset identification
mov ax,cs
mov ds,ax
mov si,offset identification
mov cx,20
repe cmpsb
pop ds
jcxz @installed
mov installed,0
@installed:
End;

Begin
no:=0;
GetIntVec(9,OldInt9);
End;
Check Inst;  {check whether already installed}
If not installed Then Begin  {if no:}
  SetIntVec(9,@Handler9);  {install new handler}
  WriteLn('Grabber installed');
  WriteLn('Activation with <alt> g');
  Keep(0);  {output message and exit resident}
End;
WriteLn('Grabber already installed');  {if already installed, message and exit}
End.

The main program is typical of all TSRs, although features such as deinstallation were omitted due to the limitations of Pascal. If the program is not already installed, a message is displayed and the program is loaded. Otherwise the user is informed that a copy already exists in memory and the program terminates normally. The installation check occurs in a procedure ID which compares the loaded program with the current keyboard handler.

The handler, which is latched onto the keyboard-interrupt (IRQ 1, i.e., Interrupt 9), now has the job of transmitting all keypresses to the old handler so the system will function normally. It also starts the Grabber with the \[\text{Alt} + G\] hotkey key combination. It does this by comparing each scan code with the \[G\] key. If a match is found, it checks the BIOS keyboard flag at address 00417h to see if the \[\text{Alt}\] key has also been pressed. If both are true, it calls procedure \text{Action}.

\text{Action} first checks whether the hotkey was pressed again while saving an image, which would lead to complications. Next, the filename is assigned using consecutive numbers in case several screenshots are made.

Before the actual procedure \text{Hardcopy} is called, the program must determine the current graphic mode since the Grabber is designed specifically for 320 x 200 pixels. This task is performed by the procedure \text{GetMode}.

First, \text{GetMode} checks the BIOS mode, which in both cases is 13h. Any other value and \text{Hardcopy} cannot run. The procedure then ends with a mode of 0. Next, the Chain 4-bit of TS-Register 4 is used to differentiate between Mode 13h and Mode X.

Since Grabber must produce the most realistic image possible with the various effects as well, a number of CRTC-registers must still be read. The first is the Linear Starting Address, which is used for scrolling. The second is the Line-Compare register for the split-line (this register is divided among three CRTC-registers which are recombined at this point).

Finally, the Row Offset is used at higher virtual resolutions (640 x 400). Through Row Offset the distance between two lines (\text{Skip}) is calculated. This value equals 0 at normal 320 x 200 resolutions and 80 at a virtual resolution of 640 x 400.

Next, \text{Action} calls procedure \text{Hardcopy}. It receives the just obtained CRTC values with the filename as parameters. \text{Hardcopy} reads the current palette from the DAC and enables Read Mode 0 if Mode X is active. The palette is brought to an 8-bit format through the procedure \text{PCXShift}, which follows the reverse path from reading GIF files (each value is shifted two bits to the left).

The completed header is now written to the file with the first 16 palette entries and filled to 128 bytes. Next, several variables are calculated. \text{Split} is given as a line and must still be converted to an offset since the
copy loop is based on offset. The start-address in Mode 13h must be multiplied by 4, considering the special (Chain4-) structure of this mode.

In the main loop, the first item calculated is another offset (aux_ofs), which is used for over-wide images (when scrolling in all directions, for example). If the current offset is still less than the split-line, the program simply multiplies the number of lines processed by the space per line. After split-line, the split-line itself must also be considered to obtain a reference to the memory-start.

The ASM block is next. It reads a byte from video memory. First es:si is loaded with a pointer to the current pixel. The offset is divided by 4 in Mode X as we mentioned.

If the split-line has already been passed, the memory-start reference is reset by eliminating the effect of the changed start-address (through subtraction) and calculating the distance to the split-line (also through subtraction). In Mode X the current plane is set as well. Next, the pixel in the variable `Value` is read and compressed as follows.

The first byte only needs to be loaded so it can be processed as the last byte the next time through. Each subsequent byte is then compared to the previous one. The counter `count` is incremented if a match occurs.

A special case arises when the counter goes past the maximum (63) that can be stored in a length-byte. In this case the counter is "emptied" into the file and the count restarts. Another special case involves the end-of-line. Originally, PCX was purely line-based which meant that character repetitions could not go past the end-of-line. You may have to consider this handicap in the compression especially if you’re using an older paint program.

When a character differs from its predecessor, the following Else branch executes. First, the program checks whether this was a repetition (count > 1) or whether the character is using the upper two bits. In both cases a compressed character must be written with a count-byte (equal to 1 in the second case) and a data-byte.

`Lastbyt` is then set to the current value, so compression occurs the next time through the loop. On the other hand if Count = 1 and the last byte is valid (upper bits empty) the Else branch of this If structure executes, which writes the byte directly to the file uncompressed and likewise sets `Lastbyt` to the current value.

After the entire image has been written to the file in this way, the palette (with ID 0ch) is added on and the file closed. The handler is then terminated.

Remember, we mentioned that although this program is not fast, speed isn’t the highest priority for a screen grabber. What is important is that we have shown the structure of a PCX file. This time our example showed writing a file rather than reading a file.

Once the program is called, it stays resident in memory and waits for the hotkey. Only a system reset will remove the program from memory. TSR programs with a deinstallation feature are normally most effective when written in Assembler, however, we were able to avoid repeat installation even in Pascal as seen in the procedure `Check_Inst`. 

55
Chapter 3

Graphic Know-how From Underground

VGA To The Last Bit

Later we'll talk about different graphic effects which rely mostly on simple register manipulations. These registers offer unlimited potential for affecting the VGA display. Many registers in the past remained unused, with apparently no purpose or function. However, by using a little imagination, you can use almost every register to create interesting effects - you just need to start experimenting.

Before you can understand these effects, we'll want to talk about all the VGA registers in detail.

In theory you can experiment with the values. However, you should be careful with the timing registers (CRTC-Registers 0-7) since prolonged operation with extreme values (the monitor will whistle like crazy) can damage the monitor.

With most other registers, the worst that can happen is a system crash. The registers can be divided into two major groups:

- Discrete registers
  These are addressed using separate port addresses.
- Indexed registers
  The term indexed simply means these registers have no port address of their own. Instead they are a component of one of the VGA chips, and are selected using a fixed port address and processed using another port address.

### Miscellaneous Output Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Vertical retrace polarity</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>Horizontal polarity (with Bit 7 vert. resolution)</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Page-select for odd/even addressing</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3,2</td>
<td>Clock select (horiz. resolution)</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Enable RAM, 1=RAM access using CPU enabled</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>I/O address, 1=monochrome (3bxh), 0=color (3dxh)</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 7,6*

The polarity of the two retrace signals determines the vertical physical resolution: 00b is reserved, 01b means 350 lines, 10b 400 lines (also 320 x 200, see CRTC-Register 9), 11b 480 lines.

**NOTE**

This feature can be used by those with malicious intent to damage someone's monitor permanently.

Therefore, we repeat: Use caution...if your monitor whistles loudly switch it off immediately...fast action may save the monitor from permanent damage. It normally takes several seconds before the monitor is damaged. If you try such experiments, we recommend keeping one finger on the ON/OFF switch at all times just in case.
**Bit 5**

Specifies Bit 0 in odd/even mode, that is, in each plane either all even addresses (Bit 5 = 0) or all odd addresses (Bit 5 = 1) are occupied, see TS-Register 4.

**Bits 3,2**

Determines horizontal resolution using pixel frequency: 00b means 640 horizontal pixels, 01b 720 pixels, 02b 800 pixels, 11b is reserved.

**Bit 1**

Regulates the CPU’s access to VGA-RAM (0= access disabled)

**Bit 0**

This bit indicates the port address of CRTC (3 x 4h/3 x 5h), Input Status Register 1 (3 x ah). A 1 in this bit means replace the x with d; a 0 means replace it with b.

### Input Status Register 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CRT-interrupt</td>
<td>RO</td>
</tr>
<tr>
<td>6-0</td>
<td>Manufacturer-specific, usually feature code etc.</td>
<td>RO</td>
</tr>
</tbody>
</table>

### Explanation

**Bit 7**

Indicates a vertical retrace, provided the retrace-interrupt has been activated (usually deactivated through hardware DIP switches). Is cleared using CRTC-Register 4.

### Input Status Register 1

Port 3dah (color) or 3bah (monochrome) depending on Bit 0 Miscellaneous Output Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,6</td>
<td>Reserved (Sometimes inverted Bit 3)</td>
<td></td>
</tr>
<tr>
<td>5,4</td>
<td>Test bits, depending on graphic card</td>
<td>RO</td>
</tr>
<tr>
<td>3</td>
<td>Vertical retrace</td>
<td>RO</td>
</tr>
<tr>
<td>2,1</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Display enable complement</td>
<td>RO</td>
</tr>
</tbody>
</table>
Explaination

**Bit 3**
A value of 1 indicates a vertical retrace is in progress. This bit synchronizes image construction so image construction and modification are not performed at the same time, which leads to flickering (see the definition for Retrace earlier in this chapter).

**Bit 0**
Enable-signal for inverted display. A 1 indicates a horizontal or vertical retrace is in progress. By using this bit you can determine the current screen line being displayed. This bit changes from 1 to 0 after a vertical retrace (Bit 3) which indicates the start of a new line.

Other registers are available, depending on the manufacturer, that reflect the capabilities of a specific card (e.g., Feature Connector). Since these are very specific registers, they’re not suited to general programming.

**Cathode Ray Tube Controller (CRTC)**

Another complex part of the VGA is the Cathode Ray Tube Controller (CRTC). It’s responsible for generating the video signal and is programmable. The cathode ray has a very wide range of motion. Therefore, a protection bit (CRTC-Register 11) protects these registers from accidental overwriting. The CRTC also controls several other registers, such as Linear Starting Address, which are unrelated to ray timing. It’s up to you on how you want to manipulate these registers.

There are two port addresses for accessing these registers:
- Port 3d4h is the index register
- Port 3d5h is the data register (on monochrome displays - Bit 0 of Miscellaneous Output Register = 0 - the addresses are 3b4h/3b5h).

A specific CRTC register is accessed by writing its number to the index register and then reading or writing to the data register. Once you have set the index, you can access the register as often as you like because the index remains valid.

A single write access can be accomplished quickly by outputting a word to the index port. The low-byte contains the register number and the high-byte the register value on the index port.

<table>
<thead>
<tr>
<th>CRTC-Register 0: Horizontal Total</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of characters per scan-line (-5 in VGA)</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation**

**Bits 7-0**
This register gives the total line length in character-units (Character Times Units). One character-unit represents either 8 pixels (TS-Register 1, Bit 0 = 1, for example in 320 x 200 modes) or 9 pixels (TS-Register
1, Bit 0 = 0, for example in Text Mode 3). The actual value for this register still be decreased by 5 (2 in EGA modes).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>End of display-enable signal in characters</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 7-0*

This register usually indicates the number of visible characters (8 or 9 pixels, see Register 0).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Start of horiz. blanking period</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 7-0*

Specifies where the CRTC deactivates the cathode ray during line construction. The blanking period comprises the retrace period and sets the frame to black at the left and right margins.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Test bit (normal operation : 1)</td>
<td>RW</td>
</tr>
<tr>
<td>6-5</td>
<td>Display-enable skew (delay)</td>
<td>RW</td>
</tr>
<tr>
<td>4-0</td>
<td>Horizontal blank end (Bits 4-0)</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 6-5*

Indicates the number of characters the CRTC "previews" in memory (the display-enable signal is delayed accordingly), so the next character is available in time, normally 0 in VGA.

*Bits 4-0*

Stores the lower 5 bits of the 6-bit end of blank period. Bit 5 of this value is found in CRTC-Register 5 Bit 7.
Chapter 3

Graphic Know-how From Underground

CRTC-Register 4: Horizontal Sync Start

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Starting position of horizontal retrace</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 7-0*

Specifies the position (in characters), where the horizontal retrace should begin.

CRTC-Register 5: Horizontal Sync End

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Bit 5 of Horizontal Blank End (Register 3)</td>
<td>RW</td>
</tr>
<tr>
<td>6-5</td>
<td>Horizontal sync skew (delay)</td>
<td>RW</td>
</tr>
<tr>
<td>4-0</td>
<td>End of horizontal retrace</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 6-5*

Specifies the delay after a horizontal retrace (see Register 3). Bits 6-5 are usually 0 in VGA.

*Bits 4-0*

Defines the end of the horizontal retrace, relative to the start. The retrace ends when the internal character counter in the lower 5 bits matches this register.

CRTC-Register 6: Vertical Total

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Total number of screen lines per image -2 (Bits 7-0)</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 7-0*

Specifies the total image height in screen lines (minus 2, minus 1 in EGA). The total height is a 10-bit value (11 bits in Super-VGA). Bits 9-8 are found in the overflow register (Register 7).
CRTC-Register 7: Overflow

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Vertical sync start - Bit 9</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>Vertical display enable end - Bit 9</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Vertical total - Bit 9</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Line compare (split-screen) - Bit 8</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Vertical blank start - Bit 8</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Vertical sync start - Bit 8</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Vertical display enable end - Bit 8</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Vertical total - Bit 8</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 7-0:**
Contains Bits 8 and 9 of most vertical registers, which do not fit in the actual registers.

CRTC-Register 8: Initial Row Address

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>6-5</td>
<td>Byte panning</td>
<td>RW</td>
</tr>
<tr>
<td>4-0</td>
<td>Initial row address</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 6-5**
Shifts the screen contents up to 3 bytes (four pixels in Mode X!) to the left. For greater flexibility, however, use the Linear Starting Address (Register 0ch/0dh).

**Bits 4-0**
Specifies the screen line with which the CRTC should begin after a vertical retrace. Normally this would be Line 0. Increasing the value makes the CRTC start on a lower line, i.e., it shifts the screen contents upward. Since the register works the same in text mode, you can use it to implement a vertical smooth-scrolling (see Chapter 5).

Another application involves 320 x 200 mode. Say you would like to slow down the vertical scroll but still maintain 70 shifts per second. To prevent jittering, proceed as follows:

First, increment the start-address every other time through. This reduces the speed, while fine-scrolling is achieved by oscillating this register between 0 and 1. This way you will be scrolling by one screen line (equals one-half line in 320 x 200 mode) each time.
### Chapter 3

#### Graphic Know-how From Underground

<table>
<thead>
<tr>
<th>CRTC-Register 9: Maximum Row Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4-0</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 7**

Setting this bit to 1 cuts the vertical clock rate in half so each line is displayed twice. This was designed for generating 200-line modes at physical resolutions of 400 lines. Most BIOSes, however, accomplish this by setting Bit 0 (see above).

**Bits 4-0**

Specifies the character height in text mode, minus 1 (15 in VGA-Mode 3, 9 x 16 pixels per character). Can also be used in graphic mode to decrease vertical resolution (multiple-display of each line if Bits 4-0 > 0). For more information see Chapter 5.

<table>
<thead>
<tr>
<th>CRTC-Register 0ah: Cursor start-line</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>7-6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4-0</td>
</tr>
</tbody>
</table>

**Explanation**

**Bits 4-0**

Specifies the screen line within character where cursor display begins.

<table>
<thead>
<tr>
<th>CRTC-Register 0bh: Cursor End - Line</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6-5</td>
</tr>
<tr>
<td>4-0</td>
</tr>
</tbody>
</table>
Explanation

**Bits 6-5**

You can insert a delay when displaying the cursor as well so it will appear at the outermost left and right margins, usually 0 in VGA.

**Bits 4-0**

Specifies the screen line within character where cursor display ends.

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Linear starting address Bits 15-8</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 7-0**

Specifies Bits 15-8 of the 16-bit start-address. The start-address gives the offset in video memory where the CRTC starts reading the image data. Changing the value lets you scroll horizontally and vertically across the screen (see Chapter 5).

In text mode, however, scrolling proceeds by characters. The fine-adjustment occurs through Register ATC 13h (Horizontal Pixel Panning) and CRTC 8 (Initial Row Address). For more information see Chapter 5. Note the true address must be divided by 2 in odd/even mode and by 4 in Chain-4 mode (e.g., 13h), before writing it to this register.

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Linear Starting Address Bits 7-0</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 7-0**

Specifies the low-order word of the screen start-address (see Register 0dh).

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Cursor position as offset (Bits 15-8)</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 7-0**

Specifies the cursor position in video memory.
CRTC-Register 0fh: Cursor Address Low

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Cursor position as offset (Bits 7-0)</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 7-0*

Specifies the low-word of cursor address (see Register 0eh).

CRTC-Register 10h: Vertical Sync Start

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Screen line where vertical retrace begins</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 7-0*

Specifies Bits 7-0 of the 10-bit screen line where vertical retrace begins. Bits 9 and 8 are in the overflow register.

CRTC-Register 11h: Vertical Sync End

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Protection bit</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Vertical retrace interrupt on (0)</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Reset vertical retrace interrupt (0)</td>
<td>RW</td>
</tr>
<tr>
<td>3-0</td>
<td>Screen line where vertical retrace ends</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bit 7*

When set, the protection bit protects CRTC-Registers 0-7 from write accesses. The sole exception is Bit 4 of the overflow register (Register 7). This bit is set by virtually every BIOS and must therefore be cleared for any timing manipulations.

*Bit 5*

If this bit is cleared and the previous interrupt reset by clearing Bit 4, the CRTC initiates an IRQ 2 at the next vertical retrace. Unfortunately, very few VGA cards generate this interrupt because they either do not have the capability or the manufacturer has configured their DIP switches (for compatibility reasons) not to initiate interrupts.
Chapter 3

Graphic Know-how From Underground

Bit 4
After each interrupt this bit must be cleared, otherwise no new interrupt can occur (see Bit 5).

Bits 3-0
The vertical retrace end is also set relative to its start because only 4 bits are used in the comparison. When the lower four bits of the internal line counter match this value, the current retrace ends.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Last displayed screen line (Bits 7-0)</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bit 7-0
After the line number given here, the CRTC disables the screen ray. Bits 9-8 of this value are in the overflow register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Offset between two screen lines</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bits 7-0
Specifies the distance between two lines in memory, i.e., their length. The unit involved here depends on which type of memory addressing is active. In doubleword mode, blocks of eight bytes are counted. For example, in Mode 13h a register value of 40 corresponds to an actual width of 40 \* 8 = 320 bytes. In word-addressing, the unit is 4 bytes (for example in Text-Mode 3, a register value of 40 means 40\*4 = 160 bytes). In byte-mode the unit is 2 bytes (for example Mode X, with the same register value of 40, 40\*2 = 80 bytes). With this register you can also do horizontal scrolling by entering a value of 80 for example. In this case the lines will begin double-spaced and there will be invisible regions of video memory "next" to the visible part. Now if you shift the screen-start by a few bytes, you can scroll horizontally over the virtual screen (see Chapter 5).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Doubleword addressing</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Linear address count by 4</td>
<td>RW</td>
</tr>
<tr>
<td>4-0</td>
<td>Line where underlining begins</td>
<td>RW</td>
</tr>
</tbody>
</table>
Chapter 3

Graphic Know-how From Underground

Explanation

Bit 6

Enables doubleword mode when set to 1. The current address in this mode is rotated two bits to the left before being sent to video memory for a read access. This way, the Offsets 0, 4, 8, etc., are first read in memory, followed by Offsets 1, 5, 9, etc. In Mode 13h, for example, doubleword mode is active.

Bit 5

When accessing video memory with this bit set, the character timing is divided by 4; usually used together with doubleword mode.

Bits 4-0

In monochrome modes, these bits determine the screen line where the ATC performs the underlining. Lowering this value lets you strike through or place a line over the characters; if the register exceeds the character height, underlining is disabled.

<table>
<thead>
<tr>
<th>CRTC-Register 15h: Vertical Blank Start</th>
<th>Bit Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Vertical Blank Start Bits 7-0</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bits 7-0

Specifies the vertical position where the CRTC should deactivate the cathode ray. During this blank period, the vertical retrace is then performed. Bits 9 and 8 of this register are found in the overflow register.

<table>
<thead>
<tr>
<th>CRTC-Register 16h: Vertical Blank End</th>
<th>Bit Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Screen line where vertical blanking ends</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bits 7-0

This register is also relative to the blanking-start because only eight bits are used. The blanking ends when the lower eight bits of the internal line counter matches this register.
**CRTC-Register 17h: CRTC Mode**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hold control</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>Word mode (0) / byte mode (1)</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Alternative setting for Address-Bit 0</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Linear address count by 2</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Line counter count by 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Alternative setting for Address-Bit 14</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Alternative setting for Address-Bit 13</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bit 7*

A value of 0 stops the entire horizontal and vertical timing.

*Bit 6*

A 0 indicates word mode is active and a 1 indicates byte mode is active. Bit 6 of Register 14h is irrelevant if it is set (doubleword mode).

*Bit 5*

When this bit is cleared, Bit 13 instead of Bit 15 is carried over in word mode to Bit 0 (rotated). This prevents word-mode overflow on EGA cards with only 64K.

*Bit 3*

Setting this bit halves the clock rate of the CRTC on video memory. This functions similarly to Bit 5 of Register 14h.

*Bit 2*

When this bit is set to 1, the line counter is incremented only every other line. This doubles the total vertical timing.

*Bits 1,0*

When cleared these two bits cause a reprogramming of Address Line 14 or 13: Bit 0 in this case is mapped onto Bit 13 and Bit 1 onto Bit 14 (only if Bit 1 of this register = 0). The result is that odd addresses are read 8K after the even ones; the two lower bits therefore determine a block. This is used to emulate the 6845 Controller, which uses this method to address video memory. In CGA Bit 0 is set to 0, in Hercules emulation Bit 1 as well.
Chapter 3

Graphic Know-how From Underground

CRTC-Register 18h: Line Compare (Split Screen)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Line Compare Bits 7-0</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 7-0**

Specifies the physical screen line where the CRTC again begins obtaining its data from the video memory-start. This allows you to implement a split-screen. The top half displays the area of memory defined by the linear starting address and the bottom half displays the memory-start (see Chapter 5). Bit 8 of this register is found in the overflow register and Bit 9 in the Maximum Row Address register.

**Timing sequencer (TS)**

The main task of the timing sequencer is memory management. Accesses to the timing sequencer itself are routed on certain planes according to the current configuration. These accesses are combined with the current character set. The TS is also internally responsible for refreshing the video memory.

Like the CRTC, the TS is also addressed using an index register (at Port 3c4h) and a data register (Port 3c5h). Here also you can set a register through a simple word-out.

TS-Register 0: Synchronous Reset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Synchronous reset</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Asynchronous reset</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bit 1**

Clearing this bit forces the TS to perform a synchronous reset, i.e., it resets all registers and then switches itself off until Bits 0 and 1 are set again. Meanwhile, RAM-refresh is deactivated so the reset can finish as quickly as possible to avoid data loss.

This reset should always be performed when TS-registers are going to be changed.

**Bit 0**

Clearing this bit initiates an asynchronous reset, which is the same as a synchronous reset, except the Font Select Register is not cleared, i.e., it retains the current character set.
Graphic Know-how From Underground

Chapter 3

TS-Register 1: TS Mode

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Screen off</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Shift 4</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Dot clocks / 2</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Video load / 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>TS state</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bit 5
When this bit is set the TS disables the screen, enabling faster RAM access by the CPU.

Bit 4
A set bit means the latches are loaded with a quarter of the timing pulse.

Bit 3
In some 320 x 200 modes this bit is set to reduce horizontal resolution from 640 to 320.

Bit 2
A set bit means the ATC latches are loaded with half the timing pulse.

Bit 0
Status 0 sets character width to nine pixels, Status 1 to eight pixels. This value is primarily used for calculating horizontal timing, whose registers are all based on character-units.

TS-Register 2: Write Plane Mask

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Write-access on Plane 3</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Write-access on Plane 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Write-access on Plane 1</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Write-access on Plane 0</td>
<td>RW</td>
</tr>
</tbody>
</table>
Chapter 3

Graphic Know-how From Underground

Explanation

Bits 3-0

You can include (1) or exclude (0) certain planes through this register during write-accesses.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Font B Bit 2</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Font A Bit 2</td>
<td>RW</td>
</tr>
<tr>
<td>3-2</td>
<td>Font B Bits 0 and 1</td>
<td>RW</td>
</tr>
<tr>
<td>1-0</td>
<td>Font A Bits 0 and 1</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 5-0*

Indicates the status of the character set in memory, according to the following code:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>16K</td>
</tr>
<tr>
<td>010</td>
<td>32K</td>
</tr>
<tr>
<td>011</td>
<td>48K</td>
</tr>
<tr>
<td>100</td>
<td>8K</td>
</tr>
<tr>
<td>101</td>
<td>24K</td>
</tr>
<tr>
<td>110</td>
<td>40K</td>
</tr>
<tr>
<td>111</td>
<td>56K</td>
</tr>
</tbody>
</table>

Font A determines the appearance of characters whose attribute-byte, Bit 3, is not set. When this bit is set Font B becomes active.
GDC-Register 0: Set / Reset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set / reset value for Plane 3</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Set / reset value for Plane 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Set / reset value for Plane 1</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Set / reset value for Plane 0</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 3**

Enables Chain-4 mode (1). It uses Address Lines 0 and 1 to select the plane, in both CPU read- as well as write accesses. This mode has application in Mode 13h - by deactivating it and switching the CRTC to byte mode, you will be in Mode X (see Chapter 4).

**Bit 2**

When this bit is cleared, odd/even mode is active. This functions similar to Chain4 mode: Bit 0 of the address line is used to select even and odd planes. In text mode this means for example, that the ASCII codes for characters are stored in Planes 0 and 2, while the attribute bytes are in Planes 1 and 3. It should always correspond to Bit 4 of GDC-Register 5 (inverted bit).

**Bit 1**

Indicates memory configuration: 0 for 64K video memory, 1 for 256K. Only when this bit is set can you use Bit 2 of the character set selection.

**Graphics Data Controller (GDC)**

This chip controls CPU memory access on a slightly higher level than the TS. It manipulates the graphics data so it can be displayed in the proper graphics mode. The GDC contains the four latches not directly accessible from outside. Each of these latches receives one byte from each plane during a CPU access. During a read-access, one plane (depending on GDC-Register 4) is sent to the CPU. During a write access, the latches are linked with this byte (depending on Register 3) and written back to the planes. Like the CRTC, the GDC is addressed using an index port (Address 3ceh) and a data port (Address 3cfh).
Graphic Know-how From Underground

Chapter 3

GDC-Register 1: Enable Set / Reset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set / reset function on (1) for Plane 3</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Set / reset function on (1) for Plane 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Set / reset function on (1) for Plane 1</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Set / reset function on (1) for Plane 0</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 3-0*

Specifies the set/reset values for individual planes, if their set/reset functions are enabled (see Register 1).

GDC-Register 2: Color Compare

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Color compare value for Plane 3</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Color compare value for Plane 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Color compare value for Plane 1</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Color compare value for Plane 0</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

*Bits 3-0*

A set bit indicates the set/reset function is activated for this plane. The corresponding latch is not linked with the CPU-byte in this case but rather with 0 (if the corresponding bit in Register 0 = 0) or 0ffh (bit = 1). This function is not available in Write Mode 1.
Explanation

**Bits 3-0**

These bits play a role in Read Mode 1. See Register 5, Bit 3 (read mode).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>4-3</td>
<td>Function select</td>
<td>RW</td>
</tr>
<tr>
<td>2-0</td>
<td>Rotation counter</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 4-3**

Indicate the type of link that occurs between the CPU-byte and the four latches:

- 0: Move (overwrite)
- 1: AND
- 2: OR
- 3: XOR

Explanation

These linkages are ignored in Write Mode 1 where a direct write is always performed.

**Bits 2-0**

Indicate how many bits a CPU-byte should be rotated to the right before being linked to the latches.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-2</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>1-0</td>
<td>Plane select</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

**Bits 1-0**

Specifies the plane addressed by a CPU read access as a 2-bit integer.
Chapter 3

Graphic Know-how From Underground

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>256-color mode</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Shift</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Odd / even Mode</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Read mode</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>1-0</td>
<td>Write mode</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 6**
Enables 256-color mode, which has a completely different plane-distribution and must therefore be activated explicitly.

**Bit 5**
Set in CGA-Mode 320 x 200, to generate four colors

**Bit 4**
Enables odd/even mode from the standpoint of the GDC (1). Should always correspond with Bit 2 of TS-Register 4 (inverted bit), see above for description.

**Bit 3**
Indicates the active read mode:

- Mode 0:
  The GDC reads the four latches and sends the contents of the latch selected in Register 4 to the CPU.

- Mode 1:
  The GDC reads the four latches and compares each of their bits with the Color Compare Register (if not excluded using Register 7 - Color Care). It first compares the combination from Bits 0 of the four latches, then from Bits 1, etc. The corresponding bit is set in the byte sent to the CPU when a complete match occurs.

**Bits 1-0**
Indicates the current write mode:

- Mode 0:
  The CPU-byte is linked with all four latches and, depending on Register 2, is written to certain planes. Only those bits are linked which were permitted by Register 8. When the set/reset function (see
Register 1) is active for a particular plane, the set/reset value is used instead of the CPU-byte. This mode is especially suited for manipulating individual bits.

- **Mode 1:**
  In this mode the latch contents are written directly to the planes selected by TS-Register 2; the CPU-byte is irrelevant. This mode is good for fast copying of large screen areas.

- **Mode 2:**
  This mode expands each of the lower four CPU bits to a byte (0 becomes 0 and 1 becomes 0ffh), and links them with the corresponding latches as in Write Mode 0.

- **Mode 3:**
  Each CPU-bit not masked using Register 8 is linked with a bit from the Set/Reset Register and written to the plane corresponding to this bit. Thus Bit 0 is first linked to Bit 0 of the Set/Reset Register corresponding to Function Select Register 3, and written to Plane 0 as Bit 0. Then Bit 0 is linked with Set/Reset Bit 1 and written to Plane 1. The same occurs for all selected bits in the CPU-byte.

<table>
<thead>
<tr>
<th>GDC-Register 6: Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>7-4</td>
</tr>
<tr>
<td>3-2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Explanation**

**Bits 3-2**

Indicate the video memory range:

- 0: 00000h-00000h
- 1: 00000h-00000h
- 2: 00000h-00000h
- 3: 00000h-00000h

**Explanation**

**Bit 1**

A 1 means that odd/even mode is active.

**Bit 0**

A 1 means a graphic mode is active, otherwise a text mode is active.
Chapter 3

Graphic Know-how From Underground

### GDC-Register 7: Color Care

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Color compare active for Plane 3</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Color compare active for Plane 2</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Color compare active for Plane 1</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Color compare active for Plane 0</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 3-0*

When a bit is set, its corresponding plane is included in a color-compare in Read Mode 1 (see Register 5); a 0 excludes it.

### GDC-Register 8: Bit Mask

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Mask for CPU-byte</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

*Bits 7-0*

A set bit means the corresponding CPU-bit is linked to the latches; a cleared bit means the latch contents are transferred unchanged.

**Attribute Controller (ATC)**

The task of the ATC is color management. It’s the second highest authority in a VGA card. Using this controller is somewhat more complicated than the others since only one port is available for write access. An index/data flip-flop occurs at 3c0h, i.e., this port alternates with each write access between index modes and data modes.

You can set the initial status of this register explicitly to index mode through a read access on Input Status Register 1 (Port 3dah in color mode, 3bah in monochrome mode). A write access is then performed. First, the index is written to 3c0h. This is followed on the same port by the data byte which is followed again by the next index.

The read access is somewhat different. After the index has been written, the data byte can be read using Port 3c1h. Here a read access on 3c0h would return the index.
Another unusual feature is the structure of the Index/Data Register at Port 3c0h. Bits 4-0 as usual give the index. However, Bit 5 has a broader meaning:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-0</td>
<td>ATC index</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Palette-RAM access</td>
<td>RW</td>
</tr>
<tr>
<td>7-6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 5**

A value of 0 enables CPU-access to the palette-RAM (Registers 0-f), yet disconnects the ATC so the image is set to the frame color. You should always reset this bit to 1 after any changes.

**Bits 4-0**

Give the index on an internal ATC register. This value is then written using Port 3c0h, and read using Port 3c1h.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>DAC color</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 7**

These registers are used to supply all EGA color values with the actual colors. The red, green and blue values in EGA have been stored here directly. However, the Digital to Analog Converter (DAC) in VGA acts as an interface. The DAC's first 16 palette entries simulate the same color scheme. Here, for example, you can set text colors to new DAC values.
**ATC-Register 10h: Mode Control**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Source for Color Circuits 4 and 5</td>
<td>RW</td>
</tr>
<tr>
<td>6</td>
<td>PelClock / 2</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Enable pixel panning</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Blinking on</td>
<td>RW</td>
</tr>
<tr>
<td>2</td>
<td>Line graphics enabled</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>Monochrome- (1) / color (0) attributes</td>
<td>RW</td>
</tr>
<tr>
<td>0</td>
<td>Graphic (1)/ text (0)</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bit 7**

In graphic modes with 16 or fewer colors this bit determines the source for Color Circuits 4 and 5. A 1 means these bits are taken from Bits 0 and 1 of Register 14 (Color Select). A 0 will load the lines with palette register values. When using the Color Select method you can shift the DAC palette area in blocks of 16. This also applies of course to EGA mode and CGA mode.

**Bit 6**

A 1 halves the speed at which pixel data are sent to the DAC. This finds application in 320 x 200 x 256 mode, since here only half the horizontal resolution is used.

**Bit 5**

A 1 prevents pixel panning below the split-line, a 0 causes panning over the entire screen.

**Bit 3**

A cleared bit enables use of all 16 ATC palette colors. A 1 enables blinking. When this bit is set, the top and bottom halves of the palette in the Intensity Plane alternate continuously. This happens in both text mode and graphic mode so only eight colors are available. The advantage is the palette can be freely constructed, enabling almost any type of blinking effect.

**Bit 2**

A 1 doubles the eighth column of characters in 9-pixel text modes (e.g. Mode 3) with ASCII codes between 0c0h and 0dfh. This allows line characters to be connected without gaps. With a 0 in this bit, the ninth column is cleared or acquired from the Intensity Plane (Plane 3).

**Bit 1**

A 1 switches to monochrome attributes (blinking, underline), a 0 to color attributes.
Bit 0
A 1 switches the ATC to graphic mode, a 0 to text mode.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>Palette-RAM access</td>
<td>RW</td>
</tr>
<tr>
<td>4-0</td>
<td>ATC index</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bits 7-0
Determine the color number used in the overscan area. This is also where EGA cards with a 6-bit RGB value entered. Color mixing with VGA occurs using the DAC.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>5-4</td>
<td>Reserved, often test bit configuration</td>
<td>RW</td>
</tr>
<tr>
<td>3-0</td>
<td>Enable plane</td>
<td></td>
</tr>
</tbody>
</table>

Explanation

Bits 3-0
Enable (1) and disable (0) the corresponding plane, excluding from the display specific color components (16-color models) or specific pixels (256-color models).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>Reserved</td>
<td>RW</td>
</tr>
<tr>
<td>3-0</td>
<td>Horizontal pixel panning</td>
<td>RW</td>
</tr>
</tbody>
</table>

Explanation

Bits 3-0
Specifies the number of pixels by which the entire image (text- or graphic mode) is shifted to the left. The actual values, however, are somewhat different. The value 8 in 9-pixel modes indicates that no shift will take place. Values from 0-7 create shifts of 1-8 pixels. This makes little sense in graphic modes since you might
as well work with the start-address. In text mode however this will allow very smooth scrolling (see Chapter 5).

<table>
<thead>
<tr>
<th>ATC-Register 14h: Color Select</th>
<th>Bit</th>
<th>Meaning</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3-2</td>
<td></td>
<td>Color Circuit 7-6</td>
<td>RW</td>
</tr>
<tr>
<td>1-0</td>
<td></td>
<td>Color Circuit 5-4</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Explanation**

**Bits 3-2**

In graphic modes with fewer than 256 colors, these bits give the upper two bits of each color value sent to the DAC. By reprogramming this register you can move the 16 EGA colors to Palette Offsets 0, 64, 128 and 192.

**Bits 1-0**

If Bit 7 is set in Mode Control Register 10h, in graphic modes with fewer than 256 colors these two bits set Color-Bits 5 and 4. In combination with Bits 3-2, EGA colors can then be moved to any color offset divisible by 16.

**Digital to Analog Converter (DAC)**

The DAC represents the highest-level authority for color generation. The DAC is where 8-bit colors, either originating directly from RAM (256-color modes) or generated from 4-bits by the ATC, are converted to analog signals using the external palette. The signals, separated by red, green and blue, are then sent to the monitor.

The "external palette" refers to the fact this RAM originally existed outside the actual chipset. Meanwhile, this chip has also been integrated. Therefore, the DAC uses an external palette encompassing 256 color values, each of which consists of three bytes (one for red, green and blue). Only the lower six bits of each byte are used, however. Therefore, there are $2^6 = 262,144$ hues available for each color.

Before you can read or set a palette, you must first specify the start-color from which you plan to read or write. Then through a series of In or Out instructions, consecutively read or write the red, green and blue components of each individual color. All values can be written directly one after the other because the DAC automatically increments the pointer. In this context, we advise against the "compatible" method of controlling the palette through the BIOS. It's not significantly more compatible and the execution time is over ten times longer.

The DAC provides access to this palette through the following five registers:
### DAC-Register

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel Mask</td>
<td>RW</td>
</tr>
<tr>
<td>Port</td>
<td>3c6h</td>
</tr>
</tbody>
</table>

**Explanation**

This register normally contains the value 0ffh. It denotes a mask, with whose help certain colors can be mapped onto others. During image generation each time the DAC wants to read a palette entry, an AND operation first occurs between this register and the color number.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel Write Address</td>
<td>RW</td>
</tr>
<tr>
<td>Port 3c8h</td>
<td>3c6h</td>
</tr>
</tbody>
</table>

**Explanation**

Prior to a write access this register specifies which color you wish to change. You can then set the palette or only parts of it using Port 3c9h (Pixel Color Value).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel Read Address</td>
<td>WO</td>
</tr>
<tr>
<td>Port 3c7h</td>
<td>3c6h</td>
</tr>
</tbody>
</table>

**Explanation**

Prior to a read access this register specifies which color you wish to read. You can then read the color values using Port 3c9h (Pixel Color Value).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel Color Value</td>
<td>RW</td>
</tr>
<tr>
<td>Port 3c9h</td>
<td>DAC-Register</td>
</tr>
</tbody>
</table>

**Explanation**

This is the port where the DAC makes color values available for a read or write, after being initiated using Pixel Read Address or Pixel Write Address.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC State</td>
<td>RO</td>
</tr>
<tr>
<td>Port 3c7h</td>
<td>DAC-Register</td>
</tr>
</tbody>
</table>

The lower two bits of this register specify the DAC status - a 0 means the DAC is making data available for a read. An 11b indicates the DAC is ready for a write access.
Super-VGA cards contain several additional registers including discrete registers and expanded controller registers. They sometimes come with entire auxiliary controllers, for example to take over the task of memory management. Since these registers are not standardized, we cannot provide a general description here.

As we have seen however, a wide variety of effects is possible even with standard registers. Experiment with the registers and see if you can create new ones.
Although graphics Mode 13h is easy to use, it has a few disadvantages. Unfortunately, these disadvantages make it difficult to program many sophisticated effects. Although hardware has undergone great technological advances over the last few years, current processors and bus architectures are still unable to paint large screen areas within a single retrace.

Mode X Offers Superior Graphic Power

Image quality usually suffers when the screen cannot be constructed within a retrace. We can prove this with a simple example. A sprite is moving horizontally across the screen. If the electron beam travels over this area while a change is in progress, the top half will contain the new image, while the bottom half will still contain the previous, unchanged image.

There will be flickering and the sprite will appear split and distorted. Therefore, if changes during image construction and other time consuming activity during retraces are avoided, only one option remains. The video pages must be preconstructed "invisibly" and activated during the vertical retrace. For this, we need a minimum of two video pages: One is displayed while the other is being built.

These pages can simply be stored adjacent to each other in video memory and selected using Registers 0ch and 0dh (Linear Starting Address). The problem in Mode 13h, however, is the total image size is already close to 64K (exactly 64,000 bytes). A single video page occupies the entire video memory (0a0000h-0affffh) within main memory. This makes it impossible for the CPU to detect the second page. Modifications are possible only through VGA segment-selectors. However, each manufacturer programs them differently. In fact, an IBM VGA card offers no way to do this. The solution to this problem is Mode X.

Another disadvantage of Mode 13h is access speed. A moving image usually has a static background. It must be recopied to the current page with each image construction. Even with fast doubleword-access in Mode 13h using a VESA local bus, it takes approximately 10% longer to copy a full video page than in Mode X. The time with other bus systems can be even greater. In addition, if you're still using word access to provide support for 286es for example, the slow speed of copying becomes unacceptable.

Mode X on the other hand, as we'll show in the following sections, copies four pixels at a time using byte-access. It also uses Read-Mode 0 and Write-Mode 1, which require no time-consuming internal address conversions and send no data to the CPU. This explains the speed advantage of Mode X over 32-bit accesses made by the CPU.
Initialization

What properties should Mode X have? The most important property of Mode X is disabling the Chain-4 mechanism, which enables access to individual planes. In addition, Odd/Even Mode (plane-selection using lowermost offset bit) must be disabled by clearing Bit 3 (Enable Chain4) in TS-Register 4 (Memory Mode) and setting Bit 2 (Odd/Even Mode):

```c
port[$3c4]:=4;
port[$3c5]:=port[$3c5] (and (not 8)) or 4;
```

Depending on the graphic card (compatibility), you must also change memory access to byte-addressing. First, clear doubleword-addressing from Bit 6 in CRTC-Register 14h (Underline Row Address), then set Bit 6 in CRTC-Register 17h (CRTC-Mode):

```c
port[$3d4]:=$14;
port[$3d5]:=port[$3d5] and (not 64);
port[$3d4]:=$17;
port[$3d5]:=port[$3d5] or 64;
```

Next, clear the video memory because the areas not used by Mode 13h (which now become visible) may still contain “garbage” from other video modes. Use Register 2 of the Timing Sequencer Write Plane Mask for this. To clear the video memory, enable all planes within this register so 32,000 word accesses or 16,000 dword accesses are enough to clear all four video pages.

Structure

As we mentioned, all plane-based graphic modes have four bytes “hidden” behind each memory address, one for each plane. The four bytes appear to lie one behind the other at each address, hence the term “plane” (level).
The planes are virtually independent storage areas which are separately addressable but are used together to display an image on the screen. In other words, data from all four planes is read simultaneously.

Aside from this however, there are major differences between 16-color modes and Mode X. Since sixteen colors can be represented by four bits, they can also be represented by the four planes. Namely, Bit 0 of a pixel is stored in the corresponding bit of Plane 0, Bit 1 in Plane 1, etc. It’s different in Mode X, however, since four bits are no longer enough to address one pixel. Each pixel occupies one byte of a particular plane in Mode X.

The planes are filled byte by byte, i.e., Pixel 0 is located at Offset 0 in Plane 0, Pixel 1 at Offset 0 in Plane 1. Plane 0 is next used for Pixel 4 (this time at Offset 1).

For addressing in Mode X, you must first calculate the offset for the pixel. Take the pixel number (calculated the same as in Mode 13h: 320 y + x) and divide by 4 (shift two to the right). You can also incorporate this division directly into the offset calculation by multiplying the y-coordinate by 80 (320 / 4) and dividing the x-coordinate by 4. The remainder (pixel number +3) determines which plane will be used. The following is how the formula notation might read:

\[
\text{Plane} := X \mod 4 \quad \text{\{gives the remainder\}}
\]

\[
\text{Offset} := Y \times 80 + X \div 4
\]

This procedure corresponds with the one normally used by Mode 13h with one difference. The offset is derived by shifting the pixel number two bits to the right and not by masking. There are, therefore, no gaps between the pixels in memory. This way four pages can fit into a 256K video memory.

It’s important to note the differences between read/write accesses when selecting the plane in Mode X. The plane number is written to Register 4 (Read Plane Select) of the GDC when reading. However, it’s possible to address several planes at once when writing (as we have seen when clearing the screen). To do this, a mask is set in TS-Register 2 (Write Plane Mask), which must first be generated from the plane number. It looks like the following in pseudocode:

\[
\text{Mask} := 1 \text{ shl Plane-Number} ;
\]

Therefore, the mask from Plane 2 is 1 shl 2 = 4 = 0100.

One more item regarding byte-wise addressing by the CPU. It’s very tempting to use 32-bit accesses on the graphic data or, at the very least, 16-bit (theoretically possible even on an XT).

Anyone who tries this, however, will soon find out differently. A completely distorted image will appear on the screen. This can be explained by how the CPU accesses memory. When the CPU copies a word (or
doubleword) using movsw, it doesn’t separate this operation into individual byte-movements. Instead, it reads the word completely and writes it back completely.

VGA, however, can accept only four bytes in its four latches, which actually function as intermediate storage. Therefore, after the read access, only the four high-bytes of the word will be present there. In the subsequent write access, the four high-bytes as well as the four low-bytes are set to the same value (the one corresponding to the latch). The result on the screen is that two (four with 32-bit accesses) adjacent blocks of four will always have the same contents. A normal image display is therefore impossible.

**Higher Resolutions In Mode X**

Mode X has the important advantage of working with four video pages. It also maintains the same resolution as Mode 13h. There are certain applications, however, that require a higher resolution. Of course, Super-VGA resolutions are available (although handled differently by each card) through VESA drivers. The larger problem arises when you want to use multiple video pages in the higher resolution. Actually, there are few VGA cards still available with less than 1 Meg of memory, so even 800 x 600 mode could theoretically be handled by two pages of video memory. Video page management similar to the Linear-Starting-Address Register is not anticipated under VESA. Direct programming again brings up compatibility problems unavoidable with Super-VGA.

In our search for a higher resolution mode which also supports the page concept, we see that Mode X supports four entire video pages, but that in many cases only two are needed. Doubling the resolution to 320 x 400 is therefore no problem. Why 320 x 400 and not 640 x 200? This has to do with the actual structure of 200-line modes on a VGA card. Namely, VGA cannot explicitly address 200 lines (Miscellaneous-Output Register, Bits 6-7 allow only 350, 400, 480 and sometimes 768 lines as values for vertical resolution).

The fact that 200-line modes are still possible is due to a feature called double-scan. This simply means that at a physical vertical resolution of 400 lines, each line is displayed twice, i.e., doubled in the y-direction. This results in a halving of the vertical resolution. Here, neither the horizontal nor the vertical timing need to be changed because physically 400 lines are still displayed for each 320 pixels.

Disabling this mechanism is very easy. You only need to clear five bits in Register 9 of the CRTC (Maximum Row Address), Bit 7 and Bits 0-3. Depending on the VGA-BIOS, doubling will occur using Bit 7 (Double Scan Enable), which is the bit actually designed for this purpose, or Bits 0-3 which show the number of scan-lines per character-line in text mode. This value minus 1 is then written to the register. This register value in graphics mode indicates how many additional copies of the line will be displayed. By increasing the value you can keep lowering the vertical resolution down to 320 x 25 or even 320 x 12.5 with Bit 7. Although this makes no sense, it does demonstrate the purpose of this register.

To get back the original 400 lines, simply clear the above mentioned bits in the register in pseudocode:

\[ \text{CRTC}[9] := \text{CRTC}[9] \text{ and } 01110000b \]

The actual Assembler code for this line is found in the procedure **Enter400** in the MODEXLIB.ASM module:
Enter400 proc pascal far
    ; switches from mode X (200 rows)
    mov dx,3d4h                   ; to extended 400 row mode
    mov al,9                      ; CRTC register 9 (maximum row address)
    out dx,al                     ; select
    inc dx                        ; read out value 2
    in al,dx
    and al,01110000b              ; clear bits 7 and 3:0
    out dx,al                     ; and write back
    ret
Enter400 endp

As we mentioned, no timing changes are necessary. Also, rewriting the Mode X routines is unnecessary since the structure remains the same. When switching to 400 lines, the 200 lines of normal Mode X are "pushed together" at the top. This clears the view for the lower 200 lines which immediately follow in VGA memory.

There is now a larger range of values in all routines for coordinates. These values are freely selectable (for speed, validity checks are omitted). The only change involves the video pages: Instead of four 64000-byte pages, now only two 128000-byte pages are available.

This affects switching in this case because the start-address (vpage) is no longer switched between 0 and 16000 but between 0 and 32000. Also, there is no longer a "reserve page" available for a background image because almost the entire 256K is used (in the end, 256K - 256000 bytes = 6K are still available and can be used for small (very small) sprite backgrounds).

Backgrounds in this mode must therefore be copied into VGA from main memory (using p13_2_ModeX, see the "Expanding The GIF Loader For Mode X" section in this chapter).

Incidentally, you must also consider this mode’s unusual page ratio of 320:400 when drawing a background. The page ratio generates very flat, wide rectangles. Circles are actually very flat ellipses. Since, as far as we know, no paint program supports the 320 x 400 resolution, it’s best to draw your images in a "square" resolution such as 640 x 480 and later convert them to 320 x 400.

In addition to 320 x 400, it’s possible to generate graphic modes with even higher resolutions. Therefore, you can have such odd resolutions as 512 x 400 or 320 x 480. Despite their highly complicated initialization, which requires a complete reprogramming of the timing registers, these modes have no significant advantage over Super-VGA modes because they greatly exceed the maximum memory use for page-programming.

Although one page can be displayed on any standard VGA, two pages will exceed the upper address limit (256 bytes). You’ll face incompatibility problems of Super-VGA cards at higher addresses so you should use a Super VGA mode from the start and have at least a reasonable page ratio.

### Using Mode X

It’s sometimes necessary, even in Mode X, to address individual pixels and change their color. However, for applications that rely almost exclusively on single-pixel modifications (for example the star-scroller), we recommend using Mode 13h, provided you can do without the four video pages. Mode 13h makes it easier to address individual pixels and it’s also faster. This is because breakdown of offsets into memory offset and plane number occurs internally on the VGA and does not have to be performed by the CPU.
This also saves you from outputting the plane number on the port-addresses of the TS and GDC, which on some motherboards can be very slow. Some chipsets on fast computers (386 and above) include several wait states when accessing the ports to give the hardware enough time for the data transfer with today’s hardware. This is, however, really not necessary.

### Setting pixels

You can use the following procedure when you need to change a few pixels. It can also be used in the Mode 13h star-scroller if you initialize Mode X instead of Mode 13h.

```assembly
Procedure PutPixel(x,y,col:word);assembler;
{sets pixel (x/y) to color col (Mode X)}
asm
  mov ax,0a000h          {load segment}
  mov es,ax
  mov cx,x               {define Write Plane}
  and cx,3               {as x mov 4}
  mov ax,1
  shl ax,cl              {set appropriate bit}
  mov ah,al
  mov dx,03c4h           {Timing Sequencer}
  mov al,2               {Register 2 - Write Plane Mask}
  out dx,ax
  mov ax,80              {Offset = Y*80 + X div 4}
  mul y
  mov di,ax
  mov ax,x
  shr ax,2
  add di,ax
  mov al,byte ptr col
  mov es:[di],al        {and set pixel}
End;
```

By masking out the lower two bits of the x-coordinate you determine the plane. This plane must still be converted to the format of the Write Plane Mask Register by setting the bit corresponding to the plane number. You then calculate the offset (shifted 2 bits to the right compared to Mode 13h) and set the color.

### Switching pages

The main advantage of Mode X is definitely its ability to use several video pages. In most cases you would be continually switching between video pages 0 and 1. A new image, with repositioned sprites for example, is drawn on the invisible page and then brought to the screen. The process then continues with the role of the pages reversed.

BIOS Function 5 is used to page in text mode. Unfortunately, this is of little use in Mode X. Even so, the task is quite simple. As we mentioned, the four video pages are arranged linearly in memory, one following the other: Page 0 occupies Offsets 0-15999, page 1 Offsets 16000-31999, etc. All you need now is a way to determine where the CRTC begins the image display. This function is performed by CRTC-Registers 0ch and 0dh (Linear Starting Address). Register 0ch contains the high-byte and Register 0dh the low-byte of the starting address.
You can usually set the start-address with the procedure `SetStart` in MODEXLIB.ASM, where the address to be set is passed as a parameter. This procedure splits the given address into high bytes and low bytes and writes it to the registers mentioned above.

The procedure `Switch` is somewhat more sophisticated. It uses the global variable `vpage` to store the start-address of the invisible page. This procedure also takes care of page management. Therefore, you can always construct your image at the page specified by `vpage`, which is currently invisible. Calling `Switch` then makes this page visible, while `vpage` points to the new invisible page.

As we mentioned, registers `0ch` and `0dh` contain the screen-start offset and not a page number. So, you can also enter intermediate values. We'll talk about this in more detail in Chapter 5.

### Expanding The GIF Loader For Mode X

We’ve mentioned the GIF format is the preferred graphic format for demos. Therefore, we’ll also need a way to load and display GIF images in Mode X. This is not a problem since the loader is designed to unpack images in main memory from where they are simply copied in Mode 13h to video memory using the Pascal Move command. Only this last step, copying, must be rewritten to conform to the altered memory organization of Mode X.

We use procedure `p13_2_modeX` in the MODEXLIB.ASM module for this purpose. It copies an image of size `pic_size*4` from main memory (pointer in VScreen) to Mode X at the starting address `start`. For example, to copy a 320 x 200 image (just loaded by LoadGIF) to video page 2, enter the call as follows:

```
p13_2_modeX(2*16000,64000 div 4);
```

The difficulty with such a copy procedure is in dividing the image among the various planes. The simplest, but by far the slowest, option is moving pixel by pixel, i.e., loading a source pixel, selecting the destination plane and then writing the byte. A more efficient option, however, is first copying all plane 0 pixels (located at Offsets 0, 4, 8 ... of the source image), then plane 1 pixels, etc. This saves you from continually changing the Timing Sequencer:

```pascal
p13_2_modeX proc pascal far start,pic_size:word
    mov dx,03ceh                ;set write mode 0
    mov ax,4005h                ;via GDC register 5 (GDC mode)
    out dx,ax
    mov b plane_l,1             ;store plane mask
    push ds
    lds si,dword ptr ds:vscreen ;load source address
    mov w plane_pos,si          ;and store
    mov ax,0a000h               ;set destination address
    mov es,ax
    mov di,start
    mov cx,pic_size             ;get number
@lpplane:
    mov al,02h                  ;TS register 2 (write plane mask)
    mov ah,b plane_l            ;mask corresponding plane
    mov dx,3c4h
    out dx,ax

    @lp1:
```

The procedure `p13_2_modeX` is part of the MODEXLIB.ASM file on the companion CD-ROM.
Chapter 4

Mode X: The "Secret" To Great Graphics

movsb ;copy byte
add si,3 ;position at next source byte
loop @lp1
mov di, start ;position at next source byte
inc w plane_pos ;get destination address again
mov si, w plane_pos ;source address to new start
mov cx, pic_size ;get size
shl b plane_l, 1 ;mask next plane
cmp b plane_l, 10h ;all 4 planes copied ?
jne @lpplane
pop ds
ret
Endp

The variable plane_l, which contains the current plane mask, is first initialized to 1 so it can address Plane 0. The source data pointer is loaded into ds:si and its offset stored in plane_pos. The plane-loop @lpplane begins after loading the destination address into es:di and the image length into cx. It selects the correct plane with the help of the current mask (Timing Sequencer, Register 2 - Write Plane Mask). The inner loop @lp1 copies one byte and moves 3 bytes forward in the source image (movsb already incremented si by 1), so it can read the next byte for the same plane.

When this loop is finished running, one plane is already at the correct location. To copy the next plane, the destination pointer is reset to the beginning and the destination address incremented by 1. So, when copying plane 1, for example, offsets 1, 5, 9, etc., will be read. Cx is reloaded and the new plane masked by rotating 1 to the left. The terminating condition is the masking of the nonexistent fifth plane, in other words when all four planes have been copied.

On the Mode X screen there now appears a video page from main memory, stored in Mode 13h format. You can copy it from this point to any page with a fast copying mode, for example to erase an old image and restore the old background.

This method of plane-wise copying is also used for displaying sprites which we’ll describe in Chapter 6. However, since special requirements apply here, a completely different procedure is used.

A Simple Text Scroller

Anyone who has worked with computers such as the Commodore 64 or Atari ST may remember early versions of scrollers - programs that make text "walk" across the screen. Here, we’ll demonstrate a simple scroller using Mode X.

Scroller in Mode X

Basically a scroller moves a text from right to left across the screen, past the "eye" of the observer. You could program something like this in just a few minutes in text mode. Take a character string and a pointer indicating the current position of the left screen edge within the string. Then, write 80 characters 70 times per second starting from this current position, perhaps onto a particular screen line. Then increment the pointer by one position. This moves the window (currently visible text section) through the string in the direction of reading. You now have your scroller.
Programming a scroller in graphic mode is somewhat more complicated. Although the principle is the same, the programmer has no BIOS routines available for character output in Mode X or very limited BIOS routines available in Mode 13h. Everything is therefore the programmer’s responsibility (which actually can be an advantage for display speed).

Another important difference from a text-mode scroller is duplicating the forward motion of the screen contents. You can display a new character string at the new starting position in text mode. However, the speed factor is much more critical in graphic mode, so you no longer have the luxury of assembling each character from the font information.

There is a much simpler way to do this, however. With this basic scroller form, the text only moves and no longer changes once it’s on the screen. You can therefore simply copy it graphically one position further by shifting columns 4-319 four pixels to the left and adding the new “text” in the “empty” columns 316-319. These four-pixel steps make effective use of the advantages of Mode X, and provide a simple way to copy four-byte blocks in Write-Mode 1.

To shift a number not exactly divisible by four (three for example), you would have to copy contents from plane to plane, which is possible only by accessing individual pixels. Copying from one offset to another without switching planes however can be used on four pixels at a time, significantly increasing the speed.

Font for scroller text

All you need to do now is display a four-pixel wide strip of new text at the right edge of the screen. Where does the data to be displayed (the font) come from? A simple solution of course is to use the VGA-ROM font, but this provides neither color nor elaborate graphic designs. Designing your own font is not for everyone but is essential for visual presentations.

Some programmers write their own editor for this purpose which includes features of a paint program and the ability to generate fonts. However, the effort required to do this usually outweighs its usefulness. Why spend time writing an editor when the demo for which you are creating the font can be finished in half the time? Also, public domain or shareware paint programs are capable of drawing basic objects such as lines, circles, section-copying, etc.

In fact, public domain, shareware or commercial paint programs are the better alternative for another reason. Saving a complete alphabet in a single image also saves valuable hard drive space since the image can be saved in compact GIF format.

How you design the characters themselves is up to you. If you’re artistically inclined you can always create a new font. Another option is to modify or edit an existing font. Remember, the characters must meet the requirements for further processing when you’re finished. The routine that appends the new columns is based on blocks of four so the characters must be arranged similarly. Both the x-coordinate and the character width must be divisible by four while also allowing for varying character widths. This also lets you use proportional fonts.

After you’ve properly arranged all the characters, comes the hard part: Entering all the character positions. The character output procedure uses these positions to access the font by reading their location in video memory and their width from the table.

To create the table, just take paper and pencil, with a paint program that continuously displays the cursor position. Be careful however, because some paint programs start counting coordinates at 1, so the upper left
corner of the screen has coordinates (1,1). Although perhaps convenient for the program's developers, it conforms neither to mathematical reality nor to the conventions of your computer. The coordinates must always conform to a 0-based system. You can always subtract 1 if necessary (this applies also when positioning in blocks of four). Based on the coordinates of the upper left corner, the offset in video memory is calculated according to our previously derived formula:

\[
\text{Offset} := \text{Y} \times 80 + \text{X} \div 4
\]

You must enter this value into the table for each letter. The plane is always 0 because that was the requirement for using the fast write mode. To determine the width, first measure the letter in pixels (either in your head or with the length function in the paint program) and round it up to the next integer multiple of four. This will give you the width in blocks of four, which is essential to the process. Enter this value into the table as well.

Once you're finished with the table, you must still transfer it to the program. The ASCII sequence should be retained to keep the programming as simple as possible. For example you could make a table from ASCII code 32 (space) to 96 (accent grave `) and assign a width of zero to unused characters (i.e., ASCII character 64 - @).

Of course you could also include the lower case letters (97-122) or the entire ASCII set if you like, because you can have a small graphic for each character. Normally however, uppercase letters, numbers, and a few special characters are enough.

The previous program uses a small demo font which we've included on the companion CD-ROM.
Chapter 5

The term "graphic effect" has assumed an entirely new meaning in the last few years. We used to consider a simple static image created with a paint program as a "graphic effect". Today graphics and graphic effects now include animation, moving sprites or other types of visual movement. These graphics effect are usually processed by the CPU. However, we'll talk about effects which can be handled independently by the VGA; whereby the CPU only gives the command to initiate the process

Basics

All the effects we'll talk about in this chapter can also be generated in some way using the CPU. However, this involves shifting large amounts of data within video RAM. It's very slow to move data this way. DMA can accelerate these movements only slightly (direct CPU access may actually be faster on most computers). In any case, accessing video memory usually ends up with a bottleneck at the data bus.

Fast access to video memory is limited on the earlier ISA-bus since even the fastest CPU's are constrained by the 8 MHz bus speed needed for 'downward' compatibility. Also, if an extra Hercules card is installed, it switches the 16-bit VGA card down to 8 bits which again doubles the access time.

Even current bus systems that use a 16-bit or even 32-bit data bus at a speed of 33 MHz can still create a "traffic jam" for the data. This is because strict protocols must be maintained, several wait states are often added and some VGAs are unable to keep up and add wait states of their own.

So, there must be another way to bring the type of effects to the screen that we see in countless demos. These demos display dazzling effects and yet still manage to play sound and perform the calculations for three-dimensional objects. All this is possible if we use the built-in capabilities of the graphic card.

PC graphic cards do not yet have programmable controllers, processors, etc., designed specifically for graphic effects (like the Amiga), where the CPU performs only control and supervisory tasks. VGA cards do, however, have many registers and combinations of register contents that if you look closely, you'll discover possibilities never considered by early designers of VGA.

However, there's no substitute for actually experimenting with the register contents. Providing you don't change the timing registers of the CRTC, your experiments cannot damage or destroy any hardware. The worst that could happen is the image or effect is distorted or doesn't appear.

One area which could lead to dangerous situations is if you change CRTC Registers 0-7. These registers usually affect horizontal timing and can "blow out" a monitor or graphic card when used incorrectly. Disordered timing can cause the electron beam to move wildly across the screen or use frequencies which are too high. This can soon overheat the monitor's deflecting coils. Inexpensive monitors can actually be destroyed this way.
Chapter 5

Split-screen And Other Hot Effects

If this happens, press the Esc key to return to the Turbo Pascal editor.

To avoid the risk of damaging your monitor don't play with these registers. We've never seen any worthwhile effects produced with them anyway...reprogramming the timing generally leads to images so distorted as to be useless.

This should not stop you from experimenting with other registers, however. As we’ll talk about in the next section, the registers offer possibilities far beyond their original design.

How Split Screen Works

One common display technique is to divide the screen into two separate areas, for example, a scrolling area in the top half and a fixed status bar on the bottom. You can, of course, assign this task to the CPU. However, if execution speed is critical you should determine what hardware programming options are available, and perhaps assign the job to the VGA.

The VGA has a register called Line-Compare (also called the Split-Screen Register) which we can use to split the screen. Although less functional than comparable PC registers, it’s still adequate for most applications. The Line-Compare Register specifies the line number that splits the screen into an upper, movable half and a lower, static half. The line number must be doubled in 200-line modes.

How this register works is quite simple. As the image is being drawn, the VGA keeps track of the current line number. This line number is always the physical line number - so even in 200-line modes such as Mode 13h and Mode X, this value ranges from 0 to 400 due to the line-doubling.

Unfortunately, this line number is located in a register is not accessible through port addresses, but which is used internally for comparison with the Line-Compare Register (CRTC, Index 18h). When the line number reaches the value contained in this register, the address counter is set to 0, that is, the display begins at this line using data from Offset 0 of the video memory.

Split-screen construction
Since the line number does not fit into 8 bits (none of the VGA modes have fewer than 350 lines, even 200-line modes are generated by 400 physical lines), the Line-Compare Register is divided among three different registers (four on Super-VGA cards). There is no more room for additional registers so all “overflow” bits of these registers are combined into two new registers (three in Super-VGA).

The first of these registers is the Overflow-Register. As its name implies, it takes up the overflow from the vertical timing registers. In addition to other items, this register contains Bit 8 of the Line-Compare Register in Bit 4. Incidentally, Bit 4 is the only bit in this register not protected by the protection bit of Register 11h.

Line-Compare Bit 9, although not yet needed at these resolutions, is located in the Maximum Row Address Register (the second overflow register) at Bit 6. So, if you want to set Line Compare, you need to use bit-shifting and masking to specify the line number among these three registers. This task is accomplished by the procedure **Split** located in MODEXLIB.ASM:

```assembly
Split proc pascal far row:byte ;screen splitting in "row" row
mov bl,row
xor bh,bh
shl bx,1                      ;*2 because of row duplication
mov cx,bx
mov dx,3d4h                   ;CRTC
mov al,07h                    ;register 7 (overflow low)
out dx,al
inc dx
in al,dx
and al,11101111b              ;load bit 4 with bit 8 of the row
shr cx,4
or al,cl
out dx,al
inc dx
mov al,09h                    ;register 9 (maximum row address)
out dx,al
inc dx
in al,dx
and al,10111111b              ;load bit 6 with bit 9 of the row
shr bl,3
and bl,64
or al,bl
out dx,al
dec dx
mov al,18h                    ;register 18h (line compare/split screen)
mov ah,row
shl ah,1
out dx,ax
ret
Endp
```

First, the procedure doubles the given value. Therefore, despite Double-Scan (line-doubling) in 200-line mode, it uses the y-coordinates which ranges from 0 to 199. After selecting the overflow register (CRTC, Register 7), Bit 4 is cleared so Bit 8 of the line number can be specified by masking and right-shifting by 4 bits. The same method is used to copy Bit 9 of the line number to Bit 6 of the Maximum Row Address Register (CRTC, Register 9). Finally, the remaining 8 bits of the line number are written to Register 18h (Line Compare or Split Screen) and the procedure is ended.
When using the Linear Starting Address Registers 0ch and 0dh to move the starting screen location to a higher address, such as the beginning of Page 2, the top half of the screen will display this area, while the bottom half (below the line specified in the Line-Compare Register) will display the contents at start of the video RAM.

You can now adjust the upper area using the Linear Starting Address and scroll in all directions (see the “Scrolling In Four Directions” section in this chapter) while the bottom area displays the same contents.

While this is somewhat limited compared to the split-screen options available on "game computers", many interesting effects are still possible.

In addition to the obvious effect of dividing the screen into two areas, you can also create other effects as well. For example, you can change the screen so the dividing line alternates between movable and fixed portions. Since the dividing line can be placed anywhere, you can use a loop to reposition it continuously. The entire bottom of the screen also moves when you change the Line-Compare Register. This is because the line always begins with Memory-Offset 0. Subtracting 2 (Double-Scan) moves the entire lower partial screen up by one graphic line, so a new line becomes visible at the bottom.

In this way, you can push an image up from the bottom over a background image, say, for example, as a transition to a new scene in a demo. All you need to do is load the background image onto Page 1 (with LoadGIF and P13_2_ModeX) and the “top” image onto Page 0. Then turn on split-screen, first at Line 400 so the top image stays invisible and only the background is visible.

Now begin moving the split-line up (decrement the Line-Compare Register by 2), the contents of Page 0 will slide up from the bottom over the background image until it hits the top of the screen (Line Compare 0). You can then switch to Page 0 and disable the split-screen. It’s important that you program these in this order. Otherwise, you might see a brief flash of the old background because disabling the split-screen brings the current page to the screen, which at this point should already be Page 0.

Of course going the opposite way (i.e., pulling an image down from the top, like a card game, so the image underneath comes into view) is also possible. This is a good effect for a slide show presentation (preprogrammed images displayed at fixed time intervals).

If you want to do this, you must store the foreground on Page 0 and the background on Page 1. First, activate the split-screen, switch to Page 1 and increment the value of the Line-Compare Register in a loop so the split-line moves downward. The contents of Page 1 (currently the active page) will then appear above this line as the background.

Like most other effects, incrementing or decrementing the Line-Compare Register should be synchronized with the vertical retrace to prevent jittering and flickering. Simply add the command WaitRetrace into the loop. This also serves as a necessary delay; the CPU would otherwise finish the entire process in a fraction of a second. Only 70 images per second are displayed with the synchronization (repeat-frequency of Mode 13h and Mode X), so the process now takes 2.8 seconds (200 lines/70 lines per second).

In experimenting with these effects you’ll quickly notice a false color display in one of the two images. This display is due to the 256-color limit in palette-based graphic modes. As you load one image, you set the palette of this image. Then the palette of the second image you load overwrites the first because only 256 colors can be displayed at one time. The only way out of this situation is to use one palette for both images.
When drawing your own images, make certain to use the same palette for both (palettes can normally be loaded and saved separately). With finished images (scanned or from graphic collections), the two different palettes must be combined into one. Several graphic conversion programs have a function for this purpose, but image quality often deteriorates in the process which results in incorrect colors and incorrectly set pixels.

A better alternative is to reduce both images separately to 128 colors. Of course, if you have two images with one more complex than the other, you could reduce one to 192 and the other to 64 colors. What is important is to make the sum equal to 256. By using a graphic converter, you can then build a complete palette without dithering. You can now load the two images into the demo, regardless of order, since the palettes are now identical.

It’s also possible (and useful) in this effect to compute the Line-Compare Register in units of one. This makes single scan-lines, i.e., half graphic-lines, the smallest unit and doubles the shift time. Because motion is involved, the user’s eye won’t notice the half-lines anyway.

In a simple split-screen, however, it might be distracting for a half graphic-line to appear in the middle of the screen. In this case, you’ll need to use odd numbers because the display of the second partial image begins on the line immediately following the one given in the Line-Compare Register. Since this must be an even number, the value in the register must be odd.

We’ll illustrate the procedure Split with a small demo program. It takes two pages (single-color for clarity), pushes them over each other and separates them again. The program first fills the two video pages and activates Page 1 (Start-Address 16000). The loops increment and decrement the Line-Compare Register, creating the desired motion, and synchronizing the entire process with the retrace.

You may wonder why we used the Exit command in the keyboard inquiry since it doesn’t fit the structured programming model of Pascal. The reason is to prevent unnecessary work, otherwise, you would need cumbersome flag variables as terminating conditions for the loops.

```pascal
Uses Crt,Gif,ModeXLib;
Var i:Word;
begin
  Init_ModeX;
  LoadGif('uwp');
  p13_2_ModeX(16000,16000);
  LoadGif('enter320');
  p13_2_modeX(0,16000);
  SetStart(16000);
  Repeat
    For i:=200 downto 0 do Begin
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
    End;
    For i:=0 to 200 do Begin
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
    End;
    Until KeyPressed;
  End.
```

You can find SPLIT.PAS on the companion CD-ROM
Furthermore, split-screen works in all graphic as well as text modes, but makes sense only when more than one video page can be displayed. In Mode 13h for example there is not enough memory for two independent screen sections. In this case the identical contents would appear, just distributed differently within the windows. Therefore, although split-screens are possible in one-page graphic modes, all statements made refer to Mode X.

### Scrolling In Four Directions

Until now we have been using the Linear Starting Address Register of the CRTC (Register No. 0ch/0dh) for switching video pages. This was done by calculating the offset of the page (Page Number x 16000) and writing it to the register. You could as easily write an intermediate value here as well, thereby freely shifting the start of the display area in RAM.

Now the electron beam builds a new image beginning at the point specified in the Linear Starting Address Register instead of obtaining the image data from the start of video RAM. This way you can shift the physical screen (the area that appears on the monitor) like a window over the entire VGA RAM, so any desired section of a 256K image can be displayed.

When you shift the beginning of the screen within RAM, the window moves in the same direction. This causes the graphic to move in the opposite direction on your monitor. Increasing the Linear Starting Address by 80 shifts the image contents up one line (320 pixels / 4 planes = 80 bytes long).

The contents of Line 0 are now outside the window and Line 200 moves up from the bottom. The visible lines are now 1-200. You can implement a vertical scrolling of the entire screen contents with minimum calculation by incrementing the register in units of 80 bytes each. The illustration on the right is an example of a video-RAM configuration with four vertical pages.

The following example easily accomplishes this task:

```pascal
Uses Crt,Gif,ModeXLib;
Var y,                      {Current value of Linear Starting Address}
   y_dir:word;             {Gives scroll direction}
Begin
  Init_ModeX;               {Enable Mode X}
  LoadGif('upfold');        {Load first image into Pages 0 and 2}
  p13_2_ModeX(0,16000);
  p13_2_ModeX(16000,16000);
  LoadGif('corner');        {Load second image into Pages 1 and 3}
  p13_2_ModeX(16000,16000);
  p13_2_ModeX(48000,16000);
  y:=80;                    {Begin with Line 1}
  y_dir:=80;                {Direction of motion +80 bytes per pass}
  Repeat
    Inc(y,y_dir);           {Motion}
    WaitRetrace;            {Wait for retrace}
    SetStart(y);            {and write new start to register}
    if (y >= 600*80)        {Border reached -> reverse direction}
      or (y <= 80) Then y_dir:=-y_dir;
End.
```
Until KeyPressed;
(Run until key is pressed)
End.

After loading the two partial images into the four video pages (you can, of course, use different images) the program sets the y-counter to 80, the starting address of Line 1. \texttt{Y\_dir} determines the scrolling direction. A positive value for the register value in Linear Starting Address increases (scroll up). A negative value will decreases (scroll down).

The starting address (y) is continually incremented (or decremented if \texttt{Y\_dir} is negative) within the main loop. The scrolling direction reverses when the borders are reached.

A scrolling text which appears on more than four video pages, for example in a header or window is more complicated. As the scrolling progresses, you must load the next video page from the hard drive and copy it to the section just read which has disappeared from the screen.

In addition to vertical scrolling, you may also find the need for horizontal scrolling. Imagine a logo four video pages in size, visible only in sections, moving back and forth “underneath” the screen.

The solution appears very simple. We previously modified the Linear Starting Address Register by 80 each time, why not go by units of one? In this case, however, whatever was scrolled out at the left would be added again at the right. This is because the lines are still directly one behind the other in memory.

For example, if you shift the screen-start by one byte (shift left 40 pixels), the CRTC displays Bytes 1 to 80 (instead of the usual 0-79) on the first screen line. Byte 80, however, actually belongs to the second line of the image so each byte that you scroll out at the left border will appear one line higher at the right border.

The solution to this problem is to enlarge the screen to a virtual width of 640 pixels. The display has only 320 pixels, but additional pixels will appear “next” to the monitor on the right. The four video pages will no longer lie one on top of the other, as in vertical scrolling, but will form a square. Pixels scrolled past the left edge will go to the extreme right (x-coordinates 636-639) of the invisible area and new pixels will scroll in from the invisible to the visible area.

In other words, we can also now move the window (which is actually what the Mode X screen represents) horizontally. Thanks to the horizontally adjacent video pages, we now have enough room to do this.

How do we generate this special mode? VGA has a register for this purpose, too. In fact, we can use Register 13h of the CRTC, Row Offset for other purposes as well. It indicates how far the internal data pointer (Linear Counter) moves up when the electron beam reaches the right border. This distance corresponds to the interval between lines within video memory and thereby gives us their length.

Row-Offset normally contains the value 40, which corresponds to a width of 80 bytes, in both Mode 13h and Mode X. This register counts by words, i.e., an 80-byte width equals 40 words. Although the lines in Mode 13h are 320 bytes long, the computational basis (largest unit addressable by the CPU) is also quadrupled from one byte to a doubleword, whereby the programmed value again becomes 320 / 8 = 40.

Of course, you can also write larger values to this register. For example, the value 80 means the lines have an interval, or length, of 160 bytes (equals 640 pixels). The 80-byte gaps which are created between individual 80-byte lines are padded with the invisible half-lines to the right.

The following illustration shows the video RAM configuration after activating 160-byte mode.
We use the procedure `Double` in the ModeXLib module to enable this line-doubling:

```pascal
double proc pascal far
    mov dx,3d4h                   ;CRTC register 13h (row offset)
    mov ax,5013h                  ;set to 80 (double width)
    out dx,ax                     ;and write
    ret

double endp
```

The only result here is that CRTC-Register 13h is loaded with the value 50h = 80d, which doubles the line length to 160 as described above.

### Using the GIF loader for large images

You may also want to load GIF images in this extended mode. Basically, you can use the normal loader because it functions independently of the screen format. In other words, the loader unpacks the compressed data in the order read: In main memory first and then to the video memory. To load an image in 640-pixel mode, it simply must exist as a 640 pixel wide image. Otherwise, every other line will be stored in the right half of the screen (not our goal).

You can also load extremely large images the same way. These images, such as the header, can be loaded in a single step by generating an image of size 320 x 800 and then loading it, for example. If you check the image size, you will notice that these images are larger than 64K and therefore cannot be stored in the variable `VScreen`. Turbo Pascal does not allow fields larger than 64K segment size.

Here are a few suggestions to help you overcome this problem:

1. Partitioning the image from the start into 64K portions. This results in an image size of 640 x 100 at double-width. This solution is neither practical nor efficient.

2. Divide the image among several pointers as you are loading it, so when one segment overflows you switch to the next variable. The programming involved here is quite complex, however.
3. Bypass the Turbo Pascal memory management completely and allocate the memory through DOS.
   In this case, when you have a segment overflow, all you need to do is increment the corresponding
   segment register by 1000h.

Solutions #2 and #3 have the same problem: Where will you get the necessary memory? A two-page image
already requires 128K, while 64K is just barely manageable.

The best solution is found in the procedure ReadGIF in ModeXLib. Immediately following the two stosb
instructions, which write the newly decoded pixel, the procedure tests for an overflow. When an overflow
has occurred, the image built-up to that point is simply copied to video RAM. Since large video pages make
sense only in Mode X anyway, the procedure calls up p13_2_ModeX, which copies VScreen to video RAM
at the position defined by VRAM_Pos and resets the destination pointer to 0. The variable VRAM_pos is then
moved to the end of the copied area in video RAM, so a possible second overflow will be added here.

At the end, the procedure ReadGIF assigns the current fill-status of VScreen (located in di) to the variable
Rest. VScreen returns the number of copied bytes because Pascal always places allocated variables at
even segment addresses. This way, by using Rest as the number of bytes to be copied, the main program
can again call p13_2_ModeX to copy the rest into video RAM.

This process may not be the optimum algorithm but it does have two important advantages:

1. The procedure ReadGIF is 100% compatible with the old one. So, an image size doesn't have to be
determined by passing a parameter or reading the GIF header. Theoretically, any image format is
   possible (even Super-VGA resolutions are no problem).

2. The procedure saves a great deal of memory because only one segment in memory is occupied. This
   segment can be removed from the heap after loading and created again when the next image is
   loaded.

Now that you have loaded a large image into video RAM, we'll start the actual scrolling. You'll need the
Linear Starting Address Register again. Since it can now be modified by any desired value, vertical,
horizontal and even diagonal scrolling are possible. The procedure is almost identical for vertical scrolling
as that described in the previous section. The Linear Starting Address Register is incremented or
decremented by one line length each time. You only need to consider the changed line length since vertical
scrolling requires 160 byte steps.

Horizontal scrolling is done by increasing or decreasing the register by units of one, whereby one byte
always represents four pixels. Scrolling is, therefore, possible only in four-pixel increments (for another
option, see Pixel-Panning Register 13h of the Attribute Controller.

Diagonal scrolling simply combines these two directions. For example, to scroll four pixels up and to the left, increment the Linear Starting Address by 641 (160 bytes per line x 4 lines + 1
byte). This program demonstrates the capabilities of this register:

```pascal
Uses Crt,Gif,ModeXLib;
Var x, y, x_dir, y_dir:word;  
Begin
  Init_ModeX;  
  SCROLL4.PAS
```
In addition to the familiar position and direction variables for the y-direction, similar variables are also needed for the x-direction. These are called \( x \) and \( x_{\text{dir}} \) and have the same purpose as their y-counterparts: \( x \) indicates the current position in the x-direction.

Note an increase or decrease of 1 represents a shift of four pixels because of the four parallel planes \( x_{\text{dir}} \) describes the x-scrolling direction. A 1 shifts the image contents four pixels to the left for each pass through the loop and a -1 shifts four pixels to the right.

After initialization, the program switches on 160-byte VGA mode by calling the double routine. This creates a virtual doubling of the screen to enable horizontal scrolling. The image is then loaded with \texttt{LoadGIF}. Note that a portion of the oversized image (640 x 400 pixels = 256,000 bytes) is copied to video RAM during the actual load process (you could also say it is paged out due to insufficient memory).

Since this paging occurs only with an overflow past the 64K limit, the procedure call to \texttt{p13_2_ModeX} copies the rest to video RAM, starting at the position after the last byte copied (vram_pos), and using the length variable \texttt{Rest}, whose value is entered by \texttt{LoadGIF}. The number of bytes must now be divided by four since \texttt{p13_2_ModeX} works with main memory data - because of the four planes, a length of 1 means that four bytes are copied and the contents of the variable \texttt{Rest} are given as real bytes (corresponding to the number of pixels).

As we mentioned, the x-direction is initialized to 1 so scrolling initially proceeds to the left. The position is likewise set to 1 so it starts at Column 1 (prior to entering the loop the screen still starts at Column 0, Line 0). The doubling of both values to 160 occurs in the y-direction variables which accounts for the doubled line length of 160 bytes.

Very little changes in the loop itself. To create movement in the x-direction another \texttt{Inc} command is used. In addition, the starting screen location is set to the sum of the x-offset and y-offset, so both directions are established (see above). Furthermore, a reversal in x-direction must occur upon reaching the right or left border, so combined with a similar reversal in the y-direction, the scroll area always stays within the rectangular virtual screen defined in video RAM.
Combining It All: Split-screen WithScrolling

Split screen and multidirectional scrolling can already generate some interesting effects. However, you can have a tremendous visual impact by combining them as you'll see in the following example (SCRL_SPT.PAS):

```pascal
uses crt,Gif,ModeXLib;
Var x,                  {current offset in x-direction}
  x_dir,               {specifies scroll direction for x}
y,                    {current offset for y-direction}
y_dir:word;           {specifies scroll direction for y}
split_line:word;      {current position of Split-Line}
split_dir:word;       {specifies direction of movement for Split-Line}
Begin
  Init_ModeX;           {enable Mode X}
double;               {enable 160 byte mode}
  Screen_Off;          {screen off}
  LoadGif_Pos('640400',160*50);{load big picture at position (0/50)}
  p13_2_ModeX(vram_pos,rest div 4); {copy rest to VGA-RAM}
  LoadGif('corner');   {load small picture at position (0/0)}
  p13_2_ModeX(0,160*50); {and copy to screen}
  Screen_On;           {screen on}
split_line:=150;      {set Split at line 150 first}
split_dir:=1;         {move Split-Line down first}
x:=1;                {x-start with column 1}
x_dir:=1;             {x-direction 1 byte per pass}
y:=160;               {y-start with row 1}
y_dir:=160;           {y-direction+160 bytes per pass}
Repeat
  Inc(x,x_dir);        {x-movement}
  Inc(y,y_dir);        {y-movement}
  Inc(Split_line,Split_dir); {move Split Line}
  WaitRetrace;         {wait for Retrace}
  SetStart(50*160+y+x);{and write new start in register,}
                        {skipping the first 50 lines}
  Split(Split_line);   {split screen at Split Line}
  if (x >= 80) or (x <= 1) Then x_dir:=-x_dir; {x-border reached -> turn x-direction around}
  if (y >= 200*160) or (y <= 160) Then y_dir:=-y_dir; {y-border reached -> turn y-direction around}
  if (split_line >= 200) or (split_line <= 150) then split_dir:=-split_dir
  until KeyPressed;    {run until key pressed}
  TextMode(3);
End

We're no longer splitting the screen into two static regions at a fixed line. Instead the line itself is also moving. The variables `Split_line` and `Split_dir` serve to describe this motion, based on the same principle as in the corresponding x- and y-variables. `Split_line` contains the line at which the screen is divided while `Split_dir` contains either a 1 for movement in the positive y-direction (down) or a -1 for moving the split-line upward.
Chapter 5

Split-screen And Other Hot Effects

After switching on Mode X and activating 160-byte lines, the screen is disabled for loading the image. Although the speed advantage is barely measurable, the overall visual effect of the demo improves if the monitor remains disabled as the image is being constructed. You can remove these commands when you need to view the load process, say, for debugging purposes. However, in the final version of a demo, the image construction should remain invisible while it's loading.

The large image, which will later scroll in all four directions, is now loaded at Position 160*50 (50 lines past RAM-start) by the procedure LoadGIF_Pos, while a smaller image is placed at RAM-start by the normal LoadGIF procedure. LoadGIF_Pos works like LoadGIF, but receives the offset to which the (large-image) overflow is copied as an additional parameter.

This 50-line shift accounts for the split-screen's mode of operation. Since at the split-line the linear counter is set to 0, the contents at RAM-start, i.e., the small image are displayed below this line. Later, when scrolling the program simply uses the start-address of the large image as the point of origin.

Next, the image is again enabled. The split-line is set to an initial value of 150 which is internally converted to scan lines (corresponds to graphic lines). Now, thanks to the split, the lower quarter-screen comes "alive". Since Split_Dir contains a value of 1, the lower part of the screen in the loop moves toward the bottom.

When the split-line reaches a value of 200, which corresponds to a complete "removed" of the lower section, the direction of motion reverses itself likewise at the top border (150), so as the upper screen is scrolling, the bottom also moves vertically.

In this version the SetStart command, which provides the scrolling, has been expanded by adding the constant 50*160. This assures the first 50 lines of video RAM never appear on the screen while scrolling. These lines contain the lower area of the split-screen, while the large image begins 50 lines later.

Door Closed: Squeezing An Image

Another application for the split-screen/scrolling combination is squeezing two halves of an image together. This effect is often used at the opening of many demos.

Here the upper half, which moves from the top to the middle, is controlled through the Linear Starting Address; in other words, it's scrolled vertically (in this case downward). Note that Video Page 1 is either empty or filled with a certain color, since half of it is visible at the beginning.

The bottom half is controlled by the splitting process. Decrementing the line number pushes the start of the split-screen up toward the middle.

Procedure Squeeze from MODEXLIB.ASM accomplishes both of these tasks:

```
squeeze proc pascal far
    ;squeezes screen together
    mov si,200*80
    mov di,199
sqlp:
    call waitretrace
    call split pascal, di
    call setstart pascal, si
```

The procedure Squeeze is part of the MODEXLIB.ASM file on the companion CD-ROM.
Registers SI and DI are used in this procedure. You may wonder why we’re not using other registers. Here, speed is not the most important factor. After all, the CPU has very little to do because all the work is handled by the VGA. Therefore, it’s sometimes more efficient to use register variables which can save storage space and processing time.

SI contains the value of the Linear Starting Address Register and so is responsible for moving the upper partial screen. DI determines the split-line, which controls the position of the lower half. These registers are assigned initial values which generate a completely separated image.

After the usual wait for a vertical retrace, the loop updates the current status with the help of the corresponding register. The movement takes place in the statements decrementing the register by 80 (Linear Starting Address, 1 line) or by 1 (Split-Line), whereby the upper part scrolls down and the split-line moves up. The image is therefore squeezed together. The terminating condition is when the split-line moves past the half-screen.

We’ll use a small example called SQUEEZE.EXE. It simply loads an image and, as its name suggests, squeezes it together. The wait for the Enter key points out something you should remember when creating an image: Notice the opening screen reverses the upper and lower screens. When press(Enter) the upper and lower screens are squeezed in the correct position.

The beginning contents of video RAM are displayed at the bottom half of the monitor. Scrolling moves data from the second half of the first video page to the top half of the monitor. To do this, you simply have to do reverse the image by cutting and copying both halves. The program should now be self-explanatory:

```
uses Crt,ModeXLib,GI;
Begin
Init_ModeX;               {switching on Mode X}
LoadGif('squeeze');      {loading of image}
P13_2_ModeX(vram_pos,rest div 4);{waiting for Enter}
Squeeze;                 {squeezing of image}
Readln;
Readln;
TextMode(3);
End.
```

Smooth Scrolling In Text Mode

Scrolling in text mode actually works the same as in graphic mode. The visible portion of video RAM is shifted using the starting address (Linear Starting Address). The one noticeable difference is scrolling is very jittery because shifting always occurs by entire characters in text mode whereas you can shift by individual pixels in graphic mode. This is due to the structure of video RAM in text mode. Data for
individual pixels no longer exist here, so the Linear Starting Address also refers to whole characters and enables only very coarse scrolling.

To prevent this, we can use a few new VGA registers: The horizontal and vertical panning registers. In panning, we’re shifting the image contents by one pixel, which these registers allow in text mode as well.

For smooth scrolling in text mode, simply shift the image contents by panning in the desired direction. Once you have moved ahead by one character, set the corresponding panning register back to its initial value and modify the Linear Starting Address Register. This is necessary because the maximum panning distance is one character width or one character height. Panning therefore is only a fine control, while coarse scrolling is achieved through the Linear Starting Address.

Vertical panning is generated by CRTC-Register 8 (Initial Row Address), where Bits 4-0 indicate the graphic line where display of the first scan line begins. Incrementing this value by 1 makes Line 1 begin inside the character set the screen contents therefore move one line up.

Horizontal panning on the other hand, is the responsibility of the Attribute Controller. Here Register 13h (Horizontal Pixel Panning) provides for smooth motion in the x-direction. The table on the right shows the somewhat unusual value assignments for this register:

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Panning by one pixel</td>
</tr>
<tr>
<td>1 to 7</td>
<td>Panning by 2 to 8 pixels</td>
</tr>
<tr>
<td>8</td>
<td>No panning</td>
</tr>
</tbody>
</table>

The necessary format conversion is done by a simple calculation (without requiring difficult IF-structures):

\[ \text{register-value} := (\text{panning-value} - 1) \mod 9 \]

The complete demonstration program SCROLLT.PAS:

```pascal
Uses ModeXLib,Crt;

Var x, x_dir, y, y_dir:Word;  {x-position in pixels, x-direction, y-position in pixels, y-direction}

Procedure Wait_In_Display;assembler;
{counterpart to Wait_In_Retrace, waits for display via cathode ray}
asm
  mov dx,3dah                  {input status 1}
@wait2:
  in al,dx
  test al,8h
  jnz @wait2                   {display on? -> then finished}
End;

Procedure Wait_In_Retrace;assembler;
{waits for retrace, also resets the ATC flip-flop by read access to input status 1}
asm
  mov dx,3dah                  {input status 1}
@wait1:
  in al,dx
  test al,8h
  jz @wait1                    {retrace active? -> then finished}
End;
```

You can find SCROLLT.PAS on the companion CD-ROM.
Procedure FillScreen;
[Fills video RAM with test image 160*50 characters in size]
var i:word;
Begin
For i:=0 to 160*50 do Begin  {character loop}
  If i mod 10 <> 0 Then      {write column counter ?}
    mem[$b800:i shl 1]:=(no, then '-')
  Else
    mem[$b800:i shl 1]:=(yes, then column number in tens)
    ((i mod 160) div 10) mod 10 + Ord('0'));
  If i mod 160 = 0 Then      {column 0 ? -> write row counter}
    mem[$b800:i shl 1]:(i div 160) mod 10 + Ord('0'));
End;
End;
Procedure V_Pan(n:Byte);assembler;
{performs vertical panning}
asm
mov dx,3D4h                   {CRTC Register 8 (Initial Row Adress)}
mov al,8
mov ah,n
out dx,ax
End;
Procedure H_Pan(n:Byte);assembler;
{performs horizontal panning}
asm
mov dx,3C0h                   {ATC Index/Data Port}
mov al,13h or 32d             {Register 13h (Horizontal Pixel Panning)}
out dx,al                     {select; Bit 5 (Palette RAM Address Source)}
mov al,n
or al,32d                     {write panning value}
out dx,al
End;
Begin
TextMode(3);                  {set BIOS mode 3 (80*25 characters, Color)}
FillScreen;                   {build test picture}
portw[$3D4]:=$5013;           {double virtual screen width(160 character)}
x:=0;
x_dir:=1;
y:=0;
y_dir:=1;
Repeat
  Inc(x,x_dir);               {movement in x and y-directions}
  Inc(y,y_dir);
  If (x<=0) or (x>=80*9)      {turn around at borders}
    Then x_dir:=-x_dir;
  if (y<=0) or (y>=25*16)    {initialize coordinates and directions}
    Then y_dir:=-y_dir;
    Wait_in_Display;            {wait until display running}
    SetStart((y div 16 *160) + x div 9);    {set start address (rough scrolling)
    Wait_in_Retrace;            {wait until retrace active}
    V_Pan(y mod 16);            {vertical panning (fine scrolling)}
    H_Pan((x-1) mod 9);        {horizontal panning (fine scrolling)}
    Until KeyPressed;             {wait for key}
    TextMode(3);                  {and set normal video mode}
End.
After setting text mode and drawing a test image, the program switches over to double-virtual-width by setting Row-Offset Register 13h of the CRTC to the value 320 bytes/4 bytes = 80 (4 due to doubleword-access). The motion itself corresponds exactly - with somewhat different values - to the same procedure in graphic mode.

One special feature is dividing WaitRetrace into two parts: Wait_in_Display and Wait_in_Retrace. This is because the registers used here are linked to different timing circuits. Linear Starting Address is loaded by VGA directly after entering into the retrace, so a switch during the retrace will usually show no effect in the next image. This was previously unimportant because the entire operation was simply delayed by one retrace. Now we also have the panning registers, whose effects become immediate after being changed.

The starting address is, therefore, set during image construction, because it has no function at this point anyway, and is guaranteed to be set correctly for constructing the next image. The panning registers however are set during the retrace because their effects are immediate and must therefore take place in the invisible area of the screen.

The procedure Wait_in_Retrace has a second purpose. Accesses to the ATC are somewhat unconventional: This port alternates between index and data register with each write-access to Port-Address 3c0h. When the program starts the current status of this register is unknown. However, reading Input-Status Register 1, as happens in this procedure, switches Port 3c0h to the index function and thereby gives it a defined status.

Since writing to the ATC (Attribute Controller) occurs almost immediately after the wait for the retrace, you can assume the index mode is active. The only possibility of this port switching over unnoticed is if a TSR "stumbles in" through an interrupt. This is highly unlikely however, and can be prevented anyway by a simple CLI instruction.

It's very important when writing to the ATC-index to always set Bit 5 at the same time. This bit controls access to the internal palette of the ATC. When Bit 5 is cleared the CPU obtains full access and the ATC disables itself; leading in the best case (if you reset the bit quickly enough) to flickering, but in the normal case to a complete system crash, where only a reboot will help.

Of course, this scrolling can also be combined with a split-screen. The procedure is exactly the same as in graphic mode. The split-screen is fully mode-independent, because you are directly programming the physical scan-line where the split occurs.

A Different Kind Of Monitor: Flowing Images

We've mentioned the double-scan mode of VGA several times. Depending on the BIOS, this double-line mode (which displays 200 graphic lines at a physical resolution of 400 lines) is implemented using either Bit 7 (Double-Scan-Enable) of CRTC-Register 9 (Maximum Row Address) or by entering 1 instead of 0 in Bits 4-0. These bits in text mode contain the number of screen lines per character line minus 1 (equal to 15 in VGA text modes), i.e., the number of screen lines in which the same information is repeatedly retrieved from video RAM.

Of course, these screen lines in text mode are still not identical because a different line of the character set is used each time. In graphic mode, however, there is no character set so the same data is actually displayed many times. A value of 1 in these bits also creates a line-doubling, whereby a copy is generated of each line.
Even more copies are generated if you enter higher values. The pixels are expanded in the y-direction, and the vertical resolution cut in half.

This effect is used in FLOW.PAS. First, it places a demo image on the screen and stretches it out through continuous incrementing of Bits 4-0. Of course, the other bits in this register cannot be touched the old contents are therefore stored in old9 and an OR operation performed on the current contents when writing to the register. The entire process is synchronized with the vertical retrace as usual.

Theoretically, this procedure is also possible in text mode but would lead to nonviewable images. This is because no character set data exists for these new lines although the VGA still doubles, triples etc., the memory contents like in graphic mode. Therefore, since only garbage appears as images in text mode, this effect is limited to graphic mode.

When selecting a suitable image, at least 13 lines must be black or some other solid color at the top. Otherwise, the image won’t flow down, it will just be stretched out. At a vertical resolution of 400 lines and a maximum value of 31 in the Maximum Row Address Register, each line will be displayed 32 times, producing $400 / 32 = 12.5$ visible lines. These must all be the same color for the entire image to be one color later.

Let’s Add More Color Please: Copper Bars Without Copying

Until now we have essentially created our effects by simply reprogramming certain VGA registers. The CPU executes only control functions, such as updating the screen start-address during scrolling.

We’ll talk about a new role for the CPU in this section. In addition to control, we’ll add some range checking tasks. We won’t, however, use the CPU to copy memory areas. This means the CPU will continuously monitor the VGA, and at certain screen lines perform modifications to the VGA registers. We can now
produce an effect called copper bars which has been a popular effect since the days of the old Commodore 64.

Copper bars are vertical bars of a specific color. They continuously move up and down in the y-direction. They not only move up-and-down but also in front of one another while moving up-and-down. Meanwhile other graphic effects, like text-scrollers, can be running at the same time and appear in front of the copper bars.

You can probably figure out how copper bars work simply by their appearance. Video RAM contains absolutely no data at those areas being displayed. It’s filled with zeros, for example, so no bars can go out from this point. For each new screen line, however, you define what color this zero actually represents: It might represent a light red color one time, while at another time it might represent a bright yellow.

Incidentally, you can also display more than 256 colors on the screen in this manner. Theoretically, each screen line can contain 256 colors and these can be different for each line. In practice, however, the range of colors is very limited because it’s not possible within a retrace to set an entire palette. The number of colors will vary depending on the processing speed of the computer. In the example in this section only Color 0 is continuously reprogrammed, which already gives us 127 new colors. Reprogramming just a few more colors expands this range even further.

By continuously changing the color as the screen lines are displayed, you can produce vertical structures such as bars. Colors other than 0 are not affected by this, so text can still be scrolling in the foreground using Colors 1-16 for example. The places in the text containing a zero are transparent so the copper bars become visible in the background.

This way, you don’t have to worry as the text scrolls through the background. Otherwise, you would have to keep saving the background contents and rewriting the text. You can simply copy blocks of data to video RAM, which in the transparent sections contain 0. The main advantage here is that you can use the fast Write Mode 1 for copying.

This process is comparable to the one used by a Genlock card. This specialized video card overlays a TV image wherever the video signal displays a certain color (usually blue). Here also, mixing occurs at a very low level and not in the slow video RAM.

How then do we generate the multicolored lines? Home computers usually have a screen-line interrupt. You can program the video controller to execute an interrupt upon reaching a particular screen line; the color can then be quickly reset in response to this event.

Although some VGA cards also have a vertical-retrace interrupt (often deactivated through dip-switches), the situation is far worse with the horizontal retrace. We’re not aware of any graphic card which supports this interrupt. You only choice is to continuously monitor the VGA status counting the lines as you go and then change the color at the desired line.

To do this, you first need to create an initial status by waiting for a vertical retrace. The next time the Display-Enable circuit is activated (read from Input-Status Register, Bit 0 - Display Enable Complement), you are guaranteed to be on Line 0 of the screen. You can count with the current line through a permanent inquiry on this bit.

Since displaying a single line takes much less time than displaying the entire image, you cannot allow any interrupts during the wait (they often last much too long).
Disabling the interrupts however causes a serious problem. Time becomes critical if you also plan to use sound during the wait for the horizontal retrace...you must prevent the sound card from requesting new data.

Even this limitation is not as significant compared to the computing time needed to copy data into video RAM. For example, if only the top half of the screen is reserved for screen lines (i.e., copper bars) instead of the entire screen, then there will be enough time remaining for scrolling and sound generation in the bottom half, too.

When it comes to timing, the order of commands is also important. In the short time it takes for the actual retrace (approximately 6 microseconds) it’s practically impossible to do the calculations for all the bars. These must therefore be done earlier, while during the retrace itself the only other thing that happens is to set the color.

The following example program is called COPPER.EXE. It consists of a Pascal section (COPPER.PAS) and an Assembler section (COPPER.ASM) to show how to build these pieces into a program.

The program first enables Mode 3 (although any text mode or graphic mode will work). Screen line counting is independent of the VGA mode. You only need to consider the various vertical resolutions of different modes where different values for y-position expand or contract the bars in the y-direction. Here, also we’re using physical resolution, which in 200-line modes (such as Mode 13h) consist of a full 400 lines (Double Scan).

Next, the border color is set to 255 using ATC-Register 11h because the copper uses Color 0 to display the bars. Simply leave the border color at 0 if you also want to display the bars in the screen border. However,
a value other than zero not only allows for clear separation of image from copper, it also extends the
duration of the horizontal retrace. This is especially true on slower computers, making the flickering at the
left border disappear.

The loop, following the familiar pattern, moves and displays the copper in the y-direction. The call to
WaitRetrace, however, is hidden in the procedure MakeCopper. This is done to guarantee immediate
proximity to the retrace and the line counting. Text is also output for demonstration purposes and shows
how easy it is to change the screen independently of the background. Also demonstrated is the fact that
when the scanning ray is outside the range of the coppers, enough computing time remains for other tasks.

The procedure MakeCopper is in the COPPER.ASM module. When the call is made, the program passes
the y-positions of both coppers (red and green) with a priority mask which controls their overlapping. The
low-byte is responsible for Copper 1 and the high-byte for Copper 2. A value of 0 means the corresponding
copper is running in the background. A value of 0ffh pages it to the foreground. An overlay occurs when
both have a value of 0 and the two colors are mixed. A mask value of 0ffffh, however, erases both coppers
at the point of overlap.

When the maximum position is reached in our example, the
mask switches, which creates a circular motion. The red copper
fades into the background as it moves up and appears in the
foreground as it moves down. The COPPER.ASM module
mostly consists of the procedure MakeCopper.

You can find COPPER.ASM on the companion CD-ROM

---

### Split-screen And Other Hot Effects

The loop, following the familiar pattern, moves and displays the copper in the y-direction. The call to **WaitRetrace**, however, is hidden in the procedure **MakeCopper**. This is done to guarantee immediate proximity to the retrace and the line counting. Text is also output for demonstration purposes and shows how easy it is to change the screen independently of the background. Also demonstrated is the fact that when the scanning ray is outside the range of the coppers, enough computing time remains for other tasks.

The procedure **MakeCopper** is in the COPPER.ASM module. When the call is made, the program passes the y-positions of both coppers (red and green) with a priority mask which controls their overlapping. The low-byte is responsible for Copper 1 and the high-byte for Copper 2. A value of 0 means the corresponding copper is running in the background. A value of 0ffh pages it to the foreground. An overlay occurs when both have a value of 0 and the two colors are mixed. A mask value of 0ffffh, however, erases both coppers at the point of overlap.

When the maximum position is reached in our example, the mask switches, which creates a circular motion. The red copper fades into the background as it moves up and appears in the foreground as it moves down. The COPPER.ASM module mostly consists of the procedure **MakeCopper**.
jns copper1_ok                ;positive, then color
xor bl,bl

;calculate color 2
mov ax,cx
sub ax,y_pos2 ;calculate position relatively
cmp ax,height/2 -1 ;2nd half
jle copper2_up
sub ax,height -1 ;then ax:=127-ax
neg ax

;positive, then color
or ax,ax
jns copper2_ok
xor al,al

;calculate overlay
mov bh,al ;bl now has color copper 1 / bh copper 2
mov ax,bx
and ax,overlay_mask ;mask out copper 1 or 2
or al,al ;copper 1 priority
je copper1_back
xor bh,bh ;then clear copper 2

;copper 2 priority
xor al,al ;copper 2 priority
je copper2_back
xor bl,bl ;then clear copper 1

;select color 0 in DAC
xor al,al ;select color 0 in DAC
mov dx,3c8h
out dx,al
or bl,bl ;if copper 1 black -> leave as is
je bl_0
add bl,(128-height) / 2 ;otherwise lighten to achieve maximum
bl_0:                           ;the same for copper 2
or bh,bh
je bh_0
add bh,(128-height) / 2

;now wait for horizontal retrace and enable copper
cli ;clear interrupts, because it is VERY time-critical
mov dx,3dah ;select Input Status Register 1

;wait for display
in_retrace:
in al,dx
mov cx,maxrow ;last row generated ?
jne next_line

;wait for (horizontal) retrace
test al,1
je in_display
mov al,bl
mov dx,3c9h ;and set
out dx,al
mov al,bl
out dx,al
xor al,al
out dx,al
cmp cx,maxrow
jne next_line

;yes -> end
mov dx,3dah
wait_hret:
before switching off, wait for retrace
in al,dx
out dx,al
je wait_hret
First, the program determines the larger of the two y-coordinates. This establishes the first screen line beyond which a copper is never displayed. Upon reaching this position you can exit the procedure and use the computer time for other tasks. The procedure then sets the line counter to 0 and waits for the vertical retrace. This also begins at line 0 of the monitor.

The loop `next_line` uses the y-positions to calculate the scan-line relative to the topmost copper-line, which indicates the color. When the electron beam reaches the second half, you must again decrement the color to guarantee symmetry. A negative color-value is generated outside the copper regardless of whether the ray is above or below the center of the copper. This value is later intercepted and reset to 0 (black).

As a result of this calculation, BL contains the color-value of Copper 1 and BH contains the color value of Copper 2. When the two coppers overlap, the program must convert these values according to the overlay mask. This is accomplished here in only two steps without tedious CMPs. An AND operation is performed on the color and the current mask so only the copper masked with 0ffh is significant. The copper masked with 0 automatically goes to the background. If both are masked with 0, no overlay occurs and the two coppers are mixed. Provided that it appears at the current position (has a color other than 0), the bar masked with 0ffh overlays the other bar by resetting its color component to 0.

At this point DAC Color Register 0 is selected so time is saved during the actual switching of the color. It’s highly unlikely that a TSR would affect the program now. A color-value is also added for the case when the copper fails to occupy the maximum height of 128 lines (specified in the variable `height`). The dark components are cut out here instead of the middle components. The copper is set to maximum brightness by summing the components.

Now, as you’ve probably assumed by the CLI instruction, the time critical part begins. The procedure waits for entry into a horizontal retrace. As with its vertical counterpart, it waits for Display Enable followed by a retrace which is just beginning. This guarantees that even on slower computers, where the electron beam is already within the next retrace, the switch never takes place during display time.

Next, Color 0 must be set using the Pixel Color Value Register of the DAC. Here, Copper 1 determines the red component and Copper 2 determines the green component. Meanwhile, blue remains at 0 in both. The loop now starts from the beginning, unless the last line (determined by the variable `maxrow`) has already been drawn. In this case, another retrace is awaited and Color 0 set to black for the remainder of the screen. The wait for the retrace is necessary, otherwise the switch would occur during display of the last line which would then fail to fully appear. You can now re-enable the interrupts and end the procedure.
Shake On The Screen: The Wobbler

In addition to copper bars, the horizontal retrace can be used for many other effects. For example, instead of changing the color, what if you changed the horizontal position within each line? Such an effect, based on a sine table, will overlay the entire screen with a wave motion. Thanks to direct programming of the CRTC registers, you can even do this effect in text mode.

The horizontal location of a screen line is best controlled through CRTC-Register 4. This register, as its name Horizontal-Sync Start implies, determines the position where the horizontal retrace begins. Since the end of the synchronization is determined relative to the start, modifying this register shifts the "location" of the retrace on the screen but not its length (you should avoid modifying its length if possible). Some monitors cannot handle too short a retrace and will respond by completely distorting the image.

The strategy behind the Wobbler again consists of waiting for a specific screen line. It then feeds Register 4 with new values in each following line. These values are obtained from a sine table, generated using procedure Sin_Gen in the unit Tools (see the "Tables" information in the "Custom Mathematical Functions" section in Chapter 2. Of course, other functions will also work but the sine waves are the most similar to water waves.

We must consider timing very carefully for this effect. Here, unlike the copper bars, the registers should be modified while the electron beam displays the image. Assigning color with the coppers had no effect during the retrace but does during the display period. So we made the switches during the retrace. With WOBBLER.PAS, on the other hand, the position of the retrace is very important during the retrace itself (Blank Time), while the register has no effect during the display of the actual image data. So this time the changes must occur during the Display Enable period. To do this, simply reverse the order of the two wait-loops.

WOBBLER.PAS is also fully independent of image content because we’re working directly with the timing at the lowest level. You only need to consider the various register defaults in each mode when generating the sine table. The value is normally 85 in Text Mode 3 and 84 in Mode 13h and Mode X. Otherwise, the fixed areas of the screen will shift to the left or right because these are determined by the zero-point of the sine table.

Second, the value of this register can vary only within a certain range. For example if you use 87 as the zero-point by mistake and add an amplitude of 4 to it, the result will appear outside the acceptable range and no effect will occur. The sine at this point will be flattened out.

We should also mention, like all registers affecting horizontal timing, the Horizontal-Sync-Start Register is protected through Bit 7 of CRTC-Register 11h (Vertical Sync End). So, you must first clear this bit by calling CRTC_unprotect (Unit ModeXLib). You should then call CRTC_protect at the end of the program since the default value for the protection bit is 1. Both procedures are very simple and require no further explanation:
Chapter 5

Split-screen And Other Hot Effects

Procedure CRTC_UnProtect;
Begin
  Port[$3d4]:=811;              {Register 11h of CRTC (Vertical Sync End)}
  Port[$3d5]:=Port[$3d5] and not $80  {clear Bit 7 (Protection Bit)}
End;

Procedure CRTC_Protect;
Begin
  Port[$3d4]:=811;              {Register 11h of CRTC (Vertical Sync End)}
  Port[$3d5]:=Port[$3d5] or $80 {set Bit 7 (Protection Bit)}
End;

The program WOBBLER.EXE consists of the main program WOBBLER.PAS. The Assembler portion WOBBLER.ASM is linked into WOBBLER.PAS:

Uses Crt,Gif,ModeXLib,Tools;
const y=246;                    {height and position are defined here}
  height=90;                {can also be variables}
Var Sine:Array[0..63] of Word;  {Sine table, will be filled later}
  i:Word;                     {temporary counter}
Procedure Make_Wob(wob_pos,wob_height,wob_offset:word);external;
{$l wobbler}
begin
  TextMode(3);                  {Wobbler functions in ANY video mode! }
  Draw_Ansi('db6.ans');         {load Ansi file}
  Sin_Gen(Sine,64,4,83);        {precalculate sine}
  CRTC_Unprotect;               {enable horizontal timing}
  ReadKey;                      {wait}
  i:=0;
  Repeat
    inc(i);                     {generate movement}
    Make_Wob(y,height,i);       {draw wobble}
    Until KeyPressed;
  End;
  CRTC_Protect;                 {protect CRTC again}
end.

This program creates a test image in Text Mode 3, generates the sine table and then unprotects CRTC-Registers 0-7 (CRTC_unprotect) to enable access to Register 4. The loop, initiated by a keypress, continually increments the variable \( i \), which gives the current offset within the sine table in the procedure call to \texttt{Make_Wob}. This creates the wavelike motion.

Finally, the protection bit of the CRTC is reset and the protection mechanism reactivated. The actual procedure \texttt{Make_Wob} is located in the module WOBBLER.ASM:

extrn WaitRetrace:far

data segment public
  extrn sine:dataptr           ;sine table
data ends

extrn WaitRetrace:far

You can find WOBBLER.PAS on the companion CD-ROM

You can find WOBBLER.ASM on the companion CD-ROM
The procedure uses the following parameters: The vertical position, height in screen lines and the offset for the first line which then affects all the other lines. Incrementing this value by 1 moves the sine wave up one line on the screen and changing it continuously produces a type of wave motion.

This procedure waits for the number of screen lines specified in `wob_pos`. Prior to the two wait-loops for horizontal synchronization, the vertical position is calculated to save time later.

To access the sine table, the specified table offset is added. The result must fall within the table; therefore, an AND operation is performed using the value 63 (see the "Variables In Assembly Language" section in Chapter 1). Next comes the table lookup. The table consists of word entries (the reason for `shl bx,1`). The table value is temporarily stored in BX.
The program now waits for the beginning of the display period and not a retrace, which is precisely what we want to avoid. If the start-line has not yet been reached the line is incremented (jb next_line), otherwise an access occurs and CRTC-Register 4 is loaded with a new (sine) value.

The value in register SI was saved earlier. It still contains the vertical position and is compared to \texttt{wob height}. You must also wait here before resetting the register to its original value, so as not to interfere with display of the last line. Note that you are waiting for the beginning of a line display.

### Real-time Animation Made Easy: Palette Effects

The palette offers an excellent way to change the entire screen easily. All pixels of a particular color-value can be changed to a new color in a single step.

#### Fade-out effect

The simplest effect that we can create this way is called a fade-out. Similar to what happens in a film, the brightness of the image is reduced from normal to zero in a relatively short time. The palette makes a fade-out very easy to accomplish. All colors are decreased by 1 within a loop and the newly calculated palette is set. You then wait for the next retrace and again decrease by 1 until the entire screen is black.

The procedure \texttt{fade_out} from the module MODEXLIB.ASM performs this task:

```assembly
fade_out proc pascal far
    ; fades out image, video-mode independent
local greatest:word          ; contains maximum possible color value
    mov greatest,63
    mov ax,ds                ; load destination segment
    mov es,ax
main_loop:                  ; main loop, run once per image
    lea si,palette           ; source and destination offset to palette
    mov di,si
    mov cx,768               ; modify 768 bytes
    lp:                       
        lodsb                  ; get value
        dec al                ; decrement
        jns set                ; if not yet negative -> set
        xor al,al             ; otherwise 0
        set:                   
            stosb           ; write destination value in "palette"
            dec cx          ; loop counter
            jne lp
        call waitretrace    ; synchronize to retrace
        call setpal         ; set calculated palette
        dec greatest        ; decrement outer loop
        jne main_loop       ; still not finished ? then continue
    ret
fade_out endp
```

Here, the variable \texttt{max} provides the terminating condition. This variable is actually only a counting variable, counting backwards from 63 to 0, but you can also consider it as the current maximum image brightness possible. Since this value starts out at 63 (maximum brightness of a red, green or blue
component), \texttt{max} is initialized to 63. As the colors are decremented, this maximum brightness also decreases until it reaches 0 and the entire image is certain to be black.

After loading the source and destination pointers in the main loop, the procedure returns the secondary loop, which decrements the entire palette by 1. Each individual value is loaded, decremented by 1 and written back, with 0 as the limit.

Once initialized in this way the palette is set by \texttt{SetPal}, synchronized of course with the vertical retrace. It's also possible to set the palette at the same time as calculating it - just initialize the DAC for write-access (\texttt{port[\$3c8]:=0}) and rewrite the data each time to Port \$3c9. There is, however, one serious disadvantage: Calculating the new values takes a relatively long time; even longer than a retrace on slower computers.

The result is flickering at the top of the screen, which you can prevent by setting the palette all at once with \texttt{SetPal} (rep outsb), a much faster process. The main loop now runs until greatest is 0 and the screen reaches its darkest level.

This palette effect also works in text mode (in the program \texttt{FADE_OUT.PAS})

```
uses crt,modeklib,Tools;
var i:word;
Begin
  GetPal;                       \{load "Palette" with current DAC-palette\}
  Draw_Ansi('color.ans');       \{load picture\}
  Setpal;
  ReadLn;
  Fade_out;                     \{fade out picture\}
  ReadLn;
  TextMode(3);                  \{normal picture again\}
End.
```

To work with the palette you must first load it from the DAC registers into the corresponding variable. Normally in graphic mode, however, the palette is already stored in the variable \texttt{Palette} because the GIF loader already placed it there. The call to \texttt{Fade_Out} is then just a formality.

This shows a major advantage of direct DAC programming over using the BIOS: Accessing the palette using the BIOS is impossible in text mode and too slow in graphic mode. The only option left in this case is direct register manipulation.

\textbf{Fade-in effects}

The opposite of fade-out is, naturally, fade-in. A fade-in begins with a black image and the brightness increases to the desired palette.

The principle is the same as with fade-out. During each pass through the loop - synchronized with the vertical retrace - you increment the color-values by 1 or more, until the target value (from the image's original palette) is reached. Naturally you can't use 0 as the terminating condition, instead you must keep comparing with the target value.

The following example program is called \texttt{FADE-IN.PAS} and is primarily for demonstration purposes. We'll talk about a fade-in which uses a more general (and much more powerful) procedure in the next section.
uses crt,ModeXLib,Tools;
var i,j:word;
destpal:Array[0..767] of Byte;
Procedure Fade_in(DPal:Array of Byte);
Begin
  For j:=0 to 63 do Begin          {64 passes, in order to fade completely}
    For i:=0 to 767 do          {calculate 768 color values}
      If Palette[i] < DPal[i]   {current value still smaller that destination
        value ?}
        Then Inc(Palette[i]);   {then increase}
      WaitRetrace;                {synchronization}
      SetPal;                     {set calculated palette}
  End;
End;

begin
  ClrScr;                       {clear screen}
  GetPal;                       {load "Palette" with current DAC-palette}
  Move(Palette,Destpal,768);    {save palette}
  FillChar(Palette,768,0);      {delete old palette}
  SetPal;                       {and set}
  Draw_Ansi('color.ans');       {load wallpaper}
  ReadLn;
  fade_in(Destpal);             {fade to destination pal (original palette)}
  ReadLn;
  TextMode(3);                  {establish normal state}
End.

The main element of this program is the procedure fade_in. It receives the target palette as a parameter. The image will have this palette when the fade is completed. The current palette is assumed to be black.

The main loop of this procedure runs 64 times, when the brightest of whites will be fully visible. The loop calculates the new palette each time through and sends it to the VGA. The calculation is very simple: Each individual color-value is checked to see whether it is still below the target value. If so, the value is incremented, otherwise it stays at the target value (which cannot be exceeded).

The main program in FADE IN.PAS simply constructs a test image which is then faded in. Before this, however, it reads the current DAC palette (the future target palette) and stores it in Destpal. The palette itself can then be switched to black, the starting point for the fade-in.

**Fading to the target palette from any source**

Until now we have faded from a palette to black or vice versa. What’s missing is fading from one palette to another. At first this doesn’t seem to make much sense, resulting only in a step-wise distortion of image colors. However, this process can produce a very interesting effect (including its use in overlays which we’ll discuss later).

For instance, what if at the very end of a demo you took the last image and slowly reduced it to black-and-white? You could then display text with the credits for example.

Now we’ll use the new procedure. We’ll still need to fade to the black-and-white palette. The first problem here is how is a black-and-white palette generated? Although we could use a BIOS function for this, it’s too slow and inflexible. Fortunately, however, we can easily copy its method.
A color consisting of red, green and blue components is recalculated to black-and-white by adding the three components and assigning this value to all three colors in the new palette. Mixing all three colors in equally always produces a shade between black and white.

So how do we combine the colors? Simply letting all three colors flow in equal proportions won't give a satisfactory result. This is because the human eye doesn't perceive all colors as equally bright. For example a blue dot of maximum brightness appears darker than a green one.

The optimal ratio, which provides the most natural-looking display, is 30% of the red component, 59% of the green component and 11% of the blue component. This simple calculation is performed by the procedure Make_bw from the MODEXLIB.ASM module:

```pascal
Procedure Make_bw;              {Reduces a palette to black-and-white}
Var i,sum:Word;                 {Valuation: 30% red, 59% green, 11% blue}
Begin
  For i:=0 to 255 do Begin
    Sum:=Round(WorkPal[i*3]*0.3 + WorkPal[i*3+1]*0.59 + WorkPal[i*3+2]*0.11);
    FillChar(WorkPal[i*3],3,Sum);  {Enter values}
  End;
End;
```

For each individual color, the loop first calculates the sum. It then uses FillChar to write it to all three components. The program simply uses the Pascal floating-point calculation To form the sum because speed is not the main object. The rest is simple percentage calculation. The fade procedure is illustrated in the FADE_TO.PAS example:

```pascal
uses crt,ModeXLib,Tools;
var i:word;
  origpal,
  destinationpal:Array[0..767] of Byte;
begin
  ClrScr;
  GetPal;                       {load "Palette" with current DAC-palette}
  Move(Palette,OrigPal,768);    {save palette}
  Move(Palette,Destinationpal,768);    {set Destination-Palette}
  Draw_Ansi('color.ans');       {load Ansi-picture}
  Make_bw(DestinationPal);             {DestinationPal to black/white}
  readkey;
  fade_to(DestinationPal,1);           {fade to black/white palette}
  ReadKey;
  fade_to(OrigPal,1);            {fade to original palette}
  Readln;
  TextMode(3);                  {normal state}
End.
```

This procedure uses a colored text screen which fades to black when a key is pressed and faces to color again when another key is pressed.

To do this, GetPal is used to load the current text mode palette. This palette is then stored in OrigPal for later use and in DestinationPal for the black-and-white calculation.

Once the screen is initialized, a call to Make_bw recalculates DestinationPal to black-and-white. At this point nothing changes on the screen because only the array DestinationPal is being modified. The
actual call to \texttt{fade\_to} occurs after a key is pressed. Passed to this procedure is the target palette, or end result. The only requirement is that the current palette displayed by the VGA is the same as the one in the variable \texttt{Palette}.

Procedure \texttt{fade\_pal} uses another parameter as the step size to be used for fading. This determines the speed of the entire process. As each image is built, each color value is incremented/decremented by the amount given. The larger the value the faster the image fades.

Procedure \texttt{fade\_to} itself is found in the MODEXLIB.ASM module of the ModeXLib unit:

```asm
fade\_to proc pascal far destinationpal:dword, length1:word, step:byte
;fades "palette" to "destinationpal", passed by Pascal as array of byte !
local greatest:word
    mov ax,63 ;calculate number of passes
    div step ;necessary to reach 63
    xor ah,ah
    mov greatest,ax ;set number of loop passes
next\_frame:
    les di,destinationpal ;get offset holen, Pascal passes arrays far !
    les si,palette ;get offset of "palette"
    mov cx,768 ;process 768 bytes
continue:
    mov al,[si] ;get value from current palette
    mov ah,[di] ;get value from destinationpal
    mov bl,ah
    sub bl,al ;difference to destination value
    cmp bl,step ;more than one step over ?
        jg up ;-> decrement
    neg bl ;difference
        cmp bl,step ;greater than negative step
        jg down
    mov al,ah ;destination reached, finally set
write:
    dec cx ;decrement color loop
    je finished ;0 ? -&gt; finished
    mov [si],al ;write value in palette
    inc si ;select next value
    inc di
    jmp continue ;and continue
down:
    sub al,step ;decrement
    jmp write
up:
    add al,step ;increment
    jmp write
finished:
    call wait\_retrace ;synchronization
    call set\_pal ;set palette
    dec greatest ;all 63 passes finished ?
    jne next\_frame ;no -&gt; continue
    ret
fade\_to endp
```

The procedure \texttt{fade\_to} is part of the MODEXLIB.ASM file on the companion CD-ROM.
Next escdi is loaded with the palette pointer passed by Pascal. Only the offset is important here since the palette lies in the data segment anyway; however, it cannot hurt to also load the ES register (or perhaps mov di, destpal + 2 looks better to you).

The loop continue, which runs 768 times (256 colors * 3 components), loads AL with the current value of the just-processed color and AH with the target value to be reached. Now only the fade direction needs to be determined - up for increasing brightness or down for decreasing brightness. A simple comparison, however, is not enough.

Imagine this situation: The initial value is 0, the target value is 15 and the step size is 2. After 7 passes through the loop the current value is 14, which is too low and therefore must be incremented. It then becomes 16, which is too high and so is set back to 14. The palette entry continuously oscillates fluctuates around a particular value and never arrives at the target. This is disastrous, especially at fast fade rates and large step sizes where the oscillations are correspondingly large as well.

The solution is to test whether the current value has approached the target value by less than one step size. To do this the procedure simply forms the difference (in Register BL: if positively greater (JG, Jump if Greater) than the step size, the value is still far above its target value and must be decremented. On the other hand if the difference is less than (the amount is greater) a negative step size, the value must be incremented (NEG BL can be used to compare with a positive step size). When neither condition exists, the target value has been reached and the two values can be set equal to each other to eliminate the final small difference.

It makes no difference if you subtract the step size from the current value at Label down, add the step size to the current value at Label up or set both values equal to each other. Either way the path leads to write, which enters the new value into the palette and moves the pointer forward. When all 768 values have been processed, the program branches from the inner loop to Label finished. Here, after synchronization, the new palette is set and if greatest is still not 0, the next palette calculated (next_frame, following new image (frame) construction).

**Film techniques: Fading from one image to the next**

When fading between images it’s much more effective to fade-out the first image and then fade-in the next image (unless you’re intentionally making abrupt changes for effect). The procedures we’ve talked about so far can already do this. However, you must create a gradual transition from one image to the next for truly professional results.

As you can imagine, there are many “morphing” programs designed for this purpose. However, morphing programs deal with fixed images where the morph process calculates the desired changes. This requires enormous resources which occupy a great deal of room on both the hard drive and in memory. Furthermore, they must be copied (slowly and tediously) into video RAM. This method is an alternative to the real-time fade ins.

Fading in palette images (the graphic modes discussed are all palette-based) is basically a simple palette-fade. It uses the procedure fade_to we talked about in the previous section. Only with complicated metamorphoses, which combine image sections while moving at the same time, would you need to use morphing software.

Basically then you just fade one palette to another. However since the images lie one on top of the other and a change of image data (as in morphing) is out of the question due to speed considerations, the same image
data must represent different images depending on the palette. First, the source palette is active so the data represents the source image. At the conclusion, the target palette is active and the same image data now represents the target image. The intermediate steps are executed by the procedure `fade_to`.

This method raises two questions:

1. How do you manipulate image data so it will represent (depending on the palette) both the original, or source, image as well as the target image?
2. How do you create a palette that generates different images from the same data?

Let's look at question 1 first. With this type of fade, we're working with a fundamentally different problem than before. Until now, either the target color (fade-out) or the initial color (fade-in) was identical for all pixels, specifically black. Now any color can be faded to any other color. Some red pixels will be green in the target image, but some will later be blue.

Anyone familiar with the field of random combinational logic in mathematics recognizes what this means for the number of colors. The number of possibilities resulting from a particular number of colors combined with an equal number of different colors amounts to the number of colors squared. Each combination must be included in the fading and therefore requires an entry in the palette. To convert a two-color image to another two-color image, 4 palette entries are needed (Color 0 to Color 0, staying the same, Color 0 to Color 1, 1 to 0 and 1 to 1). So, with only two four-color images, 16 entries are already required.

Since VGA has only one palette consisting of 256 colors, the largest number of colors you can have without changing image data is 16. For multiple flicker-free fades, however, this number (as we'll explain later) is further reduced to 15.

Based on these combination possibilities, we can now derive a method for mixing the images. Because each combination must be represented, we use N blocks of N entries each, where N represents the number of colors per image. The block number corresponds to the target color, while the index within the block corresponds to the source color. A different way is also possible but that would unnecessarily complicate the reset (explained below). To determine the color number simply use the following formula:

\[
\text{color number} := \text{target color} \times \text{number of colors} + \text{source color}
\]

You may already be familiar with this principle. The same method is used to convert a hexadecimal byte to a decimal number: Upper nibble (higher-order position)\(\times\)16 + lower nibble. With 16 colors this would increase the speed tremendously: Simply load the target color in the upper nibble and the source color in the lower nibble to obtain the color value you need. The only problem with this is that instead of 16 colors we can only use a maximum of 15, so we must stay with the difficult multiplication. Fortunately, the part of the program handling the multiplication is not time-critical and creates at most a barely noticeable delay prior to the actual fade.

The following illustration shows the block setup for two four-color images fading into each other:
Chapter 5

Split-screen And Other Hot Effects

This illustration answers the second, still outstanding, question about palette organization. The block structure corresponds exactly to the palette structure. The source palette (of size 1 block) is copied the same number of times as the number of colors - creating exactly identical source blocks - while the target palette is expanded to its full size. This is done by individually multiplying each color so all the blocks in this palette are homogeneous.

Note this effect does not require multiple video pages or copying precalculated pages to the screen. Manipulation can take place directly on the screen.

You should be aware however the visible image doesn't change during the modifications (palette-generation and mixing). Otherwise, the image would flicker. Since the image cannot change there are two points to consider: First, the active palette can be changed only in areas where colors are not currently on the screen. Second, when changing the image data the palette must already be initialized so the modified pixels can again assume their original colors. So, if a red pixel, for example of Color 3, is to be set to Color 23, you must make certain that Color 23 already contains red, so nothing changes on the screen.

Next is the order of procedure calls to be maintained when fading:

1. Initialize palettes, changing only inactive colors
2. Mix image data, no change yet visible
3. Fade palettes into each other.

The requirement that only inactive palette areas can be changed results in adding an extra color block. This block initially contains the source colors, used by the source image. Since a four-color source image normally uses Colors 0 to 3, the new block will also be at this location. To generate the source palette (in this case copy the block four times), the block is taken as the initial palette and multiplied. The block itself does not change, so at this point nothing happens on the screen.

This block, which is known as the Reset-Block, has another important task to accomplish. Once the fade has executed, the visible image now has a much larger palette than either of the original images, specifically the number of colors squared. You'll have a problem if you want to fade-to yet another image because the maximum number of colors is 15. Fortunately, the target blocks are homogeneous which means all the colors in each block have the same contents and can therefore be replaced. There are only four different colors in our example, after all, on the monitor.

To reduce these many colors to a fadeable palette, you would again use the Reset-Block, which should already contain the source palette. This is also why it's called the Reset-Block. The greatly expanded image data is reduced, or reset, to this block. After the fade the Reset Block contains the actual target palette, so only the image data needs to be recalculated.
This calculation is very simple because only the expansion needs to be reversed to go back to the actual target color. To obtain the block number and therefore also the target color, simply divide by the number of colors. As with all such operations, you must still subtract the Reset-Block.

The Reset-Block is also the reason the number of colors must be limited to 15. In addition to the palette entries required for fading \( n^2 \), \( n \) represents the number of related colors, a Reset-Block is also needed, which again must contain \( n \) colors. In the end, fading two images requires \( n^2 + n + 1 \) palette entries. With 16 colors this would amount to \( 16^2 + 16 + 1 = 272 \) palette values, which unfortunately VGA does not provide. Therefore 15 color images are the most that can be faded into each other, requiring \( 15 \times 16 = 240 \) colors.

It often makes sense to restrict the colors even further to free unused colors at the upper palette end. You can then use these colors for static image sections, such as a fixed logo underneath which texts are faded into each other. Only colors from the very top of the palette can be used for the static image because the lower ones are already used for fading.

In mixing the two images, the new palette entry is calculated according to the above formula and rewritten to VGA-RAM. Remember, in practice only another block number is written the index within the block (corresponding to the source color) does not change. Since at this point all the blocks have the same contents, nothing changes on the screen.

At the end, the current palette (which originates from copying the source palette), is faded to the expanded target palette, so all blocks are homogeneous and contain the target color. The target image then becomes visible. Only the fading itself makes the changes visible, so this is the only time-critical phase. The procedure we are using, \texttt{fade-to} , is very speedy however. For each image construction only the palette registers are reloaded; this is much faster than changing the image data itself.

We’re including the following as an introductory fade application. The program, called \texttt{FADE_OVE.PAS}, uses the \texttt{FADE.PAS} unit which we will describe below.

```pascal
uses Crt,ModeXLib,gif,fade;
Var pic1_pal, pic2_pal:Array[0..767] of Byte;     {palettes of both pictures}
    pic1, pic2:Pointer;                               {contains 1st picture}
    pic1_pal:Array[0..767] of Byte;                   {2nd picture, equals vscreen}

Begin
    Init_Mode13;                                     {Mode 13h on}
    Screen_off;                                      {screen off during loading}
    LoadGif('chess');                                {load first picture}
    GetMem(pic1,64000);                              {get memory for 1st picture}
    Move(vscreen^,pic1^,64000);                      {save in pic1}
    Move(Palette,pic1_pal,768);                      {and save the palette}
    Show_Pic13;                                      {this picture on screen}
    LoadGif('box');                                  {load next in vscreen^}
    pic2:=vscreen;                                   {pic2 used as a pointer to it}
    Move(Palette,pic2_pal,768);                      {enable palette of picture 2}
    SetPal;                                          {and set}
    Screen_on;                                       {switch screen back on now}
    ReadLn;                                          {wait}
    Fade1(pic2,pic2_pal,0,0,200);                    {and then fade in picture 2}}
    fade_ResetPic(0,200);                            {prepare renewed fading}
    ReadLn;
End.
```

You can find \texttt{FADE_OVE.PAS} on the companion CD-ROM
This program fades two completely different GIF images into each other. It requires two variables. Variable `pic1` contains a pointer to the image data of the first image in main memory (this pointer must first be allocated). Variable `pic2` points to the second image - here an allocated block of proper size already exists in main memory (vscreen). For clarity, however, we'll continue referring to the pointer `pic2`.

The two individual palettes are stored in the variables `pic1_pal` and `pic2_pal`.

After turning on Mode 13h the program disables the screen because the palette changes during loading. A professional looking demo might display a blank screen during the loading process.

The first image is loaded and moved to address `pic1` (previously allocated); its palette is stored in `pic1_pal`.

The second image is loaded after the first image has been copied to VGA RAM. This image is found in `vscreen` for the remainder of the program. However, we'll use its equivalent, the pointer `pic2`, which at this point must still be allocated. Once the second palette has also been stored in a variable, the program activates the first palette (copies it to `Palette`) and sends it to the VGA (`setpal`). The screen can now be turned on again.

The fading itself occurs in the procedure `Fade`. It receives the following parameters:

- The target image’s address
- Its palette
- The y-coordinate within the target image
- The y-coordinate to which the target image should be copied
- The height of the target image

The last three values have no direct function so far in this program because they describe the copying of a complete image. The first two parameters contain values for the second image (the image being faded to).

After fading, the image is reset to enable another fade. We also pass a y-coordinate within VGA-RAM here and a height although they still perform no function. The program then fades back to the original image and ends.

The central command of this program is the call to the procedure `Fade1`. This procedure, found in the unit `Fade`, combines all the tasks necessary for fading.
Chapter 5

Split-screen And Other Hot Effects

Procedure Fade1(Pic:Pointer;Pal:Array of Byte; Start,y,Height:Word);
Implementation
Var i,j:Word;                   {temporary counter}
   Destination_Pal:Array[0..768] of Byte;  {temporary destination palette}
Procedure fade_set(Source:Pointer;Start,y,Height:Word);external;
{mixes" source with VGA-Ram}
{use source beginning at Start line and VGA-Ram beginning at y line at height of Height}
Procedure fade_ResetPic(y,Height:Word);external;
{prepares faded picture for another fade}
{reduction of "Colors^2" to "Colors" colors}
{here again y=line in VGA-Ram, Height=Height of area to be edited}
{$l fade}
Procedure fade_CopyPal;
{multiply palette on Colors^2 (multiply non-homogenous Block 0}"
Begin
   For i:=1 to Colors do
      Move(Palette[0],Palette[i*3*Colors],Colors*3);
End;
Procedure fade_spread(Var Pal:Array of Byte);
{spread palette to Colors^2 (multiply each color separately)}
{the homogenous blocks are formed here from the colors 0..Colors-1}
Begin
   For i:= 0 to Colors-1 do      {edit each color separately}
      For j:=0 to Colors -1 do    {write Colors once each time}
         Move(Pal[i*3],Pal[(i+1)*3*Colors+j*3],3);
   End;
Procedure Fade1(Pic:Pointer;Pal:Array of Byte; Start,y,Height:Word);
{Fades from current visible picture to Pic (with Palette Pal). During
this process, in the "Start" row, the program begins copying "Height"
rows to the y-coordinate y of the current picture.}
Begin
   Colors:=15;                   {only default value !}
   End.
The only global variable in this unit is Colors, which gives the number of colors in each individual image. When fading two 15 color images into each other this value equals 15 (the default value).
The entire fade process is controlled by the procedure Fade1. To eliminate flickering, it first waits for a retrace. The procedure then takes the source palette, which is actually the current palette (in the variable Palette), and copies it to initialize it for fading. This palette is then set (no change occurs on the screen because only inactive palette entries are being changed).
The target palette is assembled by the two Move statements which follow. The current palette is taken as the source palette so static image sections, which use entries at the very end of the palette, are also
represented in the target palette. The palette passed in the variable `pal` then goes into the lower area (Colors entries of 3 bytes each). This lower area is expanded and initialized for fading by `fade_spread`.

The actual image data is mixed in `fade_set`, so depending on the palette the data will represent either the source or the target image. Finally, the actual fade to the target palette takes place, using the familiar `fade_to` procedure, and the target image slowly appears on the screen.

The unit contains two internal Pascal procedures `fade_CopyPal` and `fade_spread`. Since speed is not important here, we’ve kept these procedures in Pascal for greater clarity. `fade_CopyPal` initializes the source palette (contained in Palette) by copying the Reset-Block (Colors 0 to Colors -1) `Colors` times. The loop simply copies `Colors` *3 bytes to the block positions `Colors` times.

`fade_spread` is slightly more complicated. Each color must be expanded to one block. The program accomplishes this through two nested loops. The outer loop (counter `i`) expands each color (from 0 to Colors -1) to the necessary width. The `j` loop is inside this loop. It generates `Colors` copies of each color, lying directly one behind the other within the block. Now the block corresponding to a particular color is filled with this exact color. The Move statement is responsible for the filling, whereby the active color (at palette position `i*3`) is copied to each position (`j*3`) within the current block ((i+1)*3*Colors).

Besides these Pascal procedures, the unit also uses two Assembler procedures from the module FADE.ASM: `fade_set` and `fade_ResetPic`.

```assembly
data segment public
        extrn colors:word
        data ends
        code segment public
        assume cs:code,ds:data
        public fade_set,fade_ResetPic
        col db 0 ;code segment pendant to colors

        fade_set proc pascal near source:dword, start:word, y:word, height:word
        mov ax,colors ;colors entered in code segment variable col
        mov col,al
        push ds
        mov ax,word ptr source + 2 ;source pointer to ds:si
        mov ds,ax
        mov si,word ptr source
        mov ax,320 ;start address within the source image
        mul start
        add si,ax
        mov ax,0a000h ;destination pointer 0a000:0 to es:di
        mov es,ax
        mov ax,320 ;start address within the destination image
        mul y
        mov di,ax
        mov ax,320 ;convert height to number bytes
        imul height
        mov cx,ax
        lp:
        lods b ;destination value in al
        mul col ;calculate new color value
        add al,es:[di] ;add current value
        add al,col
        stos b ;and write back
        jmp lp
```

You can find FADE.ASM on the companion CD-ROM.
These procedures also refer to the global variable *Colors*, which is listed here in the data segment. In the meantime because of changes made to ds, *fade_set* can no longer access this variable and must therefore use the code-segment variable *col*.

To do this, *fade_set* first copies the contents of *Colors* to *Col*, whereby ds is now free to accept the target pointer (segment component). Si then receives the offset of the target image (image data). Here you must consider the starting coordinate, which gives the line from which the target image is read. This offset is calculated by multiplying by 320 and is then added to si.

Next the pointer es:di receives the VGA address, where the segment is simply the VGA-RAM start-segment. As with the target image, the offset is calculated by multiplying the y-coordinate by 320.

Finally, cx also receives a value, which is the height multiplied by 320. The procedure then enters the main loop *lp*.

Here the procedure reads the target value from the target image (dss:i). This value corresponds to the block number. It must, therefore, be multiplied by the number of colors. As an offset within the block, the source image color is read from video RAM and added on.

After writing this value back, the loop ends by decrementing cx and performing the required branch.

The second procedure in the assembler portion is *fade_ResetPic*. It reduces the visible image back to *Colors* colors once the fade is complete. Using the block number, which corresponds to the target color, the colors are reduced to the Reset-Block (color numbers 0 to *Colors*-1).

Here again, a pointer to VGA-RAM at the corresponding y-coordinate is stored in es:di, and based on the height cx receives the number of passes through the loop.
The main loop in this procedure is `res_lp`. It reads a value from video RAM, calculates its block number by dividing by the number of colors and rewrites the value. Again it must consider the Reset-Block, which keeps block numbers in the range 1 to `Colors`. They must, however, return to their actual color values in the range 0 to `Colors`. The loop then closes and the procedure ends.

In our example a full 15-color image was faded into another 15-color image, involving (almost) the entire color palette. Reducing the number of colors to 14 for example, gives you colors at the upper end of the palette which are unaffected by the fading. You can then use these colors to display static image sections which remain unchanged during the fading.

This is one way of displaying credits for a demo. A small image (say, a screen shot) is faded in for each demo section. Then text of the credits are faded into each other in the bottom half of the screen.

You can see this process in the program `FADE_TXT.PAS`:

```pascal
Uses Crt,Gif,ModeXLib,Fade;
Var
  Text_Pal:Array[0..767] of Byte;
  i:word;
Begin
  Init_Mode13;                  {use Mode 13h}
  Screen_Off;                   {screen off during loading}
  LoadGif('vflog210');          {load static part}
  Move(Palette[210*3],          {its palette part (colors 210..255)}
   Text_Pal[210*3],46*3);        {enter}
  Show_Pic13;                   {copy static picture to VGA}
  LoadGif('texts');             {load picture with texts}
  Move(Palette,Text_Pal,14*3);  {its palette part (colors 0..13)}
   {enter}
  Move(Text_Pal,Palette,768);   {set finished palette}
  SetPal;
  Move(vscreen^,                {first text can be copied}
   Ptr($a000,160*320)^,19*320);{directly to screen}
  Screen_On;                   {picture now finished -> switch on}
  Colors:=14;                   {pictures with 14 colors in this program !}
  For i:=1 to 6 do Begin        {fade to the next 6 texts, one after the other}
    Delay(500);                 {time to read}
    Fade1(vscreen,              {fade next picture to old location (y=160)}
     text_pal,i*20,160,19);     {and "reset"}
    If KeyPressed Then Exit;    {anyone who has had enough can cancel here}
  End;
  Readln;
  TextMode(3);
End.
```

This program first loads the static portion onto the screen, and writes its contribution to the total palette into the variable `Text_Pal`. The image with the texts is then loaded into vscreen and its palette section copied to `Text_Pal`, after which the palette is loaded and set.

The program next copies the first text portion directly to video RAM, because a fade is not yet required. Once the image is switched back on, the main loop begins. For each of the remaining six texts the following occurs: First, the program waits for half a second to give the viewer time to read. The text is then faded in, this time using the coordinates in the call to `Fade1`. The program copies from the target image starting at Line `i*20`,...
because in this example the texts begin at y-coordinates 0, 20, 40, etc. They’re all 19 pixels high and are faded in at screen line 160.

The same coordinates are again used in the call to **Fade_ResetPic**. At the end comes a (not very structured) wait for a keypress, enabling the user to exit.

When developing your own images for this effect, the correct palette distribution is important. For images being faded the maximum number of colors is 15 images with more colors need to be reduced. In all cases these colors must lie in the lowest area of the palette (0..Colors -1).

The following formula gives the number of colors for the static image:

\[
\text{static colors} = 256 - \text{dynamic colors} \times (\text{dynamic colors} + 1)
\]

The static colors must always lie at the upper end of the palette, because only this area remains unchanged during fading.

Therefore when fading 15-color images back and forth, at the end of the palette there are 256 - 15 * 16 = 16 colors left over for static images (Colors 240 to 255). In a 13-color fade there are 256 - 13 * 14 = 74 colors (Colors 182 to 255).

**The faster alternative: Animation through palette rotation**

So far in this chapter we’ve talked about using simple palette modifications to change large areas of the screen in a single step. We can extend this advantage of palette-based graphic modes to animations. Images in a true animation not only fade-in and fade-out but actually move across the screen.

How is this possible? As a simple example, let’s take a red pixel and move it ten pixels from left to right. To do this, just draw ten pixels next to each other, each with a different (adjacent in the palette) color, in this example Colors 0 to 9. Now if you set Color 0 to red and the other nine colors to black, only the first of the ten pixels will be visible. Now shift the palette up by one, so Color 1 is red and Colors 0 and 2 through 9 are black. The second pixel becomes visible and the first disappears.

Continue to shift the palette and the pixel will travel across the screen from left to right, without a single byte changing in video RAM (once the ten pixels are initialized). When the red entry arrives at the upper end (Color 9), you can set it back to Color 0 and thereby creating a cyclic animation where the pixel continually moves from left to right and jumps again to the left. You can also reverse the motion at the end so the pixel moves back and forth.

This method obviously makes little sense for a single pixel. Instead of two bytes being changed in video RAM (clear old pixel, set new one), you’re already resetting ten color-values using relatively slow accesses to a port. A much more efficient method with large and/or complicated images is palette rotation. We recommend using palette rotation with either a high number of pixels or complex mathematical descriptions of motion are involved.

You must format larger areas before they can be moved. These areas cannot consist of just one palette color, instead the colors must flow in the direction of motion. You are, in effect, moving line by line. The following illustration shows how this is done:
We’re scrolling up through blocks 2 pixels in height. The principle is exactly the same for larger blocks and more complex structures. The sample image contains two-and-a-half blocks, alternately colored red and blue. The number of blocks makes no difference; you can expand the image up and down as far as you like but the order of colors must be maintained. Fractions of blocks are also possible.

The structure of the overall image is periodic, in other words, it repeats itself after two blocks. Therefore two blocks are filled using a constant color progression from Colors 0 to 3. In the palette, Colors 0 and 1 are set to red and Colors 2 and 3 to blue. These initializations are best made in a paint-program, so all our program needs to worry about is the motion itself.

You generate this motion by shifting the (partial) palette one position downward over the range 0 to 3. Color 0 receives the color from Color 1, Color 1 the one from Color 2, Color 2 the one from Color 3 and Color 3 the one from Color 0, therefore creating a rotation. As you can see in the illustration, the red and blue areas are shifted up by one line, exactly as we intended.

This method of scrolling doesn’t make too much sense either, since you can just as easily do the same thing by changing the screen-start address (Linear Starting Address). The simple block structure of our example becomes interesting only when you add effects. For example, if you construct a chessboard out of these blocks and use a paint program to rotate it backward slightly, palette rotation will give you a backward scrolling or forward scrolling plane. Programming this conventionally through pixel manipulation requires far too much calculation. So leave the calculating to the paint program, and all your program needs to do is move a few bytes in memory and set the palette.

A further advantage is that objects can be placed freely in front of the scrolling-plane. Normally you would have to copy the objects there with each image construction, using a (relatively slow) sprite routine. With this technique the objects can be static within the image.

When creating such images, in all cases, you must have a palette-based paint program. Unfortunately, most of today’s Windows paint programs are based on TrueColor, which makes it impossible to tell what palette color a pixel will later assume. For coherent color progressions, you need direct control over the palette.

We created the PALROT.GIF sample image in Deluxe Paint II Enhanced. This program handles perspectives and direct palette manipulations. First, a black-and-white chessboard was drawn. We then filled all white blocks with Colors 16 to 31 (color progression red to blue) and all black blocks with Colors 32 to 47. You can
now freely manipulate this chessboard. Its graphic presentation has no affect on the palette rotation; in our example we have angled it backward so it represents a plane.

Notice the logo above the chessboard. It occupies unused areas at the upper end of the palette. Above the chessboard and to the right you see a slightly different application of palette rotation often used in adventure scenes called a fountain. The fountain uses Colors 48 to 63, with a progression from black to blue. When placed in front of a black background, this color combination creates the impression of flowing water.

The last effect in this image is a red “radar screen”. This is generated with a circular color progression from Color 64 to 88. This color progression consists of 25 concentric circles which have adjacent color-values. Only Colors 64 and 80 are set to red in the palette. The remaining colors are set to black. When these two red color-rings are rotated, they move across the rings with Colors 64 through 80, and generate expanding concentric circles.

For basic control of effects, Deluxe Paint has a function called Rotate color, which (very slowly) carries out a rotation. But how do you generate this effect in your own programs? Here we present the procedure Pal_Rot, from MODEXLIB.ASM module, which receives the area to be rotated as a parameter. In the normal case, the area in question shifts downward by one position (toward lower color numbers), and the lowest color moves back to the top. However, if the first limit passed is larger than the second, the procedure rotates upward:

```assembly
Pal_Rot proc pascal far
Start,End1:Word
;rotates palette part by 1 from start to end1
;if start < end1 : rotation down
;if start > end1 : rotation up
mov ax,ds
mov es,ax
lea si,palette
mov di,si
mov ax,3
mul start
add si,ax
mov ax,3
mul end1
add di,ax
mov bx,[si]
mov di,[si+2]
mov cx,di
sub cx,si
add si,3
;same for destination
mul end1
add di,ax
mov bx,[si]
sub cx,si
mov di,si
add si,3
;jready for forward copying
cld
or cx,cx
;if cx negative (start > end1)
jns forward
neg cx
;necessary for forward copying
sub si,4
;j to 2nd byte of penultimate color
add di,2
;di to 2nd byte of last color
add cx,2
;copy 2 bytes more,
forward:
;so that position is correct after copy loop
```

The procedure Pal_Rot is part of the MODEXLIB.ASM file on the companion CD-ROM.
Once the passed color-value is converted to byte-offsets, the starting color needs to be stored because the remaining colors will shift toward it and overwrite it.

The number of bytes to be shifted equals the difference between the two offsets, which is calculated below. To initialize the shift itself, di is set to the initial offset since the shift proceeds in that direction. Assuming a forward shift, si is set to the following color, which in the first step is copied to the staring color.

If Start is greater than End (number of bytes to be copied is negative), the copying proceeds upwards and certain registers need to be adjusted. First, the Direction-flag is set because with overlapping source and destination areas, a memory shift from bottom to top means the program must copy backwards. Otherwise, the copied values would overwrite the uncopied ones, giving you at best a very ineffective algorithm for filling a memory area.

Also, since cx is negative, you need to reverse its sign. Next si and di must be properly aligned, because si at this point is on the color-value immediately following the palette area (it was moved there previously with add si,3, which is correct for forward copying). Now si must be set to the last byte of the second-to-last color (the last byte because rep movsb copies backward). Also, di must point to the last (second) byte of the last color so this color and the ones below it from top to bottom are filled.

Due to this positioning, when the copying ends di does not point to the lowest color. You can fix this by inserting a copy-direction inquiry after the loop and adjusting di accordingly, but it’s easier to just copy two more bytes and after the copy-loop, replace them with the stored start-color bytes. This loop and the replacement of the copy-loop are again identical for both directions, and therefore follow directly after the label forward. Finally, the Direction-flag is cleared to avoid confusion in other procedures from incorrect string-directions.

The companion CD-ROM includes the sample program PALROT.PAS, which uses the image PALROT.GIF:

```pascal
Uses Crt,ModeXLib,Gif;
Var slow_flag:Boolean; {for controlling the slow progressions}

Begin
Init_Mode13; {Mode 13h on}
LoadGif('palrot'); {load and display image}
Show_Pic13;
Repeat
Pal_Rot(16,47); {move "chess board"}
If slow_flag Then Begin
   Pal_Rot(63,48); {move "fountain"}
   Pal_Rot(88,64); {move "radar"}
End;
slow_flag:=not slow_flag; {alternating "fountain" and "radar"}
WaitRetrace; {enable and lock}
End;
you can find PALROT.PAS on the companion CD-ROM
```
This program loads the image onto the Mode 13h screen and in a loop, rotates through the palette areas of each effect in the corresponding direction. To experiment with these directions, switch the values passed to `Pal_Rot`.

While the chessboard scrolls at full speed, this is too fast for the fountain and radar screen; the palette sections for these are rotated only every other pass through the loop. This is controlled by the Boolean variable `slow_flag` which alternates between TRUE and FALSE.

### Blazing Monitors: FIRE!

Watching a "blazing" fire on your screen may not seem so special today as it would have been a few years ago. These flames, however, are based on painted images played consecutively one after the other. We'll also talk about another option often used in demos which directly imitates the structure of the flames.

There are two important points to consider here: A fire continually moves upward. Also, fire starts out white at the bottom and fades more towards red as it nears the top. The program FLAMES.PAS illustrates the procedure. Two buffers are required to reproduce the video RAM (or its lower half) in fast system RAM.

The procedures `Show_Screen`, `Prep_Pal` and the main program are very easy to understand:

```
Procedure Show_Screen; {copies finished screen to VGA}
Var temp:Pointer; {for exchanging pointers}
Begin
  asm
    push ds
    lds si, Dest_Frame {finished picture as source}
    mov ax, 0a000h {VGA as destination}
    mov es, ax
    mov di, 320*100 {starting at line 100}
    mov cx, 320*100/4 {copy 100 lines as Dwords}
    db 66h {Operand Size Prefix (32 Bit)}
    rep movsw {copy}
    pop ds
  End;
  temp:=Dest_Frame; {exchange pointers to source and destination pictures}
  Dest_Frame:=Src_Frame;
  Src_Frame:=temp;
End;

Procedure Prep_Pal; {prepare palette for flames}
Var i:Word;
Begin
  FillChar(Palette, 80*3, 0); {foundation: all black}
  For i:=0 to 7 do Begin
    Palette[(i*3+2)i*2]; color 0-7: increase blue
    Palette[(i+8)*3+2]=16-i*2; color 0-7: decreasing blue
  End;
  For i:=8 to 31 do {color 8 -31: increase red}
```
The main program allocates and clears both buffers, turns on Mode 13h and initializes the palette for flames. The result is shown in the following graphic:

Pixels with lower color numbers will later represent the cooler zones within the flames. These appear at the left in the diagram above. As you read from right to left, i.e., from higher to lower temperatures, note the following:
Initially, all color components are at maximum (= white). Starting at Color 80 the blue component gradually disappears, so the appearance is somewhat more yellow. At Color 56 the green component is also diminished, so only red remains. Once this color (starting at number 32) also fades out, the result normally would be black. In this case, however, we add a small glimmer of blue (Colors 16 to 0), standing above the red flames. To get rid of this you can simply delete the first loop from the procedure.

The main loop consists of three commands: \texttt{Scroll\_Up}, \texttt{New\_Line} and \texttt{Show\_Screen}. After the new screen has been initialized in the buffer \texttt{Dest\_Frame}, \texttt{Show\_Screen} moves it to the screen through a simple (DWord) loop. The two buffer pointers are then switched so the finished screen is used as the source the next time through.

\texttt{Scroll\_Up} and \texttt{New\_Line} are the central procedures.

\begin{verbatim}
Procedure Scroll_Up; assembler;
{scrolls the picture one line up and interpolates}
asm
push ds
les di,Dest_Frame             \{load pointer to destination picture\}
lds si,Src_Frame              \{pointer to source picture\}
add si,320                    \{in source picture on line 1\}
mov cx,320*98                 \{scroll 99 lines\}
xor bl,bl                     \{required as dummy for High-Byte\}
@lp1:
xor ax,ax
xor bx,bx
mov al,[si-321]               \{get first point\}
mov bl,[si-320]               \{add next point\}
add ax,bx
mov bl,[si-319]               \{add next point\}
add ax,bx
mov bl,[si-1]                 \{etc...\}
add ax,bx
mov bl,[si+1]
add ax,bx
mov bl,[si+319]
add ax,bx
mov bl,[si+320]
adc ax,bx
mov bl,[si+321]
adc ax,bx
shr ax,3
or ax,ax                      \{already 0 ?\}
je @null
dec al                        \{if no, then decrease\}
@null:
stosb                         \{value to destination\}
inc si                        \{next point\}
dec cx                        \{other points ?\}
jne @lp1
pop ds
End;

Procedure New_Line;           \{rebuilds the bottom lines\}
Var i,x:Word;
Begin
For x:=0 to 319 do Begin       \{fill bottom 3 lines with random values\}
Dest\_Frame^[97,x]:=Random(15)+64;
End;
\end{verbatim}
Scroll_Up moves the sea of flames one position up and interpolating at the same time. In other words, each new pixel is calculated from the average of its immediate surroundings. This is how the flames become blurry and faded.

In loop @lp1, the procedure takes the eight adjacent pixels for each source pixel, sums them up and divides by 8. This average is decremented by 1 so the flames won’t flare up too high and then written to the destination buffer.

New_Line is responsible for adding new lines at the bottom screen border (the “embers” of the fire). The first loop fills the bottom three lines with random values from 64 to 79. The second loop adds “hot spots”, very hot zones with flames shooting out. To do this, blocks of nine pixels are set to 255 so as you scroll upward, they cool off very slowly and appear as glowing zones rising to the top.

The Secret Of Comanche: Voxel Spacing

A popular component of many demo programs and even a few games is voxel spacing. Using this effect, the observer moves over an imaginary landscape with hills and valleys.

Although this is a very general description, many methods exist for displaying this effect. In fact, there is likely to be as many algorithms as programmers who have already tried this effect. The procedure we show here has no complex three-dimensional images, and relies instead on simple two-dimensional relationships.

Basically, the program takes a map and tips it on its side so the observer views it at an angle from above. Altitude can now be represented by vertical lines of varying lengths.

The map, which depicts the high and low points of the landscape, is a simple 320 x 200 image with transitions as smooth as possible. Color 0 represents the lowest point and Color 255 the highest.
In practice, two-dimensional projections depend on a fairly basic principle. As we know, objects farther away appear smaller than objects closer to us. So when viewing a landscape through a frame (i.e., your computer monitor) you’ll find that more distant objects can fit in the frame than closer objects because the distant objects are smaller. What you see will always be similar to the following illustration.

Example of a landscape drawn with voxel spacing

The rectangular frame shows the entire existing landscape from a bird’s eye view. A trapezoid displays its visible portion. All you need to do now is display this trapezoid on a rectangular screen section. This rectangle must stretch the front horizontally so objects in this area appear larger. Any coordinates refer to a projection on the lower screen-half of Mode X.

The trapezoidal projection is formed by steadily decreasing the number of pixels that fit onto one line. So, the ratio of landscape size to screen-pixel size is continually decremented. The initial value is a 1:1 diagram, with 80 pixels to a line (this resolution is used for speed). As the image is drawn the ratio becomes smaller and smaller, until perhaps a 1:2 diagram is reached and only 40 pixels fit on the frontmost line.

In line-by-line display you must also remember the distance between lines always becomes smaller as you move backward. Therefore, you should continually increase the line spacing when drawing from top to bottom.

The only thing missing now is the altitude information for the image. As we mentioned, this involves taking the color for each pixel and arranging a corresponding number of pixels one on top of the other. The greater the color-value, the higher the pixel grows into a vertical line. This vertical structure also eliminates gaps arising from the ever increasing line-spacing toward the front.

To make the scenery more realistic, you’ll need to add bodies of water. Simply take all colors below a certain value and set them to this value. This creates surfaces without a vertical structure.

The easiest way to generate the landscape is to use a fractal generator that can generate plasma-clouds. FRACTINT is used for this purpose. Simply generate a plasma-field of resolution 320 x 200 and then load the palette LANDSCAP.MAP (by pressing C). The completed image can be saved with Q.

The program VOXEL.PAS redraws the scene in a loop in response to mouse-movement (control is through mouse):
Uses Crt,Gif,ModeXLib;

Var x,y:Integer;          \(\text{Coordinates of the trapezoid}\)

Procedure Draw_Voxel;external;
{$l voxel.obj}

Begin
  asm mov ax,0; int 33h End; \(\text{reset mouse driver}\)
  Init_ModeX; \(\text{enable Mode X}\)
  LoadGif('landsc3'); \(\text{load landscape}\)
  x:=195; \(\text{define start coordinate}\)
  y:=130;
  Repeat
    ClrX($0f); \(\text{clear screen}\)
    Draw_Voxel; \(\text{draw landscape}\)
    Switch; \(\text{activate completed video page}\)
    WaitRetrace; \(\text{wait for retrace}\)
    asm
      mov ax,000bh \(\text{Function 0bh: read relative coordinates}\)
      int 33h
      sar cx,2 \(\text{Division by 2}\)
      sar dx,2
      add x,cx
      add y,dx
    End;
    If x < 0 Then x:=0; If x > 130 Then x:=130;
    If y < 0 Then y:=0; If y > 130 Then y:=130;
    Until KeyPressed; \(\text{until key}\)
  TextMode(3);
End.

The central procedure is \textbf{Draw_Voxel}, found in Assembler module VOXEL.ASM:

\begin{verbatim}
data segment
extrn vscreen:dword           ;pointer to landscape data
extrn x,y: word               ;coordinates of trapezoid
extrn vpage:word              ;current video page
data ends

;variables with fractional part \(\text{lower 8 bits}\):
offst dd 0                    ;current offset
step dd 0                     ;pixel size
row_start dd 0                ;beginning of current row
row_step dd 0                 ;distance from next row
r_count dw 0                  ;counter for depth
shrink dw 0                   ;correction on lower screen border
row dd 0                      ;current screen row number
vpage_cs dw 0                 ;video page in the code segment

.386
public Draw_Voxel
Draw_Voxel proc pascal
;shows landscape on current video page
;reads data from vscreen starting from position \(x/y)\)
mov ax,vpage                  ;note number of video page
mov vpage_cs,ax
\end{verbatim}
Chapter 5
Split-screen And Other Hot Effects

The first part initializes several important variables. We should mention Offst and row_start in particular. Both form a fixed-point value in Bits 8-31 and following the decimal point in Bits 0-7), which gives the offset of the current pixel or current line. After each pixel, Offst is incremented by Step, moving you one step further in the landscape; for example if Step equals 080h (= 0.5), a new pixel is read from the landscape every two steps.

row_start and row_step are similar; row_step contains 14040, or 320.25 in decimal notation. After each line this sets row_start to the beginning of the next line and at the same time takes care of the narrowing toward the front, by starting the next line a little farther to the right.

Here the program calculates an offset before each line. The variable row contains the current line on the screen, which is converted to an offset. This line number should not be confused with row_Offset, which represents the position within the landscape. row represents the position on the screen. Even if the calculated offset lies below the screen, drawing must still occur because the vertical lines of this pixel can still project into the screen. The program handles this special case by again placing the destination pointer di in the screen and shrinking the height of the bar (by the number of lines given in shrink).

Now the increasing space between screen lines comes into play. The current y-coordinate of the screen is found in the variable row, which is multiplied here by a factor of 1.007, so the lines continue to spread out as they move forward.
For each pixel, the color at the current offset is read and the water level set to a minimum value of 99. The counter $z\_count$, which gives the distance of each line from the observer, is used to calculate the flight perspective. This process will be discussed in Chapter 7. Finally, the height (stored in al) is decremented by the value $shrink$, so the bottom screen border can be processed.

```
push di
next_fill:
    mov es:[di],cl                ;Enter color
    sub di,80                     ;Address next higher line
    dec al                        ;Decrement counter
    jne next_fill                 ;Continue ?
    pop di
weiter:
    inc di                        ;Address next byte on screen
    mov esi,step                  ;Get step size
    add esi,offst                 ;Add up
    mov offst,esi                 ;and rewrite
    dec bp                        ;Next pixel
    jne next_x
    mov esi,row_step              ;Move line-start
    add esi,row_start             ;and rewrite
    mov row_start,esi
    mov offset,esi                ;Also reload pixel-offset
    dec step                      ;Decrement scaling factor
    dec $z\_count                 ;Line counter continues
    jne next_y
    pop ds
ret
Draw_Voxel endp

code ends
end
```
The loop `next_Fill` now draws the vertical bar of the given size. The program then calculates the offset of the next pixel and, if a new line is beginning, its offset as well. In addition, the step size Step is decremented after each line to create the expansion in the foreground.

Enlarging Graphics: The Magnifying Glass Effect

In this section we’ll talk about an effect that is similar to what happens in nature when light is refracted. In MAGN.PAS use the mouse to move the cursor, shaped as a magnifying glass or “glass ball”, across a GIF image. The portion of the image immediately below the cursor is distorted, as you can see in the following illustration:

![Example of using the glass ball cursor over a GIF image](image)

You can achieve several effects by simple reprogramming the VGA register, which saves a tremendous amount of CPU time. However, VGA cannot help us with calculated images such as that under the glass ball cursor. Instead, the CPU has to control all the plotting/imaging work. However, as long as you don’t overdo it with the size of the image area to be manipulated, you can manage with this (the glass ball cursor will have a diameter of 49 pixels).

When the CPU calculates the data of the image to be displayed, make certain each point really does get processed so gaps do not appear by recalculating: You run through each point of the destination image and calculate the source point with which it corresponds. To do this, you have to reverse the mathematical allocation.

Also, you have to use double-buffering to prevent flickering when the CPU is building the image. Double buffering is a technique where data in one buffer is being processed while the next set of data is read into the second buffer. If you don’t use double buffering, you’ll come into conflict with the physical refresh from the cathode ray when you make changes to the image data.
Many algorithms can describe the desired distortion. However, the physical correctness of the refraction isn’t as important as obtaining the fastest possible execution speed. Therefore, we’ll use a simple mathematical statement:

We’ll use a unit circle (origin in (0/0), radius 1) as the shape of the magnifying glass. The radii of the original and projected magnifying glass points are in quadratic ratio to each other; as a result, the area around the center point becomes greatly lengthened while the outer area becomes compressed. Now all we have to do is resolve this ratio to the root coordinates (rg: original, even coordinates, rl: distorted magnifying glass coordinates).

Make certain the coordinates are in the interval [-1;1] by subtracting the radius of the magnifying glass beforehand and then divide by this value. At a width of 48, 12 becomes:

\[ x_l = \frac{(12-24)}{24} = -0.5 \]

You do the opposite with the result: Multiply it by the radius and then add it.

MAGN.PAS shows this effect. The algorithm is located in the assembly routine MAGN.ASM. GIF.ASM includes a GIF loader for the format 320*200*256. Although the GIF loader has limitations (i.e., reading other formats and that it cannot even read other color depths), it is very fast.

After allocating memory for the buffer and the background (Buffer and BackGnd), graphic mode 13h is set and the mouse driver is initialized. The image is loaded and stored in the background array. Next, use the circle equation:

\[ x^2 + y^2 = r^2 \]

to make certain the magnifying glass changes to a circular shape. It calculates where the drawing begins and ends for each row (arrays Circle_Start and Circle_End). The following loop takes the current coordinates from the mouse driver and checks them. Next, the background is copied to the buffer to clear the magnifying glass from the previous refresh and draw the new magnifying glass by calling MAGN.ASM. After copying to video RAM, the loop closes with a keyboard inquiry.

Assembly procedure MAGN.ASM displays the magnifying glass. The coordinates of the magnifying glass are passed to this procedure is passed as parameters zx and zy:

First, the procedure copies the circle array to the code segment because the DS register (and the data segment) won’t be available later. The outer loop begins after calculating the destination address (Pascal array Buffer) to the desired coordinates (zx/zy) and loading the source address (the background BackGnd). This loop increments the y-coordinate from 0 to 49.

By using this coordinate, the current offset is taken from array Circle_Start, added to Destination_Offset. At the same time it’s used as the initialization value for the x-counter. Also, the end coordinate of the row is loaded and stored temporarily in cur_End.

Now the x-loop implements the inferred calculation: The y-coordinate (y_count) is reduced by 24 and divided by 24. Meanwhile, the result is shifted to the left by 8 bits to allow for decimal places. The value must range from -1 to 1 which includes the (hexadecimal) numbers ff00h - 0100h (e.g., in this case 0.5 would be
The result is temporarily stored in edx. Then the x-coordinate is converted in the same way and stored in ecx.

Now the radicand of the root is calculated by squaring both coordinates separately and then adding them. Function *root* is in charge of extracting the root, and performs an approximation in accordance with the formula

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right) \]

In this formula \( a \) is the radicand.

The result becomes the desired width again by multiplying it times 24 and then stored in edi. Now the root is multiplied by \( y \) and then by \( x \), which completes the formula (Note: the results have to be shifted to the right by 16 bits to compensate for the squared 8 bit shift from before). Each of the destination coordinates \((zx/zy)\) are added and the origin is shifted back to the upper left corner. The offset is calculated from these two values and stored in esi.

Now di is also loaded and the pixel can be copied with *movsb*. Both loops are closed; after each row the destination offset is switched by one row (320 bytes).

Assemble MAGN.ASM and GIF.ASM and compile MAGN.PAS. Now move the magnifying glass across the screen with the mouse and see what happens.

Now it's your turn...it's really not too difficult to write your own small effect.
Sprites are most commonly used in video and action games. A sprite might appear as directional pointers in games or as action figures. Anything on the screen which moves differently than the background picture can be called a *sprite*. A sprite is defined as an independent graphic object which is controlled by its own bit plane. Sprites move freely across the screen and can move or collide or even pass through each other.

Now that you have a general idea of what a sprite is, we'll see how you can program sprites.

**The Basic Ideas Behind Sprites**

To create your own sprites, you must first carefully consider their internal make-up. Actually, sprites are simply graphic cut-outs that are positioned anywhere over a background picture. These sprites can be small or large depending on the required computer effect.

Sprites must meet at least one more requirement: They need to be transparent in places so the background can be seen through the sprite. This transparency is necessary for the "holes" inside the sprite as well as for areas around the sprite's edges. Every sprite that is not a true rectangle needs transparent edges. Certain forms such as circular forms or other irregular forms cannot be used due to speed considerations. Therefore, only rectangular areas are usually used because the edges are filled. Otherwise, filled up rectangles will always appear independent of their form.

Early computers such as the Commodore 64 or the Atari ST usually handled sprites quite easily with very little programming...you only needed to specify the position of the sprite. The task is more complicated when using a PC because the sprite is part of the VGA screen memory. Although this requires more programming, it has many more advantages compared to the earlier home computers. The programmer controls the layout completely and the size of the sprite is adjustable. Other advantages include scaling or rotating sprites.

The following lists the three basic operational procedures you'll need to follow when presenting sprites:

1. Erase the sprite by copying the background picture onto the current screen page.
2. Movement and presentation of the sprite.
3. Switch to the completed page.

Before a sprite can be displayed at a new position, the previous one must first be erased from the screen. You might attempt to copy the background, draw the sprite and then rewrite the background. However, since this method requires an enormous amount of time and effort, it only makes sense when using one or two sprites.
First, it is only practical to make a single copy of the background in video memory where there is reasonably fast access. However, the limited amount of video memory will eventually be exhausted. Second, the time and effort required far exceeds the cost of copying a screen page. Finally, we’re using areas in the middle of the screen that can only be copied line by line. A simple REP MOVSB of the entire area is not possible.

A complete background page is copied and positioned in video memory with a fast copy loop placing it on the current displayed page.

You have complete freedom in moving a sprite. You can move it across the screen in a line or in circles. A sprite can move as a sine wave or follow the rhythm of music. It’s your decision on how, where and when you want the sprite to move.

Before continuing, we want to introduce two screen pages. As we mentioned, it’s nearly impossible to perform extensive picture manipulations within a vertical retrace but it’s the only way to avoid flickering and “tearing” the sprites. Besides, in the procedure mentioned above, there are no sprites on the screen for a brief period (between steps 1 and 2). We know this to be true because all the sprites flicker.

You need to add a second screen page here to prepare this invisible page while the visible one remains untouched. You only switch on the new screen page after finishing all the changes and while the previously visible one is then redrawn.

Two basic concepts can be used for the presentation:

You can leave the sprite data in the graphic memory, for example leave it on Page 2 and always copy it to the current page from there. However, this has two disadvantages. The fast Write-Mode 1 is based on complete blocks of four (one byte from all four planes), so it’s not possible to copy a sprite, for example, from x-coordinate 5 (plane 1) to 6 (plane 2).

Data can be quickly copied within the plane. The only way to avoid this problem is to place every sprite in the memory four times, once in the x-position 0, once in the x-position 1, etc., which greatly limits the number of available sprites.

The second disadvantage of this concept concerns the unmasking of the transparent points. This is only possible by using separate masks that are contained in the main storage unit but which must first be created. We’ll talk about a better alternative in the next section.

---

**Reading And Writing Sprites**

A better alternative is for the sprite data to remain in the CPU and moved point-by-point to the screen. You can overcome the relatively limited speed by optimizing your routine as much as possible.

This starts when you format the sprite data in the CPU. The graphic data is divided in the four planes for positioning a sprite. Points with addresses divisible by 4 are allocated to plane 0. Addresses with a remainder of 1 are allocated to plane 1, etc. The easiest way to bring data to the screen is to place the points in the “normal” series and to constantly shift the write plane.

We saw in the procedure `p13_2_ModeX` this is not exactly the fastest solution. This procedure yields to another method which is applied to the sprites: First, copy all the data from one plane, then switch and copy the next plane.
The outcome results in the sprite format: The data does not remain here in Mode-13h format (contiguous pixels), but in one of the Mode X related series. First, the complete data from the first plane arrives. This is followed by the data from the second plane, which is followed by the data from the third plane, etc. When writing the sprite, you can access this data without additional time and effort; the starting address simply needs to be loaded in the SI-register. Then copy the first plane and, immediately thereafter, the next plane is available. Adjusting the SI-register is unnecessary.

The data contain the color values and can be copied without any difficulty to the screen memory. Only Color 0 has special significance. It represents the background and the sprite is transparent in these positions.

A problem occurs when calculating the width of the sprite because it isn’t the same in every plane. Imagine a five pixel wide sprite that should be written at x-position 0. In this case, two bytes must be copied in plane 0 (pixel 0 and 4), while the others contain only one byte.

The algorithms we discuss are for the general case, although not dependent on the x-coordinate of the final position. If the sprite is copied, for example, in the x-position 1, the 2 byte wide block physically finds itself in plane 1. Because the copy loop begins with this plane, considering the sprite data, this is always a matter of plane 0, which, moreover, is two bytes wide. In other words, with only the sprite width, you can create an array with the width of the individual planes, which the number variable provides in the copy loop. The first plane of the sprite data always two bytes wide, the others are one byte, regardless of their current position.

You need to consider the initial plane’s purpose is to communicate with the VGA.

---

**Beyond All Borders: Clipping**

In most programs, a sprite is made to move towards and eventually disappear at the edge of the screen. If you do not adjust for this, you’ll get undesirable results by copying the data for the entire sprite to video memory. If the sprite is to disappear from the right or bottom edge of the screen, part of the sprite will be copied to the next line or screen page. Since this will not produce the desired result, you have to design an algorithm which performs clipping or cutting off the “invisible” data.

One method for clipping is to verify that each sprite point falls within the range of the visible screen before drawing it. Unfortunately, this is also the slowest method. A better way is to change the margin specifications before copying the data to video memory. In doing this, you should try to perform the clipping outside of the copy loop. Doing calculations this way reduces the amount of time to write the sprite data to video memory.

Clipping is quite simple. The width, height or both of the sprite are adjusted to exclude parts that are outside the visible area. For sprites that disappear from the left or top edge of the screen, you must determine the first visible point of the sprite by calculating which point is to be drawn at x-coordinate=0 (left edge) or y-coordinate=0 (top edge).
The Unit Sprites

All these special features can be found in the procedures GetSprite and PutSprite in the Unit Sprites:

```pascal
{SG+}
Unit Sprites;
Interface
Type SpriteTyp=Record           {structure of a sprite data block}
   Addr:Pointer;                 {pointer to graphic data}
   dtx,dty:Word;                 {width and height in pixels}
   px,py,                        {current position, optional *}
   sx,sy:Integer;                {current speed, optional *}
End;

{*: optional means that the sprite routines GetSprite and PutSprite
don't use these specifications, the variables serve only the purpose
of making control easier on the part of the main program}
Procedure GetSprite(Ofs,dtx,dty:Word;var zSprite:SpriteTyp);
{read a sprite from vscreen-Offset oofs, with dtx width and dty height,
zsprite is the sprite record in which the sprite is to be saved}
Procedure PutSprite(pg_ofs,x,y:Integer;qsprite:spritetyp);
{copies sprite from RAM (position and size taken from qsprite)
to video RAM page pg at position (x/y)}
Implementation
Uses ModeXLib;
Var i:Word;
Procedure GetSprite;
Var ppp:Array[0..3] of Byte;    {table with number of pixels to copy}
   {per plane}
   Skip:word;                  {number of bytes to skip}
   Plane_Count:Word;           {counter of copied planes}
Begin
   GetMem(zsprite.addr,dtx*dty); {allocate RAM}
   zsprite.dtx:=dtx;             {note width and height in sprite record}
   zsprite.dty:=dty;
   i:=dtx shr 2;                 {number of smooth blocks of 4}
   ppp[0]:=i;ppp[1]:=i;          {equals minimum number of bytes to copy}
   ppp[2]:=i;ppp[3]:=i;
   For i:=1 to dtx and 3 do      {note "excess" pixels in ppp}
      Inc(ppp[(i-1) and 3]);      {add pixels beginning with Startplane}
   Plane_Count:=4;               {copy 4 planes}
   asm
      push ds
      les di,word ptr zsprite     {first load pointer to data block}
      lea bx,ppp
      lds si,vscreen
      add Ofs,si
      @lcopy_plane:               {will be run once per plane}
         mov si,ofs               {load si with start address of sprite data}
         mov dx,dtiy              {load y-counter with row number}
         xor ah,ah                {clear ah}
         mov al,ss:[bx]           {load al with current ppp entry}
         shl ax,2                 {blocks of 4 get moved}
         sub ax,320               {obtain difference to 320}
         neg ax                   {make 320-ax out of ax-320}
         mov skip,ax              {store value in skip}
         @lcopy_y:                {run once per row}
   End;
   Skip:=Skip+ppp[i-1];         {skip all but the first block}
End;
```
Sprites: Rapid Action On The Screen

Chapter 6

PC underground

mov cl,ss:[bx]          ; load width from ppp-array
@lcopy_x:
  movsb
  add si,3               ; to next pixel of this plane
  dec cl                 ; copy all the pixels of this row
  jne @lcopy_x
  inc bx,skip            ; after that, at beginning of next row
  dec dx                 ; copy all rows
  jne @lcopy_y
  inc ofs                ; position at next ppp entry
  dec plane_count        ; position at new plane start
  jne @lcopy_plane
  pop ds
  End;

Procedure PutSprite;
var plane_count,        ; counter of copied planes
  planemask:Byte;      ; (masked Write-Plane in TS-Register 2)
  Skip,                ; (number of bytes to skip)
  ofs,                 ; (current offset in video RAM)
  plane,               ; (number of current plane)
  Width,               ; (width of bytes to be copied in a row)
  dty:Word;            ; (height)
  source:Pointer;      ; (pointer to graphic data, if ds modified)
  clip_lt, clip_rt:integer;  ; (number of excess PIXELS left and right)
  clipact_lt, clipact_rt:Byte;  ; (with current plane active number)
  clipact_rb, clipact_rb:Word;  ; (excess BYTES)
  clip_dn,clip_up:Word;  ; (number of excess ROWS above and below)
  ppp:Array[0..3] of Byte;  ; (number of pixels per plane)
  cpp:Array[0..3] of Byte;  ; (excess BYTES per plane)
Begin
  if (x > 319) or       ; (Display unnecessary, )
    (x+qsprite.dtx < 0) or  ; (because it’s not even in the picture ?)
    (y > 199) or    ; (y+qsprite.dty < 0) then exit;
  clip_rt:=0;           ; (no clipping normally)
  clip_lt:=0;           ; -> all clipping variables to 0
  clipact_rt:=0;
  clipact_lt:=0;
  with qsprite do begin
    if y+dty > 200 then begin
      clip_dn:=(y+dty-200);  ; (first clipping case: down)
      dty:=200-y;            ; (note excess rows)
      End;
    if y<0 then begin
      dty:=dty+y;           ; (and reduce sprite height)
      y:=0;                 ; (Start-y is 0, because upper screen border)
      End;
    if x+dtx > 320 then begin
      clip_rt:=x+dtx-320;   ; (third clipping case: right)
      dtx:=320-x;           ; (note excess pixels)
      End;
    if x<0 then begin
      clipact_rb:=x;        ; (reduce width)
      plane:=plane and 3;  ; (calculate new start plane for column 0)
      plane:=plane and 3;
      End;
  end;
Chapter 6

Sprites: Rapid Action On The Screen

```
ofs:=pg_ofs+80*y+(x div 4);  {set Ofs to correct block of 4}
x:=0;                         {begin display in column}
End Else Begin              {no clipping to the right ?}
  plane:=x mod 4;           {then conventional calculation of plane}
ofs:=pg_ofs+80*y+(x div 4); {and offset}
  End;
End;

Source:=qsprite.addr;         {pointer graphic data}
dty:=qsprite.dty;             {and save height in local variables}
Width:=0;                     {preinitialize width and skip}
Skip:=0;
i:=qsprite.dtx shr 2;         {number smooth blocks of 4}
ppp[0]:=i;ppp[1]:=i;          {equals minimum number of bytes to be copied}
ppp[2]:=i;ppp[3]:=i;
For i:=1 to qsprite.dtx and 3 do(note "excess" pixels in ppp)
  Inc(ppp[(plane+i - 1) and 3]);{note "excess" pixels in ppp}
End;

asm
  mov dx,3ceh                   {GDC Register 5 (GDC Mode)}
  mov ax,4005h                  {set to Write Mode 0}
  out dx,ax
  push ds                       {save ds}
  mov ax,0a000h                 {load destination segment (VGA)}
  mov es,ax
  lds si,source                 {source (pointer to graphic data) to ds:si}
  mov cx,plane                  {create start plane mask}
  mov ax,1                      {move bit Bit 0 left by plane}
  shl ax,cl
  mov planemask,al              {save mask}
  shl al,4                      {enter in upper nibble also}
  or planemask,al
  mov plane_count,4             {4 planes to copy}
  @lplane:                        {will run once per plane}
    mov cl,byte ptr plane       {load current plane}
    mov di,cl
    mov cl,byte ptr ppp[di]     {load cx with matching ppp number}
    mov byte ptr ppp[di]        {recalculate skip each time}
    mov ax,80                   {obtain difference 80 width}
    sub al,cl
    mov byte ptr skip,al        {and write in skip}]
    mov al,byte ptr ppp[di]     {load plane specific clipping width}
    cmp clip_lt,0                {if left no clipping, continue with right}
    je @right
    mov clipact_lt,ax           {save in clip_act_lt}
    sub Width,ax                {reduce width of bytes to be copied}
    jmp @clip_rdy               {no clipping right}
  @right:
    mov clipact_rt,ax           {clipping for all planes, in clip_act}
  @clip_rdy:                   {calculate total width in bytes}
    mov ax,Width
    add ax,clipact_rt
    add ax,clipact_lt
    mul clip_up
    add si,ax
    mov cx,Width
```
In addition to the two procedures, this unit also contains the definition of **SpriteTyp**. It holds several important attributes of each sprite. It contains a pointer to the actual graphic data in main memory (`addr`); the sprite's width (`dtx`) and the sprite's height (`dty`). Procedure **GetSprite** initializes these variables before they're used by **PutSprite**. Make certain that they're not overwritten by other procedures.

Depending on the application, record **SpriteTyp** can be expanded. Since the straight line movement of a sprite is the most frequent application, four components are defined to handle future applications; these components aren’t used by the procedures in this unit:
Sprites: Rapid Action On The Screen

- \(px\) and \(py\) are the current position of the sprite
- \(sx\) and \(sy\) are the speed (step size)

Again, these procedures do not use components \(px\), \(py\), \(sx\) and \(sy\). Remember, they’re available if you want to use them.

A sprite must first be created. In other words, you have to first define a sprite. We’ll use the GIF format in these examples because it’s so universal.

After loading a picture containing a sprite, it is transferred to memory. The \(\text{GetSprite}\) procedure is used to do this.

We’ll assume the sprites are located in the video memory and that the coordinates are expressed as offsets. In other words, the coordinates are passed to procedure \(\text{GetSprite}\) as offset parameters. This is quite simple to do. Since the pictures are stored in video memory in 13h Mode, the conversion is as follows:

\[
\text{Offset} := y\text{-coordinate} \times 320 + x\text{-coordinate}
\]

Using this offset relative to video memory, \(\text{GetSprite}\) uses two other parameters \(dtx\) and \(dty\) to "cut" the sprite from the GIF image and copy it to memory.

The routine which copies the sprite to memory, \(\text{GetSprite}\), uses two other parameters \(dtx\) and \(dty\) to "cut" the sprite from the GIF image and copy it to memory.

The offset \(\text{Offset}\) is then used to load the coordinates of the source and target arrays. The source \((\text{vscreen}+\text{Ofs})\) and target \((\text{zsprite.addr})\) are loaded in es:di and ds:si; register bx points to array \(\text{ppp}\).

The outer loop \(\text{@lcopy_plane}\) executes once per plane. During each iteration, register si is set with the new staring value, register dx with the sprite height and variable \(\text{skip}\) with distance between points.

Within loop \(\text{@lcopy_y}\), executes once per sprite row. After a row is copied, \(\text{skip}\) is added to register si which takes us to the next sprite row. Register dx specifies how many times the loop is executed, depending on sprite’s height in rows.

Procedure \(\text{PutSprite}\) is similar. It uses four parameters - the offset of the screen page, the x-coordinate and y-coordinate for the upper left corner of the sprite and record \(\text{SpriteTyp}\).

In some instances, clipping is simply handled. The easiest case is for a sprite which isn’t visible on the screen. In this case, the procedure is interrupted by an Exit command and the program continues normally.

If the sprite is partially visible, \(\text{PutSprite}\) has to perform additional calculations to handle the clipped portion of the sprite.

For a sprite which has to be clipped at the bottom of the screen \((y + dty > 200)\), then the height of the sprite is reduced to fit on the screen. Since the sprite is examined four times (once for each plane), you must make sure that the pointer is reset to the beginning of the next plane.

Clipping at the top border is also simple. Again, the number of lines to be skipped is stored in \(\text{clip_up}\) and the height is truncated.

Clipping at the right border is slightly more complicated. Here, you have to consider the arrangement of the color planes. The routine makes sure that the image is clipped correctly for each plane. The number of pixels is stored in \(\text{clip_rf}\) and the width is truncated.
Clipping at the left border is different. Here, too, the number of pixels is recorded, this time in `clip_lt`. However, the width is left unchanged. Instead, the sprite copy routine adjusts for the narrower width. In addition, the routine performs additional calculations to determine the starting plane. For example, assume that our sprite extends three pixels off the left edge. The starting plane for this sprite would normally be column -3.

Of course, this is not possible; the first visible column is (-3 + 4 = 1) column 1. The offset must also be recalculated since it points to a place in the previous line when it has a negative x-value. The x-coordinate is set to 0.

The adjusted plane and offset values apply only to clipping along the left border; in other instances the values are calculated as before in Mode X.

Array `ppp` is initialized for the different plane widths.

Array `cpp` is similarly initialized. Later, `cpp` will hold the clipped image from the left or right borders.

The clipping width (`clip lt` + `clip rt`) divided by 4 gives us the number of even blocks.

The assembly language section sets up for Write Mode 0 in order to address individual points. After the source and target pointers are loaded, a mask is defined to select the current plane.

Variable `plane_count` ensures that the loop `@lplane` is performed four times. Within the loop, array `ppp` specifies how many pixels are to be copied. The step size is set into variable `skip`.

Array `cpp` is used to save the clipped image. For sprites that are clipped on the right border, variable `clipact rt` is loaded with this width from `cpp`. Correspondingly, variable `clipact lt` is used for sprites that are clipped on the left border.

Now we’re ready to perform the first clip. The total width of the line including the pixels to be clipped on the left and right sides is multiplied by the number of lines to be clipped at the top border. This value is placed into the `si` register.

In simplest terms, the loop starting at `@lcopy` simply moves the individual bytes from the source address to the target address. The majority of the code is used to handle the different planes.

The program `SPRT_TST.PAS` includes the applications for these routines:

```pascal
Uses Crt,Gif,ModeXLib,Sprites;
Const Sprite_Number=3;          {number of sprites used in the program}
Var Sprite:Array[1..Sprite_Number] of SpriteTyp;  {data records of sprites}
  i:Word;                     {counter}
Begin
  Init_ModeX;                   {enable Mode X}
  LoadGif('sprites');          {load image with the three sprites}
  GetSprite(62 +114*320,58,48,Sprite[1]);  {coordinates (62/114), width 58*48}
  GetSprite(133+114*320,58,48,Sprite[2]); {(133/114), 58*48}
  GetSprite(203+114*320,58,48,Sprite[3]); {(203/114), 58*48}
  LoadGif('wallpaper');        {load wallpaper}
  p13_2_ModeX(48000,16000);     {and copy to background page}
```
After loading a sprite, it's placed (set) in arbitrary positions and at arbitrary speeds. The repeat loop highlights the basic principle of the sprite presentation: Erase the background (CopyScreen), move the sprites, display the sprites. The WaitRetrace command is placed behind the Switch command because switching to a new page causes a delay in retrace. We're using a different logic here than we used before. First, the command to switch over is sent to the VGA. Then wait for the retrace to be certain that the screen will be changed at that instant. Now you will be able to prepare the next page.

Unfortunately, slower computers can become overtaxed with the three sprites from this demo program. If this occurs, simply set the number lower.

**Use Scrolling For Realistic Movement**

To create the illusion of depth and add a three-dimensional appearance, a sprite must be reduced in size as it "moves" towards the back of the screen. One way to do this is to define an individual sprite for each intermediary step and redraw it to the appropriate size. In terms of execution speed, this is the fastest method; but it also requires the most memory.

For some situations, you may want to use a single sprite and automate the resizing tasks. While there are general algorithms that can do this, its a very tedious process; it involves remapping each point of the sprite to a new position.

Instead, there's another method which makes the sprite appear to scroll quickly. The technique is quite simple. Rather than redraw the entire sprite, only a percentage of the sprite lines are redrawn during each
Sprites: Rapid Action On The Screen

iteration. For example, if the percentage is 50%, then every second line of the sprite is skipped as it is drawn on the screen.

Procedure **SCAL_TST.PAS** demonstrates the technique:

```pascal
{SG+}
Uses Crt,Sprites,ModeXLib,Gif,Tools;

Procedure PutScalSprt (pg_ofs,x,y,scale_y:Integer;qsprite:spritetyp);
var planecount,                  {counter of copied planes}
    planemask:Byte;             {masks Write-Plane in TS-Register 2}
    Skip,                      {number of bytes to skip}
    ofs,                       {current offset in video RAM}
    plane,                     {Number of current plane}
    Width,                     {width of bytes to be copied in a line,}
    dty:Word;                  {height}
    source:Pointer;            {pointer to graphic data, if ds modified}
    ppp:Array[0..3] of Byte;   {number of pixels per plane}
    rel_y,                     {fractional portion of rel. y-position}
    add_y:Word;                {fractional value of the addend}
    direction:Integer;         {direction of movement (+/- 80)}
    i:Word;

Begin
if (x + qsprite.dtx > 319)    {Clipping ? then cancel}
or (x < 0) or (y + qsprite.dty*scale_y div 100 > 199) or (y < 0) then exit;
add_y:=100-abs(scale_y);      {calculate addend}
if scale_y < 0 then direction:=-80 else direction:=80;
Source:=qsprite.adr;          {Pointer graphic data}
dty:=qsprite.dty;             {load local Height variable}
plane:=x mod 4;               {calculate start plane}
ofs:=pg_ofs+80*y+(x div 4);   {and offset}
Width:=0;                     {preinitialize Width and Skip}
Skip:=0;
i:=qsprite.dtx shr 2;         {number of smooth blocks of 4}
inc(ppp[(plane+i - 1) and 3]);{add pixels beginning with Startplane}
asm
    push ds                    {save ds}
    mov ax,0a000h              {load destination segment (VGA)}
    mov es,ax
    lds si,source              {source (pointer to graphic data) to ds:si}
    mov cx,plane
    mov ax,1                    {move Bit 0 left by plane}
    shl ax,cl
    mov planemask,ax           {save mask}
    shl al,4                    {enter in upper nibble also}
or planemask,al
    mov planeindex,4           {4 planes to copy}
    @lplane:                    {will run once per plane}
    mov cl,byte ptr plane
    mov di,cl
    mov cl,byte ptr ppp[di]    {load cx with matching ppp number }
    mov byte ptr Width,cl      {recalculate skip each time}
    mov ax,direction
    sub ax,cl
```

You can find **SCAL_TST.PAS** on the companion CD-ROM
Chapter 6

Sprites: Rapid Action On The Screen

```
mov rel_y,0                  ; start again with y=0,0
mov cx,Width                ; load cx with Width
or cl,cl                    ; (Width 0, then Plane finished)
jc @plane_finished         ; destination offset in video RAM to di
mov ah,planemask           ; (reduce plane mask to bit [0..3])
and ah,0fh                  ;
mov al,02h                  ; (and through TS - Register 2 (Write Plane Mask))
mov ax,rel_y                ; (and write in skip)
mov rel_y,ax                ; and write in skip
mov dx,3c4h                  ; (and through skip)
out dx,ax                    ; (and write in skip)
@lcopy_y:                     ; (and through skip)
    lodsb                      ; (and through skip)
or al,al                      ; (and through skip)
    stosb                      ; (and through skip)
    loop @lcopy_x              ; (and through skip)
@lcopy_x:                     ; (and through skip)
    lodsb                      ; (and through skip)
or al,al                      ; (and through skip)
    stosb                      ; (and through skip)
    loop @lcopy_x              ; (and through skip)
mov ax,rel_y                ; (and through skip)
add ax,rel_y                 ; (add to fractional portion)
add dx,dty                   ; (set)
@entry:                      ; (set)
    loop @lcopy_x              ; (set)
    lodsb                      ; (set)
or al,al                      ; (set)
    stosb                      ; (set)
    loop @lcopy_x              ; (set)
mov planemask,0              ; (set)
and planemask,1              ; (set)
add planemask,1              ; (set)
inc planemask                ; (set)
```

Var Logo:SpriteTyp;
Var Sine:Array[0..99] of Word;
Var Height:Integer;
Var i:Word;

Begin
    Init_ModeX;               ; (enable Mode X)
    LoadGif('sprites');      ; (load image with logo)
    GetSprite(88+ 6*320,150,82,Logo);  ; (initialize logo)
End;
```
In this program we demonstrate how to rotate a logo in front of a transparent background. The height is periodically changed (the y-coordinate in a circular motion appears as a sine oscillation from the page). Every sprite is displayed using `PutSprite` or scaled with `PutScalSprt`.

The procedure `PutScalSprt` is merely a demo and is therefore not placed in its own unit. Besides, for speed of execution, it doesn't perform any clipping. Its parameter is shared with the familiar data using `scale_y`, the percentage of the original height the scaled sprite is to show. Here, a negative value indicates a mirroring of the x-axis.
Each of the effects we have demonstrated so far have one thing in common: They function only in two dimensions. To produce pictures with snap, you might consider the third dimension. The challenge is to produce a three-dimensional presentation on a two dimensional monitor. Additionally, the monitor can only display believable depth information if we use a few tricks, such as vanishing point perspective and point of light shadings.

These tricks require an understanding of mathematics, especially geometry. Transformations, illustrations, brightness gradations, etc., are very complex calculations when you consider the number required for a presentation. These calculations must be adapted to programming logarithms to enable fast drawings in many cases. Therefore, the language used for these procedures is assembly, since a high-level language may produce code which executes too slowly.

Throughout this chapter, we will use a small main program to introduce the practical applications and methods. These programs are all based on the same assembly modules (3DASMA.SAM, POLY.ASM, BRES.ASM, TEXTURE.INC), which contain independent presentation functions (transformations, etc.) as well as special sections (point of light shading), which are activated through global variables. Using global variables for direction may not seem like a good idea at first. However, it will save both computing time and code.

Mathematics For Graphics Enthusiasts

If you’re already familiar with analytical geometry, you can probably skip this section. However, if you’d like a quick refresher, we’ll explain geometric concepts like vector, determinants and intersecting product (sum). These terms are important if you want to produce three-dimensional presentations.

The vector

What the number is to algebra, the vector is to geometry. A simple analogy for a vector is an arrow that points in a particular direction and which has a particular length. Besides, a vector can be moved arbitrarily in space and it’s defined relatively. A vector is defined as a line designated by its end points (x-y or x-y-z coordinates). A circle consists of many small vectors.

In this definition, there is no indication of where, or at which specific location in space, a vector is found. The opposite is the case with location vectors. These vectors always start at the coordinate system’s origin.
(0/0) and specify an exact point. In the vector manner of writing, the x, y and z vector components are written on top of one another:

\[
\vec{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \quad \text{Or} \quad \vec{b} = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}
\]

For example:

\[
\vec{a} = \begin{pmatrix} 2 \\ 3 \\ -5 \end{pmatrix}
\]

**Mathematical depiction of a vector**

**Calculating (computing) with vectors**

As we mentioned, a vector describes both direction and length. The length, which is often required, can be determined very easily from the components if you use the three dimensional Pythagorean theorem:

\[
\begin{align*}
\vec{a} &= \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \\
|\vec{a}| &= \sqrt{a_x^2 + a_y^2 + a_z^2}
\end{align*}
\]

For example:

\[
\begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \\
3^2 + (-2)^2 + 1^2 = 14
\]

In a way, we add vectors because the sums of their components are totaled. In a geometric definition, this addition means placing the two vectors so one is on top of the other.

Subtraction is performed similarly. Here the components are subtracted. Mathematically, the second vector is added to the first one.

**Vector subtraction**

There are three types of multiplication:

**Scaling (S-multiplication)**

One is called scaling (S-multiplication). Here, only one vector is multiplied by the number, so only its length changes. An S-multiplication with a negative number reverses the direction.
Chapter 7

The Third Dimension: 3-D Graphics Programming

Vector multiplication

The second is scalar multiplication. It multiplies two vectors with each other, whereby the product of the two is a number (a scalar). The following illustration shows the two definitions for this:

1. $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$

2. $\vec{a} \times \vec{b} = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$

For example:

$\begin{pmatrix} 1 & 3 \\ 2 & 5 \\ 3 & 1 \end{pmatrix} = 1 \cdot 3 + 2 + 5 + 3 (-1) = 10$

Scalar multiplication

The angle of the vectors can be calculated as:

$\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{a \cdot b}$

Angle between two vectors

Vector or intersecting product

The third type of multiplication is vector or intersecting product. Once again, two vectors are multiplied with each other and the product is a vector. This vector stands vertically on top of both of the multiplied vectors, which is why the intersecting product only has meaning in a three dimensional space. It’s defined in the following illustration:
Chapter 7

The Third Dimension: 3-D Graphics Programming

Presenting 3-D Figures In 2-D

The goal of this section is to describe the calculations needed to display 3-D effects on a flat 2-D screen. There are many ways to do this, from the simplest parallel projection to complex ray tracing algorithms. However, due to the huge amount of calculations required to render the objects, complex ray tracing algorithms are used less frequently. These calculations can require several minutes of computing time just to produce a single picture.

The easiest way to convert the depth information the monitor cannot display is simply to ignore it. In so doing, the two dimensional coordinates of the corner points for each surface are given three dimensional definitions from the x- and y- coordinates. Parallel lines in the three dimensional space with this method appear as parallel lines on the screen — hence the name parallel projection.

In this illustration we see the three dimensional world as it exists behind the screen. All rays emitted from an object are ultimately seen by our eyes, so their origin lies in the picture. Every coordinate for the object (here is an example of the y-coordinate for the top and bottom corners) is projected into a two dimensional screen coordinate (here y'). We can explain this using the radiation theory. It establishes a clear relationship between y, y', a (eye-screen distance) and z (object depth).

You can see that increasing depth (z) with this algorithm reduces the angle between the rays as well as the picture on the screen. Therefore, the desired vanishing point perspective is achieved. This vanishing point (the point where all the parallel lines join in the distance) can always be found in this method on the z-axis. It cannot be placed arbitrarily.
Reshaping Objects: Transformations

You can create a more realistic picture by adding movement. A rotating cube with a picture on each side is more visually exciting than a static picture of the same cube created by a paint program.

Movement basically consists of two components (although some programmers would also include scaling):

- Translation
- Rotation

A translation is simply shifting in a certain direction, for example, motion through a long corridor. Mathematically, the translation is built upon vector addition: The shifting is determined by a vector, specifically the translation vector. This vector is simply added to all location vectors (point coordinates) of the object to be shifted.

Motion by the viewer is achieved in the same manner, but the viewpoint is reversed. If you wish to move yourself, as the viewer, around a unit in the z direction, you simply shift the entire 3-D environment around the unit in the negative z direction.

Rotation is slightly more complicated, especially if you want to use matrices in the calculation. Matrices combine all the necessary computation steps for transformations, translations, rotations and scaling into one. Each component has its own matrix. The advantage here is that you can put together additional matrices and save on computing time (if it is initially clear which transformations are to be performed in which series). However, since that is rarely the case, we are suggesting the rotation matrices only to give you a better alternative.

When considering rotation, you must determine around which of the three axes you will rotate.

For rotations around the x-axis, the following algorithm applies:

\[
\begin{align*}
x' &= x \\
y' &= y \cos(a) - z \sin(a) \\
z' &= y \sin(a) + z \cos(a)
\end{align*}
\]

The corresponding matrix would appear as follows:

\[
\begin{array}{ccc}
1 & 0 & 0 \\
0 & \cos(a) & -\sin(a) \\
0 & \sin(a) & \cos(a)
\end{array}
\]

For rotations around the y-axis, the following algorithm applies:

\[
\begin{align*}
x' &= x \cos(a) + z \sin(a) \\
y' &= y \\
z' &= -x \sin(a) + z \cos(a)
\end{align*}
\]

The corresponding matrix would appear as follows:
These formulas examine only the rotation around the coordinate axes. By combining many rotations, an axis can be formed from the original extended straight lines. If you wish to bypass the limits of the original straight lines, you must combine the rotation with a translation.

In doing so, the world is pushed so the point around which the rotation occurs is located at its original position. After the rotation, they are then pushed back, for which the translation vector must be rotated in advance.

With continual rotation around many axes, the coordinates from the previous rotation must understandably be set as source coordinates in the ones that follow. Always using the actual world coordinates for your source will give unusual results that have little in common with the actual world to be presented.

Because of its many sine and cosine calculations, you can predetermine the values used in the rotation and store them in a table. Using a program language such as Pascal to perform these calculations in real time would probably produce a jerky, non-fluid movement.

These transformations are not commutative, i.e., their order is not arbitrary. For example, if you rotated the point lying on the x-axis (1/0/0) 90 degrees around the x-axis, then around the same angle at the z-axis, the result would be on the y-axis. By changing the order, it would then be on the z-axis. Therefore, you should establish a unified rotation order first around the x-, then the y- and finally around the z-axes.

Even with mixed transformations, their order must be maintained. With a translation that has rotation following it, another point emerges that is different from that which emerges in the opposite series order, which is why you should commit to a particular order. In this case, the most practical translation series is rotation, because translation values are based on the known world coordinates and not on their rotating illustrations.

In summary, the following is the recommended order:

1. Translation
2. Rotation (x, y, then z)
3. Projection onto the screen

**Wiry Figures: Wireframe Modeling**

The easiest way to get three dimensional objects onto the screen is to use wireframe models. A wireframe model represents all surfaces of a 3-D object as an outlined object. This includes the opposite sides and all internal components that you usually cannot see. It’s a less complex method for representing 3-D images.

A wireframe model is useful because three dimensional lines appear as straight lines and not as curves with both parallel projections and the vanishing point perspective. Therefore, you can limit yourself to corner
point transformations and you will not need to push, rotate and illustrate every point to the edge. If you then combine these calculated corner points, you have a realistic picture of the figure as a wireframe model.

The most important section of this model is the line drawing algorithms. The speed of display depends on these line algorithms since the transformations themselves require hardly any computational time. The fastest way to draw a line is with the Bresenham algorithm, which we'll use in this chapter.

You can find complete mathematical derivation of this algorithm in many books so we won't delve into its principles.

The algorithm is limited to angles between 0 and 45 degrees. As the line is being drawn, the algorithm decides whether each point should be placed exactly to the right of the previous point or directly above it. The decision as to which of the two points will be next is similar to the fixed decimal procedure discussed. A variable (Dist, stored in BP) is dependent on the last step (right or directly above), either raised to Add_1 (in SI) or to Add_2 (in DI) and dependent on whether Dist is positive or negative.

The angle limitations can be removed; to handle angles between 45 and 90 degrees, you can switch x and y. Negative angles are handled by reversing this direction. The exact procedure is listed as follows:

```
.data segment
extrn vpage:word              ;current video page

putpixel macro                  ;puts pixel at ax/bx
    pusha
    xchg ax,bx                    ;exchange x and y
    push ax                       ;store y for later
    mov cx,bx                     ;get x
    and cx,3                      ;mask plane
    mov ax,1                      ;and set corresp. bit
    shl ax,cl
    mov ah,2                      ;TS register 2
    xchg ah,al
    mov dx,3c4h
    out dx,ax
    pop cx                        ;get y
    mov ax,80d                    ;calculate row offset
    mul cx
    shr bx,2                      ;add column offset
    add bx,ax
    add bx,vpage                  ;write to current page
    mov b es:[bx],3               ;and set color
    popa
endm

.code segment public
assume cs:code,ds:data

bline proc    near
    ;draws line from ax/bx to cx/dx
    push bp
    push ax                       ;store x0 and
```
Chapter 7

The Third Dimension: 3-D Graphics Programming

push bx                      ;y0
mov bx,4340h                 ;prepare self modification
sub cx,ax                    ;calculate deltax
jns deltax_ok                ;negative?
seg cx                       ;yes, then reverse deltax sign
mov bl,48h                   ;and decrement ax instead of incrementing ax

deltax_ok:
  mov bp,sp                   ;addressing of y1 on the stack
  sub dx,ss:[bp]              ;calculate deltay
  jns deltay_ok               ;negative?
  neg dx                      ;yes, then reverse deltay sign
  mov bh,4bh                  ;and decrement bx instead of incrementing bx

  deltay_ok:
  mov si,dx                   ;deltay and
  or si,cx                     ;deltax = 0 ?
  jne ok
  add sp,6                   ;then ax, bx and bp from stack and end
  ret

ok:                         ;otherwise increment bx nopenpen
  mov w cs:dist_pos,bx       ;write dec/inc ax/bx to destination
  cmp cx,dx                  ;deltax >= deltay ?
  jge deltax_great
  xchg cx,dx                  ;no, then exchange deltax and deltay
  mov bl,90h                  ;and increment ax nopenpen
  jmp constants

deltax_great:
  mov bh,90h                  ;otherwise increment bx nopenpen
  mov w cs:dist_neg,bx       ;write dec/inc ax/bx to destination
  shl dx,1                    ;define add_2
  mov di,dx                    ;store in di
  sub dx,cx                    ;define start-dist
  mov bp,dx                    ;and store in bp
  mov si,bp                    ;define add_1
  sub si,cx                    ;and store in si
  mov ax,0a000h                ;load VGA segment
  mov es,ax
  pop bx                       ;retrieve stored values for x0 and y0
  pop ax
  loop_p:                      ;set pixel
    putpixel                   ;dist positive?
    or bp,bp
    jns dist_pos
  dist_neg:
    inc ax                     ;increment x (if necessary, self modification)
    inc bx                     ;increment y (if necessary, self modification)
    add bp,di
    loop loop_p
  jmp finished

  dist_pos:
    inc ax                     ;increment x (if necessary, self modification)
    inc bx                     ;increment y (if necessary, self modification)
    add bp,si
    loop loop_p
  jmp finished

  finished:                    ;next pixel
    pop bp
    ret
bline endp
  code ends
This procedure draws a line in Mode X from the point at coordinates \((ax/bx)\) to point \((cx/dx)\). The current page (offset in vpage) is considered.

The step into the third dimension next takes us to unit VAR_3D.PAS, which contains the most important global variables. The meaning of those global variables will become apparent in the following sections:

```pascal
Unit Var_3d;
Interface
Uses Tools;
Const
Txt_Number=5;             {number of used textures}
Txt_Size:                 {size specifications of textures}
Array[0..Txt_Number-1] of Word=($0a0a,$0a0a,$0a0a,$0a0a,$0a0a);

Var
vz:Word;                   {shifting into the screen}
rotx,                       {angle of rotation}
roty,
rotz:word;                  {3 degree steps}
sf_sort:Boolean;            {sort surfaces ?}
Fill:Boolean;               {true: Fill / false:Lines}
sf_shift:Boolean;           {suppress surface shift ?}
Texture:Boolean;            {use textures ?}
lightsrc:Boolean;           {use light source ?}
Glass:Boolean;              {glass surface ?}
Txt_Data:Array[0..Txt_Number-1] of Pointer;  {location of textures in memory}
Txt_Offs:Array[0..Txt_Number-1] of Word;      {offset within the texture picture}
Txt_Pic:Pointer;            {pointer to texture picture}
Sine:Array[0..149] of Word;  {sine table for rotations}

Implementation
Begin
  Sin_Gen(Sine,120,16384,0);
  Move(Sine[0],Sine[120],60);
End.
```

This unit is used by all our 3-D programs. It initializes the sine tables used for the rotations. Subsequently, the first quarter (30 entries = 60 byte) of this table is appended to the end. This has the advantage that both its sine (zero in sine[0]) and its cosine values (zero in sine[30]) can be inferred.

The Bresenham algorithm is used in the 3D_WIRE.PAS program, which directs the assembly module to clear the global variable filling, to draw wireframe models:

```pascal
Uses Crt,ModeXLib,var_3d;
Const
worldlen=8*3;              {Point-Array}
Worldconst:Array[0..worldlen-1] of Integer =
  (-200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
  -200,-200,-200,
```

169
200,-200,200,
200,200,-200,
200,200,200);                {Surface-Array}
surfconst:Array[0..surfclen-1] of Word=
{0,4, 0,2,6,4,
 0,4, 0,1,3,2,
 0,4, 4,6,7,5,
 0,4, 1,5,7,3,
 0,4, 2,3,7,6,
 0,4, 0,4,5,1,0,0);              {Surface-Array}

Var
  i,j:Word;
procedure drawworld;external;   {Draws the world on current video page}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l root.obj}
Begin
  vz:=1000;                     {solid is located at 1000 units depth}
  vpage:=0;                     {start with page 0}
  init_modex;                   {enable ModeX}
  rotx:=0;                      {initial values for rotation}
  roty:=0;
  rotz:=0;
  Fill:=false;                  {SurfaceFill off}
  sf_sort:=false;               {SurfaceSort off}
  sf_shift:=false;              {SurfaceShift suppression off}
  Glass:=false;                 {glass surfaces off}
  repeat
    clrx($0f);                  {clear screen}
    DrawWorld;                  {draw world}
    switch;                     {switch to finished picture}
    WaitRetrace;                {wait for next retrace}
    Inc(rotx);                  {continue rotating ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    Inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;             { ... until key}
  TextMode(3);                 { ... until key}
End.

You’ll find the definitions for the world and the surfaces at the start of this program and in the following
programs. The world includes only the coordinates for the corners, which are simply listed in the x, y, z
series order for every given point. A relationship between these points is first established by defining the
surface. The characteristics of each surface are established in this array.

The first word for every surface definition includes the top surface structure. We will talk more about top
surface structure in later sections but it’s simply listed here as 0. This is followed by the number of the corner
points of this surface and the numbers of the corner points themselves, whereby their numbering starts with
0.
The first square consists of the corner points 0,2,6,4, which corresponds to the coordinates (-200/-200/-200), (-200,200,-200), (200,200,-200), (200,-200,-200). The definition of the top surface is complete by specifying two consecutive zero words, otherwise it will not be recognized by the procedure.

In the main program, the global depth \( v_z \) of the object is then set (placed). Reduce this value to zoom closer. The variables \( \text{rotx} \), \( \text{roty} \) and \( \text{rotz} \) indicate the object's rotation angle in steps of 3 degrees; i.e., a rotation of 15 degrees is established with a value of 5.

The choice of graphic mode is assigned to Mode X. This mode has a few disadvantages relating to the speed at which it can address individual pixel but its most important advantage is its two screen pages. This is important with the textures, because here the painting of the screen takes longer than a retrace period.

The global directional variables are then set. In this instance, the surface (filling), the sorting (fl_sort), the hiding of the reverse side (fl_backs) and the glass top surfaces (glass) are turned off.

The loop that follows is the same for all the programs and follows this pattern if no key is pressed:

1. The screen is cleared every time.
2. The world is drawn.
3. It's switched over to the new screen page.
4. It's rotated further.

The main function of this program takes over the module 3DASM.ASM, which is written in assembly language for faster execution. Instead of listing the source code in individual sections, we've listed the entire source code at one time:

```
.286
w equ word ptr
b equ byte ptr
surfclen equ 200               ;maximum length of surface defined
Pointlen equ 4*100              ;length of point array
num_ar equ 30                  ;maximum number of areas
num_cor equ 10                 ;maximum number of corners
data segment                    ;external variables from Pascal segment
extrn vz:word                  ;total depth
extrn rotx:Word                ;angle of rotation
extrn roty:Word
extrn rotz:word
extrn worldconst:dataptr       ;array with points
extrn surfconst:dataptr         ;array with surface definitions
extrn lightsrc:word             ;flag for light source shading
extrn sf_sort:word              ;flag for surface sorting
extrn sf_shift:word             ;flag for surface shift suppression
extrn Texture:Byte              ;flag for textures
extrn Fill:Byte                 ;flag for fill / wireframe model
crotx dw 0                      ;x, y and z angle as offset to
croty dw 0
rotz dw 0                      ;specific sine value
rotx_x dw 0                     ;x,y,z to x-rot
rotx_y dw 0
rotz_x dw 0                     ;to y-rot
rotz_y dw 0
```

You can find 3DASM.ASM on the companion CD-ROM
The Third Dimension: 3-D Graphics Programming

Chapter 7

roty_z dw 0
rots_x dw 0 ; to z-rot, final
rots_y dw 0
rots_z dw 0
startpoly dw 0 ; start of definition of current area
Point dw Pointlen dup (0); receives calculated coordinates
Pointptr dw 0 ; pointer in Point-Array
Point3d dw Pointlen dup (0); receives completed 3D-coordinates (texture)
mean dw num_ar*2 dup (0); list of mean z-values
meanptr dw 0 ; pointer in Mean-Array
n dw 0,0,0,0,0,0 ; normal vector 32 Bit
n_amnt dw 0 ; amount of normal vector
extrn sine:dataptr
data ends
extrn drawpol:near ; draws area as wireframe model
extrn fillpol:near ; fills area
extrn root:near ; calculates root of ax
getdelta macro ; calculates the two surface vectors
mov ax,poly3d[0] ; x: original corner
mov delta2[0],ax ; store temporarily in delta2
sub ax,poly3d[8] ; obtain difference to first point
mov delta1[0],ax ; and delta1 finished
mov ax,poly3d[2] ; y: original corner
mov delta2[2],ax ; store temporarily in delta2
sub ax,poly3d[10] ; obtain difference to first point
mov delta1[2],ax ; and delta1 finished
mov ax,poly3d[4] ; z: original corner
mov delta2[4],ax ; store temporarily in delta2
sub ax,poly3d[12] ; obtain difference to first point
mov delta1[4],ax ; and delta1 finished
mov bp,polyn ; select last point
dec bp
shl bp,3 ; 8 bytes at a time
mov ax,poly3d[bp]
sub delta2[0],ax ; obtain difference
mov ax,poly3d[bp+2]
sub delta2[2],ax
mov ax,poly3d[bp+4]
sub delta2[4],ax
endm
setcoord macro source,offst ; sets calculated screencoord
mov ax,source ; project coordinate
cwd
shld dx,ax,7
shl ax,7
idiv cx
add ax,offst ; middle of screen is 0/0/0
mov bx,Pointptr
mov Point[bx],ax
add Pointptr,2
endm
z2cx macro tabofs ; moves z-coordinate to cx
mov cx,tabofs + 4
add cx,vz
mov bx,meanptr
add mean[bx],cx
endm
xrot macro zcoord,qcoord ; rotates qcoord by x, stores in zcoord
mov ax,qcoord
add ax,x
add ax,offst
mov bx,Pointptr
mov Point[bx],ax
add Pointptr,2
endm
Chapter 7

The Third Dimension: 3-D Graphics Programming

```assembly
mov bx, [qcoord]          ; x8, to align to point entries
shl bx, 1
mov Pointptr, bx
sub bx, [qcoord]         ; insg. x6, to align to world entries
sub bx, [qcoord]
add bx, offset worldconst ; set to world
mov ax, [bx]             ; get x
mov zcoord, ax          ; and set unchanged
mov ax, [bx+2]           ; get y
imul w ds:[bp+60d]       ; *cos rotx
shrd ax, dx, 14d
mov cx, ax              ; store in cx
mov ax, [bx+4]           ; get z
imul w ds:[bp]           ; *-sin rotx
shrd ax, dx, 14d
sub cx, ax
mov zcoord+2, cx        ; y value finished and set
mov ax, [bx+2]           ; get y
imul w ds:[bp]           ; *sin rotx
shrd ax, dx, 14d
mov cx, ax              ; store in cx
mov ax, [bx+4]           ; get z
imul w ds:[bp+60d]       ; *cos rotx
shrd ax, dx, 14d
add cx, ax
mov zcoord+4, cx

yrot macro zcoord, qcoord ; rotates qcoord by y, stores in zcoord
    mov bp, croty          ; get angle
    mov ax, qcoord+2       ; get y
    mov zcoord+2, ax       ; and set unchanged
    imul w ds:[bp]         ; *cos roty
    shrd ax, dx, 14d
    mov cx, ax            ; store in cx
    mov ax, qcoord        ; get x
    imul w ds:[bp]         ; *-sin roty
    shrd ax, dx, 14d
    add cx, ax            ; x value finished and set
    mov ax, qcoord        ; get x
    imul w ds:[bp]         ; *sin roty
    shrd ax, dx, 14d
    mov cx, ax            ; store in cx
    mov ax, qcoord+4       ; get z
    imul w ds:[bp+60d]     ; *cos roty
    shrd ax, dx, 14d
    sub ax, cx
    mov zcoord+4, ax
endm

zrot macro zcoord, qcoord ; rotates qcoord by z, saves in zcoord
    mov bx, Pointptr       ; prepare entry in 3D-Point-Array
    mov ax, crotz          ; get angle
    mov ax, qcoord+4       ; get z
    mov ax, qcoord+4, ax   ; and set unchanged
    mov Point3d[bx+4], ax ; also note in 3D-Array
    mov ax, qcoord        ; get x
    imul w ds:[bp+60d]     ; *cos rotz
    shrd ax, dx, 14d
endm
```
mov cx,ax ; store in cx
mov ax,qcoord+2 ; get y
imul w ds:[bp] ; *-sin rotz
shrd ax,dx,14d
sub cx,ax
mov zcoord,cx ; x value finished and set
mov Point3d[bx],cx
mov ax,qcoord ; get x
imul w ds:[bp] ; *sin rotz
shrd ax,dx,14d
mov cx,ax ; store in cx
mov ax,qcoord+2 ; get y
imul w ds:[bp+60d] ; *cos rotz
shrd ax,dx,14d
add cx,ax
mov zcoord+2,cx
mov Point3d[bx+2],cx
endm

get_normal macro ; calculates normal vector of an area
mov ax,delta1[2] ; a2*b3
imul delta2[4]
shrd ax,dx,4
mov n[0],ax
mov ax,delta1[4] ; a3*b2
imul delta2[2]
shrd ax,dx,4
sub n[0],ax
mov ax,delta1[4] ; a3*b1
imul delta2[0]
shrd ax,dx,4
mov n[2],ax
mov ax,delta1[0] ; a1*b3
imul delta2[4]
shrd ax,dx,4
sub n[2],ax
mov ax,delta1[0] ; a1*b2
imul delta2[2]
shrd ax,dx,4
mov n[4],ax
mov ax,delta1[2] ; a1*b2
imul delta2[0]
shrd ax,dx,4
sub n[4],ax ; cross product (=normal vector) finished
mov ax,n[0] ; x1^2
imul ax
mov bx,ax
mov cx,dx
mov ax,n[2] ; +x2^2
imul ax
add bx,ax
adc cx,dx
mov ax,n[4] ; +x3^2
imul ax
add ax,bx
adc dx,ax ; sum in dx:ax
push si
call root ; root in ax
pop si
mov n_amnt,ax ; amount of normal vector finished
endm
light macro ; determines brightness of an area
  mov ax,n[0]
imul l[0] ; light vector * normal vector
  mov bx,ax
  mov cx,dx
  mov ax,n[2]
imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
imul l[4]
  add ax,bx ; scalar product finished in dx:ax
  adc dx,cx
  mov bx,n[2] ; divide by l_amnt
  imul l[2] ; and by n_amnt
  cwd
  shld dx,ax,5 ; values from -32 bis +32
  shl ax,5d
  mov bp,startpoly ; prepare addressing of surface color
  idiv bx ; division by denominator
  inc ax
  or ax,ax
  js turned_toward ; if cos à positive -> turned away from the light
  xor ax,ax ; thus, no light
  turned_toward:
  sub b polycol,al ; cos<0 -> add to primary color
  endm

code segment
assume cs:code,ds:data
public drawworld
public linecount
public polycol
public polyn
public poly2d
public poly3d
linecount dw 0
polycol dw 3 ; current surface color
polyn dw 0 ; number of existing corners
poly2d dw num_cor*4 dup (0) ; corners of polygon to be drawn
poly3d dw num_cor*4 dup (0) ; 3D corners
public Txt_No
Txt_No dw 0 ; current texture number
delta1 dw 0,0,0 ; plane vectors
delta2 dw 0,0,0
l dw 11d,11d,11d ; light vector
l_amnt dw 19d ; amount of light vector
drawworld proc pascal ; draws three-dimensional world
push ds
push es
push bp
lea si,surfconst ; surfaces are addressed by si
mov meanptr,0 ; start in Mean-Array with 0
mov ax,ds:[rotx] ; get angle,
shl ax,1 ; convert as memory offset
add ax,offset sine
mov crotx,ax ; and store in help variables
mov ax,ds:[roty] ; exactly the same for y
shl ax,1
add ax,offset sine
mov croty,ax
The Third Dimension: 3-D Graphics Programming

Chapter 7

PC underground

mov ax,ds:[rotz] ;and z
shr ax,1
add ax,offset sine
mov crotz,ax

npoly:
    ;polygon loop
    mov startpoly,si ;store for later use
    add si,2 ;skip color
    mov cx,[si] ;get number of corners
    inc cx ;due to closed area
    mov w polyn,cx ;enter in Point-Array
    add si,2 ;move to actual coordinates
    nline: ;polygon loop
        mov startpoly,si ;store for later use
        add si,2 ;skip color
        mov cx,[si] ;get number of corners
        inc cx ;due to closed area
        mov w polyn,cx ;enter in Point-Array
        add si,2 ;move to actual coordinates
        nline: ;polygon loop
            mov ax,ds:[si] ;get color and set
            mov texture,0 ;Assumption: no texture
            cmp ah,0ffh ;texture ?
            jne no_texture
            mov texture,1 ;yes, then set
            mov b txt_no,al ;note number
            mov bx,meanptr ;calculate mean value:
            mov ax,mean[bx] ;get sum
            mov cx,polyn
            dec cx
            cwd
            div cx ;and divide by number of corners
            mov mean[bx],ax ;write back
            mov ax,startpoly ;write *number* of area also
            mov mean[bx+2],ax
            add meanptr,4 ;continue
            cmp w [si+2],0 ;polygons all finished ?
            je finished
            jmp npoly_save
        polyok: ;calculate mean value:
            mov ax,mean[bx] ;get sum
            mov cx,polyn
            dec cx
            cwd
            div cx ;and divide by number of corners
            mov mean[bx],ax ;write back
            mov ax,startpoly ;write *number* of area also
            mov mean[bx+2],ax
            add meanptr,4 ;continue
            cmp w [si+2],0 ;polygons all finished ?
            je finished
            jmp npoly_save
        finished:
            cmp b sf_sort,0 ;sort surfaces ?
            je no_quicksort
            call quicksort pascal,0,bx ;sort field from 0 to current position
        no_quicksort:
            mov mean[bx+4],0 ;set termination
            mov ax,cs ;set destination segment
            mov es,ax
            xor bx,bx ;start with first surface
            npoly_draw:
                lea di,poly2d ;destination:Poly-Array
                mov bp,mean[bx+2] ;get pointer to color and points of surface
                mov ax,ds:[bp] ;get color and set
                mov polycol,ax
                mov texture,0 ;Assumption: no texture
                cmp ah,0ffh ;texture ?
                jne no_texture
                mov texture,1 ;yes, then set
                mov b txt_no,al ;note number
            npoly_draw:
no_texture:
  mov b lightsrc,0              ;Assumption: no shading
  cmp ah,0feh                   ;shading ?
  jne no_lightsource
  mov b lightsrc,1              ;yes, then set

no_lightsource:
  add bp,2                      ;position on number
  mov cx,ds:[bp]                ;get number of corners
  mov polyn,cx                  ;write in Poly-Array

npoint:
  add bp,2                      ;get pointer to actual point
  shl si,3                      ;3 word entry !
  add si,offset Point           ;and x/y from Point-Array to Poly-Coord.
  mov ax,[si+Point3d-poly2d],ax ;set 3d-x
  mov ax,[si+Point3d-poly2d+2],ax;set 3d-y
  mov ax,[si+Point3d-poly2d+4],ax;set 3d-z
  movsw                         ;set 2D-coordinates
  movsw
  add di,4                      ;next Poly2d entry
  dec cx                        ;all corners ?
  jne npoint
  mov bp,polyn                  ;copy first corner to last
  shl bp,3                      ;position on first point
  neg bp
  mov ax,es:[di+bp]             ;and copy
  mov ax,es:[di+bp+2]
  mov ax,es:[di+bp+4]
  add di,poly3d-poly2d          ;the same for 3d-coordinates
  mov ax,es:[di+bp]
  mov ax,es:[di+bp+2]
  mov ax,es:[di+bp+4]
  mov ax,es:[di+4],ax
  cmp fill,1                    ;fill surface ?
  jne lines
  getdelta                      ;yes, then calculate Delta1 and 2
  cmp b lightsrc,0              ;light source ?
  jne shade
  jmp no_light

shade:
  push bx
  get_normal                    ;then normal vector
  light                         ;and calculate light
  pop bx

no_light:
  inc polyn                     ;increment number of corners
  call fillpol                  ;draw surface

next:
  add bx,4                      ;locate next surface
  cmp mean[bx],0                ;last ?
je _npoly_draw  ;no, then continue
jmp npoly_draw

lines:
push bx
call drawpol  ;draw polygon
pop bx
jmp next

_npoly_draw:
pop bp  ;finished
pop es
pop ds
ret
drawworld endp
public quicksort
quicksort proc pascal down,up:word
;sorts Mean-Array according to Quicksort algorithm

local key:word
local left:word
push bx
mov bx,down  ;find middle
add bx,up
shr bx,1
and bx,not 3  ;posit on blocks of 4
mov dx,mean[bx]  ;get key
mov key,dx
mov ax,down  ;initialize right and left with base values
mov si,ax
mov left,ax
mov ax,up
mov di,ax
mov dx,key

left_nearer:
cmp mean[si],dx  ;greater than key -> continue searching
jbe left_on
add si,4  ;posit on next one
jmp left_nearer  ;and check it

left_on:
cmp mean[di],dx  ;less than key -> continue searching
jae right_on
sub di,4  ;posit on next one
jmp left_on  ;and check it

right_on:
cmp si,di  ;left <= right ?
jg end_schl  ;no -> subarea sorted
mov eax,dword ptr mean[si]  ;exchange mean values and positions
xchg eax,dword ptr mean[di]
mov dword ptr mean[si],eax
add si,4  ;continue moving pointer
sub di,4
end_schl:
cmp si,di  ;left > right, then continue
jle left_nearer
mov left,si  ;store left, due to recursion
cmp down,di  ;down < right -> sort left subarea
jge right_finished
call quicksort pascal,down,di  ;continue sorting recursive halves
You've probably guessed the procedure `drawworld` is important simply by its length. Here, the complete three dimensional world is drawn, depending on the current transformation values. Parameters are not given since these are completely handled by global variables.

The current rotation angle (rotx, roty, rotz) is immediately converted to the offset relative to the data segment (crotx, crotty, crotz) so direct access is available later without requiring any further calculations. Within the loop `npoly` which is performed for every polygon, the SI-register is used for addressing within the surface array (surfconst).

Then, the color information is skipped and the numbers of corners for this polygon are read directly from the array. This value is saved in `LineCount`, which indicates the number of corners that still need to be calculated. The procedure next enters into the point loop, `nline`, which rotates the individual points one after another and projects them onto the screen coordinates. The rotations (macros `xrot`, `yrot`, `zrot`) respectively transmit the results to the one that follows, so the original coordinates are rotated one after another around the x-, y- and z-axes.

The structures of the three rotation macros is very similar. Since bp contains the angle in the offset form, the corresponding sine value can be easily read by indexing with bp from the chart. The cosine is correspondingly addressed by adding 60 (byte, i.e., 30 entries) to bp.

The number of the actual corner points is multiplied by 8 to access the three dimensional point array (with 8 byte long entries). This index is saved in variable `PointsPtr`.

Now on to the macros. For an x rotation, the x-coordinate remains unchanged. The y-and z-coordinates are changed as follows:

\[
y' = y \cdot \cos_ - z \cdot \sin_ \\
z' = y \cdot \sin_ + z \cdot \cos_.
\]

and

\[
\]

The other two macros are similar for the y- and z-coordinates. Macro `z2cx` loads the rotated z coordinates into the cx register.

Now the coordinates for the corner points are ready to be projected onto the screen. This is performed by macro `SetCoord`. It projects the coordinates listed in the source (one time `rotz_x` and one time `rotz_y`) and simultaneously moves the completed image to the middle of the screen where the offset (the second parameter) is added.

The `nline` loop is closed by advancing to the next corner point (increasing SI) and reducing the line counter.
When all the corner points have been calculated, the middle section of the polygon is calculated at label `polyok`. The wireframe model at the end of the `npoly` loop with the `cmp` instruction gets interesting. When the number of corners equals zero, the loop is ended and jumps to the label `complete`.

In later models, a surface sorter is introduced at this position. However, it's skipped here due to the setting of variable `fl_Sort`. Here, the termination is set in array `mean` by using a trailing zero. Each polygon is individually drawn in the `npoly_draw` loop. Array `Poly2` is used to transfer the coordinates to routine `DrawPoly`.

The polygon identification is retrieved from array `mean` every time, as is the color information retrieved from the `SurfConst` field and saved in `polycol`. The two instruction blocks that follow are for activating the texture and light source routines, which, in this case, are not yet required. The reading of the number of corners proceeds from the array `mean` with the label `no_light_source`. It's then saved in variable `polyn`. The `npoint` loop follows and is performed for every corner point. It then reads the number of corner points and addresses the two dimensional corner coordinates in the points array using the loop. Both of the `movsw` commands then copy the coordinates from this array into the `Poly2D` field, where the DI counter has been increased. The loop then ends.

Now the first corner must be copied to the last one to maintain a closed line. The following trick is used for this: Multiply the number of corners (`polyn`) by 8 (number of bytes per corner) to determine the total length of the array `Poly2D` in byte. This length is then subtracted by sorting out from the current position (hence `neg bp`) and the word is copied in here.

Since this program uses wireframe models, lines branch out from the label, which procedure `drawpol` calls from the module POLY.ASM and, in this manner, draws the polygon to the screen as a wireframe model. Afterwards, the next polygon in array `mean` is addressed at label `next` and jumps to the beginning of the loop `npoly_Draw`.

The polygon is drawn in the POLY.ASM module, in this case by the procedure `drawpol`:

```assembly
public drawpol
; draws wireframe model of surface in poly2d
drawpol proc near
push es
pusha
xor si,si ; index to first entry
mov bp,polyn ; get number of corners
@nline:
    mov ax,poly2d[si] ; get coordinates from table
    mov bx,poly2d[si+2]
    mov cx,poly2d[si+8]
    mov dx,poly2d[si+10d]
    push bp
    push si
    call bline ; draw line
    pop si
    pop bp
    add si,8 ; next line
    dec bp ; decrement number
    jne @nline
    popa
    pop es
    ret
drawpol endp
```

The `drawpol` procedure is part of the POLY.ASM module on the companion CD-ROM.
We are obviously using the simpler presentation procedure here. Starting and ending coordinates are read for every point from the array `Poly2D` and the procedure `bline` is called. `si` serves as an index in the array and `bp` is used for purposes other than originally intended, i.e., as a counter for the number of lines to be drawn.

## Get A Perspective: Glass Figures

The objects in the previous section consist primarily of only sides. Therefore, they don’t resemble real objects. The next step, then, is to add surfaces to these objects. In doing this, you’ll encounter one of the biggest problems in 3-D: Hidden surfaces.

If you simply erase all the surfaces that are behind each other, as they are defined, often surfaces will appear which are normally invisible. We’ll talk about hiding these surfaces from a different angle in this section.

Instead of adapting the illustrations to reality, we will first adapt reality to the illustration. We’ll start with a glass figure where all side surfaces are always visible. Nevertheless, the sides can be in color. In fact, they must be in color so we can create a picture.

If only two surfaces lie back to back, their colors are superimposed on one other and a somewhat darker blended color results (two surfaces filter out more light than one surface).

In principle, every possible surface combination must be considered, i.e., if surface A is superimposed over surface B at any angle, a blended color for the two surfaces must exist. The only way to allow for this is to reserve a bit in the color information for each surface. So, only surfaces that cannot be superimposed under any circumstances can use the same bit because blending is not possible.

When preparing the surface, the old color is not transferred, but both values are joined using OR, producing a new color value. For example, if surface A has the color 2 (bit 1 is set) and surface B has the color 16 (bit 4), this combination results in the color 18 (bit 1 and 4 set).

Naturally, the palettes for these special structures must be included. The pure colors must be maintained and every bit combination is provided with a blended color from the respective bit. The named color 18 must be, in this case, a blend of colors 2 and 16.

The palette must then be prepared at the beginning of the program. You will use the existing VGA arithmetic unit when filling the polygons. This can be switched (or operated) using GDC register 3 in the OR mode. It joins the incoming CPU data with the OR values located in the latches before they are written to screen memory. Before write access can occur, the latches must be loaded with the values from the screen memory, in which read access is given at the same memory position.

The following program, 3D_GLASS.PAS, is also based on the 3DASM.ASM model. Unlike the previous section, different variables have been set:
Chapter 7

The Third Dimension: 3-D Graphics Programming

Uses Crt,ModeXLib,var_3d;

Const

worldlen=8*3;               {Point-Array}
Worldconst:Array[0..worldlen-1] of Integer =
(-200,-200,-200,
-200,-200,200,
-200,200,-200,
-200,200,200,
200,-200,-200,
200,200,-200,
200,200,200);
surfclen=38;                {Surface-Array}
surfcconst:Array[0..surfclen-1] of Word=
(01, 4, 0,2,6,4,
 02, 4, 0,1,3,2,
 04, 4, 4,6,7,5,
 08, 4, 1,5,7,3,
 16, 4, 2,3,7,6,
 32, 4, 0,4,5,1,0,0);

Var

i,j:Word;

Procedure Glass_Pal;
{prepares the palette for glass solids}
Begin
FillChar(Palette[3],765,63);  {first all colors white}
For i:=1 to 255 do Begin      {define 255 mixed colors}
  If i and 1 = 1 Then Dec(Palette[i*3],16);
  If i and 2 = 2 Then Dec(Palette[i*3+1],16);
  If i and 4 = 4 Then Begin
    Dec(Palette[i*3],16);
    Dec(Palette[i*3+1],16);
    End;
  If i and 8 = 8 Then Begin
    Dec(Palette[i*3+2],16);
    Dec(Palette[i*3+1],16);
    End;
  If i and 16 = 16 Then Begin
    Dec(Palette[i*3],16);
    Dec(Palette[i*3+2],16);
    End;
  If i and 32 = 32 Then Begin
    Dec(Palette[i*3+1],16);
    Dec(Palette[i*3+2],16);
    End;
End;
SetPal;
End;

procedure drawworld;external;   {draws the world on current video page}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l root.obj}
Begin
vz:=1000;                     {solid is located at 1000 units depth}
vpage:=0;                      {start with page 0}
init_modeX;                   {enable ModeX}
Glass_Pal;
rotx:=0;                      {initial values for rotation}
roty:=0;
rotz:=0;
Fill:=true;                   {SurfaceFill on}
sf_sort:=false;               {SurfaceSort off}
sf_shift:=false;              {SurfaceShift suppression off}
Glass:=true;                  {glass surfaces on}
repeat
  clrx($0f);                  {clear screen}
  DrawWorld;                  {draw world}
  switch;                     {switch to finished picture}
  WaitRetrace;                {wait for next retrace}
  Inc(rotx);                  {continue rotating ... }
  If rotx=120 Then rotx:=0;
  Inc(rotz);
  If rotz=120 Then rotz:=0;
  inc(roty);
  If roty=120 Then roty:=0;
  Until KeyPressed;             { ... until key}
  TextMode(3);
End.

The surface filling algorithm is turned on by setting the variable Filling. Otherwise, Glass would be set to TRUE, whereby the GDC would be switched to the OR mode. Procedure Glass_Pal is called before the main program (discussed in the previous section). It prepares the palette for the glass figures.

For every color in which bit 0 is set, the red portion is reduced to 16. All blended colors are created in this manner when the color 1 is involved. The same is true with other 5 bit experiences, which filter out other colors respectively.

The remaining POLY.ASM modules are considered at this point. After the polygon fill algorithm, a colorful surface is created with the coordinates found in Poly2D.

There are two basic categories of fill algorithms:

- **General filling**
  This arbitrarily fills pre-drawn surfaces and is often used in paint programs

- **Coordinate filling**
  This uses coordinates to fill defined polygons. This algorithm is the fastest for our purposes.

The method described here is based on drawing of lines. This is the reason why, starting with the point having the lowest y-coordinate, the left and right edges of the polygon are keyed until the point with the largest y-coordinate is reached. As this happens, the border lines of the polygon are calculated but not drawn. If you advance a line in this fashion on both sides, a horizontal line can be drawn between the points calculated on the left and right points. You can take advantage there by being able to draw horizontal lines in Mode X at a very high rate of speed.

A filling routine must constantly calculate lines on the left as well as on the right margins of the polygon. If a line has been completely "drawn," the next one that has the last corner point as its starting point is started.

Procedure FillPol shows how this theoretical statement can be put into practice. Besides the procedure DrawPol, the complete code for the POLY.ASM module is used by the fill algorithm:
.286
w equ word ptr
b equ byte ptr
include texture.inc ;implement texture macros
setnewlinel macro ;use only ax and bx here !
local dylpos,dxlpos,dxlgreat,macro_finished
mov bx,4043h ;code for inc ax {in bh} and inc bx {in bl}
mov bp,left
mov ax,poly2d[bp+8] ;store destination coordinates
mov xl1,ax
mov ax,poly2d[bp+10d]
mov yl1,ax
mov ax,poly2d[bp] ;left x/y start in glob. var
mov xl0,ax
sub ax,xl1 ;make delta x
inc xl1 ;for the condition of truncation
neg ax
jns dxlpos ;dxl negative ?
neg ax ;then obtain amount
mov bh,48h ;code for dec ax (dec xl0)
sub xl1,2 ;extension of destination coordinate to negative
dxlgreat:
mov dyl,ax ;and store glob.
mov dxl,ax ;and store in increment flag
mov ax,poly2d[bp+2]
mov yl0,ax
sub ax,yl1 ;obtain |delta y|
inc yl1 ;for the condition of truncation
neg ax
jns dylpos ;negative ?
neg ax ;then obtain amount
mov bl,4bh ;code for dec bx (dec yl1)
sub yl1,2 ;extension of destination coordinate to negative
dylpos: ;extension of destination coordinate to negative
mov dyl,ax ;store in increment flag
cmp dxl,ax ;dx < dy
jae dxlgreat ;then sign change for increment flag
mov cs:byte ptr incxl,bh ;perform self modification
mov cs:byte ptr incyl,bl
cmp texture,1 ;textures required ?
jne macro_finished ;no, then skip
txt_makevarl ;otherwise calculate texture variables
macro_finished:
mov ax,xl0 ;use register as control variable
mov bx,yl0
mov si,incflagl
endm
setnewliner macro ;use only cx and dx here !
local dypos,dxpos,dxrgreat,macro_finished
mov cx,4142h ;code for inc cx {in ch} and inc dx {in cl}
mov bp,right
mov dx,poly2d[bp] ;get destination coordinates
mov dx,poly2d[bp+2]
mov dx,poly2d[bp+8] ;right x/y in glob. var
mov xr0,dx
sub dx,xr1 ;make |delta x|
ic xr1 ;for the condition of truncation
The Third Dimension: 3-D Graphics Programming

Chapter 7

PC underground

neg dx
jns dxrpos ;negative ?
neg dx ;then obtain amount
mov ch,49h ;code for dec cx
sub xr1,2 ;extension of destination coordinate to negative
dxrpos:
mov dxr,dx ;store in glob. var
mov incflagr,dx
mov dx,poly2d[bp+10d] ;make |delta y|
mov yr0,dx
sub dx,yr1 ;for the condition of truncation
inc yr1
neg dx
jns dxrpos ;negative ?
neg dx ;then obtain amount
mov cl,4ah ;code for dec dx
sub yr1,2 ;extension of destination coordinate to negative
dyrpos:
mov dyr,dx ;and store in glob. var
cmp dxr,dx ;dx < dy ?
jae dxrgreat ;then sign change for increment flag
dxrgreat:
mov cs:byte ptr incxr,ch ;self modification
mov cs:byte ptr incyr,cl
cmp texture,1 ;textures needed ?
jne macroFinished ;no, then skip
text_makevarr ;otherwise calculate texture variables
macro_finished:
mov cx,xr0 ;load register
mov dx,yr0
mov di,incflagr
endm

data segment public
extrn vpage:word ;current video page
extrn sf_shift :byte ;flag for surface shift suppression
extrn glass:byte ;flag for glass surfaces
;texture variables:
extrn texture:byte ;texture needed ?
extrn txt_data:dataptr ;array with pointer to graphic data
extrn txt_offs:dataptr ;array with offsets within the texture image
extrn txt_size:dataptr ;array with size specifications

d_x dd 0 ;relative x-coordinate
d_y dd 0 ;relative y-coordinate
D dd 0 ;main determinant
column1 dd 0 ;components of the main determinant
dd 0
column2 dd 0
dd 0
upper_row dw 0 ;which coordinates were used ?
lower_row dw 0
xl_3d dd 0 ;control values for 3d-coordinates when filling
yl_3d dd 0
zl_3d dd 0
xr_3d dd 0
yr_3d dd 0
zr_3d dd 0
inc_xl dd 0 ;values for addition to control values
Chapter 7
The Third Dimension: 3-D Graphics Programming

inc_y1 dd 0
inc_z1 dd 0
inc_xr dd 0
inc_yr dd 0
inc_zr dd 0

;variables for fill algorithm
high_point dw 0             ;kept in dx during search
high_y dw 0                  ;kept in bx during search
left dw 0                    ;point of left side
right dw 0                   ;point of right side
xl0 dw 0                     ;control values for left start and end points
yl0 dw 0
xl1 dw 0
yl1 dw 0
xr0 dw 0                     ;control values for right
yr0 dw 0
xr1 dw 0
yr1 dw 0
dxl dw 0                     ;delta X / Y for both pages
dyl dw 0
dxr dw 0
dyr dw 0
incflagl dw 0                ;flags, when y has to be incremented
incflagr dw 0

data ends

code segment public
assume cs:code,ds:data
extrn polycol:word            ;surface color
extrn poly:n:word             ;number of corners
extrn poly2d:word             ;array with 2D-coordinates
extrn poly3d:word             ;array with 3D-coordinates
extrn delta1,delta2:word      ;plane vectors
extrn bline:near              ;draws line
lambda1 dd 0                  ;affine coordinates
lambda2 dd 0
inc_lambda1 dd 0              ;steps
inc_lambda2 dd 0
plane dw 0002h                ;current plane to set
x0 dw 0                       ;coordinates for line
y0 dw 0
x1 dw 0
zz dw 0                       ;points still to be drawn
extrn txt_no:word             ;number of texture to be drawn

public drawpol
;draws wireframe model of surface in poly2d
drawpol proc near
    push es
    pusha
    xor si,si                     ;index to first entry
    mov bp,polyn                  ;get number of corners
    @nline:
        mov ax,poly2d[si]           ;get coordinates from table
        mov bx,poly2d[si+2]
        mov cx,poly2d[si+8]
    @nline:
mov dx, poly2d[si+10d]
push bp
push si
call bline ; draw line
pop si
pop bp
add si, 8 ; next line
dec bp ; decrement number
jne @nline
popa
pop es
ret
drawpol endp

hline proc near ; draws horiz. line ax/bx -> cx/bx
pusha
push es
mov x0, ax ; store coordinates for later
mov y0, bx
mov x1, cx
sub cx, ax ; calculate number of pixels to be drawn
jne zzok
inc cx

zzok:
    mov zz, cx
    cmp glass, 1 ; glass surface ?
    jne Solid1
    push ax ; yes, then GDC mode: OR
    mov dx, 3ceh
    mov ax, 1003h ; register 3: function select
    out dx, ax
    pop ax

Solid1:
    mov bx, 3c4h ; timing sequencer port
    mov di, 0a000h ; select VGA segment
    mov di, ax ; calculate offset
    shr di, 2 ; (x div 4) + y*80
    add di, vpage ; add current page
    mov bx, y0
    imul bx, 80d
    add di, bx ; now in di
    cmp zz, 4
    jl no_middle ; <draw 4 points -> no blocks of 4
    and ax, 11b ; two lower bits are important
    je middle ; if 0 set blocks of 4 immediately
    no_middle:
        mov bx, 0f02h ; if no_shift, then use this mask
        cmp cx, 20h ; set number of pixels in mask
        jae no_shift
        mov bx, 0102h ; prepare mask
        shl bh, cl
        ; number of pixels=number of bits to set
        dec bh
        and bh, 0fh
        no_shift:
        mov cx, ax ; shift correctly depending on start plane
        and cl, 3
        shl bh, cl
        mov ax, bx ; and mask finished
    ret
hline endp
sub zz, 4                      ; decrement pixels to be drawn
add zz, cx

start:
out dx, ax                     ; set calculated write mask
mov al, b polycol              ; get color
mov ah, es:[di]                 ; load latches, only for glass solids
stosb                         ; set
middle:
cmp zz, 4                      ; if no more blocks of 4 -> conclusion
jl close
mov ax, 0f02h                   ; select all planes
out dx, ax
mov cx, zz
shr cx, 2
mov al, b polycol
mov ah, es:[di]                 ; load latches, only glass solids
stosb                         ; and write back
dec cx
jne @lp
jmp close

solid:
rep stosb                      ; draw middle part

close:
mov cx, xl                      ; set remaining pixels
and cx, 3h
dec xx
js hline_finished               ; if nothing more there -> end
mov ax, 0102h
shl ah, cl                      ; create mask
dec ah
out dx, ax
mov al, b polycol
mov ah, es:[di]                 ; load latches, only for glass solids
stosb                         ; and draw pixels
hline_finished:                  ; GDC mode back to MOVE
mov dx, 3ceh
mov ax, 0003h
out dx, ax
pop es
popa
ret

hline endp

txt_hline                       ; macro contains procedure "hline_texture"

public fillpol

fillpol proc near
; fills polygon in mode X
push bp
pusha

cmp texture, 1                 ; textures being used ?
jne fill
; no, then simply fill
jne fillpol
; otherwise calculate main determinant

fill:
xor si,si                  ;search for highest point, select first entry
mov cx,polyn              ;number of corners
sub cx,2
mov bx,0ffffh             ;very high value, in any case lower
npoint:
    mov ax,poly2d[si+2]    ;get y
    cmp ax,bx             ;if previous minimum lower
    ja  no_min
    mov bx,ax             ;record new minimum
    mov dx,si
    no_min:
    add si,8
    dec cx                 ;next corner, if not 0ffffh
    jns npoint
    mov high_point,dx      ;record in glob var
    mov high_y,bx
    or dx,dx                ;left = 0 ?
    jne dec_valid
    mov bx,polyn
    jmp lr_finished        ;yes: right to the other end
    dec_valid:
        mov bx,dx            ;otherwise
        sub bx,8
    lr_finished:
        mov left,dx         ;record one beforehand in glob var
        mov right,bx
        ; ax/bx : start coordinates left (xl0/yl0)
        ; cx/dx : start coordinates right (xr0/yr0)
        ; si    : overflow flag left
        ; di    : overflow flag right
        ; bp    : pointer to current point
        ; setnewlinel : load line variables
        ; setnewliner
        loopl:
        cmp ax,xl1
        je new_linel          ;if end reached -> set new line
        cmp bx,yl1
        je new_linel          ;otherwise continue drawing
        or si,si               ;increment flag << 0
        jg flaglgreat
        inycl:
            ;this place is being patched !
            inc bx
            ;continue y
            txt_incl
            ;continue 3d coordinates also
            cmp bx,yl1
            ;destination reached ?
            je new_linel          ;then new line
            jmp left_increased    ;y has been increased left -> now right
        flaglgreat:
            ;deccrement inc flag
            sub si,dyl
        incxl:
            ;this place being patched !
            inc ax
            ;continue x
            jmp loopl
        finished___:
        jmp finished
    new_linel:
        mov bx,left           ;prepare increase
        cmp bx,right
        je finished___        ;same, then finished
add bx,8                      ;continue left
mov ax,polyn                  ;is left at the end of the list ?
shl ax,3
sub ax,8                      ;end defined
cmp bx,ax                     ;compare
jb left_set                   ;if yes, then set to 0
left_set:
  mov left,bx
  setnewline;
  jmp loopl
finished:_
  jmp finished
left_increased:
loop:
  cmp cx,xr1
  je new_liner                  ;if end reached -> set new line
  cmp dx,yr1
  je new_liner                  ;otherwise continue drawing
  or di,di                      ;increment flag <= 0
  jg flaggreat
  incy:
  or di,di                      ;this place being patched !
  inc dx
  add di,dxr
  txt_incr
  cmp dx,yr1
  je new_liner                  ;destination reached ?
  jmp right_increased           ;y was increased right -> now draw line
flaggreat:
sub di,dyr                    ;decrement inc flag
inxc:
  inc cx
  jmp new_liner                  ;this place being patched !
new_liner:
  mov dx,right                  ;prepare decrease
  cmp dx,left
  je finished_                  ;if same, then finished
  sub dx,8                      ;if previously at 0->set at other end
  jns right_set
right_set:
  mov right,dx
  setnewline;
  jmp loopr
right_increased:
  push ax
push cx
  cmp cx,ax                     ;correct sequence ?
  jae direct_ok                 ;then ok, otherwise:
  cmp w sf_shift,0              ;suppress surface shift ?
  je draw
  pop cx
  pop ax
  jmp finished                  ;polygon will not be drawn
draw:
  xchg ax,cx
  jmp direct_ok
  ;coordinates in correct sequence
Next, the high point (the one with the lowest y-coordinate) is determined in the loop `npoint`. BX contains the previous minimum and SI contains this point’s number. Then, variables `left` or `right` are loaded. They indicate which line will be worked on the left or on the right. If, for example, `left` indicates point 1, a line is drawn on the left polygon margin from point 1 to point 2. On the right margin, the variable is defined opposite. A variable point number of 3 means a line will be drawn from point 4 to point 3.

Now `left` is set to the determined high point and `right` is set in the coordinate table one point ahead (or at the end of the definition if `left` points to point 0). Now both macros, `setnewlinel` and `setnewliner`, are called. These specify the required variables for the left and right lines.

Similar to the Bresenham algorithm, delta x and delta y are calculated. Their values are calculated with negative signs and the corresponding position in the code is modified so it can be drawn in reverse direction. This is an effective and fast method to control the execution of a loop while avoiding variables.

The actual line drawing algorithm uses a slope defined by delta x and delta y. `Incflag` (in SI or DI) is reduced at delta y with each movement in the x direction and increased at delta x with a movement in the y direction. By recognizing the `Incflag` sign change, we can decide in which direction the next step must go. A positive flag indicates a step in the x-direction and a negative flag indicates a step in the y-direction.

For instance, if delta x = 100 and delta y = 50, the `Incflag` is initially loaded with 100. The first movement occurs to the right (`Incflag = 100`); then delta y is subtracted (`Incflag = 50`) so the second movement is also the right and delta y is again subtracted. The next movement is down since (`Incflag <= 0`). Afterwards, `Incflag` is raised to delta x so it again equals 100. Two steps to the right and one step down is performed repetitively corresponding to a slope of 0.5.

These calculations are performed in loops `loopl` and `loopr` on the left or right sides. They are then checked to see if the target point has been reached. Therefore, the target point is pushed in the `SetNewLineX` macro by one point in the x- and in the y-direction. So, this position must only be checked if one of the two coordinates is identical with the expanded target point. In this case, the original target point was bypassed.
When a line has reached its target, the next one must be taken from the list and the respective variables must be set. This occurs at labels `new_linel` or `newliner`. Variables `left` and `right` are examined to see if they are identical, in which case the polygon is completely drawn and the procedure can be exited at `finished`.

Otherwise, we advance by 8 bytes on the left side of the counter (left) and moved correspondingly back a position on the right side (right). Accordingly, the margins must also be considered, i.e., after the last point it must be positioned on the first point and vice versa.

After the counter has been loaded, the line variables are reset. `setnewline1` and `setnewline2` are called to jump back into the loop.

As we mentioned, the line is drawn after increasing both sides of the y-coordinate. This occurs at label `right_increased`. Essentially, only the procedure `hline` is called, which draws a line from point `ax/bx` to point `cx/bx`. Subsequently, it jumps back into loop `left`.

This procedure then saves the variables that are still required and calculates the length of the distance. If the glass surfaces are lengthened, as happens in this program, the GDC is needed at this point to execute an OR operation between the old latch-content and the new data.

The easiest way to draw a horizontal line is to set all the points following one another to the desired color. This isn't efficient in Mode X where addressing each individual point is very tedious. Instead, you should attempt, wherever possible, to set four points at that address.

At label `Solid1`, the target address of the first byte is then calculated. The first, incomplete block is drawn at label `no_middle`. This is either skipped if the total number of points to draw is smaller than 4 or is needed if the start plane (determined by AND of the x-coordinate with 11b) is not equal to 0. To draw this block, the number of the pixel that needs to be set is first masked in bh. Now, all the pixel from this block are masked and the actual required number of bits is set. This mask is now pushed to the correct position relative to the start plane and sent to the timing sequencer after decreasing `zz`.

Now a read access is performed at the desired offset. This is not sent from the contents of the screen memory to the CPU, but to load the latches so the write access that follows can match the new color with the correct data.

If there are fewer than four pixels to set, then branch to label `end`. Otherwise, all the planes must be turned on and the corresponding number of blocks set to the correct color. This placement is performed with an invisible figure using a REP STOSB instruction. Unfortunately, it's slightly more complicated with glass figures. Here, a loop needs to be inserted since a read access must follow each byte.

The end now sets all the pixels from the last block up to the target coordinate. In this manner, a mask is formed, set, and the color written all corresponding to the target plane. With the label `hline_complete`, only the write mode of the GDC is again directed to replace and the procedure is exited.
Hidden Lines

Glass figures have their own appeal, but they are seldom suitable for depicting real figures. Since most figures are not transparent, even computer illustrations must be careful not to show invisible back surfaces.

The problem with concealing back surfaces is one of the most complex ideas in three-dimensional presentations. It’s not just a matter of drawing only certain surfaces rather than other surfaces. To some extent, surfaces overlap each other requiring that both be drawn -- but in the correct order. We will talk about both concealment and sorting in this section.

There are many ways to hide invisible surfaces. Invisible surfaces usually form an angle between the surfaces and the line of vision (line extending from the eye to the corresponding surface). This angle determines whether the viewer is looking at the front or the back of the surface. The surface must be concealed if the viewer is looking at the back of the surface.

Although the method we will describe is based on a similar idea, it assumes that all surfaces are defined in a counterclockwise sense (mathematically positive). Therefore, the surface is independent of its position in space. If it turns itself around so the viewer sees its back surface, it appears mirror reversed and is drawn "forwards".

Now, it is easily checked with hline when the horizontal lines are drawn, if their ending point (corresponding to the definition of the surface algorithm) is to the right of the starting point. If this is not the case, you'll be looking at its back surface, which must be hidden. The program checks for this in the procedure Fillpol of the POLY.ASM module at label right_increased:

```assembly
right_increased:
push ax
push cx
cmp cx, ax        ;correct sequence ?
jae direct_ok     ;then ok, otherwise:
cmp w sf_shift, 0 ;suppress surface shift ?
je draw           ;no, then draw anyway
pop cx
pop ax
jmp finished      ;polygon will not be drawn
draw:
    xchg ax, cx   ;coordinates in correct sequence
direct_ok:
```

The AX and CX registers contain the x-coordinates of the starting and ending points. With a mathematically positive defined surface, CX must be larger than AX. If this is the case, it is drawn directly (label direct_ok), otherwise, the filling is ended (if fl_backs TRUE, otherwise the position is ignored and drawn anyway after exchanging the coordinates).

This method is adequate for convex figures and figures without "depth" (i.e., cubes) but what about concave figures such as a U-shaped object? Invisible surfaces are sorted out, but the series order is still not correct, leaving some of the surfaces completely visible, while others remain hidden.

If you now sort the surfaces, so the first surface to be drawn is the one with the largest z-coordinate, i.e., the one that is farthest back, and then always work towards the front, the newer surfaces that are in front cover the recently drawn world. Since this method is also used in painting, this is called a "painter's algorithm".
How do you go about sorting surfaces? Surface corners usually have entirely different z-coordinates. Complex (therefore, slow) algorithms attempt to make the calculations and to find connections between the corners, which makes a clear assignment possible.

Therefore, the mean value for the depth information is given for each surface and used as sorting criteria. This method can be imprecise. Especially with surfaces having a large range in the z direction, you experience very attractive and, above all, very fast results when combined with the concealment of the surface backs.

The most recently mentioned array \textit{mean} picks up this middle value and saves the corresponding registers for the surfaces. That is why completed z-values are added to the current array position in the macro \textit{z2cx}. The mean value is then formed in the label \textit{polyok}, where this total is divided from the number of corners:

```assembly
polyok:
    mov bx,meanzptr ;calculate mean value:
    mov ax,mean[bx] ;get sum
    mov cx,polyn
dec cx
cwd
    div cx ;and divide by number of corners
    mov mean[bx],ax ;write back
```

If the variable \textit{fl_sort} is set to TRUE, the procedure \textit{quicksort}, which sorts the array \textit{mean}, is called after calculating all the surfaces. The depth information (in the lower half of a doublewords) serves as a sorting criterion. The surface identification is sorted with it.

```assembly
public quicksort
quicksort proc pascal down,up:word
;sorts Mean-Array according to Quicksort algorithm

local key:word
local left:word
push bx
mov bx,down ;find middle
add bx,up
shr bx,1
and bx,not 3 ;post on blocks of 4
mov dx,mean[bx] ;get key
mov key,dx
mov ax,down ;initialize right and left with base values
mov si,ax
mov left,ax
mov ax,up
mov di,ax
mov dx,key
left_nearer:
cmp mean[si],dx ;greater than key -> continue searching
jbe left_on
add si,4 ;post on next one
jmp left_nearer ;and check it
left_on:
```

The procedure \textit{polyok} is part of the \textit{POLY.ASM} file on the companion CD-ROM.

The procedure \textit{quicksort} is part of the \textit{3DASM.ASM} file on the companion CD-ROM.
The Quicksort algorithm then divides the array into two halves. It continues to exchange elements until only values larger than that in the middle element are in the left half and smaller values on the right half. This part of the array is then sorted recursively in the same manner.

Unlike the 3D_GLASS.PAS program, 3D_SOLID.PAS does not generate a palette. This is why other colors are also defined for the surfaces. 3D_SOLID.PAS also uses the global variables differently. For example, glass is set to FALSE because glass surfaces are not desired here; besides, both the variables fl_backs and fl_sort are set to TRUE for the handling of the hidden surfaces. Otherwise, both programs are identical.

**Throwing Shadows: Light Source Shading**

Now, we have control over solid, visible figures that can move around anywhere in the space. However, illumination is one important aspect that we have overlooked. We can present the three dimensional worlds much more impressively by introducing a light source.

Today’s PC’s do not yet have the ability to perform for real time ray tracing. These types of pictures require a few minutes to generate. So, we use a faster method that does not consider every ray of light but produces very beautiful effects by using simpler methods.

If you move away from a continuous yet distant light source, all the rays of light appear to be parallel on the surface of the object. This is why you can perform ray tracing calculations using a single light vector instead of calculating each point on the surface individually. This “homogenous” illumination makes it possible to estimate the light covering each surface.
The following model demonstrates how the illumination on a surface is calculated. The "flatter" the light that strikes the surface, the darker is the illumination. The illumination is at its maximum when the light is perpendicular to the surface. This is because an equally large mass of energy is distributed over a larger surface of flat angles and is, therefore, not as dense:

\[ \frac{d}{f} = \sin(\beta) \]

\[ \beta' = \beta - 90 \]

\[ d = \sin \beta \]

**Relationship of the angle to the brightness**

The relationship \( \frac{d}{f} \) is proportional to the illumination of the surface (as shown above). This relationship is the same as the negative cosine of the angle between the light vector and the normal vector for the surface. The normal vector is a vector that stands perpendicular to the surface. It's easily determined using the intersecting product of two vectors that lie in the plane. Because the cosine of the angle is needed here, and not the angle itself, simplifies the calculation greatly since the angle determination (through the scalar product) indicates the cosine of the angle.

The procedure for determining the illumination is as follows:

- Find two vectors appearing on the surface. The easiest would be two margin vectors (from the first to the second and to the last point).
- Form normal vector (intersecting product of the surface vectors).
- Through the scalar product, angle between (form constant light vector and normal vector).
- Add the result to the color.

The result of the angle calculation is negative in the example. If the surface is turned away from the light (\( \beta < 90 \) degrees), the result will be positive. Only the basic surface color can be used because it lies in the shadow and, therefore, only receives diffused light. The 3D_LIGHT.PAS program demonstrates the capabilities of the shading routine:

```pascal
Uses Crt,ModeXLib,Gif,var_3d;
Const
    worldlen=8*3;               {Point-Array}
    Worldconst:Array[0..worldlen-1] of Integer =
        (-200,-200,-200,
        -200,-200,200,
        -200,200,-200,
        -200,200,200,
        PC underground

You can find 3D_LIGHT.PAS on the companion CD-ROM
```
200,-200,-200,
200,-200,200,
200,200,-200,
200,200,200);
surfclen=38;                {Surface-Array}
surfcconst:Array[0..surfclen-1] of Word=
($fee0, 4, 0,2,6,4,
$fee0, 4, 0,1,3,2,
$fee0, 4, 4,6,7,5,
$fee0, 4, 1,5,7,3,
$fee0, 4, 2,3,7,6,
$fee0, 4, 0,4,5,1,0,0);    { $fe = use light source, primary color in the low-byte}

Var
i,j:Word;
Procedure Shad_Pal;             {prepare palette for shading}
Begin
For j:=192 to 223 do Begin    {prepare colors 192 - 223 and 224 - 255}
i:=trunc((j/32)*43);        {determine brightness}
Fillchar(Palette[j*3],3,i+20);  {colors 192-223 to gray tones}
Palette[(j+32)*3]:=-1+20;    {colors 224-255 to red tones}
Palette[(j+32)*3+1]:=0;
Palette[(j+32)*3+2]:=0;
End;
Setpal;                       {set this palette}
End;
procedure drawworld;external;   {draws the world on current video page}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l root.obj}
Begin
vz:=1000;                     {solid is located at 1000 unit depth}
vpage:=0;                     {start with page 0}
LoadGif('logor.gif');         {load wallpaper}
init_modex;                   {enable ModeX}
Shad_Pal;                     {calculate shading palette}
rotx:=0;                      {initial values for rotation}
rotz:=0;
Fill:=true;                   {SurfaceFill on}
sf_sort:=true;                {SurfaceSort on}
sf_shift:=true;               {SurfaceShift suppression on}
Glass:=false;                 {glass surfaces off}
p13_2_modex(16000*2,16000);  {wallpaper to VGA page 2}
repeat
CopyScreen(vpage,16000*2);   {wallpaper to current page}
DrawWorld;                    {draw world}
switch;                      {switch to finished picture}
WaitRetrace;                  {wait for next retrace}
Inc(rotx);                  {continue rotating ...}
If rotx=120 Then rotx:=0;
Inc(rotz);
If rotz=120 Then rotz:=0;
inc(roty);
if roty=120 Then roty:=0;
Until KeyPressed;             { ... until key}
TextMode(3);
End.
The palette is prepared before the shading can be established. Colors 192 to 223 and 224 to 255 are filled with two color displays. The first one contains the colors ranging from gray to white. The second one contains the colors ranging from dark red to red. Now the cosine can simply be added to the basic color when filling so the correct degree of shading is maintained.

The basic color is classified in the low byte of the color information (0C0h and 0E0h). The high byte is tested against this with the first directional value. The light source shading for this surface is turned on with 0FEh. This is why it's now possible to mix shaded surfaces (directional byte 0FEh) with fixed value surfaces (directional byte < 0FEh) or with textures later.

An additional change to the 3D_SOLID program involves the background picture. A picture is loaded at the beginning, which is then calculated by a raytracer and contains a similar light vector as the rotating object, which reinforces the impression of a light source. Instead of erasing the screen before creating every picture, the background picture is simply copied each time to the corresponding page.

In procedure `drawworld` (in 3DASM.ASM), the directional byte is deciphered right at the beginning and the global variables `Lightsrc` and `Texture` are set, in this case it only refers to `Lightsrc`.

Additionally, arrays `Points3D` and `Poly3D` are factored. The first array receives three dimensional coordinates which were completely rotated and calculated in the macro `zrot`. These are then carried over to loop `npoint` in array `Poly3D` and ultimately formed into a closed line, in which the first corner is copied onto the last one. This process completely corresponds to the one executed for the two dimensional coordinates.

If the surface is now filled, the macro `getdelta` is called. It determines both surface vectors:

```assembly
getdelta macro                  ; calculates the two surface vectors
    mov ax,poly3d[0]              ; x: original corner
    mov delta2[0],ax              ; store temporarily in delta2
    sub ax,poly3d[8]              ; obtain difference to first point
    mov delta1[0],ax              ; and delta1 finished
    mov ax,poly3d[2]              ; y: original corner
    mov delta2[2],ax              ; store temporarily in delta2
    sub ax,poly3d[10d]             ; obtain difference to first point
    mov delta1[2],ax              ; and delta1 finished
    mov ax,poly3d[4]              ; z: original corner
    mov delta2[4],ax              ; store temporarily in delta2
    sub ax,poly3d[12d]             ; obtain difference to first point
    mov delta1[4],ax              ; and delta1 finished
    mov bp,polyn                  ; select last point
    dec bp
    shl bp,3                      ; 8 bytes at a time
    mov ax,poly3d[bp]             ; get x
    sub delta2[0],ax              ; obtain difference
    mov ax,poly3d[bp+2]            ; get y
    sub delta2[2],ax              ; obtain difference
    mov ax,poly3d[bp+4]            ; get z
    sub delta2[4],ax              ; obtain difference
endm
```

`delta1` is loaded here with the difference between the first and second polygon points and `delta2` is loaded with the difference between the first and last points. If no illogical surfaces are defined, both vectors will always be independently linear (not parallel) and, therefore, available.
Otherwise, the program is stopped with a Division by Zero error.

If global variable \texttt{lightsrc} is TRUE here, both \texttt{get\_normal} and \texttt{light} macros are called. The first one determines the normal surface vector from the two surface vectors, \texttt{delta1} and \texttt{delta2}. The second macro determines the lightness of the surface from the angle.

\begin{verbatim}
get_normal macro                ;calculates normal vector of an area
    mov ax,delta1[2]              ;a2*b3
    imul delta2[4]
    shrd ax,dx,4
    mov n[0],ax
    mov ax,delta1[4]              ;a3*b2
    imul delta2[2]
    shrd ax,dx,4
    sub n[0],ax
    mov ax,delta1[4]              ;a3*b1
    imul delta2[0]
    shrd ax,dx,4
    mov n[2],ax
    mov ax,delta1[0]              ;a1*b3
    imul delta2[4]
    shrd ax,dx,4
    sub n[2],ax
    mov ax,delta1[0]              ;a1*b2
    imul delta2[2]
    shrd ax,dx,4
    mov n[4],ax
    mov ax,delta1[2]
    imul delta2[0]
    shrd ax,dx,4
    sub n[4],ax                   ;cross product (=normal vector) finished
    mov ax,n[0]                   ;x1 ^ 2
    imul ax
    mov bx,ax
    mov cx,dx
    mov ax,n[2]                   ;+x2 ^ 2
    imul ax
    add bx,ax
    adc cx,dx
    mov ax,n[4]                   ;+x3 ^ 2
    imul ax
    add ax,bx
    adc dx,cx
    ;sum in dx:ax
    push si
    call root
    ;root in ax
    pop si
    mov n_amnt,ax                 ;amount of normal vector finished
endm
\end{verbatim}

The first part of this macro calculates the normal vector itself. Then, every individual vector component from the difference of two products is formed. This must follow the definition of the intersecting product and is stored in array \texttt{n}. The second part next calculates the amount of the normal vector in which all the components are squared and then added. This sum is reduced and the result is saved in \texttt{n\_amt}. The last step executes the \texttt{light} macro:
Chapter 7

The Third Dimension: 3-D Graphics Programming

After the scalar product is formed between the normal vector and the (constant) light vector (in \( l \)), the amount of the (equally constant) light vector is divided by the amount of the normal vector and the angle between both of the vectors is determined. The result of this calculation would normally be between plus and minus one. However, since whole numbers are used here, the interim result is multiplied by 32 before the second division so a value range of -32 to +32 is reached.

If the result is positive, AX is set to zero and the basic color is retained. If the result is negative, this value is subtracted from the basic color and, as a result, the amount is added. The completely calculated lightness factor is then located in variable \textbf{PolyCol}, which is used as a filling color by \textbf{FillPol}.

\begin{verbatim}
light macro ;determines brightness of an area
  mov ax,n[0]
  imul l[0] ;light vector * normal vector
  mov bx,ax ;form sum in cx:bx
  mov ax,n[2]
  imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
  imul l[4]
  add ax,bx ;scalar product finished in dx:ax
  adc dx,cx
  idiv l_amnt ;divide by l_amnt
  mov bx,n_amnt ;and by n_amnt
  cwd
  shld dx,ax,5 ;values from -32 bis +32
  shl ax,5d
  mov bp,startpoly ;prepare addressing of surface color
  idiv bx ;division by denominator
  inc ax
  or ax,ax
  js turned_toward ;if cos à positive -> turned away from the light
  xor ax,ax ;thus, no light
  turned_toward:
    mov bx,polycol,al ;cos<0 -> add to primary color
  endm
\end{verbatim}

Impressive Top Surfaces: Textures

Our final 3-D topic is \textit{textures}. By using textures, you can add a "structure" to the previously static and monotone surfaces. These textures are bitmap graphics that are projected onto the surfaces and moved in different directions with each rotation.

The bitmaps, when used, appear glued to the surfaces and can simulate a particular surface, such as wood or metal. This technique is used in role playing games such as Ultima Underworld where the walls are sometimes made out of stone, wood or other materials.

If you have a definite programming concept, you can compose a surface from many small surfaces. These small surfaces, in turn, correspond to a point from a bitmap. This technique may not be the fastest because it does not often function regularly. If such a surface should be somewhat enlarged or turned, you will notice glitches immediately because a few of the points overlap, which then become missing.
The only practical solution to this problem is to reverse the process so the surface is presented as indicated so every point is set. This point is projected back every time to the original surface to establish its location (position) within this surface. Based on the position, the point color can then be read from the texture bitmap.

Since it's impossible to close from the two dimensional screen coordinates to the three dimensional position of the point, the 3D coordinates are counted from the beginning when filling, so you know the coordinates for every point and, therefore, its position within the surface.

The 3D_TEXTU.PAS program demonstrates the texture routines. The difference between this program and the previous program is its different color values in the world definition and procedure Prep_Textures, which is called immediately:

```pascal
Uses Crt,ModeXLib,Gif,var_3d;
Const
worldlen=8*3;               {Point-Array}
Worldconst:Array[0..worldlen-1] of Integer =
(-200,-200,-200,
-200,-200,200,
-200,200,-200,
-200,200,200,
200,-200,-200,
200,-200,200,
200,200,-200,
200,200,200);
surfclen=38;                {Surface-Array}
surfcconst:Array[0..surfclen-1] of Word=
($ff00,4, 0,2,6,4,
$ff01,4, 0,1,3,2,
$ff02,4, 4,6,7,5,
$ff00,4, 1,5,7,3,
$ff03,4, 2,3,7,6,
$ff04,4, 0,4,5,1,0);  // $ff = use textures, number in the low-byte
Var
i,j:Word;
Procedure Prep_Textures;   // load texture variables
Begin
LoadGif('Texture');       // load texture image
GetMem(Txt_Pic,64000);     // get memory for this
Move(VScreen^,Txt_Pic^,64000); // and copy to memory
For i:=0 to Txt_Number-1 do Begin
  Txt_Data[i]:=Txt_Pic;   // load pointer to data
  Txt_Offs[i]:=i*64;      // determine offset
End;
End;

procedure drawworld;external;  // draws the world on current video page
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l root.obj}
Begin
vz:=1000;                    // solid is located at 1000 unit depth
vpage:=0;                     // start with page 0
init_modex;                   // enable ModeX
End;
```
The value 0FFh serves as the directional byte for all the surfaces. It activates the texture with the number listed in the low byte for the respective surfaces. Procedure Prep_Textures then loads the GIF picture with the necessary textures and moves it to another file area (TxtPic^). At this point, the arrays Txt_Data and Txt_Offs are used. Txt_Data displays a counter with the position of the texture screen for every texture, making it possible to load many independent screens with textures. Txt_Offs contains the offset for the respective texture within this screen.

In this case, Txt_Data contains a counter for all the textures in Txt_Pic^ and Txt_Offs a multiple of 64, i.e., five textures in a row with a width of 64 Pixel.

The size of the textures is determined by the constant array Txt_Size. For every texture listed here, the y-size is established with the high byte and the x-size with the low byte. A particular standard needs to be observed when calculating these sizes: The textures each has the dimension of 256 SHR (n-8) Pixel. The value of 10 (0Ah) indicates, for instance, a size of 256 SHR 2 = 64 Pixel (in the x- or y- direction).

Some of the program sections that we have already discussed, such as forming the surface vectors Delta1 and Delta2, can also be used with the textures. However, most of the required algorithms are found as macros in the include file TEXTURE.INC, which functions as a complement to POLY.ASM.

The global variable Texture is set to TRUE with texture surfaces. The primary determinants are formed immediately from the start of the procedure FillPol in the macro Txt_Maindet. With their help, the relative coordinate of the point within the surface is determined later. It's also important to know that every point is a key in the following equation:

\[
x_1 = \lambda_1 * a_1 + \lambda_2 * b_1 \\
x_2 = \lambda_1 * a_2 + \lambda_2 * b_2 \\
x_3 = \lambda_1 * a_3 + \lambda_2 * b_3
\]

Then x1-x3 are the coordinates for the point, a1-a3 the components of the first surface vector Delta1) and b1-b3 the components of the second (Delta2). lambda1 and lambda2 give the refined coordinates relative
to both of the surface vectors and can be used to gain direct access to the texture. To calculate $\lambda_1$ and $\lambda_2$, you need two of the equations. The third is then completed in every case because the point lies in the plane.

If, for example, you take the first of the two equations, the solution is easy to find using the determinants: The primary determinant equals $D = a_1b_2 - a_2b_1$, the first adjacent determinant $D_1 = x_1b_2 - x_2b_1$ and the second adjacent determinant $D_2 = a_1x_2 - a_2x_1$. The two unknowns are then identified as $\lambda_1 = D_1/D$ and $\lambda_2 = D_2/D$. The main determinant is now the same for the entire surface so it can be directly calculated here:

```assembly
.txt_maindet_macro
;calculate main determinate
xor si,si ;first attempt: rows 0 and 1
mov di,2
next:
mov ax,w delta1[si] ;calculate main determinant
imul w delta2[di]
mov bx,ax ;store intermediate result
mov cx,dx
mov ax,w delta2[si]
imul w delta1[di]
sub bx,ax ;store difference
sbb cx,dx
mov w D,bx
mov w D+2,cx
or bx,cx ;main determinant = 0 ?
jne D_finished ;then new components
add si,2
add di,2
cmp di,4 ;still within existing rows ?
jab next
xor di,di
jmp next
D_finished:
movsx eax,delta1[si] ;store used column values
mov column1[0],eax
movsx eax,delta1[di]
mov column1[4],eax
movsx eax,delta2[si]
mov column2[0],eax
movsx eax,delta2[di]
mov column2[4],eax
shl si,1 ;note used columns
shl di,1
mov upper_row,si
mov lower_row,di
endm
```

However, you are likely to notice another problem: Under certain circumstances, selecting the first of the two equations may not be what you expect or want. This indicates the main determinant equals 0. In this case, you simply need to use another combination.

This occurs in front of the label D_complete. It's simply positioned on the next row within the surface vectors (initially 0/2, then 2/4, then 4/0) and another attempt is initiated. Afterwards, the main determinant is saved in the column1 and column2 arrays and the numbers for the rows that were used in top_row and bottom_row; this data will be needed later.
As we mentioned, the filling algorithm must constantly keep pace with whatever coordinate the current point is located. Three dimensional lines are calculated parallel to touching the polygon edges, which define the polygon in its entirety. The initialization of the attached three dimensional lines must be imbedded in macros SetNewLineL and SetNewLineR. This occurs in macros txt_makefarL and txt_makevarr:

```assembly
.txt_makefarL macro
    ;reloads the 3d variables of the left side
 .386
    movsx ebx,dyl                 ;number steps
    inc ebx
    push ecx
    push edx
    movsx eax,poly3d[bp]          ;get 3d-x
    shl eax,8                     ;lower 8 bits are "fractional" part
    mov xL_3d,eax                 ;and write
    movsx ecx,poly3d[bp+8]        ;obtain difference
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx                      ;define step
    mov inc_xl,eax

    movsx eax,poly3d[bp+2]        ;get 3d-y
    shl eax,8
    mov yL_3d,eax
    movsx ecx,poly3d[bp+10]       ;obtain difference
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx                      ;define step
    mov inc_yl,eax

    movsx eax,poly3d[bp+4]        ;get 3d-z
    shl eax,8
    mov zL_3d,eax
    movsx ecx,poly3d[bp+12]       ;obtain difference
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx                      ;define step
    mov inc_zl,eax
    pop edx
    pop ecx
    endm
```

The three-dimensional coordinates for the corner points of the recently worked surfaces are located in the array Poly3D. Register bp shows the position of the current coordinates in the Poly2D array (we are still in setnewlinel or setnewliner). Since this array is constructed synchronously, the 3-D coordinates are addressed using BP. The macro then loads the variables xL_3D, yL_3D, etc., with the start values of the first corner point and calculates the difference to the second corner point, i.e., the line lengths, in x-, y- and z-direction.

The principle for these lines is that the drawing depends on every change to the y-coordinate also advancing the steps on the line. From now on, the differences must still be divided by the number of steps (i.e., the "height" of the two dimensional line in dyl or dyr). Several values are pushed to the left by 8 bit in the process so they are sufficiently precise. If the three dimensional line is, for example, seven units long (let's
say, in the z-direction) and the number of steps equals four, the value 1.75 must be added every time, which can only be possible if you reserve the bottom most byte for the after comma section (in this case, it would be inc_zl = 01C0h).

Now the variables must be recounted with every step into the next screen line. That is why shortly after the labels incyl or incyr, the macros txt_incl and txt_incr must be combined:

```assembly
.txt_incl macro
    push eax
    mov eax,inc_xl
    add xl_3d,eax
    mov eax,inc_yl
    add yl_3d,eax
    mov eax,inc_zl
    add zl_3d,eax
    pop eax
.endm

.txt_incr macro
    push eax
    mov eax,inc_xr
    add xr_3d,eax
    mov eax,inc_yr
    add yr_3d,eax
    mov eax,inc_zr
    add zr_3d,eax
    pop eax
.endm
```

These macros simply do what their names suggest: They increment the respective variables. Inc_xl is added to xl_3d, inc_yl to yl_3d, etc. In doing so, the variables block of xl_3d-yl_3d-zl_3d always contain the current 3-D coordinates for the left start point of the horizontal filling line. The same is true for the right side and the end point of the filling line.

The listing becomes interesting in the hline_texture procedure. It's located in the macro txt_hline so it can be placed in an include file. This procedure is called with texture surfaces instead of hline and does basically the same thing: It draws a horizontal line between the coordinates specified in ax/bx and cx/bx. The only difference between it and hline is the color is not the same for every point. The color is instead determined by the texture.

After saving the coordinates, the adjacent determinants for the left and for the right side are calculated with lambda1 and lambda2. The left side transmits the start values since the drawing begins on this page. The right side transmits the end value, which flow into variables inc_lambda1 and inc_lambda2:
mov bp, upper_row
mov bx, lower_row
mov eax, xr_3d[bp]                   ; determine relative x-coordinate
movsx ecx, poly3d[2]
shl ecx, 8                         ; put in "fixed point" format
sub eax, ecx
mov edx, eax
movsx ecx, w column2[0]
imul ecx                           ; multiply by Delta2 \( x \)
mov esi, eax
mov eax, xr_3d[bx]                  ; determine relative y-coordinate
movsx ecx, poly3d[0]
shl ecx, 8                         ; put in "fixed point" format
sub eax, ecx
mov edx, eax
movsx ecx, w column2[4]
imul ecx                           ; multiply by Delta2 \( y \)
sub eax, esi
mov eax, edx
imul ebx, edx
mov bx, column1[4]
imul ecx                           ; multiply by Delta1 \( y \)
sub eax, esi
mov eax, edx
imul ebx, edx
mov bx, column1[0]
imul ecx                           ; multiply by Delta1 \( x \)
sub eax, esi
mov eax, edx
mov inc_lambda1, eax                ; store for subtraction
mov eax, d_x
movsx ecx, poly3d[4]
imul ecx                           ; multiply by Delta2 \( x \)
mov esi, eax
mov eax, edx
movsx ecx, poly3d[0]
imul ecx                           ; multiply by Delta2 \( y \)
sub eax, esi
mov eax, edx
movsx ecx, poly3d[4]
imul ecx                           ; multiply by Delta1 \( y \)
sub eax, esi
mov eax, edx
movsx ecx, poly3d[0]
imul ecx                           ; multiply by Delta1 \( x \)
sub eax, esi
mov inc_lambda2, eax                ; store for subtraction

With the variables top_row and bottom_row, the necessary coordinates (depending on the main determinants used) are then selected by the right side and the difference is calculated to the beginning of the surface (point 0); with multiplying the corresponding section of the main determinant (in column1 and column2) and the subsequent subtraction of both products, the adjacent determinant is established and divided by the main determinant, the result is stored temporarily in inc_lambda1 and inc_lambda2. The left lambda values are then determined in the same manner:
The Third Dimension: 3-D Graphics Programming

mov d_x, eax
movsx ecx, w column2[4]
imul ecx           ;multiply by Delta2 y
sub eax, esi       ;obtain difference (D1)
shl ecx, 8         ;prepare division
idiv dword ptr D   ;divide by main determinant
neg eax
mov lambda1, eax   ;lambda1 determined
sub inc_lambda1, eax
mov eax, d_x       ;get relative x-coordinate
movsx ecx, w column1[4]
imul ecx           ;multiply by Delta1 y
mov esi, eax        ;place result in temporary storage
mov eax, d_y        ;get relative y-coordinate
movsx ecx, w column1[0]
imul ecx           ;multiply by Delta1 x
sub eax, esi       ;obtain difference (D2)
shl eax, 8
neg eax
mov lambda2, eax    ;lambda2 determined
sub inc_lambda2, eax

The results of these Lambda calculations are stored in the variables lambda1 and lambda2 and inc_lambda1 and inc_lambda2 account for the difference. Afterwards, both Inc-values are divided by the length of the horizontal lines, i.e., the number of steps:

movsx ecx, zz          ;calculate Lambda-Schrittweiten
mov eax, inc_lambda1   ;get total length
cdq
idiv ecx               ;and divide by number steps
mov inc_lambda1, eax
mov eax, inc_lambda2   ;get total length
cdq
idiv ecx               ;and divide by number steps
mov inc_lambda2, eax

In this manner, it is enough to add inc_lambda1 to lambda1 and inc_lambda2 to lambda2 at every step so we can have the current lambda1 and lambda2 available.

Next, we’ll prepare to address the points in the screen memory:

mov ax, 80d           ;determine offset
mov bx, y0
mul bx
mov bx, x0             ;(x div 4) + y*80
shr bx, 2
add ax, bx
add ax, vpage
mov di, ax
mov ax, 0a000h         ;load VGA segment
mov es, ax
mov cx, x0             ;mask start plane
and cx, 3
mov ax, 1
shl ax, cl
mov b plane+1, al      ;set corresponding bit
After calculating the offset for the first point, the start-plane is determined and filed in the high-byte of variable \texttt{plane}; the low-byte contains constant 2, so the entire word must be sent to the timing sequencer in order for the current plane to be selected. The plane is prepared on the rotation, in that the high- as well as the low-nibble contain the plane mask (plane 3 becomes 88h, after one rotation, then 11h, i.e., plane 0).

With the number of the current texture (in \texttt{txt\_no}), the texture size is then loaded and patched in the code. In so doing, the operand of two of the SAR instructions are addressed, so the Lambda’s values area can be limited.

The offset is also read from the corresponding array and patched with the \texttt{txt\_data-counter} offset section into the code.

The loop to draw the points can now be held relatively short. This is extremely important when considering the speed:

```assembly
lp:                           ;runs for each pixel
    add ebp,inc_lambda1     ;continue Lambda1 and 2
    add esi,inc_lambda2
    mov ax,plane           ;get plane
    out dx,ax              ;and select
    mov eax,ebp
    size_patch:            ;determine offset within the texture graphic
        sar eax,11d
        sar ebx,esi
    ofs_patch:             ;set size, being modified
        mov es:[edi+1111h],al
        dec cx                ;decrement number pixels
        je hlt\_finished      ;all pixels finished ?
        rol b plane+1,1      ;next plane
        cmp b plane+1,11h     ;plane overflow from 3 to 0 ?
        jne lp                ;no, then continue
        inc di                ;otherwise increment offset
```
The EBP register contains \texttt{lambda1} and \texttt{lambda2} within the loop. Both values must then be increased to their increments. Afterwards, the current plane is selected and the offset within the texture is determined with edx and esi. Both lambda values are then pushed to the right because of their sizes so the desired value area can be uncovered.

The deciding event occurs in the label \texttt{ofs_patch}. Here, the color is retrieved from the texture. In this, the offset that was patched in the code is added to the calculated value and the texture is selected at this point. The color is written into the screen memory and the loop is closed by further rotating the plane mask; if an overrun occurs from plane 3 to plane 0, the target address is raised by 1, for the next offset to be addressed.
Most software developers believe many more copies of their programs are being used than the number of programs that have been sold. This is, of course, due to the widespread use of illegal or pirated copies. Most experts agree those illegal or pirated copies of software cost developers many millions of dollars annually in sales and royalties. This is why some software developers use various schemes to protect their programs from illegal copying and distribution.

The most common form of software protection is password protection. It’s both the oldest form and the easiest to implement. Passwords are available in many forms. For example, you might have the option of protecting your computer by means of a ROM password which is requested on system startup. When using a network computer, you probably have a password on the network to grant or deny you access to the system. If you are a modem user, you are certainly familiar with the passwords for various BBSes and information services such as CompuServe.

We’ll talk about the available passwords in the next section and which are the most appropriate for you.

**Protecting Your Programs: Passwords**

You have several ways to protect your program. First, you must consider the purpose or reason for using a password query. We group the reasons for using a password query into three main areas:

**The password as permission to access**

The password serves to acknowledge that a legitimate user is using the program. This is the type of password you will find, for example, protecting your user account on CompuServe, on networks or even in ATM machines used by banks or corporations. Because only a few employees need access to the financial data, these records and files are often protected with a password.

**The password as registration**

This type of password is often used in shareware programs. When you register the shareware program, you are sent a password that you use to upgrade your shareware program to a complete and more powerful version. This method is used frequently for applications.
Modern Copy Protection

Chapter 8

The password as copy protection

This type of password is often used for games and entertainment software. The software developer uses this method if there is a likelihood the game or program will be copied. The password query doesn't usually appear as the program is loading or at the start of the program. Instead, the password query appears after one or more levels are completed or, more likely, when the user reloads a saved game or session.

Some earlier methods, such as changing a few sectors on a separate, yet required, startup diskette, are seldom used today because registered or otherwise "legitimate" users found these methods to be inconvenient. Also, these methods are no longer quite up to date with today's larger hard drives.

A few extremely expensive programs use a dongle (also called a hardware key). A dongle is a small hardware device containing a password or checksum which plugs into either a parallel or serial port. Some specially designed dongles even include complete program routines.

We won't talk about hardware-oriented password queries such as dongles in this chapter. Instead, we'll talk only about software solutions.

There is one basic principle which applies to all three password types we mentioned above: The better the password is hidden, and the better it is encrypted, the more secure your program will be. So, you should carefully consider where you want to store your password. Choose, if possible, an external file and not the .EXE file. If you work with several files, save the password in one of the larger files. Under no circumstances should the password be at the beginning or the end of a file...this is the first place an intruder would look and where it is easiest to find. Even when you're working with a single file, save the password in the middle of a file where it is the most difficult to locate.

If possible, don't save the actual comparison string, but an encoded version of the string. Use a supplementary checksum for additional security.

Most password queries work according to the following pattern:

1. Build screen
2. Read user password
3. Compare with user password and respond accordingly

This scheme works quite well and is easy to program. However, if you want to protect one of your own programs with a password, you should use a different technique. Dividing the individual steps will make the code less transparent.

First, build the screen and then perform the other tasks: Initialize variables, draw graphics, test checksums, etc. Read the user password only when these steps are complete. The operating speed of the PC and the keyboard buffer will prevent user entries from being lost. After you have read in the password, we recommend that you do not immediately compare the password. Instead, perform some of the other tasks we mentioned. Then load the correct password from the file and compare it.

The main goal is to frustrate any would-be hackers. When you incorporate the password query into a procedure, it becomes difficult to find on the screen. Then the jump resulting after a comparison is then turned quickly into a random jump by the would-be hackers. There's no guarantee, of course, that using this
method will result in an absolutely secure system but you will have made break-in considerably more
difficult because the query is harder to locate.

Depending on the type of password you need, you will have to build encrypting and decrypting routines
and the necessary loading and saving routines into your program. Let's first take a look at the second type.
In this case there is only the password. It is created by the programmer of the program and built into one
of the files. A routine is in the program itself with which the password is queried, read and compared.

You'll find a program for generating a coded password in PASSWGEN.PAS. In this program, the desired
password is read in, the respective letters being added to a key to be entered. Finally, the complement of
the letters is formed by means of an XOR with 255. The key is
first saved (1 char), then the password (256 char) and, finally, the
checksum (1 word) which prevents manipulation of the
password file. The program generates the file PASSWORD.DAT
containing the password, which should be inserted into a
different file.

```
program GENERATE_PASSWORDFILE;
{
  The program encrypts an entered password and saves it
  in the file PASSWORD.DAT.
  Author: Boris Bertelions
  (c)'95 Data Becker GmbH
  (c)'95 ABACUS Software, Inc
}
Uses Crt;
var password : string;
  pwf : file;
  check : word;
  key : char;
function Encrypt(pasw : string;add : char) : string;
  var li : integer;
  begin
    for li := 1 to 255 do begin;
      pasw[li] := char(255 xor (ord(pasw[li]) + ord(add)));
    end;
  Encrypt := Pasw;
end;
function Gen_checksum(pasw : string) : word;
  var sum : word;
  li : integer;
  begin
    sum := 0;
    for li := 1 to ord(pasw[0]) do begin;
      sum := sum + ord(pasw[li]);
    end;
    Gen_checksum := sum;
  end;
function Decrypt(pasw : string;add : char) : string;
  var li : integer;
  begin
    for li := 1 to 255 do begin;
      pasw[li] := char((255 xor ord(pasw[li])) - ord(add));
    end;
```

You can find PASSWGEN.PAS on the companion CD-ROM
Before you can use the password in your programs, you must first SEARCH the file where you stored the password. Then, load the key, the password and the checksum (make certain to load in that order) as you implemented it in the program printed above.
Use the `Decrypt` procedure to decrypt the password. Use `Checksum_Ok` to check whether the password was modified.

If you need the first type of password, you will have to give the user the chance to select a proper password and enter it. This procedure is comparable to the one we just described. You store an empty password with it in the file, which the user can then overwrite. It's very important the password is difficult to locate and that it is not displayed during entry. This is a necessary protection against other users who "just happen" to be watching your screen as you enter the password.

You'll see a password query in action in the following program. The default password is ABACUS.

```pascal
program PASSWORD_TYPE1;
Uses Crt, Design;

var password : string;
    passwordcheck : boolean;

procedure Secret_readln(var s : string);
var c : char;
    li : integer;
begin
    repeat
        c := readkey;
        if c <> #8 then begin
            s := s + c;
            gotoxy(24,12);
            for li := 1 to length(s) do write('*');
            end else begin
            s := copy(s,1,length(s)-1);
            gotoxy(24,12);
            for li := 1 to length(s) do write('*');
            write(' ');
            gotoxy(wherex-1,wherey);
            end;
    until c = #13;
end;

procedure Change_password;
begin
    save_screen;
    window(10,8,60,7,' Change password ',black,7);
    writeln(13,10,'Please enter your new password');
    writeln(13,12,'Password: ');
    password := ''; 
    secret_readln(password);
    restore_screen;
    textcolor(7);
    textbackground(black);
end;

Function Gen_checksum(pasw : string) : word;
var sum : word;
    li : integer;
begin
    sum := 0;
    for li := 1 to ord(pasw[0]) do begin
```
sum := sum + ord(pasw[li]);
end;
Gen_checksum := sum;
end;

function Encrypt(pasw : string; add : char) : string;
var li : integer;
begin
  for li := 1 to 255 do begin
    pasw[li] := char(255 xor (ord(pasw[li]) + ord(add)));
  end;
  encrypt := pasw;
end;

procedure Save_password;
var pwf : file;
  key : char;
  check : word;
begin
  check := gen_checksum(password);
  password := encrypt(password, key);
  assign(pwf, 'password.dat');
  rewrite(pwf, 1);
  blockwrite(pwf, key, 1);
  blockwrite(pwf, password, 256);
  blockwrite(pwf, check, 2);
  close(pwf);
end;

function Decrypt(pasw : string; add : char) : string;
var li : integer;
begin
  for li := 1 to 255 do begin
    pasw[li] := char((255 xor ord(pasw[li])) - ord(add));
  end;
  decrypt := pasw;
end;

function Checksum_Ok(pasw : string; Key: char; sum : word) : boolean;
var tsum : word;
  li : integer;
  h : char;
begin
  tsum := 0;
  for li := 1 to ord(pasw[0]) do begin
    tsum := tsum + ord(pasw[li]);
  end;
  if sum = tsum then
    Checksum_Ok := true
  else
    Checksum_Ok := false;
end;
else
  Checksum_OK := false;
end;

procedure CheckPassword;
var pwf : file;
  key : char;
  check : word;
  should_pass : string;
begin
  assign(pwf,'password.dat');
  reset(pwf,1);
  blockread(pwf,key,1);
  blockread(pwf,should_pass,256);
  blockread(pwf,check,2);
  close(pwf);
  should_pass := decrypt(should_pass,key);
  if Checksum_OK(should_pass,key) and (should_pass = password) then
    PasswordCheck := true
  else
    PasswordCheck := false;
  end;
end;

procedure RespondPassword;
begin
  save_screen;
  window(10,8,40,7,'',black,7);
  If PasswordCheck then begin
    writexy(13,11,'Password correct - Access granted');
  end else begin
    writexy(13,11,'Password WRONG ! - No access !');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

procedure Menu;
var choice : byte;
begin
  repeat
    clrsr;
    writexy(10,1,'Example program for password type 1  (c) ''95 by ABACUS ');
    writexy(20,4,'M E N U');
    writexy(20,5,'~~~~~~~');
    writexy(15,6,'1)  Change Password');
    writexy(15,8,'2)  Query Password');
    writexy(15,10,'3)  End');
    writexy(15,13,'Your Choice: ');
    readln(choice);
    if choice = 1 then begin
      Change_password;
      Save_password;
    end;
    if choice = 2 then begin
      QueryPassword;
      CheckPassword;
      RespondPassword;
    end;
    until choice = 3;
end;
In menu box 1 you can enter a new password. A password query is simulated in menu box 2. Note how it is developed into three separate procedures. In your calling program, take precautions to call them at irregular intervals and not perform them one after another.

For this third type of password, you may be asked to select from a list of passwords. You frequently see this with games where you are asked for the x-th word on a certain page. Besides words, you may also be prompted for icons, colors, etc. The principle of the query is the same, only the input and output procedures change.

You'll see such a query implemented in the following example. You have the option under menu option 1 of entering a list of ten passwords with a fictional page number. Under menu option 2 you are asked to enter one of the passwords in a simulated password query. The program then randomly selects one of the ten passwords from the password file. In this example, there is no built-in encryption of the passwords due to space limitations. However, we do recommend you encrypt the passwords. Otherwise, the words can be easily found and modified.

```pascal
program PASSWORD_TYPE3;

Uses Crt, Design;

type PasswordType = record
  page : byte;
  text : string[20];
  end;

var pwords : array[1..10] of PasswordType;
pword : PasswordType;
password : string;
PasswordCheck : boolean;
Pw_index : byte;

procedure Enter_list;
var li : integer;
begin
  for li := 1 to 10 do begin
    save_screen;
    window(10,8,60,9,' Change password ',black,7);
    writexy(13,10,'Please enter password ');
    write(li,' of 10 of the list');
    writexy(13,12,'Page of the password in the manual: ');
    readln(pwords[li].page);
    writexy(13,14,'Password ');
    readln(pwords[li].text);
    restore_screen;
    textcolor(7);
    textbackground(black);
  end;
end;

procedure Save_list;
```

You can find PASSWD3.PAS on the companion CD-ROM
var pwf : file;
begin
  assign(pwf,'Passtyp3.DAT');
  rewrite(pwf,10*sizeof(PasswordType));
  blockwrite(pwf,pwords,1);
  close(pwf);
end;

procedure Load_password(Idx : byte);
var pwf : file;
begin
  assign(pwf,'Passtyp3.DAT');
  reset(pwf,1);
  seek(pwf,Idx*sizeof(PasswordType));
  blockread(pwf,pword,sizeof(PasswordType));
  close(pwf);
  save_screen;
  window(10,8,45,7,'',black,7);
  writexy(12,10,'Please enter the password on page ');
  write(pword.page,'');
  writexy(12,12,'Password: ');
  readln(password);
  restore_screen;
end;

procedure CheckPassword;
begin
  if password = pword.text then
    PasswordCheck := true
  else
    PasswordCheck := false;
end;

procedure RespondPassword;
begin
  save_screen;
  window(10,8,40,7,'',black,7);
  If PasswordCheck then begin
    writexy(13,11,'Password correct - Access granted');
  end else begin
    writexy(13,11,'Password WRONG ! - No Access !');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

procedure Menu;
var choice : byte;
begin
  repeat
    clrscr;
    writexy(10,1,'Example program for password type 3 (c) ''94 by DATA BECKER');
    writexy(20,4,'M E N U');
    writexy(20,5,'~~~~~~~');
    writexy(15,6,'1)  Enter password list');
    writexy(15,8,'2)  Password query');
    writexy(15,10,'3)  End');
    writexy(15,13,'Your choice: ');
    readln(choice);
  until choice = 3;
end;
Modern Copy Protection

Chapter 8

High-level Language Programs At Machine Level

Now that you are familiar with some of the ways passwords are implemented in Pascal, we'll move on to machine-oriented programming in this section. More secure queries are possible using machine oriented programming because portions of a Pascal program are easily recognized. That is not the fault of your programming technique but to a few significant procedures used by the Pascal compiler.

Pascal program structure

Let's take a look first at this simple program:

```pascal
program LITTLE_PROGRAM;
Uses Crt;
var i : integer;
begin
  clrscr;
  for i := 5 downto 1 do begin
    gotoxy(10,6);
    writeln('Please wait ',i,' second(s)');
    delay(1000);
  end;
end.
```

If we use Turbo Debugger to disassemble this program, various parts of the Pascal program are easily recognized. The program in the beginning calls the initialization procedures for the included units. In this case, these are SYSTEM.TPU and CRT.TPU.

LITTLE_PROGRAM.7: begin
  cs:001C 9A0000D862 call 62D8:0000
  cs:0021 9A0D007662 call 6276:000D
  cs:0026 55 push bp
  cs:0027 89E5 mov bp,sp
```
Then, the procedure `clrsc` for the unit `Crt` is called. If you have compiled the program without comments for an external debugger, you’ll find a far address instead of the CRT.CLRSCR symbol, as in the case of initializations.

```
LITTLE_PROGRAM.8:  clrsc;
    cs:0029 9ACCO17662   call far CRT.CLRSCR
```

The next step is the initialization of the loop. The variable `i` is found again as the memory location `[0052h]` in the data segment. The value 5 is assigned to it. The assigned value is correct during the first through the loop. Then the program jumps to Pascal program line 10. In each additional pass through the loop `[0052h]`, `i` is decremented by 1.

```
LITTLE_PROGRAM.9:  for i := 5 downto 1 do begin
    cs:002E C70652000500   mov word ptr [0052],0005
    cs:0034 EB04           jmp LITTLE_PROGRAM.10 (003A)
    cs:0036 FF0E5200       dec word ptr [0052]
```

The call to procedure `gotoxy` is next in the loop. It’s given the x and the y coordinates to which the procedure is to jump. For that purpose, 000Ah for the x-coordinate and 0006h for the y-coordinate are pushed onto the stack. The procedure is called using a `far` call.

```
LITTLE_PROGRAM.10:  gotoxy(10,6);
    cs:003A 6A0A           push 000A
    cs:003C 6A06           push 0006
    cs:003E 9A1F027662     call far CRT.GOTOXY
```

The next line includes the output of the message of procedure `writeln`. It has the peculiarity that several arguments can be passed to it, i.e., that you can output either a string or even a combination of strings or variables. It’s clear how this is translated by the Pascal compiler in this example: A separate procedure is called for each argument passed to it.

A distinction must be made there between three different types. First, procedure `0615h` of the SYSTEM.TPU outputs a text string. Its address on the stack, and not the string to be displayed, is saved.

```
LITTLE_PROGRAM.11:  writeln('Please wait ',i,' second(s)');
    cs:0043 BF6801         mov di,0168
    cs:0046 1E             push ds
    cs:0047 57             push di
    cs:0048 BF0000         mov di,0000
    cs:004B 0E             push cs
    cs:004C 57             push di
    cs:004D 6A00           push 0000
    cs:004F 9A1506D862     call 62D8:0615
```

Next is the return of the number `[0052h]`. It’s copied into the ax:dx register and passed to procedure `06D9h` of SYSTEM.TPU. Finally, there is the procedure for generating the line feed. Its inclusion represents the only difference regarding the write procedure. It’s found at position 0582h of SYSTEM.TPU.
The call to the procedure `Delay` follows the display of the message in the loop. The number of milliseconds to wait is saved on the stack. In this case, 03E8h (1000 decimal) milliseconds.

A check for the end of the loop is performed and if not complete control jumps back to the beginning of the loop and continue decrementing the [0052h] variable.

However, if the ending value is reached, the program arrives at the last program line. There, the stack frame is first removed. Finally, the ax register containing the "result value" of the program is removed and the termination procedure 0116h of the SYSTEM.TPU is called.

You have seen that several significant procedures are already found in this small program. Because these procedures are present, an .EXE file is easily identified as one generated by Turbo Pascal. Besides the unit initialization, a procedure like `DELAY` from the CRT unit is also of particular interest. Let's look closer at the procedure `DELAY`:

First, the ms counter is loaded in cx. Therefore, when
appears at this point, only one more pass would be made through the loop. Finally, program execution will cycle through the loop until the specified number of ms has lapsed.

In the case of important procedures like wait loops or password queries, it’s best to avoid using reserved Pascal procedures. Instead, we recommend developing original and more specialized procedures by building on the available procedures. Some command scanners can easily be tricked by rearranging the sequence of commands and a few little modifications.

However, we want to avoid using `DELAY` for a wait procedure and develop our own small wait routine. It’s as effective as `DELAY` but isn’t likely to be recognized by anyone looking to break your program.

The solution we’re talking about uses the timer interrupt for its own routine. To make it possible to work precisely, the timer interrupt is reprogrammed to 1,000 calls per second. We’ll use the DOS routines for interrupt handling. These routines will allow us to manage the program better.

```pascal
program WAIT;
Uses Crt, Dos;
var oldtimer : pointer;
    mscount : longint;
    msready : boolean;
    i        : integer;
procedure WaitInt; Interrupt;
begin
    dec(mscount);
    if mscount = 0 then msready := true;
    port [$20] := $20;
end;
procedure Wait1(ms : longint);
begin
    GetIntVec(8,oldtimer);
    SetIntVec(8,@waitint);
    asm
    cli
    mov dx,43h
    mov al,36h
    out dx,al
    sub dx,3
    mov al,169
    out dx,al
    mov al,4
    out dx,al
    sti
    msready := false;
    mscount := ms;
    repeat until msready;
    asm
    cli
    mov dx,43h
    mov al,36h
    out dx,al
    sub dx,3
    xor ax,ax
    out dx,al
    out dx,al
```
A Protected Password Query

The method for a protected password query we are introducing in this section is not the best protection imaginable. Nevertheless, it can prevent would-be hackers from deciphering the password query. This version stores the passwords and related data as constants. This should be replaced by one of the methods which we'll introduce when used in your own programs. We use them here only for clarity.

Instead of creating insurmountable obstacles for hackers, we will use several little tricks in the program. It all starts with the MEM command. It's not implemented (we really don't need much memory) but is included to prevent the program from being debugged with the normal Turbo Debugger.

This trap can of course be avoided by appropriately reducing the value in the .EXE header, offset 0Ah, to 0D430h for 200,000 bytes for example. The program can then be debugged with the normal Turbo Debugger, although further refinements await the intruder. The main loop, which asks for the password, is kept in assembler.

The first thing we do in this loop is switch off the keyboard in similar fashion to a hardware lock. This triggers an INT 3h interrupt. This interrupt will have no effect if a debugger was not installed. However, if Turbo Debugger is running in the background, there will be a program break at this point. This is similar to the case for the normal Turbo Debugger because it's no longer possible to manipulate the program with the keyboard turned off. At best, Turbo Debugger can still be used with a mouse, but even the mouse interrupt 33h can be switched off.

If would-be hackers should overcome this obstacle, they'll encounter the PIQ trick that we will introduce in the next section. The main reason we have included the unnecessary jumps is to cause further confusion. Should the would-be hacker overcome all these obstacles, our only hope is that they're discouraged by the prospect of untangling the entire mess not only once, but several times. If that does not stop them, we're probably no longer dealing with a would-be hacker but a professional hacker. There may not be an appropriate defense for a professional hacker.

Back to our starting point: The memory command. Our hacker will use Turbo Debugger 386 if they don't know how to decipher it. Besides a small memory requirement, this also offers the advantage of a complete virtualization of the PC. This means our interrupt misdirection and switching off the keyboard tricks occur on the virtual computer and will have little effect on Turbo Debugger.
However, protected mode is one small detail which Turbo Debugger 386 does not recognize. We, therefore, deliberately switch into protected mode at the most dissimilar areas in the program and then switch back again immediately. Although this has no effect on our program, the Turbo Debugger 386 terminates with an Exception Error.

The indirect procedure calls present a final difficulty. The procedures of the Pascal program are not called with their true names, but by means of a pointer containing the address of the procedure. This makes reading the program several times more difficult.

Here, now, is the source code for the password program:

```
program passwordquery;

uses crt, design;

const Passwords : array[1..10] of string =
('Abacus','Underground','Graphics','Sprites',
 'Dimension','Protection','Mode X','Assembly','Glass','VGA');
Pw_Pages : array[1..10] of word =
(17,47,94,158,167,211,89,29,181,35);

Var pw_no : byte;
remaining_passes : byte;
Password_correct : word;
New_Pass : string;
PNew_Password_select : pointer;
PInput_Box_draw : pointer;
PPassword_query : pointer;
PSystem_stop : pointer;
unnecessary_Variable1 : word;
unnecessary_Variable2 : word;

procedure Query_Loop; far; external;

procedure New_Password_select;
begin
  pw_no := random(10)+1;
  unnecessary_Variable1 := 1;
  unnecessary_Variable2 := 2;
end;

procedure Input_Box_draw;
var pws : string;
begin
  str(Pw_Pages[pw_no]:2,pws);
  asm int 3; end;
Window(6,10,52,6,'Please type the password on page '+pws+' of the manual',black,7);
  unnecessary_Variable1 := 1;
  unnecessary_Variable2 := 2;
gotoxy(23,12);
end;

procedure Password_query;
begin
  readln(New_Pass);
  unnecessary_Variable1 := 1;
```

You can find QUERY.PAS on the companion CD-ROM
unnecessary_Variable2 := 2;
if New_Pass = Passwords[pw_no] then
  Password_correct := 1
else
  Password_correct := 0;
end;

procedure System_stop;
begin;
  textbackground(black);
  textcolor(7);
c1rscr;
  writeln('We probably would have been better off buying an original ...');
halt(0);
end;

procedure Main_Program;
begin;
  textbackground(black);
  textcolor(7);
c1rscr;
  gotoxy(20,12);
  writeln('Welcome to the main program !');
  gotoxy(20,22);
  write('Enter to exit ... ');
  readln;
  halt(0);
end;

begin;
  textbackground(black);
  textcolor(7);
c1rscr;
  remaining_passes := 57;
PNew_Password_select := @New_Password_select;
PInput_Box_draw := @Input_Box_draw;
PPassword_query := @Password_query;
PSystem_stop := @System_stop;
randomize;
Query_Loop:
  end.

The following program listing is from PWMODUL.ASM.

```assembly
.P386p
.MODEL TPASCAL

keyb_off macro
  push ax
  in  al,21h
  or  al,02h
  out 21h,al
  pop ax
endm

keyb_on macro
  push ax
  in  al,21h
  and  al,0Fdh
  out 21h,al
endm
```

You can find PWMODUL.ASM on the companion CD-ROM.
pop ax
endm

.DATA
extrn remaining_passes
extrn pnew_password_select : dword
extrn pinput_box_draw : dword
extrn ppassword_query : dword
extrn psystem_stop : dword
extrn password_correct : byte
extrn unnecessary_variable1 : word
extrn unnecessary_variable2 : word

.CODE
extrn main_program : far
public query_loop

query_loop proc pascal
keyb_off

;PIQ - Trick
int 3
mov cs:word ptr [@int_21_funct1],4CB4h ; function end prg.
@int_21_funct1:
  mov ah,30h ; function get DOS vers.
  int 21h

@query_loop:
keyb_off

call dword ptr pnew_password_select
cmp unnecessary_variable1,5
jbe @unnecessary_destination1a

;PIQ - Trick
int 3
mov cs:word ptr [@int_21_funct2],4CB4h ; function end prg.
@int_21_funct2:
  mov ah,30h ; function get DOS vers.
  int 21h
  mov cs:word ptr [@int_21_funct2],30B4h ; function end prg.
call dword ptr pinput_box_draw
jmp @unnecessary_destination1b
@unnecessary_destination1a:

;PIQ - Trick
int 3
mov cs:word ptr [@int_21_funct2],4CB4h ; function end prg.
@int_21_funct2a:
  mov ah,30h ; function get DOS vers.
  int 21h
  mov cs:word ptr [@int_21_funct2a],30B4h ; function end prg.
call dword ptr pinput_box_draw
@unnecessary_destination1b:
keyb_on

cmp unnecessary_variable2,10
jbe @unnecessary_destination2a
dec byte ptr remaining_passes

; Protected MODE Trick
pusha
cli ; disable interrupts
mov eax,cr0 ; switch to Protected mode
or eax,1
mov cr0,eax
jmp PROTECTION_ENABLED ; clear execution pipe
PROTECTION_ENABLED:
and al,0FEh ; switch back to Real mode
mov cr0,eax
jmp PROTECTION_DISABLED
PROTECTION_DISABLED:
sti ; enable interrupts again
popa

call dword ptr ppassword_query
jmp @unnecessary_destination2b

@unnecessary_destination2a:

dec byte ptr remaining_passes

; Protected MODE Trick
pusha
cli ; disable interrupts
mov eax,cr0 ; switch to Protected mode
or eax,1
mov cr0,eax
jmp PROTECTION_ENABLED2a ; clear execution pipe
PROTECTION_ENABLED2a:
and al,0FEh ; switch back to Real mode
mov cr0,eax
jmp PROTECTION_DISABLED2a
PROTECTION_DISABLED2a:
sti ; enable interrupts again
popa

call dword ptr ppassword_query

@unnecessary_destination2b:

cmp byte ptr password_correct,1
je @query_was_OK
jmp @query_was_not_OK
@query_was_OK:
call main_program

@query_was_not_OK:

cmp byte ptr remaining_passes,54
ja @query_loop

call dword ptr psystem_stop

ret

query_loop endp

END
Most experts agree using a dongle is currently the best protection against illegal copying. However, using a dongle to protect programs is impractical. The dongle, if it is to provide reliable protection, could cost several times more than the program; low cost dongles provide only minimal security.

A more reliable method is using a CD-ROM. The CD offers a certain degree of security for at least two reasons:

1. Although it's possible to write directly to a CD, the hardware required is quite expensive. Unless the program is illegally copied by a professional, the CD offers a certain degree of security.

2. An enormous quantity of data can be stored on one CD-ROM. Most users will think twice about copying 150 Meg onto their hard disk for only one program.

The disadvantage of using a CD-ROM is that many users still do not have a CD-ROM drive. Therefore, your program cannot be used by those who do not have a CD-ROM drive.

Therefore, if using a dongle or a CD-ROM is not possible, you have only one more possibility: You must build copy protection into your program. Remember, there is no copy protection that a dedicated hacker cannot defeat. However, you can make a hacker's life much more difficult.

The protection tricks we'll talk about in this chapter become more important in this age of rising economic crime. No one likes to see their ideas and technology stolen.

The tricks we'll talk about probably will not protect your programs from the professional hacker, but they will frustrate the would-be debugger expert. You will also find these tricks used in many professional programs.

Dissecting Programs

There are many tools that can be used to debug. We'll introduce a few of the most popular ones to you in this section.

Your first thought when we mention “tools” is probably programs like PC Tools or the Norton Utilities. However, we are not talking about this type of program here. Instead, we'll discuss several useful programs that will make your life with your computer easier. By the time you finish reading this section, you'll be familiar with the following types of programs:
Chapter 9

Protect Your Know-how: Protection Tricks

- Compression/decompression programs
- Debugging programs
- Hex editors

Removing the camouflage: Compression/decompression programs

Using compression/decompression programs like PKZIP or WinZip, you can compress files so they do not occupy as much space on the hard drive. Many compression programs compress .EXE and .COM files into self-extracting EXE files or COM files that decompress automatically when they are run and do not require a separate decompression program.

The only area of caution with compression programs is the possibility of a virus. If you compress a file that is infected with a virus, the virus itself is also compressed with the program and cannot be recognized by virus detection software.

We cannot discuss all the specific compression/decompression programs here. For more in-depth information on compression/decompression programs see PKZIP, LHARC & Co. from Abacus (6th printing).

Debugging programs: Turbo Debugger

Turbo Debugger, bundled by Borland with C(++) and Pascal, is one of the most popular debugging programs. Using Turbo Debugger, you can step through both your source code and .EXE or .COM files. You are also able to set breakpoints with Turbo Debugger. You can allow your program to execute to these points and then, from this point onward, work through the program one step at a time.

In the following paragraphs we'll explain how you can analyze a program with this powerful tool and monitor specific variables. However, if you need more detailed information on Turbo Debugger, refer to any of the several books available for either Borland C(++) or Borland Pascal.

Running Turbo Debugger

First, you must determine which version of Turbo Debugger you would like to use. You can use the standard Turbo Debugger. This works with any memory manager, but also requires a large amount of memory. Alternatively, you can also use Turbo Debugger386. However, for this debugger you have to enter the following line in your CONFIG.SYS:

```
Device = TDN386.SYS
```

The driver is not compatible with EMM386 and QEMM. So, if your system requires EMS memory, you will not be able to use this debugger. However, Turbo Debugger can be used to debug even very large, memory-intensive programs. If you can manage without EMS memory, Turbo Debugger386 is the best choice to use.
Command line parameters

You can configure Turbo Debugger individually with each session by using command line parameters. There are, however, several options that you will need each time. You can save them in a configuration file. Create this file using Turbo Debugger’s Option/Save Options command. Give the configuration file a name that you can easily remember, for example, MYCONFIG.TD. Later, use the command line parameter -c to load the file directly each time you start Turbo Debugger. The call could then look similar to the following:

```
TD386 -cMYCONFIG.TD the prg
```

where the `prg` represents the program to be debugged.

Additionally, there are many other options that you can select:

- `-d` - screen conversion
- `-do` - use two screens

Select the option with a Hercules video card that has two monitor outputs. The debugger output is displayed on the Hercules monitor and the normal output on the VGA monitor. The advantage of this arrangement is that you are always able to see just what your program displays on the monitor.

- `-dp` - page-flipping (particularly for text mode)

By selecting this option, Turbo Debugger uses two virtual screen pages. This option works only in text mode and only if the program to be debugged uses a single screen page and does not change the starting address of the video RAM. When debugging programs in graphics mode, you’ll encounter many graphics errors if you switch between the graphics screen and the debugger screen. These errors are usually tolerable; because this mode is the quickest, it is the one preferred.

- `-ds` - screen swapping

By selecting this option, Turbo Debugger saves the screen content of the program to be debugged in a buffer before restoring the debugger screen. It then restores it again following user entries. However, this swapping takes time and is therefore rather awkward when you are debugging a program step-by-step. Select this mode only if you really need it.

- `-k` - keyboard record

This is a useful option. It records all user entries in the program to be debugged in a file. This lets you return later to a given point in a program if you have debugged “too far”.

- `-l` - start in assembler mode

When you start the debugger in assembler mode, the CPU window is displayed on start-up instead of the source code. The procedures executed on start-up are not performed automatically. This means that you can also view the loading process of the program.

- `-m` - set heap size

Turbo Debugger uses an 18K heap for internal purposes. However, if you then find yourself seriously low on memory, you can use this parameter to reduce the size of the heap to as low as 7K. The size of the memory to be used is expressed in kilobytes (K) directly following `-m`. 
Chapter 9

Protect Your Know-how: Protection Tricks

-v - video RAM setting
-vg - backup entire video RAM

This option always saves the entire 8K of video RAM. Additional 8K screen RAMs are, of course, used in this case, but you can debug programs that use this memory and whose output would otherwise be erased.

-vn - no 50 line mode

If you're certain the 50-line mode is not needed, use this parameter. This will help free more memory.

-vp - backup color palette

Many programs modify the VGA color palette. For these programs you should use -vp. This saves the palette, and you'll likewise always have the correct colors when you step through a program.

-y - set TD overlay size

Turbo Debugger swaps part of itself using overlays. Depending upon how much memory is needed by the program to be debugged, you can adjust the size of this overlay. If your program is quite memory intensive, select the minimal request using -y20. Although you will now have much more free memory, you will paying for it with correspondingly longer loading times. If the program to be debugged needs less memory, you can select the maximum memory size of 200K with -y200. This keeps the entire debugger in memory.

Targeted searches with Turbo Debugger

In this book we're not concerned with how Turbo Debugger debugs a program for which the source code is available. Instead, our interest is in finding certain variables or commands in the program to be debugged.

In the following paragraphs we'll see how you can, for example, find the variables for "lives" or "points" in an .EXE file in the following small Pascal program:

```pascal
{$M 16384,0,655360}
program SHOW_HOW_TO_DEBUG;
uses Crt;
var inputvar : word;
lives     : byte;
factor   : real;
procedure Init_variables;
begin
  lives   := 4;
  factor := 0.27
end;
procedure User_input;
begin
  textcolor(14);
gotoxy(10,3);
  write('                                                     ');
  gotoxy(10,3);
  write('Please enter your new test value : ');
  readln(inputvar);
end;
```
procedure Nonsense_evaluation;
begin
   if (inputvar * factor) < 10 then 
      dec(lives);
end;

procedure Write_status;
begin
   textcolor(15);
   gotoxy(10,10);
   write('Input : ',inputvar:5,'      ==>    Lives : ',lives:3);
end;

begin
   clrscr;
   Init_variables;
   repeat
      User_input;
      Nonesense_evaluation;
      Write_status;
      until lives = 0;
end.

The program inputs a value from the user. If the value is smaller than 38, the user loses a "life". This continues until the user has no more lives left.

After you have compiled the program (you will find it on the companion CD-ROM), you can start Turbo Debugger. Type the following:

TD td_test

and you’ll see the following screen:

![Image of Turbo Debugger interface]

The initialization procedures of the units are at the top of the program. Press [F8] to move through the program step-by-step. You’ll then encounter the calls to the individual subroutines. Instead of following these procedures, let’s try to determine their function.
You’ll see that you’re in a loop in which user input is first requested. The meaning of the next procedure is not clear yet. The third one echoes the value entered and displays the lives remaining on the screen. Finally, there is a very interesting comparison. The [0054] variable is checked for a value of zero. If it is not zero, another pass is made through the loop.

Let’s take a closer look at this important variable using the Watch window where specific memory areas can be checked.

Now we’ll continue through the program. You’ll see the variable has the same value as the number of “lives” in the program. Therefore, this is the “lives” variable. The significance of the second procedure now becomes clear. If a value less than 38 is entered, the variable is then decremented.

Now we know where we can find the “fellow” who counts down the number of lives. The next time you get to the second procedure, don’t pass over it again, but go through it one step at a time. To do that, press the
When you hit the \( \text{F7} \) key when you’re at that procedure. You will now be in the subroutine. Step through it again with \( \text{F8} \).

Our “fellow” who decrements the lives variable is near the end of the procedure, specifically:

\[
\text{dec byte ptr[0054]}
\]

To check whether \texttt{lives} is really decremented here, you can overwrite the 5 bytes of the decrement instruction with 5 NOPs (no operation). To enter a different instruction, press the \( \text{ [Spacebar] } \) and enter the instruction. Note the length of the previous instruction must coincide with the length of the new instruction.

At the very least, you’ll confirm the variable [0054] is really the variable lives for which you are searching. If you’re writing a game trainer, you can do one of the following:
Chapter 9

Protect Your Know-how: Protection Tricks

1. Change the memory location where the variable is stored.

or

2. Change the memory location of the place in your computer where variable \texttt{lives} is decremented.

Turbo Debugger has offers many other functions besides watches and stepping through a program. These include stopping execution of the program at various points or executing the program to a specific point and other search and jump functions.

\textbf{F2} - set breakpoint

You can set a breakpoint with the \texttt{F2} key. When the program reaches a breakpoint, control is given to Turbo Debugger so you can inspect memory, variables, etc. Move to the location in the program where you would like to insert a breakpoint and press the \texttt{F2} key.

\textbf{Alt+F2} - setting a breakpoint at a specific location

By using the \textbf{Alt}+\textbf{F2} key combination, you can set a breakpoint at a specific memory location. To do this, enter the segment and the offset position of the desired breakpoint. This does not have to be in the program just loaded but can be anywhere in memory. This is very useful whenever you want to, for example, debug interrupts or drivers.

\textbf{F9} - run program

To start the loaded program, simply press the \texttt{F9} key. You can usually interrupt the program to be debugged at a “critical point” with \texttt{Ctrl-Break} and then debug step-by-step.

\textbf{F4} - execute up to current position

When you happen to find yourself at a certain point in the program, and want to test program execution again to this point, you don't have to debug your way through the entire program again to this point, step-by-step. Instead, you can reset it with \texttt{Ctrl}+\texttt{F2} and cause the program to execute to your current position in the same with \texttt{F4}.

\textbf{Alt+F9} - execution up to a location in memory

This key combination is similar to the \textbf{F4} mentioned above. You enter the location (segment and offset) in the program where the program (starting from the current position) is to be executed.

\textbf{Alt+F5} - display user screen

When debugging, you should be able to check which outputs were last displayed on the screen or at which entry the program was stopped by a breakpoint. An application running in graphics mode is switched automatically to this mode. If your program is using Mode X, you'll probably see some rather bizarre graphics. This is because reprogramming of a few registers in the VGA card is ignored.

\textbf{Ctrl+S} - search for an expression

Whenever you're searching for a specific statement in the program to be examined, you enter the search criteria using \texttt{Ctrl}+\texttt{S}. Enter the assembler statement for which you are searching in the input window which appears, for example:
Turbo Debugger then searches from the current position in the program for these instructions. Be careful with short commands: It’s possible that a part of a statement is interpreted as this instruction. In this case, you should scroll up and down for a few lines to check whether the statement found is really the one you are searching for or part of a different one.

- go to place

Turbo Debugger shows the current position in RAM to the far left in the CPU window. Note this position when you have found an interesting place in the program. You can jump to it at any time using the keys. This makes testing the program at various points easier.

- return to procedure to be called

If you have jumped into a procedure, and it appears that what you are searching for is not there, you do not have to step through all the way to the end. Instead, you can use to return to the procedure call and then skip it with.

Now that you are familiar with the basic functions of the Turbo Debugger, you should be able to analyze any program you like. Perhaps you would like to try your luck with RAIDER from the companion CD-ROM as an example for trainer programming. You can debug and change this program according to your own ideas.

Not hexing: Hex editors

A hex editor is not used like a text editor to edit ASCII text files. Instead, it’s used to address the numeric code that represent machine language instructions and data. These numeric values are entered in hexadecimal form, hence the name hex editor.

Why a hex editor?

A hex editor lets you edit application files and other files that cannot be accessed by a text editor. Unlike text editors, hex editors let you edit actual program code. The best examples of such files can be found in entertainment software where you have discovered yourself short of money, points, weapons, etc. You can save the current game and look at the program code using a hex editor.

For example, in a business simulation game, note how much money you have before you save the current game. Convert this sum into a hexadecimal number (i.e., 1000 gold pieces would equal03E8h gold pieces). The CPU saves a number, but usually the least significant bytes first followed by its most significant bytes. Therefore, you have to reverse the sequence. You now haveE803h. Now use the hex editor to search for this number. When you have located E803h, simply replace it with, for example, 9999h and restart the game. When you reload the game and have found the right place, you can then enjoy your 39,321 gold pieces.

If you see a different amount, search again using the hex editor to see whether E803h turns up somewhere else. In this way, you can also increase lives, ammunition, extra weapons, etc. You can in this way search for and replace different byte combinations. What is required, of course, is a good hex editor.
The Hexcalibur hex editor

Using the Hexcalibur editor, you not only can overwrite data, but you can also insert, remove and copy data. To start the editor, you must also enter the name of the file to be edited, for example:

Press any key and you’ll be in the editor. The editor defaults to the insert mode. To edit a saved game files, activate the overwrite mode by pressing the \( i \) key. You move from one file location to another using the arrow keys. Switch between the hexadecimal window on the left and the ASCII window on the right by pressing the \( F_2 \) key.

The following is an overview of the defined keys:

- **AWSZ**
  - You can use the arrow keys to move from one file location to another within the respective window.

- **Pg Up / Pg Do**
  - Use these keys to jump one "page" up or down. One page represents 256 characters.

- **Home**
  - Press this key to move to the beginning of the current line.

- **End**
  - Press this key to move to the end of the current line.

- **A** 0 or **A** 1
  - Press one of these two key combinations to move to the beginning of the file.

- **A** 1 to **A** 9
  - Press one of these key combinations to move to approximate positions within a file. For example, press **A** 1 + 1 to jump to the position representing 10% of the file, press **A** 1 + 2 to jump to the position representing 20% of the file, etc.

- **A** E
  - Press one of these key combinations to move to the end of the file.

- **A** G
  - Press this key combination to jump to a specific position on the disk. To do this, enter the desired sector and the number of the first byte and then press **Enter**.

- **F1**
  - Displays a summary of the available editor functions.

- **F2**
  - Switches between the hexadecimal window and the ASCII window.

- **Ins**
  - Switches between the insert mode and overwrite mode.
Deletes the character beneath the cursor. The character is in both the ASCII window and in the hexadecimal window. Caution: The loss of just this one character can have fatal consequences in program files.

Press this key combination to highlight the beginning of a block with which you would like to carry out one of the following operations.

Copies the highlighted block.

Moves the highlighted block.

Removes the highlighted block.

Selects the highlighted block for insertion.

Press this key combination to find text. Simply press press and enter the text for which you want to search. The search begins from the current position in the file.

Press this key combination to replace one character string with a different one. The character strings do not need to have the same length.

Press this key combination to save an edited file.

Press this key combination to exit the program. You'll be asked if you want to save any changes to the file.

Press to cancel all changes since the last time the file was saved.

Editing a saved game with Hexcalibur

How do you edit a saved game with Hexcalibur?

Let's assume you would like to edit the file called SAVEGAME.001. You proceed as follows:

First, start the editor with HC SAVEGAME.001 which will then load the saved game into the editor. Now look for the value that needs to be changed. For example, if you had 1,234 gold pieces, first press enter D2 04 and press . HC will now search for this byte sequence. When the editor finds it, it marks the found area red and ends the search.
This is the location which you need to change. Enter the hexadecimal value of the desired sum in the hex window, e.g., 3075 for 30000. Now press $$\text{Alt} + \text{S}$$ to save the changes. Confirm the prompt, and the program saves the modified version. HC creates a backup copy using the .BAK extension by default.

You’re finished with the task and can exit the program by pressing $$\text{Alt} + \text{X}$$. If the first occurrence of the byte sequence is not the desired value, then rename the backup copy of the file to the original filename and search for another occurrence of the value. To do this, press $$\text{Alt} + \text{F}$$ after Hexcalibur found the first value.

The Debug Interrupts

To protect your own programs from being themselves debugged, it helps to know how a debugger works. Depending on the type of debugger, it will use the same virtual machine as the program being debugged or will emulate a virtual CPU. All debuggers use interrupt 3 (Debug interrupt). If this interrupt is called, a debugger is switched on which may run in the background. Of course, if no interrupt is set, the program will continue to run.

We now introduce a neat little trick which we’ll build into our program: A few calls to Int 3 which appear unconnected and in random locations in the program will help frustrate our would-be hackers.

Masking interrupts

It’s definitely more effective to mask Int 3 using hardware such as the keyboard. This is done through the interrupt controller (port 20h), bit 1, which has interrupt 09h. If we now simply block the channel for this interrupt, the routine associated with the interrupt will not be called, however. In fact, often the interrupt itself may be called. Unfortunately, the debug interrupt has one small disadvantage. Since it has a value less than 8, it’s called an NMI (nonmaskable interrupt). Simply masking the interrupt won’t work. However, masking is a simple method for disabling serial ports, for example, which, of course, can mean that without a mouse many applications simply can’t be used.

Changing the debug interrupt

A further option for making the interrupts useful is to point them to your own routine. You can, for example, swap an important routine which isn’t call too often, but which is required for program execution, into a procedure which you declare as an interrupt. You then coerce interrupt 3 to your own routine and call interrupt 3. This method works only for debuggers which do not create a virtual CPU with its own interrupt table. You cannot use Turbo Debugger for such a task.

Hiding data in an interrupt vector

Another way to protect a program is by hiding data in an unused interrupt vector. Each vector is a DWORD. You can hide a checksum or special code here that you later test at runtime to ensure no one has tampered with your program.

The interrupt table is arranged so a given interrupt is located at 0:4*interrupt number. The address of the standard interrupt handler is located here. After your program is finished, you must restore the original interrupt vector.
You can also directly modify the interrupt vector table. Many commercial products do this. For example, many sound cards do not modify the SoundBlaster interrupt by using functions 25h and 35h of Int 21h, but by directly accessing the vector table. These tricks, however, only protect against debugging if the debugger does not use a virtual machine.

The following is an assembler example of this method in NODEB.ASM. The 64-bit value is saved by Check_for_vector in the vector table at the position for Int 3.

```
public Check_for_vector
Check_for_vector proc pascal check : dword;
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, es:[bx]
  mov oldint3, eax
  mov eax, check
  mov es:[bx], eax
  ret
Check_for_vector endp
```

Vector_ok lets you check whether a certain longint value is saved at the position of Int 3 in the vector table. A 1 or TRUE is returned if the value in the table agrees with the value returned, otherwise you get 0 or FALSE.

```
public Vector_ok
Vector_ok proc pascal check : dword;
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, es:[bx]
  cmp eax, check
  je @check_ok
  mov al, 0
  jmp @check_end
@check_ok:
  mov al, 1
@check_end:
  ret
Vector_ok endp
```

Procedure Restore_checkvector restores the changed Int 3 to its original state. You should execute this procedure before ending the program, if you have modified the vector.

```
public restore_Checkvector
restore_Checkvector proc pascal
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, oldint3
  mov es:[bx], eax
  ret
restore_Checkvector endp
```
Protect Your Know-how: Protection Tricks

Chapter 9

Substituting interrupts

A last trick with interrupts is to substitute one interrupt for another. This means that instead of calling interrupt 21h, you call the single-step interrupt (01h) or debug interrupt (03h). This method is useful when you program in assembler.

First, determine the address of Int 21 handler and then save it in the desired vector. Now, when you call Int 21h, write the number of the appropriate vector instead of 21h. You should have no problems programming it because you can take care of replacing the interrupt when your program is completed. However, this method may not work every time and it reduces the legibility of the code significantly.

An example of how you can substitute interrupt 21h for int 3h is found in assembler procedure Copy_int21_int3. The old interrupt is saved in the variable old_interrupt3 and must be restored before terminating the program.

```
public Copy_int21_int3
Copy_int21_int3 proc pascal
    mov bx,0
    mov es,bx
    mov bx,18
    mov eax,es:[bx]
    mov old_interrupt3,eax          ; store old int3
    mov bx,84                       ; load int 21
    mov eax,es:[bx]
    mov bx,18                       ; store in int3
    mov es:[bx],eax
    ret
Copy_int21_int3 endp
```

Fooling The Debugger

Another approach to protection is to interfere with the debugger intentionally.

The memory trick

The simplest method is a memory trick. You reserve all the available memory and not just the amount of memory required by your program. You can do this easily in Turbo Pascal with the following command:

```
{8M 800,550000,65000}
```

If you now try to debug this program using Turbo Debugger, you'll get an error message saying there is not enough memory available to run the program. If you set the number too high, you may even cause Turbo Debugger386 to quit with the same error message. However, in any case, don't push Turbo Debugger386 too far. Remember, not all computers are configured with 630K available. You'll never know if your program will still run on all configurations of computers.

Ambiguous instructions

Another option for fooling the debugger is using ambiguous instructions, for example, a jump to the middle of another instruction. Most debuggers become disoriented with such an instruction and jump to the next one which can be recognized.
The trick, then, is to write a DOS interrupt instruction for ending the program at the next statement. If the debugger operates step-by-step, it will stop at this point. In normal operation, however, it will jump to the instruction addressed and find there is another jump using the end program DOS interrupt 21h subfunction 4Ch.

Here's an example of such a program:

```assembly
No_stepping proc near
  push ax
  jmp @Nostep+2
@Nostep:
  mov ds:byte ptr [06EBh],00
  mov ax,4C01h
  int 21h
  pop ax
  ret
No_stepping endp
```

### Stopping TD386

The methods we've talked about so far have all been designed to protect against prying eyes using Turbo Debugger. However, Turbo Debugger386 uses hardware mechanisms which can defeat these protection schemes. Fortunately, there is another method we can use: Turbo Debugger386 cannot tolerate protected mode.

The trick we'll use is only a few lines long. We won't be concerned with the entire way protected mode operates; we need only switch to protected mode and then immediately back again. We do this by using the cr0 register.

Now for a procedure which switches you to protected mode and then back again immediately into real mode. It runs with the EMS driver installed.

```assembly
public protected_stopping
protected_stopping proc pascal
  pusha
  cli                      ; disable interrupts
  mov eax,cr0             ; switch to protected mode
  or eax,1
  mov cr0,eax
  jmp PROTECTION_ENABLED  ; clear execution pipe
PROTECTION_ENABLED:
  and al,0FEh               ; switch back to real mode
  mov cr0,eax               ; don't reset CPU
  jmp PROTECTION_DISABLED   ; clear execution pipe
PROTECTION_DISABLED:
  sti                       ; enable interrupts again
  popa
  ret
protected_stopping endp
```

This procedure is from the NODEB.ASM file on the companion CD-ROM.
Self-modifying Programs

Another interesting and widely used way to protect a program is to write one that modifies itself. A self-modifying program changes its code during execution so it no longer looks like it did when it was first loaded into memory.

Checksums

If you’ve used a modem, then you’re probably familiar with checksums. A checksum is a method used to ensure that a file has been transmitted error-free. A checksum can detect missing or erroneous data. They serve, for example, to check whether a file has the correct size or contains the correct data. A checksum is basically a value used to make certain that data is transmitted without error. It’s necessary when transmitting data by modem to check the entire file sent.

The same may be true for passwords inserted in programs. These places cannot normally be so easily located, but a little trick can be used: Declare a local constant at the beginning of the procedure, if possible, with an easily recognized string. For example, if you use the longint constant 12345678h, it is then easily found in the finished compilation. Now simply note the position in the .EXE file and determine the checksum for the next 100 to 500 bytes. You can therefore be certain that no one has modified the program.

The same method also works for a program in memory. The address of the interesting procedure can be easily determined. Check the next 100 to 500 bytes, starting from this address, and you can be sure the procedure has likewise not been modified by a TSR crack. Your checksum procedure should also be backed up at least once by other checksum procedures. You then call these procedures at random in your program. Execute all the tests only after a certain time has elapsed or when the user wants to save his/her work.

Encryption algorithms

Encryption algorithms are a very broad field and could fill a book. Therefore, we only touch on encryption algorithms. There are basically two approaches which interest us. The first is modifying a character by means of a specific key, a different character. A simple but effective method is adding the key to the character. Any overflow can be handled by the processor, so data is not lost even if the key has a high value.

Another way is to XOR the characters and the key. This method can be used effectively if combined with a random number. The method works because most random numbers are not really random numbers. When they use the same initialization, the numbers always appear in the same sequence. You’re therefore able to use them without difficulty.

A different approach to encryption is using your own alphabet so each character is replaced by a character from your alphabet. This method is somewhat more difficult to unravel than that described above because decryption requires not only a key but an entire alphabet. Of course, decryption is also correspondingly more time consuming. However, if you encrypt only certain parts of a file using the method presented we described in this section, the method makes a lot of sense.
The PIQ technique

The PIQ technique is another example of how you can get any debugger to stop. The trick is based on the prefetch queue of the CPU. To enhance performance, the CPU reads both the current instruction and the next one.

This technique changes the memory location following the next instruction. For example, you change an instruction for checking the DOS version number into an instruction for stopping the program.

The instruction is already loaded during normal execution by the PIQ. This means the memory location is changed but the instruction was already fetched by the processor.

If the debugger is active, the instructions are executed one step at a time and the PIQ is not active. Therefore, the memory location is changed and when the modified instruction is reached, the program encounters the stop instruction.

The following shows this technique:

```assembly
PIQ_Stop_System proc near
    push ds
    push ax
    push bx
    push cs
    pop  ds       ; CS to DS
    mov  cs:word ptr [@int_21_funct],4CB4h  ; End function Prg.
    @int_21_funct:
    mov  ah,30h       ; Function Get DOS-Vers.
    int  21h
    pop  bx
    pop  ax
    pop  ds
    ret
PIQ_Stop_System endp
```

Using compression programs

Compressing your program has two important advantages:

1. The .EXE file will not require as much room on your hard drive.
2. The file will be "encrypted".

The .EXE file cannot be modified when it's compressed. Of course, you can decompress the file, modify it and recompress it again, but a different file size usually results. If you've put a check for the file size into the program, the changed size will be recognized. Then you can stop the program and display an error message.

The protection techniques we've discussed are irritating nuisances for any hacker to overcome. However, they're the basis for some protection if combined creatively. Write the code in assembler and not in high level procedural code which generate predictable code. A few techniques used liberally should frustrate most would-be hackers.
Train 'em: Game Trainers And Debuggers

Game trainers are another of the fascinating areas in computing. A trainer uses a program, either a TSR or a loader, to add functions for the user. These functions are activated by using a specific key combination. The user can then affect the behavior of a different program, usually an arcade or adventure game. These game trainers, unfortunately, are also being misused by crackers for dismantling program protection.

In this section, we'll talk about game trainers. Many of you've probably been involved in a great action game in which you've fought your way to the final opponent only to discover that you have no more weapons or only a single life remaining. In such a case, you'd be fortunate to have a trainer available. By just pressing a few keys, you immediately find yourself equipped with all weapons and have all your lives back.

The problem is that such trainers are not available in every game. There's only one thing you can do in that case: Create your own. You'll find the information you'll need in this section to write your own trainer. First, consider which programs are best suited for a trainer. Programs and games which are not suited for trainers usually include role games and adventure games. These games use a form of script language which is translated by an interpreter. In this case, it's better to save a game and inspect the "save" file with a hex editor (see the "Not hexing: Hex editors" information starting on page 237). In the extreme case, you can even write a savegame editor.

However, action games are quite suitable for training, especially action games where lives are lost, extra weapons distributed and points collected. These are all managed directly within the game, usually by WORD or DWORD variables. All you need for a trainer in these cases is to find the location of these variables in memory. Then write a small TSR which will change these variables to new values when a certain key combination is pressed.

Rummage around: Finding variables

It's not so easy to write game trainers, otherwise there would already be dozens of trainers for every game. The biggest problem is finding the variables used. To find the variables, you'll need to use a debugger such as Turbo Debugger. First, you should search for the structures usually used to increment or decrement variables. Let's say, you're looking for that place in the program where you lose a life. When you find this place, you have also found the "lives" variable. To find it, there are two methods:

Method A - look for frequently used structures

This is the method you should try first because, if successful, it leads quickly to your target. In our example, look for a machine code which decrements the content of a byte variable. A byte variable is used because you probably won't get more than 256 lives. The structure most frequently used for decrementing is:

\[
\text{dec byte ptr [xxyy]}
\]

where xxyy is the offset-position of the variable in the current data segment. The assembler translates this instruction into:

\[
\text{FE 0E yy xx}
\]

That means you'll probably want to look for the byte combination FE 0E because you won't know the value of yy xx. Don't panic if the program is long or if there are many duplicate combinations. First, some combinations will be eliminated because they're part of an instruction consisting of several bytes. Then,
other combinations involve the same variable. Write down the variables which are found and then check their contents while the program is running. You'll soon find the desired variable.

If you are unable to find the decrement instruction, you should look for a different combination. The following table shows an overview of the most commonly used structures:

<table>
<thead>
<tr>
<th>Assembler instruction</th>
<th>Translation</th>
<th>Assembler instruction</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec word ptr [xxyy]</td>
<td>FF 0E yy xx</td>
<td>sub [xxyy],dx</td>
<td>29 16 yy xx</td>
</tr>
<tr>
<td>dec byte ptr [xxyy]</td>
<td>FE 0E yy xx</td>
<td>inc word ptr [xxyy]</td>
<td>FF 06 yy xx</td>
</tr>
<tr>
<td>sub word ptr [xxyy],zz</td>
<td>83 2E yy xx zzzz</td>
<td>inc byte ptr [xxyy]</td>
<td>FE 06 yy xx</td>
</tr>
<tr>
<td>sub byte ptr [xxyy],zz</td>
<td>80 2E yy xx zz</td>
<td>add word ptr [xxyy],zz</td>
<td>83 06 yy xx zzzz</td>
</tr>
<tr>
<td>sub [xxyy],ax</td>
<td>29 06 yy xx</td>
<td>add byte ptr [xxyy],zz</td>
<td>80 06 yy xx zz</td>
</tr>
</tbody>
</table>

Be aware during your search that many programs use several data segments or save data in the code segment. If you don't have any luck in the current segment, then you should look through the other segments used in the program.

If all the variables found in these targeted searches failed to contain the desired value or, very simply, too many variables were found, you'll have to use method B.

Method B - following the operation of the program

The tracing of programs has become more important since the lawsuit between Microsoft and Stack, Inc. However, if method A is unsuccessful, this is the only option for locating the variables searched. First, locate a procedure which rebuilds the screen. Now step through this procedure and see which variables are used for which screen structures. Here, you'll usually find something quickly and have a series of trainable variables. If you still cannot find the desired variable for life or the program break, follow the main loop until you reach the comparison of a certain variable with 0. This variable is usually (but not always) the variable for lives or the program end.

Handling the corner data

After you have found the variables used by the program, you can begin writing the trainer. You'll need some corner data from the program to do this. First, you'll have to note the data segment of the respective variables. Then you must determine whether the program uses its own routine for handling the keyboard. Search in the program for the command:

```
in al,60h.
```

Now set a breakpoint on this command. Then execute the program and press a key at that point. If the program breaks when you press a key, you have found the handling routine if you're at the breakpoint position. Write down the current code segment (CS) and the offset where the command stands. You can find this value in the instruction pointer (IP).

In case the program is not interrupted, you'll have to keep on searching for the command. If you do not find it, check whether your program has several code segments. You must look through all code segments if you have no luck in the base segment. It's also a good idea to look very carefully at the ES and DS segments often.
Programming trainers

How, then, must a trainer be constructed? A trainer consists in principle of two essential parts:

**Installation**
This doesn't refer to the loading of the program as a TSR but integrating it into the program to be trained.

**Interface**
This part checks whether a key of the trainer was activated and correspondingly reacted.

We'll describe both in more detail below.

**The installation part**

During installation, you must determine if the program uses its own keyboard routine or accesses the DOS routines. If the DOS routines are used, then you can just change interrupt 9h to point to your routine. Check to see if the key combination is for the game trainer; if so, respond appropriately and then call the original interrupt.

If the program uses its own keyboard routine, the steps are slightly more complicated. First, find and unused interrupt, such as interrupt 65h. We then proceed to a frequently used interrupt, preferably interrupt 21h. This interrupt will in any case be called by the program to be trained. You can use it to make a slight modification in the program code. That is legal, because you are not modifying the program on the hard disk, only a region of memory.

The place to be modified is that point in the program where a character is read from the keyboard using the following:

```
in al,60h.
```

In machine code, the instruction reads E4 60.

If an interrupt is then generated, the processor saves the following:

- The flags
- The code segment where the call originated
- The current instruction pointer on the stack

Now save the current values of BP, BX and DS on the stack. Then move the stack register SP to BP. You'll now be able to determine the code segment using [BP+10], and the position of the IP using [BP+8]. Then move the value of the code segment to DS and the IP to BX. You can now check whether E4 60 is at the memory location addresses by DS:[BX]. If so, the interrupt was triggered by the program to be trained and you have found the place to be modified. Now replace the two bytes with CD 65. Interrupt 65h will now be called instead of having the program reading in a character using in al,60h.
In interrupt 65h, no registers except for the al register can be changed. You must also read a character from
the keyboard using in al,60h. Check now whether the character is a key used by your trainer. If yes, the
corresponding code is handled by the trainer.

The interface part

The interface part is really the most interesting part of a trainer. We’ll distinguish between programs that
use a single data segment and programs that use several data segments.

The first type is typically programs that are generated by a language, such as Borland Pascal, which allow
only one data segment. In this case you don’t have to worry whether you’re using the correct data segment:
You'll automatically have the right one. If the interrupt which you changed is called, check first which key
was pressed. If it’s one of the trainer keys, the routine will first jump to where the corresponding variable
is changed and then to the routine which correctly terminates the interrupt. This will be an IRET or even
a call to the original interrupt, depending on which interrupt you used. If the key pressed is not a trainer
key, the jump is made directly to this routine.

Now, in the variable-modifying part, write the new value for the variable at the position you determined.
Don’t always write the maximum value for the variable type. You should instead determine, for example,
at which value the energy is completely restored or the maximum complement of weapons is available. A
typical instruction might look like the following:

```
Full_energy:
mov ds:[1234],80
jmp End_routine
```

Your work will be slightly more complicated when your program has more than one data segment.
Hopefully, you wrote down the respective data segment of the variables to be modified and you know the
code segment in which the keyboard is queried. You can now identify the data segment based on that
information. In the trainer interrupt, determine the code segment where the interrupt is called. Now add
to or subtract from this value the difference between the data segment of the variable and the code segment
of the interrupt call. You already have the correct code segment. You can now modify your variables entirely
normally. Here, now, is an example of how such a routine could look:

```
Full_weapons:
pusha
mov bx,[bp+8]
sub bx,3K43h
mov ds,bx
mov ds:[1234],255
popa
jmp End_interrupt
```

Now you know the secrets of writing a trainer. You can try your hand at writing a trainer for the minigame
RAIDERS which we’ve included on the companion CD-ROM.

Trainer for the RAIDERS game

We hope that you’ve tried your luck with RAIDERS already. It will serve as a good introduction to creating
a trainer.
You'll find the complete source code for a trainer for RAIDERS on the companion CD-ROM. We've included the source code as a basic kit for your own project. We'll take a closer look at the part containing the actual trainer programming.

The trainer first defines a few variables to check whether the trainer is already resident.

```assembly
.286
w equ word ptr
b equ byte ptr
code segment public
public insthand
public handler9
public reslim
public oldint21
public oldint65
public ID
public IDlen
public check_inst
assume cs:code,ds:code
ID: ab 'ABACUS'
oldint21: dd 0
oldint65: dd 0
procedure: dd ?
IDlen equ offset oldint21 - offset ID
```

The new handler for the 9h interrupt is installed next. The latter reads a character from the keyboard and checks whether it is a trainer character. If the answer is yes, the corresponding variable is adjusted. The variables were determined during a small Turbo Debugger session.

```assembly
handler9 proc pascal
pushf
push bp
```

You can find RAIDTRAI.ASM on the companion CD-ROM.
push ds
push bx

in al,60h ; read character
cmp al,63 ; F5 key
je Full_Shoots_j
cmp al,64 ; F6 key
je Full_Lives_J
cmp al,65 ; F7 key
je Weapon_new_j ;
cmp al,66 ; F8 key
je Weapon_new_j ;
cmp al,67 ; F9 key
je Weapon_new_j ;
cmp al,68 ; F10 key
je More_Points_J
End_Keyb:
pop bx
pop ds
pop bp
popf
iret

Full_Shoots:
pushf
PUSHA
sub bx,0 ; since already correct CS
mov word ptr procedure+2,bx
mov bx,1401h ; es:[bx] = 14EF:1401
mov word ptr procedure,bx

mov ds:byte ptr [0DA3h],20h
mov ax,20h
push ax
call dword ptr [procedure]
POPA
popf
jmp End_Keyb

Full_Lives:
pushf
pusha
sub bx,0 ;
mov word ptr procedure+2,bx
mov bx,1317h ; es:[bx] = 14EF:1317
mov word ptr procedure,bx
---------
mov ds:byte ptr [0DA3h],0009
mov ax,9
push ax
call dword ptr [procedure]
popa
popf
jmp End_Keyb
Chapter 9
Protect Your Know-how: Protection Tricks

More_Points:
pushf
pusha
sub bx,0 ;
mov word ptr procedure+2,bx
mov bx,1BD0h ; es:[bx] = 14EF:1BD0
mov word ptr procedure,bx
---------
mov ax,1000
push ax
call dword ptr [procedure]
popa
popf
jmp End_Keyb

handler65 endp

The code which now follows serves to install or uninstall the trainer.

instand proc pascal
reslim label byte
push ds
pop ds
mov ax,3521h ; store old INT 21
int 21h
mov w oldint21,bx
mov w oldint21 + 2,es
mov ax,3565h ; store old INT 65h
int 21h
mov w oldint65,bx
mov w oldint65 + 2,es
mov ax,2521h ; bend/deflect INT 21h to custom routine
lea dx,handler21
int 21h
mov ax,2565h ; INT 65h to custom keyboard routine
lea dx,handler65
int 21h
ret
instand endp

check_inst proc near
mov ax,3521h ; get interrupt vector
int 21h
mov di,bx
mov si,offset ID
mov di,si
mov cx,IDlen
repe cmpsb ; check for ID
ret
check_inst endp

code ends
end

The RAIDERS program has only one data segment and does not use its own keyboard routine. This trainer is therefore a good introductory trainer.
The following trainer represents a small "tidbit" because it still calls the procedure for updating the display of the altered variable. Then you can see quite clearly, how the handling looks for those programs which program the keyboard directly.

First, the new procedure checks for interrupt 21h. It checks whether the in al,60h instruction is in the program and if necessary replaces it.
Now for the actual trainer routines. The handler first determines the code segment at the time the interrupt is called. Because the code segment for keyboard handling is the same as the procedures for updating the screen, it can be transferred directly. Otherwise, an adjustment would have to be made using one of the methods described above. Whenever a variable is to be trained, the pointer to the procedure is first set to the correct values with the respective screen update. The variable is then called and the screen update executed. Finally, the procedure jumps into the original handler.

```
;  *************************************************************************
;  ***                                                                   ***
;  ***  Int 65h procedure. It reads in a character via "in al,60h"       ***
;  ***  and checks whether the read character was defined as the         ***
;  ***  Trainer key. If the answer is yes, the allocated memory          ***
;  ***  changes and procedure calls are executed. Enter your training    ***
;  ***  variables here !!!                                               ***
;  ***                                                                   ***
;  *************************************************************************
handler65 proc far
  pushf
  push bp
  push ds
  push bx
  mov  bp,sp
  mov  bx,[bp+10] ; cs at time of interrupt to BX
  in  al,60h     ; read character
  cmp al,63      ; F5 key
  je  Full_Shoots_j
  cmp al,64      ; F6 key
  je  Full_Lives_J
  cmp al,65      ; F7 key
  je  Weapon_new_j
  cmp al,66      ; F8 key
  je  Weapon_new_j
  cmp al,67      ; F9 key
  je  Weapon_new_j
  cmp al,68      ; F10 key
  je  More_Points_J
End_Keyb:
  pop  bx
  pop  ds
  pop  bp
```
popf
  iret

Full_Shoots_j:
  jmp Full_Shoots
Full_Lives_j:
  jmp Full_Lives
More_Points_j:
  jmp More_Points
Weapon_new_j:
  jmp Weapon_new

Full_Shoots:
  pushf
  pusha
  sub bx,0 ; since already correct CS
  mov word ptr procedure+2,bx
  mov bx,1401h ; es:[bx] = 14EF:1401
  mov word ptr procedure,bx
  ;--------
  mov ds:byte ptr [0DA3h],20h
  mov ax,20h
  push ax
  call dword ptr [procedure]
  popa
  popf
  jmp End_Keyb

Full_Lives:
  pushf
  pusha
  sub bx,0 ;
  mov word ptr procedure+2,bx
  mov bx,1317h ; es:[bx] = 14EF:1317
  mov word ptr procedure,bx
  ;--------
  mov ds:byte ptr [0DA3h],0009
  mov ax,9
  push ax
  call dword ptr [procedure]
  popa
  popf
  jmp End_Keyb

Weapon_new:
  pushf
  pusha
  sub bx,0 ;
  mov word ptr procedure+2,bx
  mov bx,1454h ; es:[bx] = 14EF:1454
  mov word ptr procedure,bx
  ;--------
  sub al,65
  mov ah,0
  mov ds:byte ptr [0DA2h],al
  push ax
  call dword ptr [procedure]
  popa
  popf
  jmp End_Keyb

More_Points:
pushf
pusha
sub bx,0 ;
mov word ptr procedure+2,bx
mov bx,1BD0h ; es:[bx] = 14EF:1BD0
mov word ptr procedure, bx
;------------
mov ax,1000
push ax
call dword ptr [procedure]
popa
popf
jmp End_Keyb

handler65 endp

insthand proc pascal
reslim label byte
push ds
pop ds
mov ax,3521h ; store old INT 21
int 21h
mov w oldint21,bx
mov w oldint21 + 2,es
mov ax,3565h ; store old INT 65h
int 21h
mov w oldint65,bx
mov w oldint65 + 2,es
mov ax,2521h ; bend/deflect INT 21h to custom routine
lea dx,handler21
int 21h
mov ax,2565h ; INT 65h to custom keyboard routine
lea dx,handler65
int 21h
ret
insthand endp

check_inst proc near
mov ax,3521h ; get interrupt vector
int 21h
mov di,bx
mov si,offset ID
mov di,si
mov cx,IDlen
repe cmpsb ; check for ID
ret
check_inst endp

code ends
end

By working with these two examples, you should be able to write a trainer. However, we also recommend that you keep practicing with other examples.
Most PCs today have at least 4 Meg of memory. However, regardless of the amount of RAM, you can only access 640K of this memory directly from DOS. In this chapter, we'll show you some ways of using the memory above this 640K barrier.

First, we'll review a few of the basic structures.

**Conventional DOS Memory**

Under DOS, conventional memory is the first 640K. As we've discovered, this isn't enough for many applications. Here we'll show you how to get around the 640K barrier and address up to 4 Gigabytes (4,000,000,000 bytes) of memory.

Conventional DOS memory has a maximum size of 640K. This memory range is not be addressed as a single large block. Instead, it's divided into segments of 64K. A segment starts at a paragraph address that is a physical address divisible by 16. If you want to address a memory location, the computer has to know which segment to access. The exact address is specified within the segment using an offset. An offset is always one word long because exactly 64K can be addressed with one word. Specifying an address is as follows:

\[
\text{addr} := \text{SEGMENT:OFFSET}. 
\]

If you program in a high-level language like C or Pascal and you don’t need more than the prescribed data types, you don’t need to worry about addressing memory. C or Pascal does this automatically for you. However, if you get into more advanced system programming, you will need to have an understanding of the different ways to access memory.

In earlier 286 processors, a PC had four segment registers:

<table>
<thead>
<tr>
<th>Code segment</th>
<th>Specifies the segment in which the current executable code is located.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data segment</td>
<td>Specifies the memory range in which the program data is stored.</td>
</tr>
<tr>
<td>Extra segment</td>
<td>Specifies a segment for storage and copy operations.</td>
</tr>
<tr>
<td>Stack segment</td>
<td>Specifies the stack segment.</td>
</tr>
</tbody>
</table>

The 386 and later processors also have FS and GS registers.
There are different ways to access the memory. In Pascal, you can use the MEM command. The following line is an example of using MEM to copy a byte from memory into a variable:

```
Result := MEM[Segment:Offset];
```

For example, if you want to read the 50th pixel of the screen in MCGA mode, then write the following:

```
Byte_50 := MEM[$A000:0050];
```

There are several ways to do the same thing in assembler. Probably the most common method (especially for reading large blocks of data) is to use the LODSB/STOSB/MOVSB instructions or their corresponding WORD or DWORD equivalents. This method is slightly longer because the data segment is saved so it can access the program data again.

```
push ds
mov si,0a0000h
mov ds,si
mov si,50
lodsb
mov byte ptr byte_50,al
pop ds
```

During execution, you may need additional memory to store data, for example. You can ask DOS to allocate this additional memory.

To reference this additional memory you use a pointer. A pointer is a data type containing both the segment and offset to a range of memory. A pointer is normally initialized using the procedure Getmem, but it can also be built. You can use Freemem to free the allocated area of memory. The memory used by Pascal is only accessible to the Pascal program. When a program begins execution, Pascal reserves the minimum amount of memory indicated. This memory is not available to others until the program has completed execution.

If you do not want to use the memory reserved by Pascal, but you would rather use the conventional DOS memory, you need to use the 48h and 49h functions of DOS interrupt 21h. Reserve a block of memory with function 48h and free it with function 49h. It’s important to know the blocks are always allocated on paragraph boundaries so you have to round all sizes up to the next 16 bytes.

An example of how these routines might look is shown by the assembler procedures dos_getmem and dos_freemem.

```
dos_getmem proc pascal pointer:dword,amount:word
; **************************************************************************
; *** Allocates a (max. 64K) memory area in DOS RAM ***
; **************************************************************************
push ds
mov bx,amount
shr bx,4
inc bx
mov ah,48h
int 21h
mov bx,w [pointer+2]
mov ds,bx
mov bx,w [pointer]
mov bx,w [pointer+1],0
mov w [bx+2],ax
pop ds
```

The following procedures are part of the GUSASM.ASM file on the companion CD-ROM.
EMS: A First Step Towards More Memory

Originally, the memory range for DOS was limited to 640K. Some applications required more memory than this "640K barrier". The three industry leaders at the time (Lotus, Intel and Microsoft) created a standard called the "Expanded Memory Specification", or LIM EMS, to go beyond this barrier.

To use EMS, an application must be specifically written to use it (Lotus 1-2-3 Version 2.x, AutoCAD, etc.) or the application must run in an environment that supports it, such as DESQview.

EMS is installed in the old XT and ATs by plugging in an EMS memory board and adding an EMS driver. In systems using a 386 or above, EMS is simulated by software (EMM) software that turns extended memory into EMS.

The original LIM EMS consisted of an expansion card that could be inserted into a PC/XT to give the computer 8 Meg of additional memory. This memory was switched into the conventional memory address in banks up to 64K in size (four 16K pages) in the Upper Memory Area (640K to 1 Meg area). This memory was called the EMS frame. The switched banks are divided into four pages of 16K each and they can be accessed like normal memory.

Access to the pages is controlled by an Expanded Memory Manager (EMM). It converts extended memory into EMS in computers using the 386es or higher (we'll forego the technical details of the EMM in this book).

EMM Functions

The EMM makes its functions available using interrupt 67h. If an error occurs in one of its functions, EMM returns an error code listed in the following table:
Now onto the functions of EMM. The functions we’ll describe are through Version 3.0 because later versions weren’t widely distributed.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>80h</td>
<td>Error in software interface</td>
</tr>
<tr>
<td>81h</td>
<td>Error in EMS hardware</td>
</tr>
<tr>
<td>82h</td>
<td>EMM is busy</td>
</tr>
<tr>
<td>83h</td>
<td>Invalid handle</td>
</tr>
<tr>
<td>84h</td>
<td>Invalid function number</td>
</tr>
<tr>
<td>85h</td>
<td>No additional handles available</td>
</tr>
<tr>
<td>86h</td>
<td>Error when saving or resetting the display between logical and physical pages</td>
</tr>
<tr>
<td>87h</td>
<td>Too few free pages</td>
</tr>
<tr>
<td>88h</td>
<td>Invalid number of pages</td>
</tr>
<tr>
<td>89h</td>
<td>An attempt was made to allocate 0 pages</td>
</tr>
<tr>
<td>8Ah</td>
<td>Invalid page number</td>
</tr>
<tr>
<td>8Bh</td>
<td>Invalid physical page number</td>
</tr>
<tr>
<td>8Ch</td>
<td>Mapping cannot be saved</td>
</tr>
<tr>
<td>8Dh</td>
<td>Mapping is already saved</td>
</tr>
<tr>
<td>8Eh</td>
<td>Mapping was not saved</td>
</tr>
<tr>
<td>8Fh</td>
<td>Wrong subfunction number</td>
</tr>
</tbody>
</table>

Function 40h - Read status

**Input:** AH = 40h

**Output:** AH = Status of the EMM

If the function returns a value of 0, the EMM is working correctly. Otherwise, the error can be determined from the status code using the error table.

Function 41h - Determine the segment address of the frame

**Input:** AH = 41h

**Output:**

AH = Status of the EMM

BX = Segment address of the window

The segment address is only valid if the status code contains the value of 0. Otherwise, an error is indicated as listed in the table. The segment address is the address where the frame is to be found in the main memory.
Function 42h - Determine the number of EMS pages

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 42h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>AH = status of the EMM</td>
</tr>
<tr>
<td></td>
<td>BX = number of free EMS pages</td>
</tr>
<tr>
<td></td>
<td>DX = total number of EMS pages</td>
</tr>
</tbody>
</table>

An EMS page is 16K in length. The amount of free EMS memory can be calculated by multiplying 16 x 1024. The values are only valid if the status code has the value of 0.

Function 43h - Allocate the EMS memory

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 43h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BX = number of 16K pages to reserve</td>
</tr>
<tr>
<td>Output:</td>
<td>AH = status of the EMM</td>
</tr>
<tr>
<td></td>
<td>DX = handle number</td>
</tr>
<tr>
<td></td>
<td>DX = total number of EMS pages</td>
</tr>
</tbody>
</table>

The allocated pages are addressed using the returned handle. It’s only valid if the status code has the value of 0. Make certain not to lose the handle and to free all the allocated pages at the end of the program. A maximum of 512 pages can be allocated.

Function 44h - Set mapping

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 44h</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL = number of the page in the EMS frame</td>
<td></td>
</tr>
<tr>
<td>BX = number of the page in the reserved block</td>
<td></td>
</tr>
<tr>
<td>DX = handle of the reserved block</td>
<td></td>
</tr>
<tr>
<td>Output:</td>
<td>AH = status of the EMM</td>
</tr>
</tbody>
</table>

The page number in an EMS frame can have a value between 0 and 3. The page number in the reserved block is relative to the start of the reserved block (e.g., the sixth page of the handle 1Ch). The indicated range is immediately merged into the DOS EMS frame. If the status code has a value of 0, the function is working correctly.

Function 45h - Free pages

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 45h</th>
</tr>
</thead>
<tbody>
<tr>
<td>DX = handle</td>
<td></td>
</tr>
<tr>
<td>Output:</td>
<td>AH = status of the EMM</td>
</tr>
</tbody>
</table>
Chapter 10

Memory Management

This function frees the reserved EMS pages. All the pages of the handle are deallocated. It’s very important you call this function at the end of the program for all the handles you have allocated. Otherwise, other programs cannot access the allocated EMS memory. A status code of 0 shows there were no errors in execution.

<table>
<thead>
<tr>
<th>Function 46h - Determine EMS version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: AH = 46h</td>
</tr>
<tr>
<td>Output: AH = status of the EMM</td>
</tr>
<tr>
<td>AL = version number</td>
</tr>
</tbody>
</table>

The version number is stored in BCD-format. The main version number is located in the upper four bits. The minor version is located in the lower four bits. There is an error if the status code is not 0.

<table>
<thead>
<tr>
<th>Function 47h - Save mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: AH = 47h</td>
</tr>
<tr>
<td>DX = Handle</td>
</tr>
<tr>
<td>Output: AH = Status of the EMM</td>
</tr>
</tbody>
</table>

The allocation of the mapping within the EMM is saved as a setting with this function. The function is working error-free if the status code equals 0.

<table>
<thead>
<tr>
<th>Function 48h - Deprotect mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: AH = 48h</td>
</tr>
<tr>
<td>DX = handle number</td>
</tr>
<tr>
<td>Output: AH = status of the EMM</td>
</tr>
</tbody>
</table>

With this function, an allocation made using the function 47h can be unprotected. There is an error if the status code does not equal 0.

<table>
<thead>
<tr>
<th>Function 4Bh - Determine handle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: AH = 4Bh</td>
</tr>
<tr>
<td>Output: AH = status of the EMM</td>
</tr>
<tr>
<td>BX = number of handles currently allocated</td>
</tr>
</tbody>
</table>

This function determines the number of handles currently allocated by the EMM. A status code other than 0 indicates an error.
Function 4Ch - Determine allocated pages

<table>
<thead>
<tr>
<th>Input:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH = 4C</td>
<td></td>
</tr>
<tr>
<td>DX = handle number</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH = status of the EMM</td>
<td></td>
</tr>
<tr>
<td>BX = number of allocated pages</td>
<td></td>
</tr>
</tbody>
</table>

With this function you can determine how many 16K pages of EMS memory are allocated by the given handle.

Function 4Dh - Determine allocated handles

<table>
<thead>
<tr>
<th>Input:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH = 4Dh</td>
<td></td>
</tr>
<tr>
<td>ES = segment of the address of the data structure</td>
<td></td>
</tr>
<tr>
<td>DI = offset of the address the data structure</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH = status of the EMM</td>
<td></td>
</tr>
<tr>
<td>BX = number of pages allocated all together</td>
<td></td>
</tr>
</tbody>
</table>

This function loads the numbers of all active handles and the number of all allocated pages into a data structure. The structure has the length “number of handles * 4”. The entries in the table each consist of two words. The number of handles is stored in the first word and the number of pages is stored in the second word.

**Using EMS**

Now we can see how to use EMS. First, we need to make certain EMS is installed. This can be done by checking the label of the interrupt 67h. An EMS driver is installed if the string ‘EMMXXXX’ appears in the label. Next we can check the version number. Determining the version number is especially important if you want to use certain functions that have not been implemented in Version 3.0, for example. The following is an example of an initialization procedure:

```pascal
procedure Check_for_EMS;
var emsseg : word;
  emsptr : pointer;
  emshead : EMS_Header;
begnin;
asm
  mov ax,3567h
  int 21h
  mov emsseg,es
end;
move(ptr(emsseg,0)^,emshead,17);
if emshead.ID = 'EMMXXXX' then begin;
  EMS_Available := true;
asm
  mov ah,40h { get EMS driver status }
end;
```

The following functions are part of the MEMORY.PAS file on the companion CD-ROM.
Memory Management

Next, determine the address where the EMS frame is inserted into the main memory. This is important because all memory accesses refer to this address. Use function 41h of EMM to do this:

Function EMS_Segment_determine(VAR Segment : word) : byte;
VAR hseg : word;
fresult : byte;
begin;
  asm
    mov ah,41h
    int 67h
    cmp ah,0
    jne @EMS_Segdet_Error
    mov hseg,bx
    mov hseg,0
    jmp @EMS_Segdet_End
@EMS_Segdet_Error:
  mov fresult,ah
@EMS_Segdet_End:
  Segment := hseg;
  EMS_Segment_determine := fresult;
end;

To see how much memory is available, use the function to determine the number of EMS pages. Function 42h provides the total number of pages available and the number of pages still free (which is the one we’re interested in). In this function, this value is stored in the global variable EMS_Pages_Free. This function is an auxiliary function for EMS_Free.

Function EMS_Get_PageNumber : byte;
var fresult : byte;
begin;
  asm
    mov ah,42h
    int 67h
    cmp ah,0
    jne @EMS_GetPages_Error
    mov EMS_Pages_Free,bx
    mov EMS_Pages_Tot, dx
@EMS_GetPages_Error:
  mov fresult, ah
end;
If your program needs the value for the amount of available EMS memory, use function `EMS_free`. It returns the number of bytes that are available in EMS and uses the auxiliary function `EMS_Get_PageNumber` to do so.

```pascal
function EMS_free  : longint;
  var    help : longint;
  begin;
    EMS_Get_PageNumber;
    help := EMS_Pages_Free;
    EMS_free := help SHR 14;
  end;
```

To allocate EMS memory from within your program, you can program the EMM directly. However, it is easier to allocate memory using the function `Getmem_EMS`. This function’s first parameter is the target handle under which the block is to be accessible. The second parameter is the size (in bytes) of the block to be allocated. The function converts your request into the appropriate number of pages and allocates the memory using function 43h.

```pascal
Function Getmem_EMS(VAR H : EMSHandle; Size : longint) : byte;
  var Fresult : byte;
    EPages : word;
    Hhandle : word;
  begin;
    EPages := (Size DIV 16384) + 1;
    asm
      mov ah,43h
      mov bx,EPages
      int 67h
      cmp ah,0
      jne @Getmem_Ems_Error
      mov Hhandle,dx
      mov fresult,0
      jmp @Getmem_Ems_End
    @Getmem_Ems_Error:
      mov Fresult,ah
    @Getmem_Ems_End:
      H := Hhandle;
      Getmem_EMS := Fresult;
    end;
```

To free the allocated EMS memory later (something you should always do at the end of the program) use function `Freemem_EMS`. Using the EMM function 45h, the function `Freemem_EMS` frees up the memory block which was managed using the provided handle.

```pascal
To free the allocated EMS memory later (something you should always do at the end of the program) use function `Freemem_EMS`. Using the EMM function 45h, the function `Freemem_EMS` frees up the memory block which was managed using the provided handle.
```
Chapter 10

Memory Management

Function Freemem_EMS(H : EMSHandle) : byte;
var Fresult : byte;
begin;
  asm
    mov ah, 45h
    mov dx, H
    int 67h
    mov Fresult, ah
  end;
Freemem_EMS := Fresult;
end;

The function **EMS_Allocation** is very important. Use it to determine which pages from the EMS will be displayed in the EMS frame. For the first parameter, provide the function with the handle by which the page block in EMS is addressed. Next, inform the function as to which of the pages are to be allocated in the EMS frame (0 to 3). The number of the page in EMS should be provided as the last parameter.

Function EMS_Allocation(H : EMSHandle; PagePage, EMSPage : word) : byte;
VAR Fresult : byte;
begin;
  asm
    mov ah, 44h
    mov al, byte ptr PagePage
    mov bx, EMSPage
    mov dx, H
    int 67h
    mov Fresult, ah
  end;
EMS_Allocation := Fresult;
end;

To protect the memory allocated in EMS for a particular handle, use function **EMS_Protect_Allocation**. A protected page of EMS memory cannot be changed until it has been first unprotected.

Function EMS_Protect_Allocation(H : EMSHandle) : byte;
VAR Fresult : byte;
begin;
  asm
    mov ah, 47h
    mov dx, H
    int 67h
    mov Fresult, ah
  end;
EMS_Protect_Allocation := Fresult;
end;

Use **EMS_Unprotect_Allocation** to remove the protection on a previously protected EMS page. This function uses the EMM function 48h. Protected memory must be unprotected before it can be changed.

Function EMS_Unprotect_Allocation(H : EMSHandle) : byte;
VAR Fresult : byte;
begin;
  asm
    mov ah, 48h
    mov dx, H
    int 67h
    mov Fresult, ah
  end;
EMS_Unprotect_Allocation := Fresult;
end;
Function **RAM_2_EMS** provides an example of how you can swap out the data to EMS. Use it to copy a block from DOS RAM to EMS. Use this function as an example for your own routines or try it out directly. If time is a very important factor, you should formulate the routine for each application because the method we show here only serves to swap out the data, and not to directly access the pages.

```pascal
Function RAM_2_EMS(q : pointer; H : EMSHandle; Size : longint) : byte;
VAR fresult : byte;
   EMSseg : word;
   hp      : *byte;
   li      : word;
begin
   EMS_Segment_determine(EMSseg);
   hp := q;
   if Size > 16384 then begin
      { More than one page required }
      for li := 0 to (Size SHR 14)-1 do begin
         EMS_Allocation(H,0,li);
         move(hp^,ptr(EMSseg,0)^,16384);
         dec(Size,16384);
         inc(hp,16384);
      end;
      EMS_Allocation(H,0,li+1);
      move(hp^,ptr(EMSseg,0)^,16384);
      dec(Size,16384);
      inc(hp,16384);
   end else begin
      EMS_Allocation(H,0,0);
      move(hp^,ptr(EMSseg,0)^,16384);
      dec(Size,16384);
      inc(hp,16384);
   end;
end;
```

Copy a block from the EMS memory back into the DOS RAM using the function **EMS_2_RAM**.

```pascal
Function EMS_2_RAM(q : pointer; H : EMSHandle; Size : longint) : byte;
VAR fresult : byte;
   EMSseg : word;
   hp      : *byte;
   li      : word;
begin
   EMS_Segment_determine(EMSseg);
   hp := q;
   if Size > 16384 then begin
      { More than one page required }
      for li := 0 to (Size SHR 14)-1 do begin
         EMS_Allocation(H,0,li);
         move(ptr(EMSseg,0)^,hp^,16384);
         dec(Size,16384);
         inc(hp,16384);
      end;
      EMS_Allocation(H,0,li+1);
      move(ptr(EMSseg,0)^,hp^,16384);
      dec(Size,16384);
      inc(hp,16384);
   end else begin
      EMS_Allocation(H,0,0);
      move(ptr(EMSseg,0)^,hp^,16384);
      dec(Size,16384);
      inc(hp,16384);
   end;
end;
```
In your program, use function 4Ch of the EMM if you need to know how many EMS pages are allocated by a particular handle. The first parameter you provide to the function `EMS_Pages_allocated` is the number of the handle to be analyzed. The second parameter is the variable in which the number of pages is to be stored.

```pascal
Function EMS_Pages_allocated(H : EMSHandle;var Pages : word) : byte;
var fresult : byte;
    Hs : word;
begin;
    asm
        mov ah,4Ch
        mov dx,H
        int 67h
        mov Hs,bx
        mov fresult,ah
    end;
    Pages := Hs;
    EMS_Pages_allocated := fresult;
end;
```

One last function is used to determine the allocated EMS handle. Unfortunately, you can’t access as many EMS handles as you might want; the number is usually limited to 32. This is also the reason why you should make sure you always reserve blocks that are as large as possible and which can be addressed using a handle.

```pascal
Function EMS_Handles_assign(Var Number : word) : byte;
Var Fresult : byte;
    Han       : word;
begin;
    asm
        mov ah,4Bh
        int 67h
        mov Han,bx
        mov Fresult,ah
    end;
    Number := Han;
    EMS_Handles_assign := Fresult;
end;
```

**XMS: Thanks For The Memory**

The XMS standard (Extended Memory Specification) was developed by Microsoft, Lotus, Intel and AST Research. Unlike the EMS standard, XMS is not based on a hardware expansion card.

The XMS standard provides several functions which reserve, release and transfer data to and from extended memory without conflict (including the HMA or high memory area). These functions are made available by an XMS driver which uses interrupt 2Fh. The most commonly used XMS driver is HIMEM.SYS which is loaded through the CONFIG.SYS file. The HIMEM.SYS driver is included with both DOS and Windows.
XMS Error codes

The functions of the driver are addressed as a far call rather than by an interrupt. To determine the address of the XMS driver, you must first test whether an XMS driver has been installed by calling interrupt 2Fh with the value 4300h in the AX register. A driver is installed if the value 80h is returned in the AL register. In this case, the interrupt must be called again using the value 4310h in the AX register. The segment address of the driver is returned in the ES register and the offset address of the XMM in the BX register.

Once you have determined the address of the XMM (Extended Memory Manager), you can call its functions with a far-call. If the function was able to finish successfully, it usually returns the status code 0001h in AX. If an error occurred during execution, AX will contain the value 0000h. In this case the BL register will contain an error code. The following table defines the error codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>80h</td>
<td>Function is not known</td>
<td>A4h</td>
<td>Source offset is invalid</td>
</tr>
<tr>
<td>81h</td>
<td>VDISK ramdisk found</td>
<td>A5h</td>
<td>Target handle is invalid</td>
</tr>
<tr>
<td>82h</td>
<td>Error on the A20 address line</td>
<td>A6h</td>
<td>Target offset is invalid</td>
</tr>
<tr>
<td>8Eh</td>
<td>General driver error</td>
<td>A7h</td>
<td>Invalid length for move function</td>
</tr>
<tr>
<td>8Fh</td>
<td>Irreparable error</td>
<td>A8h</td>
<td>Inadmissible overlap at move function</td>
</tr>
<tr>
<td>90h</td>
<td>HMA not available</td>
<td>A9h</td>
<td>Parity error</td>
</tr>
<tr>
<td>91h</td>
<td>HMA already filled</td>
<td>AAh</td>
<td>UMeg not blocked</td>
</tr>
<tr>
<td>92h</td>
<td>The amount of memory indicated in DX is too small</td>
<td>ABh</td>
<td>UMeg still blocked</td>
</tr>
<tr>
<td>93h</td>
<td>HMA not allocated</td>
<td>ACh</td>
<td>Overrun of the UMeg blocking counter</td>
</tr>
<tr>
<td>94h</td>
<td>A20 address line is still on</td>
<td>ADh</td>
<td>UMeg cannot be blocked</td>
</tr>
<tr>
<td>A0h</td>
<td>No more XMS available</td>
<td>B0h</td>
<td>Smaller UMeg available</td>
</tr>
<tr>
<td>A1h</td>
<td>All XMS handles are allocated</td>
<td>B1h</td>
<td>No more UMeg available</td>
</tr>
<tr>
<td>A2h</td>
<td>Handle is invalid</td>
<td>B2h</td>
<td>Invalid UMeg segment address</td>
</tr>
<tr>
<td>A3h</td>
<td>Source handle is invalid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XMS Functions

The function number of the XMS functions is always passed in the AH register. In this section we’ll provide basic information on the XMM functions. A comprehensive list of all XMS functions is found in PC Intern (5th edition) from Abacus.
Chapter 10
Memory Management

Function 00h - Determine version number

<table>
<thead>
<tr>
<th>Input</th>
<th>AH = 00h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>AX = XMS version number</td>
</tr>
<tr>
<td></td>
<td>BX = internal revision number</td>
</tr>
<tr>
<td></td>
<td>DX = status of the HMA</td>
</tr>
<tr>
<td></td>
<td>BX = number of pages allocated all together</td>
</tr>
</tbody>
</table>

Use this function to determine the version number of the driver. Make certain the driver you’re working with is at least Version 2.0. If the value 1 is found in the HMA status, the HMA is accessible; otherwise no access to it is possible.

Function 03h - Globally activate A20

<table>
<thead>
<tr>
<th>Input</th>
<th>AH = 03h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>AX = error code</td>
</tr>
</tbody>
</table>

If you want HMA to be used reliably in read mode as well, this function must be called before or after the request to the HMA. Remember to close the A20 address line again before the program ends to avoid segment overruns with subsequent programs.

Function 04h - Globally close A20

<table>
<thead>
<tr>
<th>Input</th>
<th>AH = 04h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>AX = error code</td>
</tr>
</tbody>
</table>

You close the A20 address bus again with this function.

Function 05h - Locally activate A20

<table>
<thead>
<tr>
<th>Input</th>
<th>AH = 05h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>AX = error code</td>
</tr>
</tbody>
</table>

Use this function to activate the A20 address line locally. This means it will only be activated if the function has not already been called. This test is performed using a call counter, which is incremented using function 05h and is decremented using function 06h.

Function 06h - Locally disable A20

<table>
<thead>
<tr>
<th>Input</th>
<th>AH = 06h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>AX = error code</td>
</tr>
</tbody>
</table>

The counterpart to function 05h disables the A20 address line locally.
Memory Management

Function 07h - Query the status of A20

| Input:   | AH = 07h |
| Output:  | AX = status of the address bus |

Returns the value 0001h in the AX register if A20 address line is available; otherwise, it returns the value 0000h.

Function 08h - Determine free XMS

| Input:   | AH = 08h |
| Output:  | AX = length of the largest free block |
|          | DX = total size of the XMS inK |

Use this function to determine the amount of free XMS memory. Note the value that is returned is always 64K too large. This is due to the 64K size of the HMA which is also counted.

Function 09h - Allocate Extended Memory Block

| Input:   | AH = 09h |
|          | DX = size of the block in K |
| Output:  | AX = error code |
|          | DX = handle for further access to the EMB |

This function reserves an Extended Memory Block (EMB) in the XMS memory block. The allocation is only successful if a sufficiently large block is still free in the XMS. The allocated block can be addressed using the handle which is returned. Remember to free all allocated memory before ending the program. Otherwise, it won’t be available for subsequent programs.

Function 0Ah - Free Extended Memory Block

| Input:   | AH = 0Ah |
|          | DX = handle |
| Output:  | AX = error code |

Use this function to free an allocated XMS. After this function is invoked, the handle becomes invalid. The data is deleted.
Function 0Bh - copy memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Function</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Length of the block</td>
<td>1 Dword to be copied</td>
</tr>
<tr>
<td>04h</td>
<td>Handle of the source block</td>
<td>1 Word</td>
</tr>
<tr>
<td>06h</td>
<td>Offset in the source block</td>
<td>1 Dword the copying starts from here</td>
</tr>
<tr>
<td>0Ah</td>
<td>Handle of the target block</td>
<td>1 Word</td>
</tr>
<tr>
<td>0Ch</td>
<td>Offset in the target block</td>
<td>1 Dword the copying starts from here</td>
</tr>
</tbody>
</table>

Use this function to copy memory into, out of, or within the XMS. The following structure specifies the area involved in this operation:

Function 0Ch - Protect EMB against being moved

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 0Ch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DX = handle</td>
</tr>
<tr>
<td>Output:</td>
<td>AX = error code</td>
</tr>
<tr>
<td></td>
<td>DX: BX = The linear 32-bit address of the EMB in memory</td>
</tr>
</tbody>
</table>

If necessary, the XMM will move some blocks of memory so there aren’t any holes in the XMS. Use this function to protect a block against being moved. This can be important if you need to access the memory directly. The result is the linear address under which you can access the memory in DX:BX.

Function 0Dh - Deprotect blocked EMB

<table>
<thead>
<tr>
<th>Input:</th>
<th>AH = 0Dh</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DX = handle</td>
</tr>
<tr>
<td>Output:</td>
<td>AC = error code</td>
</tr>
</tbody>
</table>

Use this function to unprotect the block that was protected with the function 0Ch.
Function 0Eh - Fetch information about EMB

Input:
- AH = 0Eh
- DX = handle

Output:
- AX = error code
- DX = length of the EMB

These functions allow you to fetch information about a block of memory and to determine the number of XMS handles that are still free. If there were no errors, AX will contain the value 0001h. In that case, BH will contain the block counter of the EMB. The blocking counter is incremented each time the function 0Ch is called and it is decremented each time the function 0Dh is invoked. Therefore, if the value of BH is greater than 0, you have to call the function 0Dh BH-times to unprotect the block. BL contains the number of free EMB handles.

Make certain to allocate memory blocks that are as large as possible since the number of XMS handles is limited. An error occurred if AX contains the value 0000h. The cause of the error can then be determined using the error table. When the function has successfully completed, DX contains the size of the EMB in K.

Function 0Fh - Change EMB size

Input:
- AH = 0Fh
- BX = new size in K
- DX = handle

Output:
- AX = error code

You can use this function to change the size of an EMB later, if necessary. The block should not be protected if you want to change its size. If you reduce the size of the block, the data are irretrievably lost.

Now that you know the most important functions of the XMM, we’ll turn to their practical application. We’ll develop routines that you can use to allocate and deallocate XMS memory and to copy data into the memory or to fetch it from memory.

Let’s start with a function with which you can verify whether XMS is installed. The procedure checks first to see whether the XMM is present and then determines the entry point of the driver if necessary. Then it checks to see whether the driver is out-of-date (having a version number smaller than 2.0). An appropriate value is then assigned to the variable XMS_Available.

```asm
Procedure Check_for_XMS; assembler;
asm
  mov ax,4300h  ; Check whether driver installed
  int 2Fh
  cmp al,80h
  jne @No_XMSDriver
  mov ax,4310h  ; Get entry point address of driver
  int 2Fh
  mov word ptr XMST + 2,es
```

The procedures/functions on pages 273-276 are from MEMTEST.PAS or RMEM.PAS on the companion CD-ROM.
Chapter 10
Memory Management

Before you start using XMS memory, you should check to see how much memory is available. Use the function `XMS_free` for this purpose.

```pascal
function XMS_free : longint;
var xms_in_kb : word;
   xms_long: longint;
begin;
   asm
     mov ax,0800h                 { 8 = Get free memory }
     call dword ptr [XMST]
     mov xms_in_kb,dx
   end;
   xms_long := xms_in_kb;
   XMS_free := xms_long * 1024;
end;
```

You can allocate the memory once you know how much is available by using the function `Getmem_XMS`. It's similar to the Pascal function `Getmem`. The first parameter you pass to it is the variable in which the handle for accessing the allocated memory should be stored. The second parameter is the size of the block to be reserved in bytes.

```pascal
Function Getmem_XMS(VAR H : XMSHandle; Size : longint) : byte;
var bsize : word;
   Fresult : byte;
   xmsh : word;
begin;
   bsize := (size DIV 1024) + 1;
   asm
     mov ax,0900h                 { 9 = Allocate memory area }
     mov dx,bsize
     call dword ptr [XMST]
     cmp ax,1
     jne @Error_GetmemXms
     mov xmsh,dx
     mov Fresult,0
     jmp @End_GetmemXms
   @Error_GetmemXMS:
     mov Fresult,bl
   @End_GetmemXms:
     h := xmsh;
     Getmem_Xms := Fresult;
   end;
```

Use the function `Freemem_XMS` to free all allocated XMS memory again at the end of the program. The only parameter the function needs is the number of the handle of the EMB.

```
@No_XMSDriver:
   mov XMS_Available,1
@End_XMS_Check:
end;
```
Function Freemem_XMS(H : XMSHandle) : byte;
var fresult : byte;
begin;
  asm                            { A = deallocate memory area }
    mov ax,0a00h
    mov dx,h
    call dword ptr [XMST]
    cmp ax,1
    jne @Error_FreememXms
    mov fresult,0
    jmp @End_FreememXms
  @Error_FreememXms:
    mov fresult,bl
  @End_FreememXms:
  end;
end;

The allocated memory is no use to us if we cannot copy data into it or from it. Function Ram_2_XMS copies data from RAM to the XMS. The function XMS_2_Ram copies data from the XMS to RAM. Note that you must allocate the XMS memory first using function Getmem_XMS before you can copy data into the memory.

Function RAM_2_XMS(q : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := 0;              { 0 = RAM }
  XC.Q_Offset := q;
  XC.Z_Handle := h;
  XC.Z_Offset := nil;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Error_RAM2XMS
    mov fresult,0
    jmp @End_Ram2XMS
  @Error_RAM2XMS:
    mov fresult,bl
  @End_Ram2XMS:
  end;
end;

Function XMS_2_Ram(d : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := h;
  XC.Q_Offset := nil;
  XC.Z_Handle := 0;              { 0 = RAM }
  XC.Z_Offset := d;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Error_XMS2RAM
    mov fresult,0
    jmp @End_XMS2Ram
  @Error_XMS2RAM:
    mov fresult,bl
  @End_XMS2Ram:
  end;
end;
Chapter 10
Memory Management

The last function is **XMS_2_XMS**. It lets you copy data within the XMS. This can be useful, for example, if you load a picture from the disk and then you want to duplicate it. The function requires for the first parameter the handle of the source block, and for the second parameter the handle of the target block. You enter the size in bytes for the last parameter.

```pascal
Function XMS_2_XMS(d : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
    XC.Size     := Size;
    XC.Q_Handle := h;
    XC.Q_Offset := nil;
    XC.Z_Handle := 0;              { 0 = RAM }
    XC.Z_Offset := d;
    asm
        mov si,offset XC
        mov ax,0B00h
        call dword ptr [XMST]
        cmp ax,1
        jne @Error_XMS2RAM
        mov fresult,0
        jmp @End_XMS2Ram
    @Error_XMS2Ram:
        mov fresult,bl
    @End_XMS2Ram:
    end;
end;
```

**The Flat Memory Model: The Solution To Your Memory Problems**

In many case, using XMS is an acceptable solution to the memory problem. However, it has a slight disadvantage: You can't access the XMS directly. Instead, you always have to copy block-by-block into or from the XMS which can require a lot of time.

So, if you need to directly access the memory, or if speed is important, you will have to use the flat model. The flat model makes it possible to linearly access the entire memory of the PC, and it is also supported by the new Borland Pascal 8.0.

The idea for the flat model is based on following technique: The processor, which must be at least a 386, is switched into Protected Mode. There the segment limits for the FS and GS registers are adjusted to 4 Gigabyte (Gbyte). Then we switch the processor back to Real Mode without resetting it. That way we have, at least in theory, up to 4 Gbyte available.

**The technical background for the flat model**

So how does this all work? First, we need to talk about memory management in protected mode. The memory in protected mode is not addressed directly, instead uses selectors. Selectors are pointers to a descriptor. All the information about a given segment is stored in a descriptor: Size, physical address and
access permissions. The size of a segment is set by default at 0FFFFh when the computer is switched on. These are the "magic" 65,536 bytes which normally is the most you can address under DOS. You won't face this limit in protected mode. You can address a maximum of 4 Gbyte of memory. You simply need to modify the descriptor for the particular segment appropriately. This modification is controlled by the Global Descriptor Table (GDT). It contains the segment information.

So, if you want to switch to the flat model, first you have to construct a GDT. In principle, the GDT can be considered to be an array of descriptors. The following table shows how a descriptor is created:

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>Length of the segment</td>
</tr>
<tr>
<td>24 bits</td>
<td>Physical address</td>
</tr>
<tr>
<td>8 bits</td>
<td>Access permissions</td>
</tr>
<tr>
<td>16 bits</td>
<td>Additional bits for segment length</td>
</tr>
</tbody>
</table>

For our purposes, the GDT must contain a null segment at first (all values are = 0). Next comes the modified segment for the access of up to 4 Gbyte. The lower 16 bits are set to 0FFFFh, and the last 16 bits for the segment length receive the value 0FFCFh. We set the physical address to 0. That's how the segment at the top of the memory starts. The access permissions are allocated using a value of 92h with highest priority for a data segment.

After we have finished constructing a GDT, we only need to load it using the command lgdt. After we have masked out the interrupts using cli so we don't get any interference, we can switch to Protected Mode. In Protected Mode, the first thing to do is to delete the execution pipe of the processor using a jump. That's important because otherwise the machine will crash, since it still has commands in the prefetch queue. Now adjust the segments to 4 Gbyte by assigning the appropriate selector to the segment registers. After you have taken care of that, you can switch back to Real Mode without resetting the processor. Here, too, a jump must occur directly after switching over. Don't forget to activate the interrupts again at the end.

### How the flat model is programmed

This an assembler procedure for initializing the flat model. The GDT is declared externally:

```
Enable_4Giga proc pascal
    mov GDT_Off[0],16
    mov eax,seg GDT
    shl eax,4
    mov bx,offset GDT
    movzx ebx,bx
    add eax,ebx
    mov bx,eax
    mov bx,offset GDT
    movzx ebx,bx
    add eax,ebx
    mov dword ptr GDT_Off[2],eax
    lgdt pword ptr GDT_Off ; load GDT

Enable_4Giga endp
```

The procedures on pages 277-279 are from RMEMASM.ASM on the companion CD-ROM.
Next, we need a procedure to access the memory. Unfortunately, the RMEM (real memory) can't be addressed using the lodsb/stosb/movsb instructions. Instead, the RMEM must be addressed using its physical address. To do that, we store each desired address in a Longint variable which we can then use to address the memory. An access might look like this:

```
mov ebx, variable position
mov al, gs: [ebx]
```

Before you access the memory, you still need to make sure that the 20th address bit is cleared. Normally this causes things to be switched back to the beginning of the memory. If you are accessing the expanded memory using an XMS driver, for example, the driver itself takes care of the matter.

However, we must clear the bit. Direct programming here would involve a lot of effort since you have to use the keyboard controller. However, we can simply use the functions provided by the HIMEM driver since we're using it anyway.

Procedure `mem_Write` is one example of how to access RMEM. By using `mem_Write`, you copy a block from the main memory into the RMEM. The first parameter it requires is the target in RMEM. For the second parameter, you indicate the offset of the pointer to the source region in main memory, followed by the segment of the pointer. The last parameter represents the length of the block to be copied.
Procedure `mem_read` is the opposite of `mem_Write`. Use it to read a block from the RMEM back into RAM. The first parameter you provide it is the source position in the RMEM; the second is the offset and then the segment of the pointer to the target region in RAM. The last parameter indicates the length to be copied in bytes. You can even indicate the video RAM of the VGA card for the target region.

```
procedure Rmem_read(source:longint; destination:pointer;length:word);
{
    ;***************************************************************************
    ;***                                                                   ***
    ;***      Copies a block from RMEM to RAM                              ***
    ;***                                                                   ***
    ;***************************************************************************

begin
    if source + length < Rmem_Max then begin
        Segm:=seg(destination^);
        Offs:=ofs(destination^);
        inc(Segm,Offs div 16);
        Offs:=Offs mod 16;
        inc(source,Mbyte1);
```

The procedures on pages 279-282 are from RMEM.PAS on the companion CD-ROM
Before you apply the RMEM procedures, you have to verify every time whether the appropriate memory is even available. The worst scenario for the unit is one where a memory manager like EMM386, QEMM or something similar is already installed. The unit isn't compatible with these memory managers because they work based on a similar principle. Function Multitasker_active lets you easily determine whether one of these programs is active.

You can verify whether sufficient memory is available using the procedure Memory_Checks. It uses the HIMEM driver to examine any free main memory available and the XMS memory.
Memory Management

var xmsfree, mainfree : word;
begin;
{ Get Free XMS - memory }
xmsfree := xms_free;
{ Get Main Memory }
mainfree := memavail div 1024;
{ Message, if not enough free memory }
if (xmsfree < minxms) or (mainfree < minmain) then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Sorry, not enough memory available !');
  writeln('You need
           Available');
  writeln('XMS :     ',minxms :6,' KB         ',xmsfree:4,' KB');
  writeln('Main:     ',minmain:6,' KB         ',mainfree:4,' KB');
  halt(0);
end;
end;

You should set up your own management unit for managing the RMEM. Our solution here is only valid if you know exactly what you're doing with the memory. That will likely be the case if you are writing code for a demo, but if you should make the RMEM handling available to other, perhaps less-experienced users, you should consider using Freemem and sort routines.

function Rgetmem(Var rpos : longint; rsize : longint) : boolean;
{
*************************************************************************
***                                                                   ***
***        A simplified Getmem-Procedure for RMEM                     ***
***                                                                   ***
*************************************************************************
}
begin;
if Rmemposi+rsize > Rmem_max then begin;
  Rgetmem := false;
end else begin;
rpos := Rmemposi;
  inc(Rmemposi,rsize);
  Rgetmem := true;
end;
end;

Finally, we would like to give you two other procedures that you can use to switch your system into RMEM mode and back again without problems from within Pascal. You merely have to provide the procedure enable_Realmem with the desired RMEM memory in K; the procedure takes care of the rest for you. It tests to see whether a HIMEM driver is installed, whether the requested memory is available and whether a multitasking program is active. Then it allocates the needed memory and switches into RMEM mode.

procedure enable_Realmem(Min : word);
{
*************************************************************************
***                                                                   ***
***               Switches to RMEM - Mode                             ***
***    There must be "MIN" KB free XMS-Memory available !             ***
*************************************************************************
}
begin
{ Check for Multitasker ... }
if multitasker_active then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Processor already in V86 mode !');
  writeln('HIMEM.SYS is required ! ');
  halt(0);
end;

{ XMS driver installed ? }
if not XMS_available then begin;
  asm mov ax,0003; int 10h; end;
  writeln('No XMS or Himem-driver available');
  writeln('Please reboot your System using HIMEM.SYS !!!');
  halt(0);
end;

{ Allocate required memory }
error := Getmem_XMS(My_XmsHandle,min*1024);
if error <> 0 then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Error during memory-allocation !');
  writeln('We need at least ','Min,' KB of free XMS Memory !!!');
  writeln('Please reboot your System using HIMEM.SYS');
  writeln;
  halt(0);
end;

{ Get physical Start position }
Rmemposi := XMS_lock(My_XmsHandle);
if rmemposi < 1000000 then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Error during memory-fixing !');
  writeln('We need at least ','Min,' KB of free XMS Memory !!!');
  writeln('Please reboot your System using HIMEM.SYS');
  writeln;
  halt(0);
end;

{ Enable }
Enable_4Giga;
end;

Of course, you should “clean up” the program by freeing the allocated XMS memory; otherwise, it cannot
be used by other programs. Use procedure Exit_Rmem for this. This procedure must be invoked at the end
of the program if you have used RMEM.

procedure Exit_Rmem;
{***************************************************************************
 *** Exit-Procedure of RMEM, MUST be called !
 ***
***************************************************************************}
begin;
  { unlock block }
  XMS_unlock(My_XmsHandle);
  { Free memory }
  Freemem_XMS(My_XmsHandle);
end;
In this chapter we'll talk about other PC components such as the timer, the interrupt controller, the real-time clock chip (RTC) and the DMA controller. These components let you use your computer's hardware to the fullest extent possible.

**Interrupts**

As the name suggests, an interrupt causes the computer system to stop its normal processing so it can attend to other tasks. An interrupt can be generated by several circumstances:

*Software interrupt*

This type of interrupt is caused by an instruction in the program.

*Hardware interrupt*

This type of interrupt is the result of some action by a hardware device, such as pressing a key or moving a mouse.

*NMI* (NonMaskable Interrupt) This is a high-priority interrupt that cannot be disabled. It's used to report problems like parity, bus or other errors.

Let's talk about the basic principles of interrupt programming and show you several different applications.

When an interrupt is invoked, the processing of the current program is interrupted. A section of code (routine) that is defined to handle this interrupt is executed. This routine ends with the assembler instruction IRET, which represents Interrupt Return. When the interrupt is invoked, the FLAG register, CS and IP are stored on the stack. Using these within the interrupt, you can easily determine the point in the main program where the interrupt was triggered.

A table in RAM called the interrupt vector table contains the addresses of the routines that are called if a corresponding interrupt is invoked. These addresses are simply pointers that you might recognize from
earlier chapters. For example, if you want to assign a pointer to the address of your routine in Pascal, you can use the following:

```
My_Pointer := ADR(The_Procedure);
```

A shorter version is:

```
My_Pointer := @The_Procedure;
```

Pascal supports this simple means of working with interrupts. To determine the address of a specific interrupt, such as the timer interrupt, you can use this statement:

```
VAR Pointer_to_Timer : pointer;
GetIntvec (8,Pointer_to_Timer);
```

You can change the contents of an interrupt vector to go to your own routine by using the following:

```
SetIntvec(8,@My_Timer_Procedure);
```

Important: You must include the word `Interrupt` after the procedure declaration of the interrupt routine. This is necessary so Pascal executes the IRET instruction at the end of the interrupt routine and not the normal RET instruction. The declaration might look like the following:

```
Procedure My_Timer_Procedure; Interrupt;
begin;
  writeln ('Hello! This is an interrupt!');
  port[$20] := $20;
end;
```

Note that Interrupt 8 is a hardware interrupt. At the end of a hardware interrupt you must let the interrupt controller know the interrupt has been handled and is terminated. To do this, you send the command EOI (End of Interrupt, = 20h) to the controller at port 20h.

You might be asking yourself, "So what good does all this do me?" Let's answer that question by using the timer interrupt as an example. Let's say you want to write a program that displays ASCII character #1 every half-second and ASCII character #2 every half-second. You could, of course, use a loop in the main program to check the clock time continuously and display the desired character every half-second. However, that would make it difficult to process other parts of the program.

Another option is to divert the normal timer interrupt, which is called up 18.2 times per second, to your own routine. In this routine count how often the interrupt was invoked. Every ninth interrupt you display the desired character and reset the counter. Now you can process the other parts of your program and display the desired screen characters every half second. You'll have to reset the interrupt at the end of the program to its original routine, which you determined using `GetIntvec`. Trying to debug an interrupt-driven program step-by-step usually ends in disaster.

Now that you know how to program your own interrupts in Pascal, let's talk about interrupt programming in more detail. One question is: "What do the procedures `GetIntvec` and `SetIntvec` really do?" They use the DOS functions 25h and 35h of the DOS interrupt 21h. That brings up another phenomenon: An interrupt can have not only a routine but also several subroutines.

DOS provides interrupt 21h for all important functions of the operating system. To tell the interrupt which subroutine you want to execute now, there is almost no way to get around the (inline) assembler. You must
pass the number of the desired interrupt function to the AH register, and any other required parameters to other registers, which will be different from one function to another. With function 25h you have to initialize the following registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents/ Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>25h</td>
</tr>
<tr>
<td>AL</td>
<td>Number of the interrupt which is to be set</td>
</tr>
<tr>
<td>DS</td>
<td>Segment address of your interrupt procedure</td>
</tr>
<tr>
<td>DX</td>
<td>Offset address of your interrupt procedure</td>
</tr>
</tbody>
</table>

With function 35h, not only do you have to pass values, you'll also receive return values. After all, it is determining the address of the current interrupt routine, and you have to get the result somehow. Here is how the registers are allocated:

<table>
<thead>
<tr>
<th>Input</th>
<th>AH 35h</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Number of the interrupt to be determined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>ES Segment address of the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BX Offset address of the routine</td>
</tr>
</tbody>
</table>

So what would an assembler routine for determining an interrupt look like?

```pascal
.get_interrupt proc pascal int_nr: BYTES
  mov ah,35h
  mov al,int_nr
  int 21h
  mov int_ofs,bx
  mov bx,es
  mov int_seg,es
  RET
.get_interrupt endp
```

Your work will be more complicated if instead of completely overwriting the original interrupt, first execute your own routine and then call the original interrupt. In that case, first determine the address of the original interrupt and save it in a DWORD variable. Now you can reset the interrupt to your own routine.

In this routine you must first save all the registers that you’ll change and that are important for the target interrupt. Now you can perform your own processing. For example, you can check to see whether a key was pressed or whether a certain value is located at a particular place in memory.

Remember to reset the registers after your code has been processed. Now you jump directly to the original interrupt routine. You don't have to invoke IRET; the original routine takes care of that for you. The code might look like the following:
The Programmable Interval Timer (PIT)

The PIT is like an alarm clock. It can be set to remind your program to perform a certain task. We use the counter 0 of the PIT for that purpose. This counter controls the timer interrupt of the PC, which otherwise works at 18.2 interrupts per second.

Let’s take a close look at the timer hardware.

The PIT hardware

The timer chip that is used is an 8254 PIT. It contains three independent 16-bit counters. Each of these three counters can perform one of these six logical operations:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Interrupt generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 0</td>
<td>Interrupt generator</td>
</tr>
<tr>
<td>Mode 1</td>
<td>Programmable monoflop</td>
</tr>
<tr>
<td>Mode 2</td>
<td>Cycle generator</td>
</tr>
<tr>
<td>Mode 3</td>
<td>Rectangular wave signal generator</td>
</tr>
<tr>
<td>Mode 4</td>
<td>Triggering the exit through software</td>
</tr>
<tr>
<td>Mode 5</td>
<td>Triggering the exit through hardware</td>
</tr>
</tbody>
</table>

We’re especially interested in operating modes 1 and 3.
Each of these three channels is controlled by its own counter. The counter can be set to work in either binary or in BCD mode. Channel 0 controls the system timer, which is responsible for operations like the system time, time of day or disk timeout. The system uses channel 1 to control the DRAM-refresh. Finally, channel 2 is used to generate sound through the PC speaker.

The channel consists of several registers. We’re interested in the counter register, the two counter input registers ZEL and ZEH and the control register. There is also a status register and two counter output registers, ZAL and ZAH.

The following table summarizes these different counters:

<table>
<thead>
<tr>
<th>Counter 0</th>
<th>Gate 0Ch</th>
<th>System Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk 0Ch</td>
<td>Generates IRQ0</td>
<td></td>
</tr>
<tr>
<td>Out 0Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter 1</td>
<td>Gate 1Ch</td>
<td>DRAM refresh</td>
</tr>
<tr>
<td>CLK 1Ch</td>
<td>REFTIME</td>
<td></td>
</tr>
<tr>
<td>OUT 1Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter 2</td>
<td>Gate 2Ch</td>
<td>Loudspeaker frequency</td>
</tr>
<tr>
<td>CLK 2Ch</td>
<td>Loudspeaker on</td>
<td></td>
</tr>
<tr>
<td>OUT 2Ch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**The control register**

The control register controls the activity of the individual channels. It’s addressed through port 43h. You assign the number of desired channels, the type of command, the operating mode and the counter format to the control register. You use the status register to read the current settings of the control register. The following table describes the purpose of these bits:

<table>
<thead>
<tr>
<th>Bit 7-6</th>
<th>Meaning</th>
<th>Bit 3-1</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>counter 0</td>
<td>000</td>
<td>mode 0, interrupt generation</td>
</tr>
<tr>
<td>01</td>
<td>counter 1</td>
<td>001</td>
<td>mode 1, programmable monoflop</td>
</tr>
<tr>
<td>10</td>
<td>counter 2</td>
<td>010</td>
<td>mode 2, cycle generator</td>
</tr>
<tr>
<td>11</td>
<td>read back mode</td>
<td>011</td>
<td>mode 3, rectangular wave signal generator</td>
</tr>
<tr>
<td>Bit 5-4</td>
<td>command type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>latch counter</td>
<td>100</td>
<td>mode 4, triggering the output through software</td>
</tr>
<tr>
<td>01</td>
<td>read/write the counter low byte</td>
<td>101</td>
<td>mode 5, triggering the output through hardware</td>
</tr>
<tr>
<td>10</td>
<td>read/write the counter high byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>read/write the counter low byte then the counter high byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 0</td>
<td>select the counter format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16-bit binary (default)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BCD-format</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The counter register

The counter register is a 16-bit register that counts from 0 to 0xffff. A write access to one of the counter input registers starts the counter. A typical initialization of the counter 0 would be as follows:

\[
\begin{align*}
\text{port}[43] & := \$36; \\
\text{port}[40] & := 12; \\
\text{port}[40] & := 34;
\end{align*}
\]

Rereading the registers

The data in the counters can be reread from the respective counter registers. The read-back mode is selected for this. Next, you specify whether you want to read the counter or the status byte. The following table explains the structure:

<table>
<thead>
<tr>
<th>Bit 7-6</th>
<th>Meaning</th>
<th>Bit 3-0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 5-4</td>
<td>Indicate read back command</td>
<td>Bit 3-0</td>
<td>Select counter</td>
</tr>
<tr>
<td>11</td>
<td>Must have this value</td>
<td>1000</td>
<td>Select counter 2</td>
</tr>
<tr>
<td>10</td>
<td>Read current counter value</td>
<td>100</td>
<td>Select counter 1</td>
</tr>
<tr>
<td>10</td>
<td>Read status byte</td>
<td>10</td>
<td>Select counter 0</td>
</tr>
<tr>
<td>11</td>
<td>Read nothing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value of the status byte has the following meaning:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Status of the OUT signal</th>
<th>Bit 3-1</th>
<th>Select the operating mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OUT signal 0 (low)</td>
<td>0</td>
<td>Mode 0</td>
</tr>
<tr>
<td>1</td>
<td>OUT signal 1 (high)</td>
<td>1</td>
<td>Mode 1</td>
</tr>
<tr>
<td></td>
<td>Counter flag NIL-COUNT set?</td>
<td>10</td>
<td>Mode 2</td>
</tr>
<tr>
<td>0</td>
<td>The counter was via the input registers and can be read</td>
<td>11</td>
<td>Mode 3</td>
</tr>
<tr>
<td>1</td>
<td>The control register was written but the expected counter values haven’t yet appeared.</td>
<td>100</td>
<td>Mode 4</td>
</tr>
<tr>
<td></td>
<td>Type of transfer command</td>
<td>101</td>
<td>Mode 5</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td></td>
<td>Choice of the counter format</td>
</tr>
<tr>
<td>1</td>
<td>Read/write low byte</td>
<td></td>
<td>0 16-bit binary (default)</td>
</tr>
<tr>
<td>10</td>
<td>Read/write high byte</td>
<td></td>
<td>1 BCD format</td>
</tr>
<tr>
<td>11</td>
<td>First read/write low byte, then high byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the PIT

The PIT is most often used to synchronize multiple events. For example, you may need to play music through a sound card at periodic intervals during a program or to perform sophisticated screen display based at fixed time intervals.

An example is when you have to update or switch a screen display every second or third retrace. How can you make a switch like this? First, you must determine the frequency at which the screen is updated. This is 70 Hz for most display cards in normal 320 x 200 mode. So we have to generate an interrupt shortly before the end of the display of a picture to be able to synchronize with the retrace and then to perform the switch. In our case, this means that to generate the interrupt shortly before, its frequency should be higher than 70 Hz. How much higher depends on the amount of calculations within your routines.

TIMER.PAS is an example of such a program. First, it determines the frequency of the screen display by using the timer interrupt to check how long it can change the value of a variable while waiting for the retrace. The timer is next switched from the rectangular mode into the monoflop mode. The interrupt routine has to program the timer from scratch every time. To illustrate where the interrupt occurs, the screen is first switched to red and then to green by the interrupt. This is the program listing:

```
program test_timer;
uses crt,dos;
Var OTimerInt    : pointer;
   Timerfreq    : word;
   Orig_freq    : word;
   Sync_counter : word;
   Ticounter    : word;
procedure SetColor (Nr, R, G, B : BYTE);
begin;
   asm
      mov   al,Nr
      mov   dx,03C8h
      out   dx,al
      mov   dx,03C9h
      mov   al,r
      out   dx,al
      mov   al,g
      out   dx,al
      mov   al,b
      out   dx,al
   end;
end;
procedure waitretrace;
begin;
   asm
      MOV   DX,03dAh
      @MD_R:  
      IN    AL,DX
      TEST   AL,8d
      JE    @MD_R
      @MD_D:  
      IN    AL,DX
      TEST   AL,8d
      JNZ   @MD_D
   end;
```

You can find TIMER.PAS on the companion CD-ROM

289
procedure SetTimerOn(Proc : pointer; Freq : word);
  var icounter : word;
  oldv : pointer;
begin;
  asm cli end;
  icounter := 1193180 DIV Freq;
  Port[$43] := $36;
  Port[$40] := Lo(ICounter);
  Port[$40] := Hi(ICounter);
  Getintvec(8,OTimerInt);
  SetIntVec(8,Proc);
  asm sti end;
end;

procedure New_Timerfreq(Freq : word);
  var icounter : word;
begin;
  asm cli end;
  icounter := 1193180 DIV Freq;
  Port[$43] := $36;
  Port[$40] := Lo(ICounter);
  Port[$40] := Hi(ICounter);
  asm sti end;
end;

procedure SetTimerOff;
  var oldv : pointer;
begin;
  asm cli end;
  Port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  SetIntVec(8,OTimerInt);
  asm sti end;
end;

procedure Syncro_interrupt; interrupt;
begin;
  inc(Sync_counter);
  Port[$20] := $20;
end;

procedure Synchronize_timer;
begin;
  Timerfreq := 120;
  SetTimerOn(@Syncro_interrupt,Timerfreq);
  Repeat
    dec(Timerfreq,2);
   wartrace;
    New_Timerfreq(Timerfreq);
    Sync_counter := 0;
    waitretrace;
    until (Sync_counter = 0);
end;

procedure Timer_Handling;
begin;
  setcolor(0,0,63,0);
The Programmable Interrupt Controller (PIC)

The Programmable Interrupt Controller is another important component for controlling a PC's interrupt electronically. Technically, a PC has two interrupt controllers. The first is found at the address 20h, the second at A0h. However, the second controller is joined with the first through “cascading” (a series of chips or other devices that are connected). This cascade is implemented through channel 2 of the controller at 20h.

The PIC works based on a priority logic. That means the interrupt with the lowest value, interrupt 0, has the highest priority. As a result, interrupt 7 has the lowest priority and not interrupt 15, like you might think. This is because the interrupts 8 to 15 are connected through interrupt 2, and therefore are higher than interrupt 3 in the priority logic. The following table summarizes this priority logic:
The PIC uses this priority logic to simplify the operation of an interrupt driven operating system which helps to reduce the time spent managing interrupts.

As this is an overview of the PIC operation, if you need more information, refer to a book on system programming.

The PC distinguishes between four types of interrupts:

- Maskable interrupts (MI)
- Nonmaskable interrupts (NMI)
- Hardware interrupts
- Software interrupts

**Maskable interrupts**

Maskable interrupts are triggered by the PC hardware. They can be blocked by using the following assembler command:

```
cli ; clear all maskable interrupts
```
Conversely, they can be reactivated using this instruction:

```assembly
sti ; turn interrupts back on
```

Furthermore, it's possible to selectively switch off the interrupts. This is done by masking them at a given port. For interrupts IRQ 0 to IRQ 7, this is port 020h; for other interrupts this is port 0A0h.

### Nonmaskable interrupts

Our work with interrupts becomes slightly more complicated with Nonmaskable Interrupts (NMI). They're invoked, for example, if a parity error appears or if the BIOS invokes the interrupt. Even these interrupts can be switched off. This is useful in the situation where you are executing operations that are very critical with respect to the timing, and you don't want them to be interrupted. You can switch off the NMIs with the following assembler instruction:

```assembly
mov al,80h
out 70h,al
```

You can later reactivate them using these instructions:

```assembly
mov al,00h
out 70h,al
```

### Hardware interrupts

These are the interrupts triggered by hardware devices. The hardware usually generates an interrupt on one of the interrupt channels. They are treated like maskable interrupts.

### Software interrupts

Software interrupts are the interrupts that are triggered by software. They can be system interrupts of the operating system, like the int 21h for example. They can also include interrupts that the user installed to control hardware or other applications. You can switch them off using the `cli` instruction and reactivate them using `sti`.

### The DMA Controllers

By using the DMA controller, large blocks of data can be quickly transferred from a peripheral device to memory or vice versa. The method we'll use is somewhat radical: The DMA controller simply switches off the processor and then transfers the data.

A PC has two DMA controllers. The first controller is responsible for 8-bit transfers; the second controller regulates 16-bit transfers. The second controller provides a cascade connection to the first controller using channel 0.

The following table summarizes each channel:
Let’s look at a practical example: How to program the DMA controller to send data to the SoundBlaster.

The general procedure is listed in the following seven steps:

1. Block DMA channel
2. Set transfer mode
3. Clear flip-flop
4. Write address of the data block
5. Write page of the data block
6. Write transfer length
7. Release DMA channel again

Masking a DMA channel

Let’s see how we could implement these steps. First, let’s address the masking and unmasking of the DMA channel. There are two ways to mask a DMA channel:

1. Masking using the Single Mask register

You can mask and unmask only one channel with this register. The register is found at port 0Ah for channels 0-3 and at port 0D4h for channels 4-7. The bits have the following meaning:
To mask out channel 1 of controller 1 (which you need to do for the standard SoundBlaster), you write the value 5 to port 0Ah. Write the value 1 to restore the channel after programming the DMA controller.

2. Masking using the All Mask register

You can mask or unmask all the DMA channels of a controller at one time with the All Mask register. The advantage is that you don't have to mask each channel individually like you do with the Single Mask register. However, you must indicate whether each channel should be masked or unmasked. So, if you’re uncertain about each channel, you should use Single Mask. The All Mask register is at port 0Fh found for the first controller and at port 0DEh for the second controller.

The following table shows the allocation of the bits:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bit 1</td>
<td>Mask/unmask channel 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Channel - switch on 1 mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Channel - switch off 1 mask</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Mask/unmask channel 3</td>
<td></td>
<td>Channel - switch on 3 mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Channel - switch off 3 mask</td>
</tr>
<tr>
<td>Bit 2</td>
<td>Mask/unmask channel 2</td>
<td></td>
<td>Channel - switch on 2 mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Channel - switch off 2 mask</td>
</tr>
</tbody>
</table>

Setting the transfer mode

The DMA controller handles different transfer modes. Before a DMA transfer, you first set the desired mode. To do so, you send the mode bit to either port 0Bh for the first DMA controller or to port 0D6h for the second controller. First, the following table describes the meaning of the bits here:
<table>
<thead>
<tr>
<th>Bit 7-6</th>
<th>Meaning</th>
<th>Bit 3-2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Request mode</td>
<td>00</td>
<td>Verify transfer</td>
</tr>
<tr>
<td>01</td>
<td>Single mode</td>
<td>01</td>
<td>Writing transfer</td>
</tr>
<tr>
<td>10</td>
<td>Block mode</td>
<td>10</td>
<td>Reading transfer</td>
</tr>
<tr>
<td>11</td>
<td>Cascade mode</td>
<td>11</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Bit 5</td>
<td>Increment/decrement addresses</td>
<td>Bit 1-0</td>
<td>Channel selection</td>
</tr>
<tr>
<td>1</td>
<td>Address is decremented</td>
<td>00</td>
<td>Use channel 0</td>
</tr>
<tr>
<td>0</td>
<td>Address is incremented</td>
<td>01</td>
<td>Use channel 1</td>
</tr>
<tr>
<td>11</td>
<td>Switch on autoinitialization</td>
<td>10</td>
<td>Use channel 2</td>
</tr>
<tr>
<td>0</td>
<td>Switch off autoinitialization</td>
<td>11</td>
<td>Use channel 3</td>
</tr>
</tbody>
</table>

Let’s take a closer look at the meaning of the individual bits:

**Mode selection**

Bits 7 and 6 specify the transfer mode to be used. There are four modes:

- Request mode
- Single mode
- Block mode
- Cascade mode

The mode in which we’re most interested is the single mode. Exactly one byte is transferred in this mode. Then the address counter is incremented or decremented, depending on the mode that was selected. If the counter has reached its target value, autoinitialization is performed if this was specified.

**Incrementing/decrementing addresses**

Bit 5 specifies whether the defined block should be transferred forward (incremented) or backward (decremented).

**Autoinitialization on/off**

Bit 4 specifies whether autoinitialization will occur at the end of a transfer. Autoinitialization resets the address register and counter registers to their initial programmed values.
Transfer selection

Bits 3 and 2 specify the direction of the transfer. A "write transfer" means a transfer from the source to the addressed main memory. A "read transfer" means a transfer from the addressed main memory to the desired target. A "verify transfer" verifies the completed transfer.

Channel selection

Bits 1 and 0 specify the channel to be used for the transfer. Remember, DMA channel 4 corresponds to channel 0 of the second controller. Therefore the differences between the first and the second controller are really only in the different port addresses.

Clearing the DMA flip-flop

When programming the DMA controller, most action require you first to clear the flip-flop. Do this with the "Clear flip-flop" register. This is located at port 0Ch for the first controller and at 0D8h for the second controller. Clear this register by writing the value 0 to the appropriate port.

Specifying the address of a data block

You specify the address of the data block to be transferred as a physical page. This is calculated using the following formula:

\[ \text{adr} := 16 \times \text{longint}(\text{Seg}(\text{Block}^*)) + \text{ofs}(\text{Block}^*) \]

The 32-bit number derived using this formula contains the page in the upper 16 bits and the offset in the lower 16 bits. To set the address, first the offset must be written to the DMA address register. This is located in the ports listed in the table to the right (depending on the channel):

At the appropriate port, first write the low byte of the offset, then the high byte.

Now you can transfer the page address to the controller. There are two different registers for this: A lower page register for the lower 8 bits of the address and an upper page register for the upper 4 bits of the address. An address up to 256 Meg can be transferred using this method. This is enough for the typical situation. The lower page register overwrites the contents of the upper page register with 0 if it is programmed. Therefore, it has to be programmed before the upper page register. The following table lists the ports of these registers:

<table>
<thead>
<tr>
<th>Channel</th>
<th>Lower Page</th>
<th>Upper Page</th>
<th>Channel</th>
<th>Lower Page</th>
<th>Upper Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>87h</td>
<td>487h</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>83h</td>
<td>483h</td>
<td>5</td>
<td>8Bh</td>
<td>48Bh</td>
</tr>
<tr>
<td>2</td>
<td>81h</td>
<td>481h</td>
<td>6</td>
<td>89h</td>
<td>489h</td>
</tr>
<tr>
<td>3</td>
<td>82h</td>
<td>482h</td>
<td>7</td>
<td>8Ah</td>
<td>48Ah</td>
</tr>
</tbody>
</table>
Adjusting the size of a DMA transfer

Before specifying the size of the block to be transferred to the DMA controller, you first have to clear the flip-flop. Then you transfer the block size - low byte first, then the high byte to the DMA Count register. This specifies the amount of data to be transferred. The table on the right lists the ports of the counter register.

The following is a small unit for programming the DMA controller. It includes all the required procedures. You can either program all the steps individually or perform the entire initialization using the procedure `DMA_Init_Transfer`.

The procedure's first parameter is the number of the channel to be used. The next parameter is the desired mode. The third parameter is a pointer to the data block in RAM. Finally, the fourth parameter is the size of this data block.

```
unit DMA;
interface
TYPE DMAarray = array[0..7] of byte;
CONST
  { DMA controller addresses }
  DMA_Address           : DMAarray = ($00,$02,$04,$06,$C0,$C4,$C8,$CC);
  DMA_Count             : DMAarray = ($01,$03,$05,$07,$C2,$C6,$CA,$CE);
  DMARead_status_Reg    : DMAarray = ($08,$08,$08,$08,$D0,$D0,$D0,$D0);
  DMAWrite_status_Reg   : DMAarray = ($08,$08,$08,$08,$D0,$D0,$D0,$D0);
  DMAWrite_requ_Reg     : DMAarray = ($09,$09,$09,$09,$D2,$D2,$D2,$D2);
  DMAWr_single_mask_Reg : DMAarray = ($0A,$0A,$0A,$0A,$D4,$D4,$D4,$D4);
  DMAWr_mode_Reg        : DMAarray = ($0B,$0B,$0B,$0B,$D6,$D6,$D6,$D6);
  DMAClear_Flipflop     : DMAarray = ($0C,$0C,$0C,$0C,$D8,$D8,$D8,$D8);
  DMARead_Temp_Reg      : DMAarray = ($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
  DMAMaster_Clear       : DMAarray = ($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
  DMA_Clear_Mask_Reg    : DMAarray = ($0E,$0E,$0E,$0E,$DC,$DC,$DC,$DC);
  DMA_Wr_All_Mask_Reg   : DMAarray = ($0F,$0F,$0F,$0F,$DE,$DE,$DE,$DE);
  DMA_Lower_Page        : DMAarray = ($87,$83,$81,$82,$00,$8B,$89,$8A);
  DMA_Higher_Page       : Array[0..7] of word
    = ($487,$483,$481,$482,$00,$48B,$489,$48A);
  { Mode Register DMA_Wr_mode_Reg }
  Request_mode  = $00;
  Single_mode   = $40;
  Block_mode    = $80;
  Cascade_mode  = $C0;
  Addresses_Decrement = $20;
  Addresses_Increment = $00;
  AutoInit_Enable = $10;
  AutoInit_Disable = $00;
  Check_transfer = $00;
  Write_Transfer = $04;
  Read_Transfer = $08;
  Set_Request_Bit = $04;
  Clear_Request_Bit = $00;
  Set_Mask_Bit   = $04;
  Clear_Mask_Bit = $00;
procedure DMA_Mode_set(Channel,Mode : byte);
procedure DMA_NormMode_set(Channel,Mode : byte);
procedure DMA_Clear_Flipflop(Channel : byte);
```

You can find DMA.ASM on the companion CD-ROM
procedure DMA_Startaddress(Channel : byte; Start : pointer);
procedure DMA_Blocksize(Channel : byte; size : word);
procedure DMA_Channel_maskon(Channel : byte);
procedure DMA_Init_Transfer(Channel,Mode : byte; p : pointer; s : word);

implementation

TYPE
  pt = record                           ( makes simple pointer )
    ofs, sgm : word;                   ( handling possible )
  end;
procedure DMA_Mode_set(Channel,Mode : byte);
begin;
  port[DMAWr_Mode_Reg[Channel]] := Mode;
end;
procedure DMA_NormMode_set(Channel,Mode : byte);
begin;
  port[DMAWr_Mode_Reg[Channel]] := Mode+Addresses_Increment+Read_Transfer+
    Autoinit_Disable+Channel;
end;
procedure DMA_Clear_Flipflop(Channel : byte);
begin;
  port[DMAClear_Flipflop[Channel]] := 0;
end;
procedure DMA_Startaddress(Channel : byte; Start : pointer);
begin
  l := 16*longint(pt(Start).sgm)+pt(Start).ofs;
  pn := pt(l).sgm;
  offs := pt(l).ofs;
  port[DMA_Address[Channel]] := lo(offs);
  port[DMA_Lower_Page[Channel]] := lo(pn);
  port[DMA_Higher_Page[Channel]] := hi(pn);
end;
procedure DMA_Blocksize(Channel : byte; size : word);
begin;
  DMA_Clear_Flipflop(Channel);
  port[DMA_Count[Channel]] := lo(size);
end;
procedure DMA_Channel_maskon(Channel : byte);
begin;
  port[DMAWr_single_mask_Reg[Channel]] := Channel + Set_Mask_Bit;
end;
procedure DMA_Init_Transfer(Channel,Mode : byte; p : pointer; s : word);
begin;
  DMA_Channel_maskon(Channel);
  DMA_Startaddress(Channel,p);
  DMA_Blocksize(Channel,s);
  DMA_NormMode_set(Channel,Mode+Channel);
  DMA_Channel_maskoff(Channel);
end;
begin;
end;
The Real Time Clock (RTC)

The Real Time Clock is another interesting chip (either a Dallas 1287 or compatible). It has a battery backup and works in a low-power mode, which protects the system from loss of data during boot-up and shutdown. The system configuration is stored in the CMOS-RAM (64 bytes). The chip also has a clock, a calendar and a programmable periodical interrupt.

The time and the calendar are updated periodically. Under normal operation, the RTC’s second counter is increased each second. On an overflow, the other registers that are affected are incremented. During the overflow, bytes 0 through 9 of the RTC’s RAM are not available to the CPU. The speed of the update cycle is adjusted by the divider bits 2-0 of status register A and the SET bit 7 of status register B.

Accessing the RTC’s RAM is very easy. First, at port 70h, the index number of the desired register is written. Then data can be read or written at port 71h. All registers can be read and written except for the following read-only bits:

- Status registers C and D
- Bit 7 of status register A
- Bit 7 of the seconds byte (Index 00h)

The first 14 bytes of the RTC’s RAM are used for the clock and the four status registers; the remaining 50 bytes are used for the system configuration. The following table shows the purpose of each byte:

<table>
<thead>
<tr>
<th>00h</th>
<th>Seconds</th>
<th>01h</th>
<th>Seconds alarm</th>
<th>02h</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>03h</td>
<td>Minutes alarm</td>
<td>04h</td>
<td>Hours</td>
<td>05h</td>
<td>Hours alarm</td>
</tr>
<tr>
<td>06h</td>
<td>Day of the week</td>
<td>07h</td>
<td>Day of the month</td>
<td>08h</td>
<td>Month</td>
</tr>
<tr>
<td>09h</td>
<td>Year</td>
<td>0Ah</td>
<td>Status A</td>
<td>0Bh</td>
<td>Status B</td>
</tr>
<tr>
<td>0Ch</td>
<td>Status C</td>
<td>0Dh</td>
<td>Status D</td>
<td>0Eh</td>
<td>Diagnosis status</td>
</tr>
<tr>
<td>0Fh</td>
<td>Shutdown status</td>
<td>10h</td>
<td>Floppy type</td>
<td>11h</td>
<td>Reserved</td>
</tr>
<tr>
<td>12h</td>
<td>HD type</td>
<td>13h</td>
<td>Reserved</td>
<td>14h</td>
<td>Configuration</td>
</tr>
<tr>
<td>15h</td>
<td>Low base memory</td>
<td>16h</td>
<td>High base memory</td>
<td>17h</td>
<td>Low extended memory</td>
</tr>
<tr>
<td>18h</td>
<td>High extended memory</td>
<td>19h</td>
<td>HD 1 extended type byte</td>
<td>1Ah</td>
<td>HD 2 extended type byte</td>
</tr>
<tr>
<td>1Bh</td>
<td>Reserved</td>
<td>1Ch</td>
<td>Reserved</td>
<td>1Dh</td>
<td>Reserved</td>
</tr>
<tr>
<td>1eh</td>
<td>Reserved</td>
<td>1Fh</td>
<td>Installed features</td>
<td>20H</td>
<td>HD 1 Low cylinder number</td>
</tr>
<tr>
<td>21h</td>
<td>HD 1 High cylinder number</td>
<td>22H</td>
<td>HD 1 Heads</td>
<td>23h</td>
<td>HD 1 Low pre-compensation start</td>
</tr>
</tbody>
</table>
Clock functions

The CPU retrieves the time and date by reading the appropriate bytes within the RTC. To set the time, you first switch on the SET bit of status register B and then write the desired values to the RTC’s RAM. The data within the clock are stored in BCD format. This means the tens position is stored in the upper four bits and the ones position is stored in the lower four bits. The following is the organization of these bytes:

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
<th>BCD Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>00</td>
<td>00 to 59</td>
</tr>
<tr>
<td>Seconds alarm</td>
<td>01</td>
<td>00 to 59</td>
</tr>
<tr>
<td>Minutes</td>
<td>02</td>
<td>00 to 59</td>
</tr>
<tr>
<td>Minutes alarm</td>
<td>03</td>
<td>00 to 59</td>
</tr>
<tr>
<td>Hours</td>
<td>04</td>
<td>(12 hour mode) 01 to 12 (AM) 81 to 92 (PM) (24 hour mode) 00 to 23</td>
</tr>
<tr>
<td>Hours alarm</td>
<td>05 (12 hour mode)</td>
<td>01 to 12 (AM) 81 to 92 (PM) (24 hour mode) 00 to 23</td>
</tr>
<tr>
<td>Day of the week</td>
<td>06</td>
<td>01 to 07</td>
</tr>
<tr>
<td>Day of the month</td>
<td>07</td>
<td>01 to 31</td>
</tr>
<tr>
<td>Month</td>
<td>08</td>
<td>01 to 12</td>
</tr>
<tr>
<td>Year</td>
<td>09</td>
<td>00 to 99</td>
</tr>
</tbody>
</table>
Status registers

The RTC has four status registers that are used to control and indicate the status of the chip.

Register A

Register A is especially interesting because of its ability to specify the speed at which the periodic interrupt is invoked. Bit 7 is the UIP (Update In Progress) bit. When it contains a value of 1, the time is being updated at that moment. Bits 6 to 4 contain the timebase. The default value 101b represents a value of 32,768 Hz. Bits 3 to 0 selects the rate according to the following formula:

\[ \text{rate} = \frac{65536}{2^{\text{RS}}} \]

So the default value of 1024 Hz is calculated from \( \frac{65536}{2^6} \) (101b).

Register B

Register B has several functions.

Bit 7 is the set bit. Normally this is set to 0 which means the clock is updated once every second. If set to 1, the update cycle is interrupted and you can then initialize the clock to a new time and date.

Bit 6 is the PIE (Periodic Interrupt Enable). When set to 1, the periodic interrupt is invoked with a frequency specified by Register A. The interrupt is not triggered when set to 0.

Bit 5 is the AIE (Alarm Interrupt Enable). This interrupt is enabled when set to 1.

Bit 4 is the UIE (Updated Ended Interrupt Enable). This interrupt is enabled when set to 1.

Bit 3 is the DM (Date Mode). When set to 1, the values are in binary format. When set to 0, the values are in BCD format.

Bit 2 is the 24/12 (24 or 12 Hour mode). When set to 1, the 24 hour mode is selected. When set to 0, the 12 hour mode is selected.

Register C

Register 0Ch is a flag register.

Bit 7 is the Interrupt Request flag. If set to 1, then one of the conditions that triggers an interrupt is true and the corresponding interrupt flag is also set.

Bit 6 is the periodic interrupt flag.

Bit 5 is set if the alarm time and current time are the same of if the alarm is off,

Bit 4 is the is the Updated Ended Interrupt flag. If set to 1, the update code of the RTC is completed.

Bits 3 to 0 are reserved.
Register D

Register D is used to monitor the battery. If bit 7 is set, then the battery is fully functional. If bit 7 is clear, the battery is defective. Bits 6 to 0 have no function.

Configuration bytes

The system configuration is found memory starting at 0Eh. In the following pages we'll describe the purpose of the individual bytes. If you need more information, refer to more in-depth information books on system programming such as PC Intern from Abacus.

Diagnostic status byte 0Eh

This byte is used to validate the system configuration at boot-up. The following table shows the meaning of the individual bits:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Status of the RTC</td>
<td>4</td>
<td>Test for change in memory size</td>
</tr>
<tr>
<td>1</td>
<td>Current is off</td>
<td>1</td>
<td>Memory size has changed</td>
</tr>
<tr>
<td>0</td>
<td>Current is on</td>
<td>0</td>
<td>Memory size has not changed</td>
</tr>
<tr>
<td>6</td>
<td>Checksum test</td>
<td>3</td>
<td>Hard disk test</td>
</tr>
<tr>
<td>1</td>
<td>Checksum is not valid</td>
<td>1</td>
<td>Hard disk or controller error</td>
</tr>
<tr>
<td>0</td>
<td>Checksum is valid</td>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>5</td>
<td>Correct configuration information?</td>
<td>2</td>
<td>Time status</td>
</tr>
<tr>
<td>1</td>
<td>Configuration is not valid</td>
<td>1</td>
<td>Time is not correct</td>
</tr>
<tr>
<td>0</td>
<td>Configuration is valid</td>
<td>0</td>
<td>Time is correct</td>
</tr>
</tbody>
</table>

Shutdown status byte 0Fh

This byte is set during a reset of the CPU. The reset code serves as an indicator of the type of reset. The following allocations are possible:

| 00h | Normal system reset            |
| 09h | Software reset (return from protected mode) |

Type of connected floppies - 10h

Information about the installed floppy disk drive can be determined using this byte.
Chapter 11
Programming Other PC Components

<table>
<thead>
<tr>
<th>Bit 7-4</th>
<th>Meaning</th>
<th>Bit 3-0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>No drive installed</td>
<td>0000</td>
<td>No drive installed</td>
</tr>
<tr>
<td>0001</td>
<td>360K 5.25-inch drive</td>
<td>0001</td>
<td>360K 5.25-inch drive</td>
</tr>
<tr>
<td>0010</td>
<td>1.2 Meg 5.25-inch drive</td>
<td>0010</td>
<td>1.2 Meg 5.25-inch drive</td>
</tr>
<tr>
<td>0011</td>
<td>720K 3.5-inch drive</td>
<td>0011</td>
<td>720K 3.5-inch drive</td>
</tr>
<tr>
<td>0100</td>
<td>1.44 Meg 3.5-inch drive</td>
<td>0100</td>
<td>1.44 Meg 3.5-inch drive</td>
</tr>
<tr>
<td>0101</td>
<td>2.88 Meg 3.5-inch drive</td>
<td>0101</td>
<td>2.88 Meg 3.5-inch drive</td>
</tr>
</tbody>
</table>

**Type of the installed hard drives - 12h**

This byte is used to determine the type of installed hard drive. You'll probably need to read bytes 19h and 1Ah for further information. The byte has the following meaning:

<table>
<thead>
<tr>
<th>Bit 7-4</th>
<th>Meaning</th>
<th>Bit 3-0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>No hard drive installed</td>
<td>0000</td>
<td>No hard drive installed</td>
</tr>
<tr>
<td>0001-1110</td>
<td>Hard disk of type 1 to 14</td>
<td>0001-1110</td>
<td>Hard disk of type 1 to 14</td>
</tr>
<tr>
<td>1111</td>
<td>Type information in byte 19h</td>
<td>1111</td>
<td>Type information in byte 1Ah</td>
</tr>
</tbody>
</table>

**Definition of the equipment - 14h**

This byte is accessed for the hardware self test.

<table>
<thead>
<tr>
<th>Bit 7-6</th>
<th>Meaning</th>
<th>Bit 3-2</th>
<th>Meaning</th>
<th>Bit 1</th>
<th>Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>One disk drive</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Two disk drives</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 5-4</td>
<td>Type of the graphics card</td>
<td>Bit 0</td>
<td>Disk drive installed?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Extended functionality controller</td>
<td>00</td>
<td>Drive is installed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Color display with 40 columns</td>
<td>01</td>
<td>Drive is installed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Color display with 80 columns</td>
<td>10</td>
<td>Drive is installed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Monochrome display</td>
<td>11</td>
<td>Drive is installed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Not used</td>
<td>00</td>
<td>No disk drive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Not used</td>
<td>01</td>
<td>No disk drive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Not used</td>
<td>10</td>
<td>No disk drive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Not used</td>
<td>11</td>
<td>No disk drive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Low/High Base Memory - 15h/16h

The size of the installed main memory can be read from the low base memory byte and the high base memory byte. A value of 0200h represents 512K of RAM and 0280h represents 640K.

Low/High Extended Memory - 17h/18h

The size of the installed extended memory (above 1 Meg) can be read from the low and high extended memory byte. For example, 0400h represents 1024K extended RAM, 0C00 for 3072K.

Extended Type Byte for Hard Disk 1 - 19h

This byte contains the type of the first hard drive if the bits 7-4 of the byte 12h have the value 0Fh.

Extended Type Byte for hard drive 2 - 1Ah

This byte contains the type of the second hard drive if the bits 3-0 of byte 12h have the value 0Fh.

Installed Features Byte - 1Fh

This byte reports error messages.

<table>
<thead>
<tr>
<th>Bit 7-3</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 2</td>
<td>Report disk drive error</td>
</tr>
<tr>
<td>Bit 1</td>
<td>Report video display error</td>
</tr>
<tr>
<td>Bit 0</td>
<td>Report keyboard error</td>
</tr>
</tbody>
</table>

CPU Speed Byte - 34h

The speed of the CPU is determined by this byte. Bits 7 to 1 are reserved. If bit 0 has the value 0h, the CPU is in the slow mode; if the bit is set, the CPU is working in turbo mode.

The RTC in practice

This program demonstrates how to access the RTC and evaluate the registers. It can serve as basis for a RTC-control unit.

```pascal
program rtc_unit;
uses crt,dos;
const
  Rtc_Seconds         = $00;
  Rtc_Seconds_alarm   = $01;
  Rtc_Minutes         = $02;
  Rtc_Minutes_alarm   = $03;
  Rtc_Hours           = $04;
  Rtc_Hours_alarm     = $05;
  Rtc_Weekday         = $06;
  Rtc_Day_of_Month    = $07;
  Rtc_Month           = $08;
  Rtc_Year            = $09;
```
function wrhexb(b : byte) : string;
const hexcar : array[0..15] of char =
( '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F');
begi
  wrhexb := hexcar[(b shr 4)] + hexcar[(b AND $0F)];
end;

function wrhexw(w : word) : string;
begi
  wrhexw := '$' + wrhexb(hi(w)) + wrhexb(lo(w));
end;

procedure write_rtc(Reg,val : byte);
{ Writes a value to the RTC register specified in Reg }
begi
  port[70] := Reg;
  port[70] := val;
end;
function read_rtc(Reg : byte) : byte;
{ Reads a value from the RTC register specified in Reg }
begin;
  port[$70] := Reg;
  read_rtc := port[$71];
end;

Function Write_Floppy;
{ Outputs information about the installed floppy drives }
Var Fl : byte;
  Fls : array[1..2] of byte;
begin;
  Fl := Read_Rtc(Rtc_Floppy_Typ);
  Fls[2] := Fl AND $0F;
  Fls[1] := Fl SHR 4;
  for Fl := 1 to 2 do begin;
    write('Floppy ',Fl,': ');
    case Fls[Fl] of
    0 : begin
      writeln('No Floppy        ');
      end;
    1 : begin
      writeln('5.25-inch Floppy, 360K');
      end;
    2 : begin
      writeln('5.25-inch Floppy, 1.2 Meg');
      end;
    3 : begin
      writeln('3.5-inch Floppy, 720K');
      end;
    4 : begin
      writeln('3.5-inch Floppy, 1.44 Meg');
      end;
    end;
  end;
end;

Procedure Write_Hd;
{ Outputs the type of installed HD }
Var Hd : byte;
  Hds : array[1..2] of byte;
begin;
  Hd := Read_Rtc(Rtc_HD_Typ);
  Hds[2] := Hd AND $0F;
  Hds[1] := Hd SHR 4;
  If Hds[1] = $F then Hds[1] := Read_Rtc(Rtc_HD1_extended);
  If Hds[2] = $F then Hds[2] := Read_Rtc(Rtc_HD2_extended);
  writeln('HD 1 : Typ ',Hds[1]);
  writeln('HD 2 : Typ ',Hds[2]);
end;

Procedure Write_Memory;
{
Chapter 11
Programming Other PC Components

Outputs available memory

```pascal
var base, extended : word;
begin;
  Base := 256 * Read_Rtc(Rtc_Hi_BaseMemory) +
            Read_Rtc(Rtc_Lo_BaseMemory);
  extended := 256 * Read_Rtc(Rtc_Hi_ExtendedMemory) +
               Read_Rtc(Rtc_Lo_ExtendedMemory);
  writeln('Base memory: ',Base,' KB');
  writeln('Extended memory: ',extended,' KB');
end;
```

procedure Write_Display;

```pascal
{ Outputs the type of graphic card and tells whether a coprocessor
  is installed }

var dtyp : byte;
  Copro : byte;
begin;
  dtyp := Read_Rtc(Rtc_Equipment);
  Copro := (dtyp AND 3) SHR 1;
  dtyp := (dtyp AND 63) SHR 4;
  case dtyp of
    0 : begin;
      writeln('Extended functionality GFX controller');
    end;
    1 : begin;
      writeln('Color display in 40 column mode');
    end;
    2 : begin;
      writeln('Color display in 80 column mode');
    end;
    3 : begin;
      writeln('Monochrome display controller');
    end;
  end;
  if Copro = 1 then
    writeln('Coprocessor found')
  else
    writeln('Coprocessor not found');
end;
```

procedure write_shadow;

```pascal
{ Outputs which areas of shadow RAM are supported }

var shadow : byte;
begin;
  shadow := read_Rtc(Rtc_Options1);
  shadow := shadow AND 3;
  case shadow of
    0 : begin;
      writeln('Shadow System AND Video BIOS');
    end;
    1 : begin;
      writeln('Shadow System BIOS');
    end;
    2 : begin;
      writeln('Shadow disabled');
    end;
  end;
end;
```
procedure write_cpuspeed;
  { Indicates whether the CPU is in Turbo mode }
  var speed : byte;
  begin
    speed := read rtc(Rtc_CPU_speed);
    if speed = 1 then
      writeln('CPU in Turbo mode')
    else
      writeln('CPU not in Turbo mode');
  end;

  var speed : byte;
  begin
    clrscr;
    Write_Floppy;
    Write_Hd;
    Write_Memory;
    Write_Display;
    Write_Shadow;
    Write_CPUSpeed;
  end.
Most people agree the Sound Blaster card is now the de facto standard for sound on the PC. The Sound Blaster has been improved many times since it was introduced by Creative Labs in 1989. Early versions of the Sound Blaster included SBPro and the SB16; the current version is the AWE 32. At first, most of the cards were "backward" compatible. An exception is the built-in DSP chip (Digital Signal Processor).

This chapter shows the basics of programming sound cards.

**Sound Blaster Card Components**

The heart of a Sound Blaster card is the Digital Signal Processor (DSP). It performs the function of an AD/DA converter (Analog to Digital/Digital to Analog) and is responsible for the input and output of digital data. The card also includes an OPL chip, which is responsible for the FM synthesis. The SBPro also contains an additional mixer chip, which performs other useful functions such as controlling volume level or balance.

**Programming the signal processor: The DSP**

The DSP used in the Sound Blaster series is the CT-DSP-1321. It’s located at port address 2x0h (the x here represents the appropriate base address). It’s programmed by writing a value to one of its registers. The base address can be selected in increments of 10h and may lie between 210h and 280h. The following table shows the available registers:
A register is accessed in the following order: First, the command value and then any necessary parameters are written to the Sound Blaster through command port 2xCh. Read access is through data port 2xAh.

However, before sending various commands to your Sound Blaster, you must first reset it. Write the value 1 to the reset port, wait a moment, and then write the value 0 to the reset port. You can check whether the reset was successful by checking if the data port contains the value $AA after about 100 ms. If it does, the reset was successful.

An example of the reset function is contained in `Reset_sb16`.

```pascal
FUNCTION Reset_sb16 : BOOLEAN;
{ Function resets the DSP. If the reset was successful, TRUE is returned, otherwise FALSE }
CONST ready = $AA;
VAR ct, stat : BYTE;
BEGIN
  PORT[dsp_addr+$6] := 1;                { dsp_addr+$6 = Reset function }
  FOR ct := 1 TO 100 DO;
    PORT[dsp_addr+$6] := 0;
    stat := 0;
  ct := 0;                               { Comparison ct < 100, since }  
  WHILE (stat <> ready)                  { initialization takes }           
    AND (ct < 100) DO BEGIN               { approx. 100ms }               
      stat := PORT[dsp_addr+$E];          
      stat := PORT[dsp_addr+$a];          
      INC(ct);                            
    END;
  Reset_sb16 := (stat = ready);
END;
```

The function `Reset_sb16` is part of the `MOD_SB.PAS` file on the companion CD-ROM.
Knowing that the value 0AAh is found in the data register after a reset can be used to your advantage when you’re trying to find the base port. Simply use a loop to check all ports between 210h and 280h, in increments of 10h, as possible Sound Blaster base ports. If the reset is successful, then you’ve found the correct base port. Function Detect_sb16 is an example of how a detect procedure might appear. The function uses this approach and returns the value TRUE if it was able to detect a Sound Blaster card.

FUNCTION Detect_Reg_sb16 : BOOLEAN;
{ Function returns TRUE if a SoundBlaster was initialized, otherwise FALSE. The dsp_addr variable is set to the base address of the SB. }
VAR
  Port, Lst : WORD;
BEGIN
  Detect_Reg_sb16 := SbRegDetected;
  IF SbRegDetected THEN EXIT; { Exit, if initialized } }
  Port := Startport; { Possible SB addresses bet- }
  Lst := Endport; { ween $210 and $280 ! }
  WHILE (NOT SbRegDetected)
    AND (Port <= Lst) DO BEGIN
    dsp_addr := Port;
    SbRegDetected := Reset_sb16;
    IF NOT SbRegDetected THEN
      INC(Port, $10);
    END;
  END;
  Detect_Reg_sb16 := SbRegDetected;
END;

By using this function, you can find a Sound Blaster card and determine at which port it can be addressed. However, this function doesn’t tell us which of the many Sound Blaster versions was detected.

To determine which version, use a function of the Sound Blaster card that returns the card’s version number. Send a command to the Sound Blaster that writes the version number to the data register. How do you send a command to the Sound Blaster?

All you actually need to know is that commands are always sent over the command port 2xCh. Note, however, you cannot write to the port while bit 7 of the command port is set. Therefore you have to use a loop to wait until the port may be written to and you can send the appropriate value to the port. The following is an example of such a procedure:

procedure wr_dsp_sb16(v : byte);
{ Waits until the DSP is ready to write and then writes the byte passed in “v” to the DSP }
begin;
  while port[dsp_addr+$c] >= 128 do ;
  port[dsp_addr+$c] := v;
end;
Since you not only want to send values to the Sound Blaster, but also read values from it, you’ll need a function that reads a value from the Sound Blaster. To read a value from the data port, wait until it no longer contains the value 0AAh. At that point the value can be read. Let’s look at this simple process in practice: Sound Blaster `ReadSB` function uses the described principle and returns the current byte in the data port.

```pascal
FUNCTION SbReadByte : BYTE;
{ Function waits until the DSP can be read and returns the read value }
begin
  while port[dsp_addr+$a] = $AA do ;  { wait till DSP ready   }
  SbReadByte := port[dsp_addr+$a];  { write value        }
end;
```

Now back to the problem of determining the version number of the Sound Blaster card. Use the command $E1 to determine the version number. Once this command is sent, read the version number from the data port. The version number consists of two bytes. The first indicates the main version number. By reading the port a second time, you’ll also receive the sub-version number. Procedure `SbGetDSPVersion` determines the version number of a Sound Blaster card.

```pascal
PROCEDURE SbGetDSPVersion;
{ Determines the DSP version and stores the result in the global variables SBVERSMAJ and SBVERSMIN, as well as SBVERSSTR. }
VAR i : WORD;
  t : WORD;
  s : STRING[2];
BEGIN
  wr_dsp_sb16($E1);         { $E1 = Version query     }
  SbVersMaj := SbReadByte;
  SbVersMin := SbReadByte;
  str(SbVersMaj, SbVersStr);
  if SbVersMin > 9 then begin
    str(SbVersMin, s);
    SbVersStr := SbVersStr +       s
  else
    SbVersStr := SbVersStr + '0' + s;
END;
```

The following table describes the other Sound Blaster DSP commands:
## Experience Sonic Worlds: The Sound Blaster Card

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>10h</td>
<td>8 bit output direct</td>
</tr>
<tr>
<td>14h</td>
<td>8 bit output through DMA</td>
</tr>
<tr>
<td>16h</td>
<td>Output of 2 bit comp. samples using DMA</td>
</tr>
<tr>
<td>17h</td>
<td>Output of 2 bit comp. samples with reference byte using DMA</td>
</tr>
<tr>
<td>20h</td>
<td>Direct digitalization</td>
</tr>
<tr>
<td>24h</td>
<td>Digitalization of 8 bit samples through DMA</td>
</tr>
<tr>
<td>30h</td>
<td>Direct MIDI input</td>
</tr>
<tr>
<td>31h</td>
<td>MIDI input through interrupt</td>
</tr>
<tr>
<td>32h</td>
<td>Direct MIDI input with timestamp</td>
</tr>
<tr>
<td>33h</td>
<td>MIDI input through interrupt with timestamp</td>
</tr>
<tr>
<td>34h</td>
<td>MIDI UART mode, direct</td>
</tr>
<tr>
<td>35h</td>
<td>MIDI UART mode through interrupt</td>
</tr>
<tr>
<td>37h</td>
<td>MIDI UART through interrupt with timestamp</td>
</tr>
<tr>
<td>38h</td>
<td>Send MIDI code</td>
</tr>
<tr>
<td>40h</td>
<td>Set sample rate</td>
</tr>
<tr>
<td>45h</td>
<td>DMA multiblock continue SB16</td>
</tr>
<tr>
<td>48h</td>
<td>Set block size</td>
</tr>
<tr>
<td>74h</td>
<td>Output of 4 bit comp. samples using DMA</td>
</tr>
<tr>
<td>75h</td>
<td>Output of 4 bit comp. samples with ref. byte using DMA</td>
</tr>
<tr>
<td>76h</td>
<td>Output of 2.6 bit comp. samples using DMA</td>
</tr>
<tr>
<td>77h</td>
<td>Output of 2.6 bit comp. samples with ref. byte using DMA</td>
</tr>
<tr>
<td>80h</td>
<td>Define silence block</td>
</tr>
<tr>
<td>91h</td>
<td>8 bit output through DMA in high speed</td>
</tr>
<tr>
<td>99h</td>
<td>8 bit input through DMA in high speed</td>
</tr>
<tr>
<td>B6h</td>
<td>Output 16 bit data on SB16 using DMA</td>
</tr>
<tr>
<td>B9h</td>
<td>Digitalize 16 bit data on SB16 using DMA</td>
</tr>
<tr>
<td>C6h</td>
<td>Output 8 bit data on SB16 using DMA</td>
</tr>
<tr>
<td>C9h</td>
<td>Digitalize 8 bit data on SB16 using DMA</td>
</tr>
</tbody>
</table>
Let's take a closer look at the individual commands:

10h  8 bit output direct

To output data periodically, for instance through the timer interrupt, you can use the function 10h. The two steps must be repeated at a constant rate!

1. Send the command 10h.
2. Send the data byte.

14h  8 bit output through DMA

The output over DMA is fairly complex, especially if you're unfamiliar with the process. It's important to program the DMA controller correctly. Otherwise, you'll have many different results, but not the desired one: The output of the sample data. You'll find more information on the DMA controller and how to program it in Chapter 11.

If you want to output data through the DMA, you'll first need to divert the Sound Blaster interrupt to a separate procedure and then program the DMA controller correctly. First, you'll need to determine and set the sample rate. Then perform the following steps:

1. Send the command 14h.
2. Send the low byte of the sample length -1.
3. Send the high byte of the sample length -1.

The output will begin when you've completed Step 3. Please note that you cannot output blocks larger than 64K. Such blocks must be divided into several sub-blocks.

In practice you should proceed like this: First, calculate the physical address of the block you want to output. You can use the following formula:

\[ \text{Physpos} := 16 \times \text{sgm(Block)} + \text{ofs(Block)} \]

The upper word of the physical address is then the page which you'll need to pass to the DMA controller. The lower word contains the offset that you'll need to set. First, lock the DMA channel to prevent unwanted accesses. Then clear the flip flop and output the write mode of the DMA controller.
Experience Sonic Worlds: The Sound Blaster Card

The $28+ number mode of the DMA channel is required to output data. In other words, $49 if you're using DMA channel 1. Next, you'll set the low byte, followed by the high byte of the offset of the physical address. Then you need to send the low byte followed by the high byte of the block size of the block you wish to transfer, before sending the page of the physical address. With this, the DMA controller is now ready for the transfer, and we can turn to the programming of the Sound Blaster card. First, send the command $14 for 8 bit output through the DMA. Send the low byte and then the high byte of the size of the block that is to be transferred. The Sound Blaster is now also programmed. Once you unlock the DMA channel, the sound output will start.

The procedure **Games_Sb** is an example of how this process works correctly. The first parameter required by the procedure is the page of the physical address of the sample block. The second parameter consists of the offset portion of the physical address of the sample block, while the third contains its length.

```pascal
procedure Games_Sb(Segm, Offs, dsz, dsize : word);
{ This procedure plays back the block addressed from Segm:Offs with
  the size dsize. Remember, the DMA controller CANNOT
  transfer more than one page ... }
var li : word;
begin;
  port[$0A] := dma_ch+4;                { lock DMA channel          }
  Port[$0c] := 0;                       { Buffer address   }
  Port[$0B] := $48+dma_ch;                     { for sound output           }
  Port[dma_addr[dma_ch]] := Lo(offs);    { to DMA controller          }
  Port[dma_addr[dma_ch]] := Hi(offs);
  Port[dma_wc[dma_ch]] := Lo(dsz-1);  { size of block (block-  }
  Port[dma_wc[dma_ch]] := Hi(dsz-1);  { size) to DMA controller }
  Port[dma_page[dma_ch]] := Segm;
  wr_dsp_sb16($14);                     { size of block to       }
  wr_dsp_sb16(Lo(dsize-1));             { DSP                    }
  wr_dsp_sb16(Hi(dsize-1));             {               }
  Port[$0A] := dma_ch;                  { release DMA channel       }
end;
```

16h  output of 2-bit comp. samples using DMA

The output of 2-bit compressed samples is similar to the output of 8-bit samples. You only need to replace the command $14 for programming the Sound Blaster with the command 16h.

17h  output of 2-bit comp. samples with reference byte using DMA

The same basic information for the command 16h applies to command 17h. The only difference is the first byte is interpreted as a reference byte. In other words, it's the starting value from which the differences are calculated. Command 17h should is usually used to first output a block with a reference byte, followed by clocks without reference bytes.

20h  direct digitizing

20h allows you to directly record 8-bit sample data. Follow these steps to record data using this command:
1. Send the command 20h
2. Read a sample byte

Note that both steps must be repeated periodically and at a constant rate. This method will only work for digitizing at a low sampling rate. Use the DMA method for higher-quality recordings.

24h digitize 8-bit samples through DMA

To record data through the DMA, first change the Sound Blaster interrupt to a separate procedure and then program the DMA controller appropriately. Afterwards, set the sample rate and do the following:

1. Send the command 24h
2. Send the low byte of the sample length -1
3. Send the high byte of the sample length -1

The recording will begin when you've completed the third step. Note that you cannot record blocks longer than 64K. These blocks must be divided into several sub-blocks.

In practice you should proceed like this: First, calculate the physical address of the block to record. Use the following formula:

\[
\text{Physpos} := 16 \times \text{sgm(Block^)} + \text{ofs(Block^)}
\]

The high word of the physical address is the page which you'll need to pass to the DMA controller. The lower word contains the offset which you'll need to set. First, lock the DMA channel to prevent unwanted accesses. Then clear the flip flop and output the write mode of the DMA controller.

A value equal to $44 + \text{DMA channel number}$ is used to output data. For example, $45$ if you're using DMA channel 1. Next, set the low byte, followed by the high byte of the offset of the physical address. Then send the low byte followed by the high byte of the block size of the block to transfer, before sending the page of the physical address.

Now, the DMA controller is ready for the transfer. We can now begin programming the Sound Blaster card. First, send the command $24$ for 8-bit output through the DMA. Send the low byte and then the high byte of the size of the block to be transferred. The Sound Blaster is now also programmed. The recording will start once you unlock the DMA channel.

Procedure Games_SbPro is an example of how this process works. The first parameter is the page of the physical address of the sample block. The second parameter is the offset portion of the physical address of the sample block. The third parameter contains its length.

```pascal
procedure Games_SbPro(Segm, Offs, dsz, dsize : word);

This procedure plays back the block addressed from Segm:Offs with the size dsize. Remember the DMA controller CANNOT transfer more than one page ...

var li : word;
begin;
```

The procedure Games_SbPro is part of the MOD_SB.PAS file on the companion CD-ROM
Experience Sonic Worlds: The Sound Blaster Card

Chapter 12

PC underground

319

port[$0A] := dma_ch+4;                  { lock DMA channel          }
Port[$0c] := 0;                         { Buffer address  }
Port[$0B] := $48+dma_ch;                { for sound output           }
Port[dma_addr[dma_ch]] := Lo(offs);     { to DMA controller          }
Port[dma_addr[dma_ch]] := Hi(dsz-1);    { size of block (block-   }
Port[dma_wc[dma_ch]] := Lo(dsz-1);      { size) to DMA controller }
Port[dma_page[dma_ch]] := Segm;
wr_dsp_sb16($48);
wr_dsp_sb16(Lo(dsize-1));               { size of block to       }
wr_dsp_sb16(Hi(dsize-1));               { DSP                    }
wr_dsp_sb16($91);
Port[$0A] := dma_ch;                  { release DMA channel        }
end;

30h  direct MIDI input

Command 30h is for the direct recording of MIDI data. Do this as follows:

1. Write 30h to the DSP
2. Read the status port until bit 7 is set
3. Read data from the data port

If bit 7 in the data byte is set, it is a control byte. In this case, steps 2 and 3 are repeated to record the actual note; otherwise, the byte is the note.

31h  MIDI input through interrupt

Command 31h reads MIDI data triggered by an interrupt.

First, change the Sound Blaster interrupt to your own procedure. Then send the command 31h. If there is any MIDI data, the Sound Blaster interrupt is triggered, and you can process the MIDI data. Use the following three steps:

1. Read and store the data byte from the data port
2. Read the status port
3. End the interrupt

32h  direct MIDI input with timestamp

To time MIDI data precisely, capture the data using a timestamp. A timestamp is a 24-bit value that indicates the number of milliseconds that have passed since the command 32h was triggered. Follow these steps:

1. Send the command 32h
2. Check whether bit 7 of the status port is set (has the value 1)
3. Read the low byte of the timestamp
4. Read the middle byte of the timestamp
5. Read the high byte of the timestamp
6. Read the MIDI code

Repeat steps 2 through 6 as needed.

**33h MIDI input with timestamp through interrupt**

This command works similar to 31h except you must first read the three timestamp bytes using command 33h before you can read the MIDI code.

**34h MIDI UART mode, direct**

After you've switched to this mode, use command 34h to read and write MIDI data. You can write directly to the DSP with this command.

**35h MIDI UART mode through interrupt**

This command is basically the same as 34h, except that it is controlled by an interrupt. Refer to the description for command 32h for an example of how to apply the command.

**37h MIDI UART through interrupt with timestamp**

MIDI data is controlled by this interrupt. This procedure is similar to command 35h, except that you'll also need to read the three timestamp bytes.

**38h Send MIDI code**

Use the following method to send a MIDI code to control one of the connected devices:

1. Send command 38h
2. Read the command port until bit 7 is no longer set (=0).
3. Write the desired MIDI code

**40h Set sample rate**

This command sets the sampling rate of the Sound Blaster card. Use this command in any program that uses digital output. Follow these steps:

1. Send the command 40h
2. Send the TC constant

This TC constant is calculated differently for normal speed modes and for high speed modes (starting with SBPro). For normal speed modes use the following formula:

$$TC := 256 - (1.000.000 \div \text{frequency})$$

And, for high speed mode, use the following formula:

$$TC := 65536 - (256.000.000 \div \text{frequency})$$

Note than only the upper byte of the TC constant is sent. The lower byte is ignored.
Experience Sonic Worlds: The Sound Blaster Card

Chapter 12

45h  DMA Multi block continue Sound Blaster 16

Use this command to continue the transmission of a Sound Blaster 16 block rather than using commands B6h, B9h, C6h, and C9h. In this case, you don't have to retransmit the length and the transfer mode but instead send the command 45h directly. Another block of the same length as the previously transmitted block is then sent.

48h  Set block size

Command 48h sets the block size of the sample block transmitted in high speed mode. You'll need to do the following three steps:

1. Send the command 48h
2. Send the low byte of the block size -1
3. Send the upper byte of the block size -1

74h  output of 4-bit compressed samples using DMA

The same method applies to the output of 4-bit compressed samples as the method described for command 14h.

75h  output of 4-bit comp. samples with reference byte using DMA

See the instructions for command 17h.

76h  output of 2.6-bit compressed samples via DMA

See command 74h.

77h  output of 2.6-bit comp. samples with reference byte using DMA

See command 75h.

80h  define silence block

Command 80h defines and sends a "silence block". Follow these steps:

1. Set the Sound Blaster interrupt to your own routine.
2. Set the desired sample rate (command 40h)
3. Send the command 80h
4. Send the low byte of the block size -1
5. Send the high byte of the block size -1

The Sound Blaster card will generate an interrupt once the silence block has been completely sent.

91h  8-bit output through DMA in high speed mode

Command 91h plays sample data with a sampling rate up to 44 KHz. You'll need to follow similar steps as for regular output:
1. Divert the Sound Blaster interrupt to your own routine.
2. Initialize the DMA controller.
3. Set the desired sample rate using the command 40h.
4. Send 48h to set the block size.
5. Send the low byte of the block size -1.
6. Send the high byte of the block size -1.
7. Send the command 91h.

Data will be output immediately after command 91h is sent. At the end of the output the Sound Blaster card will generate an interrupt.

99h  **8-bit input through DMA in high speed mode**

Command 99h records data in high speed mode. The only difference is that you must send command 99h instead of 91h as the last step.

B6h  **output 16-bit data using DMA with SB16**

Command B6h plays 16-bit samples on a Sound Blaster 16. The method is the same as the conventional playback through the DMA. However, with SB16 you need not specify whether mono or stereo data is to be processed through the mixer chip. Instead, information can is passed to the DSP during the initialization. Perform the following steps:

1. Change the SB interrupt to your own routine
2. Initialize the DMA controller
3. Send command B6h
4. Send 00h to play mono data or 20h for stereo data
5. Send the low byte of the block size -1
6. Send the high byte of the block size -1

Playback will start immediately after you’ve sent the block length. After the data is played, the Sound Blaster generates an interrupt.

B9h  **Record 16-bit data through DMA on SB16**

Command B9h records 16-bit samples. The technique is similar to playback except that you send command B9h instead of B6h.

C6h  **output 8-bit data through DMA on SB16**

Command C6h plays 8-bit samples on a Sound Blaster 16. The technique is similar to playback of 16-bit samples except that you send command C6h.
Experience Sonic Worlds: The Sound Blaster Card

C9h  Record 8-bit data through DMA on SB16

Command C9h records 8-bit samples on a Sound Blaster 16.

D0h  Stop DMA

Command D0h pauses data transfer using DMA. You can continue the transfer using command D4h.

D1h  switch loudspeaker on

Command D1h switches on the Sound Blaster loudspeaker.

D3h  switch loudspeaker off

Command D3h switches off the Sound Blaster loudspeaker.

D4h  continue DMA

Command D4h continues DMA transfer that was interrupted by command D0h.

D8h  check loudspeaker status

Command D8h checks the loudspeaker status if a Sound Blaster Pro. Send command D8h and then read a byte from the DSP. If it has the value 0, the loudspeaker is switched on. If it has the value 1 it is switched off.

E1h  version inquiry

Command E1h identifies the version number of a Sound Blaster card. Do this as follows:

1. Send E1h
2. Read the main version number of the Sound Blaster
3. Read the sub-version number of your Sound Blaster

Detecting the Sound Blaster card

When you install a Sound Blaster card, the installation program adds the BLASTER environment variable to your AUTOEXEC.BAT. You can normally specify the Sound Blaster card parameters using this variable.

However, this environment variable may not always be set. In this case, the card through hardware. Unfortunately, performing such a card-check is not entirely risk-free. Some network cards or S3 chips installed in your computer system react unpredictably when you write to their registers. Therefore, be very careful when performing hardware checks. Although you won’t physically damage the hardware, it’s very likely you’ll crash the computer.

While determining the base port of the Sound Blaster is rather easy, you’ll need to use a trick or two to identify the interrupt. With the DMA channel, however, there’s not much you can do other than let the user specify their channel number if the Sound Blaster is not set to 1 (which is most often the case).

Recognizing the base port of the Sound Blaster is actually quite simple. The possible addresses of the Sound Blaster range from 200h to 280h in increments of 10h. Check these addresses to see whether a Sound Blaster is present. If one is present, even at 280h, either there is no Sound Blaster card in the PC (perhaps it’s a GUS
Experience Sonic Worlds: The Sound Blaster Card

Chapter 12

with SBOS) or maybe our detection routine is faulty. This won’t happen unless the DMA channel is incorrect.

To determine whether a Sound Blaster is at a specific address, we’ll try to reset the card. If the reset is successful, then we can conclude that there’s a Sound Blaster at this address. Otherwise, we’ll need to check another address.

You’ve already encountered the two procedures that we’ll use to detect a card: Reset_SBCard and Detect_SBReg.

Once you know the Sound Blaster base address, you can attempt to identify the card’s interrupt as follows: Whenever a block is transferred over the DMA, an interrupt is triggered at the end of the transfer. A small text procedure transfers a short block using DMA and waits a moment to give the Sound Blaster enough time to send an interrupt. This procedure is repeated in a loop. For each interrupt to be checked, the corresponding vector is changed and handled by your procedure. If the procedure is accessed, a flag is set indicating the interrupt has been identified. If the flag is set, you can jump out of the loop; otherwise you continue to check the next interrupt.

Procedure detect_sbIRQ demonstrates this approach. Make certain to save the original interrupt vectors and to restore them once the procedure is finished. You may want to turn the loudspeaker off before the block is transmitted; you probably won’t want to listen to the test data during the routine.

```pascal
procedure detect_sbIRQ;
{
This routine detects the IRQ of the SoundBlaster card. To achieve this,
the routine tests all possible interrupts. For this purpose, short blocks are
output via DMA. It has found the right one if the routine jumps to the set interrupt at the end of
output.
}
const possibleIRQs : array[1..5] of byte = ($2,$3,$5,$7,$10);
var i : integer;
h : byte;
begin;
getintvec($8+dsp_irq,intback);         { Store values !            }\nport21 := port[$21];\nfillchar(buffer1^,1200,128);\nset_Timeconst_sb16(211);\nwr_dsp_sb16($D3);                      { Speaker off           }\ni := 1;\ninterrupt_check := true;\nwhile (i <= 5) and (not IRQDetected) do\nbegin;
dsp_irq := possibleIRQs[i];      { IRQ to be tested           }\ngetintvec($8+dsp_irq,oldint);      { Deflect/Bend interrupt       }\nselintvec($8+dsp_irq,0dsp_int_sb16);\nirqmsk := 1 shl dsp_irq;\nport[$21] := port[$21] and not irqmsk;\nSampling_Rate := 211;\nblocksize := 1200;                { test output          }\ndsp_block_sb16(blocksize,blocksize,buffer1,true,false);\ndelay(150);\nsetintvec($8+dsp_irq,oldint);      { Interrupt back again     }\nport[$21] := Port[$21] or irqmsk;\nh := port[dsp_addr+$E];
```

The procedure detect_sbIRQ is part of the MOD_SB.PAS file on the companion CD-ROM
Port[$20] := $20;
inc(i);
end;
interrupt_check := false;
wr_dsp_sb16($D1);                      { Speaker back on    }
setintvec($8+dsp_irq,intback);         { Old values back !!!      }
port[$21] := port21;
dsp_rdy_sb16 := true;
end;

Mixing sound data: The mixer chip

The Sound Blaster Pro also includes a mixer chip. This mixer regulates the inputs and outputs of the card. It also controls the volume level and balance.

The mixer is addressed through the two registers at 2x4h (the mixer's selection port) and 2x5h (the mixer's data port).

To program the mixer, first select the register to be changed through the selection port. Then read or write the desired data through the data port.

Let's take a look at the SBPro mixer chip:

Register 00h - Reset

This register returns the default settings of the mixer. Follow these steps:

1. Write the value 0 to the index register
2. Wait about 100 ms
3. Write the value 0 to the data register

Register 02h - DSP volume level

This register sets the volume level of the DSP. Bits 7-5 control the left channel volume level. The volume level can range from 0 to 7. Bits 3-1 are for the right channel volume level. Do the following to set the desired volume level:

1. Write the value 02h to the index register
2. Determine the volume for the SBPro according the formula:
   \[ \text{volume} := (\text{left} \text{ shl} 5) + (\text{right} \text{ shl} 1); \]
3. Write the volume level to the data register

Register 0Ah - microphone volume level

This register sets the microphone volume level. The value can range from 0 to 3. The microphone level is adjusted in the following steps:

1. Write the value 0Ah to the index register
2. Determine the microphone level according to the formula
3. Write the volume level to the data register

Register 0Ch - input filter settings

This register controls the settings for recording. You select a filter and an input source. The filter is turned on if bit 5 is reset, otherwise it's off. Bit 3 controls the transparency level of the filter. It's low if bit 3 is set, otherwise it's high. Bits 2-1 control the input source. A value of 0 specifies microphone input, 1 specifies CD ROM input, and 3 specifies line input. You set the registers as follows:

1. Write the value $0C to the index register.
2. Determine the setting value:
   - If filter on, then \( Ew := (1 \text{ shl } 5) \)
   - If transparency = low, then \( Ew := Ew + (1 \text{ shl } 3) \)
   - \( Ew := Ew + (\text{Quelle } \text{shl } 1) \)
3. Write the setting value to the data register.

Register 0Eh - output filter and stereo selection

This register has two functions. The first selects the output filter. The second function selects mono or stereo output.

If bit 5 is 1, the output filter is not selected. If bit 5 is 0, the output filter is selected. The register is controlled with the following steps:

1. Write the value $0Eh to the index register
2. If with filter, then \( \text{Reg-Wert} := (1 \text{ shl } 5) \)
   - If stereo, then \( \text{Reg-Wert} := \text{Reg-Wert} + (1 \text{ shl } 1) \)
3. Write the register value to the data register

Register 22h - master volume level

This register sets the overall volume level. Bits 7-5 control the volume of the left stereo channel. The volume level ranges between 0 and 7. Bits 3-1 control the volume for the right channel. Set the volume levels as follows:

1. Write the value 22h to the index register
2. Determine the volume level
   - \( \text{volume} := (\text{left } \text{shl } 5) + (\text{right } \text{shl } 1) \)
3. Write the volume level to the data register

Register 26h - FM volume level

This register sets the volume level of the FM channel. It controls the volume levels the same way as command 22h. Set the FM channel volume levels as follows:

1. Write the value 26h to the index register
2. Determine the volume level using the formula
   \[ \text{volume} = (\text{left} \text{ shl} 5) + (\text{right} \text{ shl} 1) \]
3. Write the volume level to the data register

**Register 28h - CD volume level**

This register sets the volume level of the CD channel. It works the same as command 22h. Set the CD channel volume levels as follows:

1. Write the value 28h to the index register
2. Determine the volume level for the SBPro:
   \[ \text{volume} = (\text{left} \text{ shl} 5) + (\text{right} \text{ shl} 1) \]
3. Write the volume level to the data register

**Register 2Eh - Line volume level**

This register sets the volume level of the line-in channel. It works the same as command 22h. Follow these steps:

1. Write the value 2Eh to the index register
2. Determine the volume level:
   \[ \text{volume} = (\text{left} \text{ shl} 5) + (\text{right} \text{ shl} 1) \]
3. Write the volume level to the data register

**SB16 (ASP) mixer chip registers**

Now that you’re familiar with the SBPro mixer chip, you’ll be able to understand the SB16’s improved mixer. The following tables show an overview of its registers:

<table>
<thead>
<tr>
<th>Register 48: Left Master Volume</th>
<th>Register 49: Right Master Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit number</strong></td>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>Master Volume</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Master Volume</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Master Volume</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Master Volume</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Master Volume</td>
</tr>
</tbody>
</table>
### Experience Sonic Worlds: The Sound Blaster Card

#### Chapter 12

Register 50: Left Voice Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 51: Right Voice Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 52: Left MIDI Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>MIDI Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>MIDI Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>MIDI Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>MIDI Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 53: Right MIDI Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>MIDI Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>MIDI Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>MIDI Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>MIDI Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 54: Left CD Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

Register 55: Right CD Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>
## Experience Sonic Worlds: The Sound Blaster Card

### Register 56: Left Line Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

### Register 57: Right Line Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

### Register 58: Mic Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

### Register 59: PC Speaker Volume

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

### Register 60: Output Control Register

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>Mic</td>
<td>1</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>CD right</td>
<td>1</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>CD left</td>
<td>1</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>Line right</td>
<td>1</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Line left</td>
<td>1</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>- - -</td>
<td>0</td>
</tr>
</tbody>
</table>

### Register 61: Left-In Control Register

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>Mic</td>
<td>1</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>CD right</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>CD left</td>
<td>1</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>Line right</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Line left</td>
<td>1</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>MIDI right</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>MIDI left</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>- - -</td>
<td>0</td>
</tr>
</tbody>
</table>
Register 62: Right-In Control Register

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 63: Left Input Gain Factor

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Volume</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Volume</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Volume</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 64: Right Input Gain Factor

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0-5:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Gain Factor</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Gain Factor</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 65: Left Output Gain Factor

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0-5:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Gain Factor</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Gain Factor</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 66: Right Output Gain Factor

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0-5:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Gain Factor</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Gain Factor</td>
<td>1</td>
</tr>
</tbody>
</table>

Register 67: AGC ON/OFF (Automatic Gain Control)

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>AGC On/Off</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1-7:</td>
<td>- - -</td>
<td>0</td>
</tr>
</tbody>
</table>
Experience Sonic Worlds: The Sound Blaster Card

## Playing VOC Files

Normally VOC files are played using one of the Sound Blaster drivers. For those of you interested, we'll show you how to program the card directly. This section shows you how to build a small VOC player.

The procedures described here also form the basis for the Sound Blaster MOD player (see Chapter 15).

The player uses the procedures introduced in this chapter. The complete source code is on the companion CD-ROM. Here, we'll only talk about the most important procedures for VOC playback.

### Register 68: Left Treble

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Treble Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Treble Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Treble Control</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Treble Control</td>
<td>1</td>
</tr>
</tbody>
</table>

### Register 69: Right Treble

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Treble Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Treble Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Treble Control</td>
<td>1</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Treble Control</td>
<td>1</td>
</tr>
</tbody>
</table>

### Register 70: Left Bass

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Bass Control</td>
<td>1</td>
</tr>
</tbody>
</table>

### Register 71: Right Bass

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 1:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 2:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3:</td>
<td>- - -</td>
<td>0</td>
</tr>
<tr>
<td>Bit 4:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 5:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 6:</td>
<td>Bass Control</td>
<td>0</td>
</tr>
<tr>
<td>Bit 7:</td>
<td>Bass Control</td>
<td>1</td>
</tr>
</tbody>
</table>
Let’s start with the `Init_Voc` procedures. This procedure requires the file name and, if necessary, the path of the file you wish to play. The file’s header is then verified and the first block of the VOC file is interpreted. The procedures can determine from this block the sample rate and output mode (mono or stereo).

Next, the data from the VOC file is read and transferred to a double buffer. Playback starts by transferring the buffer contents using DMA. The interrupt procedure handles sending additional data.

Writing a VOC player isn’t very difficult. You might consider adding the capability for the interrupt to handle additional blocks. However, VOC files are usually recorded as one piece and are also played back as such.

```pascal
procedure Init_Voc(filename : string);
const VOCid : string = 'Creative Voice File'+#$1A;
var ch : char;
   IDstr : string;
   ct : byte;
   h : byte;
   error : integer;
   srlo,srhi : byte;
   SR : word;
   Samplinsz : word;
   stereoreg : byte;
begin;
   PLAYING_MOD := false;
   PLAYING_VOC := true;
   VOC_READY   := false;
   vocstereo := stereo;
   stereo := false;
   assign(vocf,filename);
   reset(vocf,1);
   if filesize(vocf) < 5000 then begin;
      VOC_READY   := true;
      exit;
   end;
   blockread(vocf,voch,$19);
   IDstr := voch.IDstr;
   if IDstr <> VOCid then begin;
      VOC_READY   := true;
      exit;
   end;
   Blockread(vocf,inread,20);
   vblock.ID := inread[2];
   if vblock.ID = 1 then begin;
      vblock.SR := inread[6];
   end;
   if vblock.ID = 8 then begin;
      SR := inread[6]+(inread[7]*256);
      Samplinsz := 256000000 div (65536 - SR);
      if inread[9] = 1 then begin; {stereo}
         if sb16detected then samplinsz := samplinsz shr 1;
         stereo := true;
      end;
      vblock.SR := 256 - longint(1000000 DIV samplinsz);
   end;
```

The following procedures are part of the MOD_SB.PAS file on the companion CD-ROM.
if vblock.ID = 9 then begin;
    Samplinsz := inread[6]+(inread[7]*256);
    if inread[11] = 2 then begin; (stereo)
        stereo := true;
        if sbprodetected then samplinsz := samplinsz * 2;
        vblock.SR := 256 - longint(1000000 DIV (samplinsz));
    end else begin;
        vblock.SR := 256 - longint(1000000 DIV samplinsz);
    end;
end;
if vblock.SR < 130 then vblock.SR := 166;
set_timeconst_sb16(vblock.SR);
blocksz := filesize(vocf) - 31;
if blocksz > 2500 then blocksz := 2500;
blockread(vocf,vocb1^,blocksz);
ch := #0;
fsz := filesize(vocf) - 32;
fsz := fsz - blocksz;
Block_active := 1;
if fsz > 1 then begin;
    blockread(vocf,vocb2^,blocksz);
    fsz := fsz - blocksz;
end;
wr_dsp_sb16($D1);
lastone := false;
if not sb16Detected then begin;
    if Stereo then begin;
        stereoreg := Read_Mixer($0E);
        stereoreg := stereoreg OR 2;
        Write_Mixer($0E,stereoreg);
    end else begin;
        stereoreg := Read_Mixer($0E);
        stereoreg := stereoreg AND $FD;
        Write_Mixer($0E,stereoreg);
    end;
end;
pause_voc := false;
dsp_block_sb16(blocksz,blocksz,vocb1,false,true);
end;
procedure voc_done;
var h : byte;
begin;
    lastone := true;
    { repeat until dsp_rdy_sb16;}
close(vocf);
    Reset_Sb16;
    stereo := vocsstereo;
end;
procedure voc_pause;
begin;
    pause_voc := true;
end;
procedure voc_continue;
begin;
pause_voc := false;
if block_active = 1 then begin
  dsp_block_sbl6(blocksz,blocksz,vocb2,false,true);
  block_active := 2;
end else begin;
  dsp_block_sbl6(blocksz,blocksz,vocb1,false,true);
  block_active := 1;
end;
end;

**************************************************************************
***                                                                    ***
***    More SB Routines                                               ***
***                                                                    ***
**************************************************************************

procedure dsp_int_sbl6; interrupt;
{ The interrupt generated at the end of a block transfer jumps to this
procedure. If the last_output flag is not set, new output begins }
var h : byte;
begin;
if interrupt_check then begin;
  IRQDetected := true;
end else begin;
  if PLAYING_MOD then begin;
    h := port[dsp_addr+$E];
    dsp_rdy_sbl6 := true;
    if not last_output then begin
      if bsw then
        dsp_block_sbl6(blocksize,blocksize,buffer1,true,false)
      else
        dsp_block_sbl6(blocksize,blocksize,buffer2,true,false);
      bsw := not bsw;
      phase_1 := false;
      phase_2 := true;
    end;
  end;
end;

IF PLAYING_VOC then begin;
  h := port[dsp_addr+$E];
  if (fsz > blocksz) and not lastone then begin
    lastone := false;
    if block_active = 1 then begin
      if not pause_voc then
        dsp_block_sbl6(blocksz,blocksz,vocb2,false,true);
      blockread(vocf,vocb1^,blocksz);
      fsz := fsz - blocksz;
      block_active := 2;
    end else begin;
      if not pause_voc then
        dsp_block_sbl6(blocksz,blocksz,vocb1,false,true);
      blockread(vocf,vocb2^,blocksz);
      fsz := fsz - blocksz;
      block_active := 1;
    end;
  end else begin;
if not lastone then begin;
  if block_active = 1 then begin
    if not pause_voc then
      dsp_block_sb16(blocksz,blocksz,vocb2,false,true);
    lastone := true;
  end else begin;
    if not pause_voc then
      dsp_block_sb16(blocksz,blocksz,vocb1,false,true);
    lastone := true;
  end;
end else begin;
dsp_rdy_sb16 := true;
wr_dsp_sb16($D0);
VOC_READY := true;
end;
end;
Port[$20] := $20;
end;
Virtually all demos and computer games being produced have sound accompaniment - in fact, some of the most successful games also have great sound. In this chapter we'll describe some of the important aspects of sound programming and the various sound file formats.

We'll also present MOD players for the two most popular sound cards, the Sound Blaster and the GUS sound card from Gravis UltraSound on the companion CD-ROM.

The MOD File Format

The MOD file format, developed originally for the Commodore Amiga, was introduced to the PC world with the Sound Blaster card. This format is used for digital music using relatively modest storage requirements.

A MOD file is a series of natural sound samples which can be replayed as various instruments. The sampling method produces more realistic sound than the Sound Blaster’s FM synthesis.

The MOD format

In the early days of MOD format development each programmer tended to "do his own thing." Therefore, several different types of the MOD format exist today. We'll talk about what is usually regarded as the standard format. This MOD file format is divided into three parts:

1. The MOD header

The header is 1084 bytes long and contains information about the sound clip that follows:

<table>
<thead>
<tr>
<th>BYTE 0-19</th>
<th>Song name</th>
</tr>
</thead>
</table>

The first 20 bytes of the header contain the song name. It's stored as a C-string in ASCII format (the last byte must be zero).

<table>
<thead>
<tr>
<th>BYTE 20-41</th>
<th>First instrument name</th>
</tr>
</thead>
</table>

These 22 bytes contain the name of the first instrument. It is stored as a C-string in ASCII format (the last byte must be zero).
These two bytes contain the length of the first instrument in words. The value must be multiplied by 2 to obtain the number of bytes. Because the MOD format was first developed on the Amiga, the high and low bytes must be reversed when using MOD files on the PC. This is because the 68000 processor uses the sequence high/low as opposed to the low/high sequence of 80x86 processors.

Only the lower four bits are used. This field represents a value from -8 to 8 for fine tuning the instrument. This option is seldom used.

This byte contains the default volume of the instrument. Acceptable values range from 0 to 64.

Loop (repeat) MOD file playback. These two bytes indicate start of a loop. These bytes are reversed high/low.

These two bytes indicate the length of the loop, high/low.

The information in bytes 20 through 49 are repeated for additional instruments (30 in the new format, 14 in the old). We assume the new MOD format here.

This byte indicates the length of the song in patterns (the actual song length, not the number of patterns defined).

This byte is Amiga-specific and is not used for PC applications.

Each of the 128 bytes can have a value from 0 to 63, designating the pattern to be played. From these 128 bytes and the song length, you derive the song arrangement.

These four bytes contain a MOD-file identifier. The values M.K. and FLT4 indicate a 4-voice MOD file with 31 instruments. PC files may also contain newer formats, such as 6CHN for 6-voice and 8CHN for 8-voice MOD files.
2. The MOD patterns

BYTE 1084-2107

These bytes contain the first pattern. Each note of a pattern consists of four bytes. A pattern has 64 lines of four notes per line, yielding a total pattern length of 1024 bytes.

To determine the number of patterns, perform the following: First, subtract the header length from the size of the MOD file. Then add the lengths of the individual samples and subtract this total from the file size. Divide the remainder by the pattern size (1024) to arrive at the number of patterns. Now you can read the patterns in without a problem.

3. The MOD sample data

BYTE n1-nx

After the patterns, you’ll find the sample data in signed raw-data format. This 8-bit format can have values ranging from -127 to 128. However, a PC defines the range as 0 to 255 so the sample data must be converted prior to output.

The format of a MOD note

Although the format of a MOD note may at first seem complicated, it’ll become clearer as you continue reading.

BYTE 1

The upper four bits (7-4) are the high bits of the instrument number. The lower four bits (3-0) are part of the pitch.

BYTE 2

Byte 2, with bits 3-0 of Byte 1, indicate the pitch (in samples per minute).

The following table shows the standard sample rates:
BYTE 3

The upper four bits (7-4) are the low bits of the instrument number. The lower four bits contain the effect command.

BYTE 4

Byte 4 contains the operands of the effect command.

Effects

**Effect 00**  **Arpeggio (two operands)**

The arpeggio effect causes a note to be played as a sequence of three different pitches, rather than one continuous pitch. The first parameter indicates the difference in half-steps between the first pitch and the second pitch; the second parameter indicates the difference in half-steps between the second pitch and the third pitch.

**Effect 01**  **Portamento Up (one operand)**

This effect causes a continuous increase in pitch. The operand determines the rate of increase.

**Effect 02**  **Portamento Down (one operand)**

This effect causes a continuous decrease in pitch. The operand determines the rate of decrease.

**Effect 03**  **Portamento To Note (one operand)**

Similar to Effects 01 and 02, but includes a target pitch. The operand determines the rate of change.
Sound Support For Your Programs

Effect 04  
**Vibrato (two operands)**

Creates a vibrato effect. The first operand indicates the rate of the vibrato and the second operand indicates its depth.

Effect 05  
**Portamento To Note + Volume String (two operands)**

This combines the two effects Portamento To Note and Volume Sliding. The upper part of the operand (Xx) gives the portamento rate; the lower part (xX) gives the volume reduction rate.

Effect 06  
**Vibrato + Volume String (two operands)**

This combines the two effects Vibrato and Volume Sliding. The first operand (Xx) gives the rate of the vibrato; the second operand (xX) gives the volume reduction rate.

Effect 07  
**Tremolo (two operands)**

The tremolo effect is very similar to the vibrato but the vibration involves a change of volume rather than pitch. Since this technique is frequently used to simulate a vibrato, the playback routines for the two effects are often the same. The first operand (Xx) gives the tremolo rate; the second operand (xX) gives its depth.

Effect 08  
**Not yet defined**

This is the only undefined command. It is available for use in custom routines.

Effect 09  
**Set Sample Offset (one operand)**

This effect lets you play a sample from a given offset position within the sample data, rather than from the beginning. The value of the operand refers to the first two digits of the sample length.

Effect 10  
**Volume Sliding (one operand)**

This effect increases or decreases the volume of an instrument. An operand of the type n0 increases the volume at rate n; an operand of the type 0n decreases the volume at rate n.

Effect 11  
**Position Jump (one operand)**

Interrupts playback of the current pattern and continues with the pattern specified by the operand. The operand gives the line where output should be resumed.

Effect 12  
**Set Note Volume (one operand)**

This effect sets the volume of a note. The value of the operand is a volume setting from 0 to 63.

Effect 13  
**Pattern break (no operands)**

This effect interrupts playback of the current pattern and continues with the next pattern.
Effect 14 has several subfunctions, each producing a different effect. The first operand specifies the subfunction; the second serves as a parameter for the subfunction.

**Effect 14.1**  
*Fineslide Up*

This effect is similar to the Portamento Up. Instead of a continuous rise in pitch, however, there is just one initial slide. The new pitch is sustained after this initial slide.

**Effect 14.2**  
*Fineslide Down*

This effect works like Effect 14.1, except the pitch slides down instead of up.

**Effect 14.3**  
*Glissando Control*

Glissando works with the Portamento commands. If Glissando is enabled, the change in pitch occurs in half-step intervals instead of a continuous gradient. The value 1 in the operand enables the effect, 0 disables it.

**Effect 14.4**  
*Vibrato Waveform*

This effect lets you select the waveform for a vibrato. The default operand value of 0 designates a sine wave. Other values are 1 for continuously decreasing, 2 for a rectangular wave and 3 for a random form.

**Effect 14.5**  
*Set Finetune*

This function lets you change the Finetune value, as indicated by the following table:

<table>
<thead>
<tr>
<th>Finetune:</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>0</th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
<th>-4</th>
<th>-5</th>
<th>-6</th>
<th>-7</th>
<th>-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>09</td>
</tr>
</tbody>
</table>

**Effect 14.6**  
*Pattern Loop*

This effect lets you loop a specified part of a pattern. The parameter 0 indicates the loop start; each subsequent parameter (n) plays the loop from the start position to the current position n times, before continuing.

**Effect 14.7**  
*Tremolo Waveform*

This effect lets you select the waveform for a tremolo. The default operand value of 0 designates a sine wave; other values are 1 for continuously decreasing, 2 for a rectangular wave and 3 for a random form.

**Effect 14.10**  
*Fine Volume Sliding Up*

This effect produces an initial volume increase. The new volume is sustained after this initial volume increase. It’s similar to normal volume sliding, but the increase in volume is a one-time change rather than a continuous change.
Sound Support For Your Programs

Effect 14.11  
*Fine Volume Sliding Down*

This effect is like Effect 14.10, but the volume slides down instead of up.

Effect 14.12  
*Note Cut*

This effect shortens the sounding of a note. The operand gives the number of ticks for which the note is sustained. The Set Speed command determines the normal speed in ticks, where F06 means six ticks, for example. The value 4, in that case, sustains the note for 2/3 its normal time.

Effect 14.13  
*Note Delay*

This effect produces a delay before sounding a note. The operand gives the delay time in ticks. The technique is useful for creating echo effects.

Effect 14.14  
*Pattern Delay*

This effect delays the sounding of a pattern for the amount of time specified in the operand.

Effect 15  
*Set Speed (one operand)*

This effect sets the playback speed for the MOD file. Valid values for the operand range from 0 through 31.

The 669 format

In addition to the standard MOD format, there are several others, most of which have been established by the leading demo crews. The 669 format comes from Renaissance. It resembles the MOD standard, but is somewhat more compact. Like the standard format, it has three parts:

The 669 header

Unlike the standard fixed-length header, the 669 header is variable. Its length depends on the number of samples defined.

BYTE 0-1  
*Ident*

These two bytes identify the file as a 669 file. They contain the value $6669.$

BYTE 2-110  
*Song message*

These 108 bytes contain informational text about the song, in three lines of 36 characters each. This feature is unique to the 669 format.

BYTE 111  
*Number of samples*

This byte contains the number of samples stored. Valid values are from 0 to 64, allowing a maximum sample count of double that of the standard MOD file.
Chapter 13

Sound Support For Your Programs

<table>
<thead>
<tr>
<th>BYTE 112</th>
<th>Number of patterns</th>
</tr>
</thead>
</table>

This byte contains the number of patterns stored. It is a convenient addition and eliminates the need to derive the number by calculation.

<table>
<thead>
<tr>
<th>BYTE 113</th>
<th>Repeat start</th>
</tr>
</thead>
</table>

This byte indicates the pattern where a repeat loop begins.

<table>
<thead>
<tr>
<th>BYTE 114-242</th>
<th>Song arrangement</th>
</tr>
</thead>
</table>

This 128-byte array contains the song arrangement. Each entry acts as an index for the pattern to be played. Valid values are from 0 to 128.

<table>
<thead>
<tr>
<th>BYTE 243-371</th>
<th>Tempo list</th>
</tr>
</thead>
</table>

This 128-byte array contains the tempos at which the corresponding patterns should be played.

<table>
<thead>
<tr>
<th>BYTE 372-500</th>
<th>Break Position List</th>
</tr>
</thead>
</table>

This 128-byte array contains the break positions within the individual patterns. An entry in the list indicates the line where pattern processing should end. Valid values are from 1 to 64.

<table>
<thead>
<tr>
<th>BYTE 501-501+(Sample number *$19)</th>
<th>Sample information</th>
</tr>
</thead>
</table>

The next section contains the sample information. The length of the section depends on the number of samples declared. The record length is $19 bytes for each sample. It has the following structure:

  13 bytes File name of the instrument as an ASCII string
  1 dword (= longint) Instrument length
  1 dword Start position of instrument loop, from Inst.-Start
  1 dword End of instrument loop, from Inst.-Start

The 669 pattern

<table>
<thead>
<tr>
<th>BYTE 501+(Sample number &quot;$19)</th>
<th>Sample information</th>
</tr>
</thead>
</table>

The next part defines the individual patterns. A pattern consists of 64 lines of 8 notes each. A note is 3 bytes long, giving a total length of 1536 bytes per pattern. The information in these three bytes is packed and crosses byte boundaries. The individual bits have the following meanings:

<table>
<thead>
<tr>
<th>BYTE 0</th>
<th>BYTE 1</th>
<th>BYTE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>76543210</td>
<td>76543210</td>
<td>76543210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 0, Bit 7 to Byte 0, Bit 2</th>
<th>Note value</th>
</tr>
</thead>
</table>

These 6 bits designate the note to be played. You can compute it with the following formula:
NOTE VALUE = (12*Octave)+Note

The note D1 therefore has the note value 12*1+2 = 14. Five octaves can be addressed in this way.

<table>
<thead>
<tr>
<th>Byte 0, Bit 1 to Byte 1, Bit 4</th>
<th>Instrument number</th>
</tr>
</thead>
</table>

These six bits give the number of the instrument to be played.

<table>
<thead>
<tr>
<th>Byte 1, Bit 3 to Byte 1, Bit 0</th>
<th>Volume</th>
</tr>
</thead>
</table>

These four bits give the volume of the note to be played. The resulting range of only 16 possible volume levels is normally quite adequate, since the ear can barely distinguish finer differences.

<table>
<thead>
<tr>
<th>Byte 2, Bit 7 to Byte 2, Bit 4</th>
<th>Command</th>
</tr>
</thead>
</table>

These four bits give the command to be executed. The following values are used:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Portamento Up</td>
<td>3</td>
<td>Pattern Break</td>
</tr>
<tr>
<td>1</td>
<td>Portamento Down</td>
<td>4</td>
<td>Vibrato</td>
</tr>
<tr>
<td>2</td>
<td>Portamento To Note</td>
<td>5</td>
<td>Set Speed</td>
</tr>
</tbody>
</table>

Note the command operand for the vibrato can be interpreted according to the following formula:

VIBRATO VALUE = Operand SHL 4 + 1.

<table>
<thead>
<tr>
<th>Byte 2, Bit 3 to Byte 2, Bit 0</th>
<th>Command operand</th>
</tr>
</thead>
</table>

The last four bits store the operand values for the effect.

Special values

The following byte combinations have a special meaning:

<table>
<thead>
<tr>
<th>Byte 0 = $FE:</th>
<th>No note, only volume change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0 = $FF:</td>
<td>No note and no volume change</td>
</tr>
<tr>
<td>Byte 2 = $FF:</td>
<td>No command</td>
</tr>
</tbody>
</table>

**The Scream Tracker file format**

Another very popular format is the Scream Tracker format from Future. Unfortunately, information about this format is difficult to obtain. In Scream Tracker format, a distinction is made between songs and modules, based on whether the instruments are also saved. A module includes the instruments, while a song does not. Since only the module is in widespread use, we consider this form primarily in our discussion.
The STM header

The header of an STM file created with Scream Tracker V2.24 is as follows:

<table>
<thead>
<tr>
<th>BYTE 0-19</th>
<th>Song name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE 20-27</td>
<td>Tracker name</td>
</tr>
<tr>
<td>BYTE 28</td>
<td>Version</td>
</tr>
<tr>
<td>BYTE 29</td>
<td>File type</td>
</tr>
<tr>
<td>BYTE 30-32</td>
<td>Version number</td>
</tr>
<tr>
<td>BYTE 32</td>
<td>Tempo</td>
</tr>
<tr>
<td>BYTE 33</td>
<td>Pattern count</td>
</tr>
<tr>
<td>BYTE 34</td>
<td>Global volume</td>
</tr>
<tr>
<td>BYTE 36-47</td>
<td>Reserved</td>
</tr>
<tr>
<td>BYTE 48-80</td>
<td>Instrument number 1</td>
</tr>
<tr>
<td>BYTE 48-59</td>
<td>Instrument name</td>
</tr>
</tbody>
</table>

The first 20 bytes contain the song name in ASCII format. You use Array of char to reference the name for processing.

These 8 bytes contain the name of the Tracker that created the song. The string is not 0-terminated.

This byte contains the version number, which is presently $1A.

This byte indicates the file type, 1 for a song (without samples), 2 for a module (with samples).

These two bytes contain the version number, with the main number in Byte 30 and the subversion number in Byte 31.

This byte indicates the tempo at which the file should be played.

This byte contains the number of patterns stored, which is needed later for loading the patterns. You don’t have to compute it like you do for a standard MOD file.

This byte gives the volume at which the file should be played. Valid values are from 0 to 63.

Bytes 36 through 47 are reserved for internal use by the Scream Tracker program.

The next 32 bytes contain data for the first instrument. The instrument data is divided as follows:

Bytes 48 through 59 contain the name of the first instrument as ASCII text.
### Sound Support For Your Programs

#### Chapter 13

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td><strong>Instrument name zero</strong></td>
</tr>
<tr>
<td>61</td>
<td><strong>Instrument drive</strong></td>
</tr>
<tr>
<td>62-63</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>64-65</td>
<td><strong>Instrument length</strong></td>
</tr>
<tr>
<td>66-67</td>
<td><strong>Loop start</strong></td>
</tr>
<tr>
<td>68-69</td>
<td><strong>Loop end</strong></td>
</tr>
<tr>
<td>70</td>
<td><strong>Volume</strong></td>
</tr>
<tr>
<td>71</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>72-73</td>
<td><strong>Speed for C1</strong></td>
</tr>
<tr>
<td>74-77</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>78-79</td>
<td><strong>Segment address / length in paragraphs</strong></td>
</tr>
</tbody>
</table>

Byte 60 is always 0 and terminates the instrument name (ASCIIZ).

This byte indicates the drive where the instrument is stored. Songs (which do not include samples) require this information.

These two bytes are reserved and are used internally for the segment address.

These two bytes give the length of the instrument, expressed in bytes.

Bytes 66 and 67 give the start position of the loop within the sample data where the instrument can be played.

These two bytes contain the end position of the loop within the sample data.

This byte gives the default volume of the instrument. The value can be from 0 to 63 and can be overwritten at run time.

This byte is reserved.

These two bytes are supposed to contain the speed for C1. They usually contain the value 0.

These four bytes are reserved for internal use.

These bytes contain the internal segment address in a song. In the much more common module, they contain the length in paragraphs (length in bytes DIV 16). The same information now repeats for the remaining 30 instruments. Then the structure continues as follows:
Chapter 13

Sound Support For Your Programs

**BYTES 1028-1091**  
**Arrangement**

This 64-byte array contains the arrangement of the piece. Each entry indicates the number of the corresponding pattern.

**BYTES 1092-2115**  
**Pattern 1**

Bytes 1092 through 2115 contain the first pattern. It consists of 64 lines of 4 notes each. A note contains 4 bytes. The resulting pattern size is therefore 1024 bytes (64*4*4). Notes are constructed as follows:

**BYTES 0, Bits 7 to 4**  
**Octave**

These four bits contain the octave number of the note to be played.

**BYTES 0, Bits 3 to 0**  
**Note**

This is the note to be played; 0 = C, 1 = C#, 2 = D etc.

**BYTES 1, Bits 7 to 3**  
**Instrument**

These five bits give the number of the instrument to be played.

**BYTES 1, Bit 2 to BYTE 2, Bit 4**  
**Volume**

These bits contain the current volume for the instrument.

**BYTES 2, Bits 3 to 0**  
**Effect**

These four bits contain the effect number. Effects are numbered as explained in the standard MOD-file layout.

**BYTE 3**  
**Operands**

This byte contains the effect operands. Subsequent patterns follow the same structure. There are as many patterns as indicated by the pattern count in the header. If the file is a module, the sample data follows the patterns. Its length is also given in the header.

**The S3M format**

The S3M format is the one currently used by Future. It originated with Scream Tracker 3.0 and is one of the most flexible sound formats. AdLib pieces as well as sample modules can be stored in this format. Again, a distinction is made between modules, which include samples, and songs, which contain only the arrangement.

**The S3M header**

**BYTES $00-$1B**  
**Song name**

The header begins with the name of the piece, terminated in 0.
**Sound Support For Your Programs**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1C</td>
<td>Ident $1A</td>
</tr>
</tbody>
</table>

This byte always contains the value $1A. It is for identification only and has no other function.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1D</td>
<td>File type</td>
</tr>
</tbody>
</table>

This indicates whether the file is a module or a song. The value 16 indicates a module; the value 17 indicates a song.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1E-$1F</td>
<td>Not used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$20-$21</td>
<td>Arrangement length</td>
</tr>
</tbody>
</table>

This word contains the variable arrangement length. It must be an even number.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$22-$23</td>
<td>Number of instruments</td>
</tr>
</tbody>
</table>

This word contains the number of instruments used in the piece.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$24-$25</td>
<td>Number of patterns</td>
</tr>
</tbody>
</table>

This is the number of patterns defined. The value is also needed for determining the length of the pattern parapointer.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$26-$27</td>
<td>Flags</td>
</tr>
</tbody>
</table>

The flags affect the initialization of various effects. The bit meanings are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ST2 vibrato</td>
<td>2</td>
<td>ST2 tempo</td>
</tr>
<tr>
<td>3</td>
<td>Using Amiga sliding</td>
<td>4</td>
<td>Optimize 0 volumes</td>
</tr>
<tr>
<td>5</td>
<td>Observe Amiga boundaries</td>
<td>6</td>
<td>Enable filter and sound effects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$28-$29</td>
<td>Version information</td>
</tr>
</tbody>
</table>

These two bits contain the version information. The Tracker number is in the upper four bits, and the version number is in the lower twelve. The value for Scream Tracker 3.0 is $1300.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2A-$2B</td>
<td>Version of file format</td>
</tr>
</tbody>
</table>

There are currently two versions of the file format. The value 1 denotes the original format. The value 2 indicates the samples are in unsigned format, as used by the PC.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2C-$2F</td>
<td>Ident</td>
</tr>
</tbody>
</table>

These bytes contain the identifier ‘SCRM’, indicating the Scream Tracker format.
Chapter 13

Sound Support For Your Programs

<table>
<thead>
<tr>
<th>Byte $30</th>
<th>Master volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>This byte contains the initial master volume.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte $31</th>
<th>Start speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>This byte contains the initial speed (Command A).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte $32</th>
<th>Start temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>This byte contains the initial tempo (Command T).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte $33</th>
<th>Master multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>The value of the master multiplier is contained in the lower four bits. You use AND $0F to evaluate it. A value &lt;&gt; 0 in the upper four bits indicates that stereo is used.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte $34-$3F</th>
<th>Not used</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Byte $40-$5F</th>
<th>Channel settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>These 32 bytes contain the settings for up to 32 channels. Each byte is interpreted as follows:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sample on left channel (S1)</td>
<td>1</td>
<td>Sample on left channel (S2)</td>
</tr>
<tr>
<td>2</td>
<td>Sample on left channel (S3)</td>
<td>3</td>
<td>Sample on left channel (S4)</td>
</tr>
<tr>
<td>4</td>
<td>Sample on right channel (S5)</td>
<td>5</td>
<td>Sample on right channel (S6)</td>
</tr>
<tr>
<td>6</td>
<td>Sample on right channel (S7)</td>
<td>7</td>
<td>Sample on right channel (S8)</td>
</tr>
<tr>
<td>8</td>
<td>AdLib Melody voice, left (A1)</td>
<td>9</td>
<td>AdLib Melody voice, left (A2)</td>
</tr>
<tr>
<td>10</td>
<td>AdLib Melody voice, left (A3)</td>
<td>11</td>
<td>AdLib Melody voice, left (A4)</td>
</tr>
<tr>
<td>12</td>
<td>AdLib Melody voice, left (A5)</td>
<td>13</td>
<td>Adlib Melody voice, left (A6)</td>
</tr>
<tr>
<td>14</td>
<td>AdLib Melody voice, left (A7)</td>
<td>15</td>
<td>AdLib Melody voice, left (A8)</td>
</tr>
<tr>
<td>16</td>
<td>AdLib Melody voice, left (A9)</td>
<td>17</td>
<td>AdLib Melody voice, right (B1)</td>
</tr>
<tr>
<td>18</td>
<td>AdLib Melody voice, right (B2)</td>
<td>19</td>
<td>AdLib Melody voice, right (B3)</td>
</tr>
<tr>
<td>20</td>
<td>AdLib Melody voice, right (B4)</td>
<td>21</td>
<td>AdLib Melody voice, right (B5)</td>
</tr>
<tr>
<td>22</td>
<td>AdLib Melody voice, right (B6)</td>
<td>23</td>
<td>AdLib Melody voice, right (B7)</td>
</tr>
</tbody>
</table>
## Sound Support For Your Programs

### Chapter 13

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>AdLib Melody voice, right (B8)</td>
<td>25</td>
<td>AdLib Melody voice, right (B9)</td>
</tr>
<tr>
<td>26</td>
<td>AdLib Basedrum, left (AB)</td>
<td>27</td>
<td>AdLib Snare, left (AS)</td>
</tr>
<tr>
<td>28</td>
<td>AdLib Tom, left (AT)</td>
<td>29</td>
<td>AdLib Cymbal, left (AC)</td>
</tr>
<tr>
<td>30</td>
<td>AdLib Hihat, left (AH)</td>
<td>31</td>
<td>AdLib Basedrum, right (BB)</td>
</tr>
<tr>
<td>32</td>
<td>AdLib Snare, right (BS)</td>
<td>33</td>
<td>AdLib Tom, right (BT)</td>
</tr>
<tr>
<td>34</td>
<td>AdLib Cymbal, right (BC)</td>
<td>35</td>
<td>AdLib Hihat, right (BH)</td>
</tr>
<tr>
<td>&gt; 128</td>
<td>Not active</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bytes $60-($60+Arrangement length)</th>
<th>Arrangement</th>
</tr>
</thead>
</table>

Here again is the arrangement of the piece. The entry values indicate the patterns to be played.

<table>
<thead>
<tr>
<th>Bytes xxxx</th>
<th>Instrument parapointers</th>
</tr>
</thead>
</table>

Following the arrangement you’ll find the parapointers (paragraph pointers) to the individual instruments. The value of a parapointer is the offset from the beginning of the header DIVided by 16. Each parapointer is one word long, so the total length of the instrument parapointers is (Number of instruments) * 2.

<table>
<thead>
<tr>
<th>Bytes xxxx</th>
<th>Pattern parapointers</th>
</tr>
</thead>
</table>

After the instrument parapointers are the parapointers to the individual patterns. They indicate where each pattern begins within the file. The total length of the pattern parapointers is (Number of patterns) * 2.

### The S3M patterns

The length of a pattern is determined as follows:

Number of channels used * 320 bytes per channel

The notes of the individual lines are stored consecutively. Each note consists of five bytes. Their meanings are as follows:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>Bit7-4 = Octave</td>
</tr>
<tr>
<td></td>
<td>Bit4-0 = Note</td>
</tr>
<tr>
<td></td>
<td>255 = Empty</td>
</tr>
<tr>
<td></td>
<td>254 = Key Off (for AdLib)</td>
</tr>
</tbody>
</table>

| Byte 1 | Instrument #. 255 = Empty                    |
| Byte 2 | Volume. 255 = Empty                          |
| Byte 3 | Command. 255 = Empty                         |
| Byte 4 | Command parameters                           |
The S3M instruments

The S3M format distinguishes between two types of instruments. One type is the sample instrument that we have already seen with the standard MOD file. The other type supported is the AdLib instrument. Both types start with a $40-byte header. This is followed by the sample data for sample instruments. We’ll look first at the structure of the header for a sample instrument:

- **Byte $0** **Instrument type**
  The value 1 in this byte indicates a sample instrument. Otherwise the instrument is an AdLib instrument.

- **Byte $1-$C** **DOS file name**
  The name of the sample file is stored in these bytes. This is not a 0-terminated string!

- **Byte $D-$F** **Storage segment**
  This field contains the paragraph position (Offset DIV 16) of the sample data.

- **Byte $10-$13** **Sample length**
  This double word contains the length of the sample data. We recommend checking whether the length exceeds the Word area and if necessary, working with Word variables.

- **Byte $14-$17** **Loop start**
  This double word contains the loop start position within the sample data.

- **Byte $18-$1B** **Loop end**
  This double word contains the loop end position within the sample data.

- **Byte $1C** **Volume**
  This byte contains the default volume of the instrument.

- **Byte $1D** **Disk**
  This byte indicates the disk where the instrument resides.

- **Byte $1E** **Pack type**
  This byte tells whether the sample data is packed. The value 0 indicates unpacked 8-bit data; 1 indicates the use of the DP30ADPCM1 packing algorithm.

- **Byte $1F** **Flags**
  The flags give additional information about the sample data. If Bit 1 is set, looping is enabled. Bit 2 indicates stereo data, and Bit 3 denotes a 16-bit sample in Intel format.
This value is used for tuning the instrument. It indicates the speed at which the sample must be played to obtain the tone C2.

This word is used only for Scream Tracker 3.0. It gives the position of the sample in the Gravis UltraSound RAM DIV 32.

This is the name of the instrument as a 0-terminated ASCII string of up to 28 characters.

This is the identifier “SCRS” for Scream Tracker Sample.

Now we look at the AdLib instruments. The header is much the same as that of the sample instruments:

This gives the instrument type, as follows:

The DOS file name of the AdLib instrument is found here.

Bytes $D through $F contain the value 0
Chapter 13

Sound Support For Your Programs

| Byte $10-$1B | FM synthesis table |

Bytes $10 through $19 describe first the modulator and then the carrier for the FM synthesis. Unfortunately, some of the information given here is technically incomplete.

The byte values derive from the following formulas:

- Bytes $10 & $11 = (Frequency multiplier) + (Sustain * 32) + (Pitch vibrato * 64) + (Volume vibrato * 128)
- Bytes $12 & $13 = (63 - Volume) + ((Levelscale AND 1) * 128) + ((Levelscale AND 2) * 64)
- Bytes $14 & $15 = (Attack * 16) + decay
- Bytes $16 & $17 = ((15 - Sustain) * 16) + Release
- Bytes $18 & $19 = Wave select
- Byte $1A = (Modulation feedback * 2) + Additive synthesis
- Byte $1B = Not used
- Byte $1C = Volume

This byte contains the default volume for the instrument.

| Byte $1D | Disk |

This byte gives the disk where the instrument resides.

| Byte $1E-$1F | Not used |

| Byte $20-$23 | C2 Speed |

This value is used for tuning the instrument. It indicates the speed at which the instrument must be played to obtain the tone C2.

| Byte $24-$2F | Not used |

| Byte $30-$4B | Instrument speed |

This contains the name of the instrument as a 0-terminated ASCII string of up to 28 characters.

| Byte $4C-$4F | Ident |

This contains the identifier "SCRI" for Scream Tracker Instrument.

A MOD Player For The Sound Blaster Card

One way to bring your programs to life is to add sound. The easiest approach is to use a VOC player. With up to 16,000 bytes of data to process per second, however, a 2.5 minute demo uses 2.3 Meg of sound data. Even with looping and compressing data, the size cannot be reduced sufficiently to make VOC files practical except for CD-ROM applications.

We'll need an alternative way to produce high quality sound but which has a much smaller storage requirement. Although a MOD file is acceptable, these players are highly hardware-dependent. In this section we'll look at a player for the Sound Blaster card. The next section describes a player for the Gravis UltraSound, which is much better suited to this purpose.
The Sound Blaster is the most widely used sound card. This card remains a "standard", and although it may take some work, you'll want to support it as much as your CPU speed and storage space allow.

You'd think that playing MODs on the Sound Blaster should be trouble-free. After all, sample output using DMA is much simpler and takes virtually no computing time. Unfortunately, this is only partly true. Let's look at the Amiga where the MOD format originated. It has four DACs for sound output. MOD playing was trouble-free with these four channels because the only manipulations needed were for volume and frequency control. However, it's different for the Sound Blaster. It has only one DAC (two with the stereo models). As a result, we can't simply "play" the data...everything has to be calculated. And with an output frequency of 22 KHz, this means 22,000 "pieces" of data per second.

Therefore, we'll need a truly fast player to handle background sound for a graphics demo without affecting its performance. This presents us with the next limitation: Because of speed requirements, we're forced to use fixed-point arithmetic, while the 16-bit register width of a 286 cannot accommodate this requirement for 24-bit resolution. True, a few tricks can get us around this problem, but they take extra resources and impact performance accordingly. Since the 286 is a "dinosaur", this approach is really not justified.

Writing the player in Turbo Pascal, the language in which many demos are programmed, brings up the next question, namely, how to effectively handle 32-bit instructions. Unfortunately, 32-bit code is not supported in Turbo Pascal before Version 8, and native Pascal is much too slow. Our only choice is to incorporate an external assembler module. This can give us all the capabilities and the complete instruction set of the 386 processor.

**Basic principles of a MOD player**

Let's see how a MOD player works. How do we get different pitches?

The data in a MOD file represents the changing amplitude of the sampled oscillation. On the Amiga, this data takes on values from -128 to 127, while the corresponding PC range is from 0 to 255. The value for a period of silence, when no oscillation is present, is 0 on the Amiga and 127 on the PC. To convert a sample from Amiga to PC format, we increase each value by 127. By storing only the amplitude, we simplify the task. To produce differences in pitch, we use a principle from the days of the old record player. If you play a recording too fast, it sounds too high. Played too slow, a soprano becomes a bass. This is exactly what we do with the samples. Controlling the speed of playback controls the pitch.

In practice we cannot constantly adjust the sample rate for the Sound Blaster card. This works for a single voice, but not for the four or more voices that a MOD file is likely to contain. So, we'll need to use a little trick. Suppose we play our MOD file at a frequency of 16 KHz. This means that if we play a 16 KHz sample, we will hear a tone at its original pitch. We simply advance byte by byte in our sample to output a total of 16000 bytes. However, now if we want to play the sample at 32 KHz, we have to double the speed. This means that we can no longer process all the bytes in the time available, so we take only every other byte. To play at 8 KHz, we have extra time available, so we play each byte twice.

Now for a little music theory: For any half-step interval, the frequency ratio is always the 12th root of 2. Consequently, to play a tone that is one half-step higher than its predecessor, the playback rate must be $1.059463094$ times as fast. To go a half-step lower, the rate is reduced by a factor of $0.943874312$. Our program will have to make use of this information. But instead of performing the necessary calculations for each interval during execution, we reference the appropriate values from a predefined table. The factor by which the values differ is always the 12th root of 2.
Volume is the next characteristic to consider. We’ve already mentioned the sample values represent oscillation amplitudes. An amplitude of 0 (or 127) denotes silence and the stronger the amplitude, the louder the sound. The volume values for the voices in a MOD file range from 0 to 63. So we simply take the sample byte and multiply it by the volume, then divide by the maximum volume of 64. Since 64 equals 2, the division can be performed by a SHR 6 instruction, which saves 13 clock cycles each time.

A vibrato effect can be produced by manipulating the volume. Here again we read values from a table and apply them to the base volume to alternately lower or raise it. A similar effect can be obtained by periodically alternating the sample speed. You can choose whichever method you prefer. The volume method requires slightly less computing time (you work only with bytes), but it sounds a little less clean. It can also lead to problems when used with such other effects as volume sliding.

Mixing sound data: Building a mixing procedure

Now that we understand a little sound theory, let’s turn our attention to the player program. The main work is performed by a mixing procedure. For a single output byte, this procedure must be executed once for each active voice. If a voice is inactive, we can skip this processing or call a dummy procedure. The following illustration shows how the mixing procedure looks schematically:

<table>
<thead>
<tr>
<th>Voice position &lt; End position for voice ?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Does voice loop?</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Position of start value</td>
<td>Load mixingproc to emptyproc and emptybyte</td>
</tr>
<tr>
<td>Load byte from sample</td>
<td></td>
</tr>
<tr>
<td>Adjust value for correct volume</td>
<td></td>
</tr>
<tr>
<td>Add to mixbuffer</td>
<td></td>
</tr>
<tr>
<td>Increment voice position by Incfactors</td>
<td></td>
</tr>
</tbody>
</table>

Output of mixing procedure

At the start of the procedure, we check to see if the end of the voice has been reached. Rather than looking for the very end, we look for a value just before it. The few bytes that are lost will not be noticed. Otherwise, we almost always overshoot the exact end which will mix extraneous data into our output, producing a clicking noise.

If the end of the voice has not been reached, we can proceed with normal processing. If it has been reached, we must check to see if the voice loops. If so, we reposition the voice to the start of the loop. Otherwise, we load an empty byte (amplitude = 0) and call the dummy procedure we’ve mentioned already instead of the mixing procedure.

For normal processing, we get a new byte from the sample. Next we adjust the value as described to arrive at the correct volume. Now the byte can be added to the mixer buffer. The buffer is then divided by the number of voices to interpolate the correct overall volume.
Finally, to advance to the next position within the sample, we increment the current position by the value of Inc. The position and the Inc factor should be 32-bit values, giving the computation a precision of 16.16 bits.

The Sound Blaster MOD player

Developing a MOD player is a major undertaking. Even experienced programmers can be intimidated by such a time-consuming and error-prone project. To prevent these problems, we’ve included a fairly flexible unit called MOD_SB which you can incorporate into your programs.

One way to "drive" the program is by timer interrupts. You call the procedure 50 times per second using this method. Output is controlled by the SB interrupt. This is the most efficient solution and works on any card that can handle the SB DMA transfer.

An alternative method is by polling. Computation occurs here at a precise time (as in the case of a retrace) and not periodically. This method is less efficient, since the SB interrupt may execute before the sound data computation is ready. Still there are areas in a demo that require polling, so we include it although the quality is not as good.

The unit is quite fast. To output an 8-voice MOD file at 22 KHz on a 486DX-33, it requires less than 15% computing time. For a 4-voice MOD file, this figure drops to about 13%. Depending on how computation-intensive your program is, you can use either 4-voice or 8-voice files. If you don’t use all channels of an 8-voice file, the output will be correspondingly faster.

As a bonus, the unit also contains a routine to play VOC files. So here’s a universal player for the Sound Blaster card.

The MOD386 program shows how the unit is used. It’s a small MOD and VOC playing program that lets you type the name of the file to be played or select it from a menu. Use the -r parameter on the command line to start the program and in Repeat mode. You can play different music clips (MOD and VOC) in succession in Repeat mode.

To see how much computing time you have left for a MOD file, press P for Performance. The program then displays the remaining time and system performance (it takes about 3 seconds).

The system performance is determined by checking how much time the system has left before a vertical retrace. If it’s simply waiting for the retrace and incrementing the test variable, the available performance is 100%. This test is performed every time the program starts.

When the performance test is called again, the loop seems to do nothing more than wait for the retrace. The difference, however, is the MOD file plays in the background through the timer. Processing the sound data takes from 10% to 30% of system performance, depending on the computer and the sample rate. This accordingly lowers the value that is counted in the loop and used to derive the performance.
Variables required for the MOD player

Now you should have a basic idea on what to expect from the MOD_SB unit. To take full advantage of the unit and perhaps extend it (a track for sound effects would be one possibility), you can become familiar with its data and procedures. Therefore, we’ll describe the essential variables and their functions before proceeding with the actual programming.

First, we look at the types provided by the unit:

VocHeader

Basic information about a VOC file is stored in VocHeader. This begins with the 20-byte identifier “Creative Voice File”+1Ah. This is followed by the position offset for the start of the sample data as a word. Next is the VOC file version number, first the byte with the high number, then the byte with the low number. The encoded version number at the end is unimportant to our work.

VoiceBlock

A VOC file is divided into blocks. Each block begins with a header, which is read into this datatype. The first byte contains the identifier, the next three bytes the length, the next byte the sample rate, and the last byte the packing type.

Pt

Pt is a simple data type that facilitates working with pointers. It includes first the word ofs, the offset portion of the pointer, then sgm, the segment portion.

These are all the types used. Now we’ll need to describe the constants:

Incfacts : array[1..99] of longint;

Incfacts is a table of precalculated factors containing tone intervals as fixed-point values. The lower 16 bits contain the fractional portion of the interval, and the upper 16 bits contain the integral portion.

outfading : boolean = FALSE;

If you don’t want the MOD file to end abruptly, set this variable to TRUE. The output volume is then reduced gradually until it reaches 0.

outvolume : byte = 63;

Outvolume specifies the volume at which the MOD file will be played. The maximum value is 63, the minimum is 0 (= silence). This variable can change, as when outfading is set to TRUE, for example.
This variable is intended especially for using the unit in demos or games. There you'll want to have several MOD files packed into a single large sound file. Since you must be able to recognize each MOD file individually, this variable is set to the size of the MOD file after packing occurs. A value of 0 indicates the MOD file is not part of a packed file.

Mastervolume is a variable that is used for cards starting with SB Pro. These cards include a mixer chip that allows volume control at the hardware level. The maximum volume is 31, the minimum is 0.

The SoundBlaster card is recognized automatically. This is done by testing the ports from Startport to Endport in increments of $10. If you don't want a port tested (this can cause problems with a network card, for example), you can set the Startport and/or Endport values accordingly. Similarly, if you know which port to use and don't need to test any others, set the Startport and Endport both to this value.

You can turn off base port detection by setting this variable to TRUE. The default value of 220h is then assumed.

This global constant tells the SB-Interrupt procedure not to output the next block, because an interrupt check is in progress.

The timer_per_second variable contains the number of timer calls to be executed per second.
This variable is set to TRUE when the mod_waitretrace procedure is active. This is necessary to turn off the Mixing procedure when synchronizing with the vertical retrace. Otherwise, the Mixing procedure would be called during the wait, making synchronization impossible.

The variable dsp_irq contains the number of the SB-IRQ. This interrupt must be diverted to a custom handling routine. When IRQ detection is disabled, you can preassign an interrupt by setting this variable to the desired value.

This is the number of the DMA channel used. You must change this variable if the Sound Blaster card is using a channel other than 1.

This variable contains the base address of the Sound Blaster card. It's set to the proper value automatically by the initialization procedure. If you disable this initialization, you must set the correct value manually.

The Detect routine uses these two variables to save the version number of the Sound Blaster card that it finds. SbVersMaj contains the major version number, and SbVersMin contains the minor number. The major number is more important; you'll use it to identify the card.

The Initialization procedure modifies this variable. If it is set to TRUE, a Sound Blaster card was found. Otherwise, you can abandon further processing of the SB routines, since no Sound Blaster is installed.

This variable is also set by the Initialization procedure. If an SB Pro or an SB 16 card is found, it is set to TRUE.

If an SB 16 (ASP) is installed in the computer, the Initialization procedure sets this to TRUE. In this case, SbProDetected also is TRUE.
Sound Blaster cards (starting with the SB Pro) include a mixer chip. If such a card is installed, this variable is set to TRUE.

Voices contains the number of voices used in the MOD file. In this version of the unit, the variable can have a value of 4 or 8.

The values in this array correspond to the pitches stored in the MOD file. To determine the pitch value for a note, you use the note's position within the array.

DMA channels 0 through 3 are supported. This variable is needed for direct programming of DMA transfer and avoids having to link another unit.

See dma_page.

When you output data to an SB 16 using a DMA transfer, you should use the DMA Continue command starting with the second data block. This minimizes crackling. The method works only with fixed-length blocks, however. The size of the last block that was output will be found in this variable. If the next block to be output agrees with this size, the DMA Continue command can be used.

You set this variable to TRUE to halt DMA output. Then when the Interrupt procedure runs, it will not initiate output of a new block.

Those were the constants used in the unit. Now we finally come to the variables. We'll only talk in detail about the most important variables (those that are not used as counters or buffers).

The size of the block to be output is stored here. This data is always computed during an interrupt execution.
This flag is set to TRUE when data transfer via DMA is finished. While data is being transferred, its value is FALSE. The somewhat misleading sb16 suffix exists for historical reasons. The variable is also used for other cards in addition to the SB16.

The variable OldInt saves the address of the old timer interrupt, so it can be reset at the end of the program. You can also use this variable to call the old interrupt.

This variable is needed for masking the Sound Blaster interrupt.

Different procedures are called depending on whether a MOD file has four or eight voices. The decision of which procedures to use is made only once at which time the above pointers are initialized to the appropriate procedure addresses. This will make the program simpler and faster. The procedures are subsequently accessed through the pointers.

This variable is used to control a simple equalizer. The values in the array represent the amplitude of the associated channels. This is set to 500 each time a new note is struck.

This array manages the individual patterns. The pointers index the start positions of the corresponding patterns in memory.

You should be able to understand this name even if all the other names are unfamiliar. The arrangement of the song is found here. The individual patterns will be played in the sequence indicated by this arrangement.

This is an array of pointers to the individual MOD samples in RAM.

This field gives the length of each sample.
Sound Support For Your Programs

These fields contain the loop start position and loop length for a voice.

This field gives the currently instrument for each voice.

This is the pattern count of the MOD file. This is not the song length, but rather the number of patterns defined.

This tells how many time the Mixing procedure is executed per call.

When this variable is set to TRUE, the Interrupt procedure stops music computation and playback.

Notvol contains the volumes of the respective channels. Values are from 0 to 64.

This variable is needed for synchronizing the old timer interrupt, which, as always, must be called 18 times per second.

Old_TCounter is now needed for counting purposes in the new timer interrupt to determine how long it takes until the call of the old interrupt.

The name of the MOD file is stored here.

The names of the individual instruments within the MOD file are contained here. The field is often used for brief messages as well.

This contains the default volumes of the MOD-file instruments.
Sound Support For Your Programs

Chapter 13

**Songlength : byte;**

This variable holds the length of the song in patterns. The arrangement contains the corresponding number of defined entries.

**vocf : file;**

The unit can also play VOC files. VOC files are accessed using this file handle.

**voch : vocheader;**

This variable is used to access the VOC-file header. We talked about its structure in the Types section.

**vblock : voiceblock;**

This references the active block within a VOC file.

**SaveExitProc : pointer;**

The Unit uses a custom exit procedure, which resets variables that have been changed and releases allocated memory. This variable saves the address of the old ExitProc. Don’t forget to also call this variable.

**Portamento_Up_Voice : array[1..8] of longint;**

The fixed-point value for a possible Portamento Up is stored here.

**Portamento_Do_Voice : array[1..8] of longint;**

The fixed-point value for a possible Portamento Down is stored here.

**Mixing_Proc : array[1..8] of pointer;**

Each voice has its own mixing procedure. This procedure is called according to the active effect. Referencing the procedure through a pointer simplifies its access.

**EmptyVoice : pointer;**

This is a pointer to a procedure that does nothing. When no tone is currently playing for a voice, this dummy procedure is called instead of the Mixing procedure.

**Effect_Voice : array[1..8] of byte;**

A different effect can be active for each voice. The number of the currently active effect for each voice is saved here.
Voice_Length : array[1..8] of word;
The voice lengths are stored here. They’re required for comparing positions in the assembler module.

loop_length_voice : array[1..8] of word;
The lengths of the voice loops are saved in Voice_loop_length. A loop is only recognized if it spans more than ten bytes.

loop_start_voice : array[1..8] of word;
This indicates the position within the sample where looping begins. The end position is obtained by adding the loop length to this position.

position_voice : array[1..8] of longint;
The positions within the voices are stored here as fixed-point values. The upper 16 bits give the position within the sample. The lower 16 bits are the fractional position.

segment_voice : array[1..8] of word;
To access the sample we need the appropriate segment. This word variable is loaded into the segment register used for accessing the sample.

notvol_voice : array[1..8] of word;
This variable is used in the assembler module for accessing the individual voice volumes.

incval_voice : array[1..8] of longint;
These are the increment values to be added to the sample positions for the voices as the sample is read. They are fixed-point values.

This completes the list of main variables used in the unit. It should serve as a reference to help you understand how the unit works.

Controlling the MOD player timing: The timer routines

A MOD player is a “cyclic” program. The routines for voice mixing and output are called at regular intervals. The output routine is controlled through the SB interrupt. Although the same method could theoretically be used to control the mixing, it’s too slow and would lead to problems such as erratic scrolling.

How can we guarantee, however, that processing will be done when the SB interrupt occurs? The answer is to use the timer interrupt. First, we divert the interrupt handler to our own routine. Rather than 18 calls per second, we need a frequency between 200 and 1000 interrupts, so the timer chip must be reprogrammed. To do this, we compute a value to be passed to the chip as follows:
Value := 1193180 DIV calls_per_second

First, we write the value 3h to port 43h to indicate that a new timer value is being written. Then we send first the low byte and then the high byte of the computed value to port 40h. This programs the timer to its new frequency. Now we change the normal timer routine to our own interrupt handler, initialize certain variables that will be used there, and we’re ready to go. This is the procedure:

```pascal
procedure SetTimerOn(Proc : pointer; Freq : word);
var ICounter : word;
OldV : pointer;
begin;
  asm cli end;
  ICounter := 1193180 DIV Freq;
  Port[$43] := $36;
  Port[$40] := Lo(ICounter);
  Port[$40] := Hi(ICounter);

  Getintvec(8,OldV);
  setintvec(OldTimerInt,OldV);
  SetIntVec(8,Proc);
  Old_tCounter := 1;
  SecCounter := 0;
  OldintCounter := 0;
  asm sti end;
end;
```

When we’re finished using the timer, it needs to be reset to its original frequency. First, we write the value 36h to port 43h to inform the timer that it is getting a new frequency. Then we write the value 0 to port 40h twice. The default frequency is then restored. Finally, we redirect the timer interrupt handler back to its original routine.

```pascal
procedure SetTimerOff;
var OldV : pointer;
begin;
  asm cli end;
  Port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  GetIntVec(OldTimerInt,OldV);
  SetIntVec(8,OldV);
  asm sti end;
end;
```

Now that we’ve reprogrammed the timer, let’s proceed to the actual work that our interrupt handler needs to do. First, we must take care of the old timer interrupt. It must still be called 18.2 times per second. This means that if our new interrupt frequency is 1050, the old timer routine must be called every 58th interrupt. We check for this value in OldintCounter. As an extra feature, we also measure the run time for the song here. Finally, the most important procedure of the interrupt is called: The procedure that performs the calculations for music output. The retrace check is necessary because exact synchronization is not possible when these calculations are taking place during the wait.

```pascal
function Notes_Nr(height : word) : integer;
var nct : byte;
found : boolean;
begin;
  nct := 1;
```

366
Sound routines

You may be familiar with the routines for controlling the Sound Blaster card from the section on Sound Blaster programming. We won’t talk about these routines again but only point out the most important procedures and functions.

**procedure wr_dsp_sb16(v : byte);**

This procedure outputs a byte to the Sound Blaster command port.

**Function SbReadByte : Byte;**

This reads a data byte from the Sound Blaster card.

**Function Reset_Sb16 : boolean;**

This resets the Sound Blaster card. If the reset was successful, TRUE is returned, else FALSE. This function can be used to get the base address of the SB.

**Function Detect_Reg_Sb16 : boolean;**

This function takes over the determination of the base address mentioned above. It sets the variable dsp_adr to this address. If a base address was found, TRUE is returned, else FALSE.

**Procedure Write_Mixer(Reg,Val : Byte);**

This procedure writes the value of Val to the mixer register Reg. It runs only on the SB Pro and subsequent cards.

**Function Read_Mixer(Reg : Byte) : byte;**

This function reads a value from the mixer register Reg.
Chapter 13

Sound Support For Your Programs

Procedure Filter_On;

On SB Pro and later cards, this procedure enables accentuation of the bass range.

Procedure Filter_Mid;

This procedure enables normal sound filtering on the SB Pro and later cards.

Procedure Filter_Off;

This procedure accentuates the treble range.

Procedure Set_Balance(Value : byte);

This procedure sets the output balance on stereo cards. Value can be from 0 to 15, with 0 for full left and 15 for full right.

procedure Set_Volume(Value : byte);

This procedure sets the volume at which the MOD file is played. For the SB Pro and later cards, this occurs using the hardware-level mixer. On the normal SB, volume is controlled at the software level with the outvolume variable.

procedure Reset_Mixer;

Like the DSP, the SB mixer must be reset to its default value. This procedure performs the task by passing the value 0 to the mixer's reset register 0.

function Detect_Mixer_sb16 : boolean;

This function detects the mixer chip. It returns TRUE if a mixer chip was found, else FALSE. The function is important because it lets you determine the type of card installed. A return value of TRUE indicates the SB Pro or later.

procedure SbGetDSPVersion;

The version number of the card is used to distinguish between the SB Pro and the SB 16. If the version is less than 4, the card is an SB Pro, else it is an SB 16.

procedure Set_Timeconst_sb16(tc : byte);

You need this function to set the timer constant on the Sound Blaster. The constant is computed as follows:

\[ tc := 256 - \left( \frac{1,000,000}{\text{Sample-frequency}} \right) \]

The derived constant is transferred using the $40 command. You pass the constant to the procedure.
procedure Exit_Sb16;

Make absolutely certain to call this procedure at the end of the program. It resets the SB interrupt to its initial value and restores the interrupt masking.

procedure Play_SB_SbPro_Sb16(Segm, Offs, dsize : word);

This procedure starts the output of the block addressed by Segm and Offs. Segm is the page number in memory and Offs is the offset. The size of the block is given in dsize. Depending on which card is installed (SB, SB Pro or SB16), control branches to the appropriate procedure. Refer to the section on the Sound Blaster for more information.

procedure dsp_block_sb16(sz : word; bk : pointer; b1, b2 : boolean);

This function is for DMA transfer. It outputs the block of size sz referenced by the pointer bk via DMA. The values of b1 and b2 depend on the type of output desired. If you are playing a MOD file, set b1 = TRUE and b2 = FALSE; for a sample or VOC file, set b1 = FALSE and b2 = TRUE.

Handling MOD files

The unit plays a MOD file as follows: The procedure periodic_on diverts the timer interrupt to the custom routine and reprograms it to 50 calls per second. The new routine increments some variables and calls the procedure calculate_music. This makes sure that another routine is not still processing and then calls the Sound_handler, which checks to see whether the next block must be calculated. If so, the procedure inner_proc performs the calculation. Otherwise a test determines whether the calculated block has been transferred. If it has, a new block can be processed.

First, however, new parameters must be loaded. This is accomplished with procedures nmw_proc and Initialize.mixer. Procedure nmw_proc processes the MOD file line by line and controls any sound effects.

Now that you know the general logic of the unit, it's time for some hands-on work. To make it easier to follow the source code, we'll discuss the procedures in the same order as you'll encounter them in the program.

The first procedure of interest to us is get_pctunel. It determines the interval for the next pitch to be played. First, we search an array for the pitch that has been read from the MOD file. The position within the pitch table then serves as an index to a table of increment factors (intervals).

```pascal
procedure get_pctunel(height : word; Var vk : longint);
{
The procedure determines the places before and after the decimal point
required for frequency manipulation from the passed pitch
(as it is in the MOD file).
}
var nct : byte;
```
found : boolean;
begin;
nct := 1;
found := FALSE;
while (nct <= 70) and not found do { until found or last } begin; { value in table }
  if height > Modoctave[nct] then
    found := TRUE;
  inc(nct);
end;
if found then begin;
  vk := Incfacts[nct-tpw+12];
end else begin;
  vk := 0; { get values from table. }
end;
end;

Function Notes_Nr works similar to get_pctunel. It references the pitch table to obtain information needed for effects.

function Notes_Nr(height : word) : integer;
var nct : byte;
  found : boolean;
begin;
nct := 1;
found := FALSE;
while (nct <= 70) and not found do { until found or last } begin; { value in table }
  if height > Modoctave[nct] then
    found := TRUE;
  inc(nct);
end;
if found then begin;
  Notes_nr := nct-1;
end else begin;
  Notes_nr := -1;
end;
end;

Now we come to two very important procedures called inner_loop_4 and inner_loop_4_stereo. They do the actual mixing of the MOD file voices. The size to be calculated is determined first. This is taken from blocksize because we always do one playback block at a time. Because we’re working with double buffering, we next determine the current target buffer.

Next the mixer buffer is cleared of any content remaining from the previous mixing process. Then the assembler mixing procedure is called for all the voices. This adds the sample data for each voice to the mixer buffer. After placing the sample data into the mixer buffer in this manner, we must still transfer it to the output buffer. First, however, we divide its value by the number of voices, because the SB can output only 8-bit data. This step can be skipped on the SB 16, but true 16-bit DMA transfer of the data is then required. Just before the procedure writes to the output buffer, the current outvolume is computed. This can be repeatedly lowered to produce a fading-out effect.

The following is the source code for this procedure:
procedure inside_loop_4;
{
  The actual mixing of data takes place here. The buffer is filled with the calculated data. This is the MONO version of the routine.
}
begin;
  calc_size := blocksize;
  if Proc then
    destination := pt(buffer1)
  else
    destination := pt(buffer2);
  fillchar(mixed_data^,8000,128);
  asm
    mov cx,1
    @Voices_loop:
    mov mixed_posi,0
    mov current_Voice,cx
    mov si,cx
    dec si
    shl si,2
    call dword ptr Mixingprocs[si]
    inc cx
    cmp cx,Voices
    jbe @Voices_loop
    mov mixed_posi,0
    mov cx,calc_size
    @Mixed_2_blk:
    les di,mixed_data
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfactor
    shr ax,cl
    pop cx
    add mixed_posi,2
    mov bx,destination.sgm  
      { write byte to destination }
    mov es,bx
    mov bx,destination.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc destination.ofs
    loop @mixed_2_blk
  end;
end;

Procedure inside_loop_4_stereo works almost the same as inside_loop_4. It needs two mixer buffers, however: One for the left channel and one for the right. The existing buffer mixed_data is used for the left channel and a new one called mixed_data_st is used for the right channel. The mixing of voices occurs as usual. The allocation of each voice to the appropriate buffer is controlled by nmw_proc. When transferring the data from the mixer buffer to the output buffer, remember the SB requires alternating values for the left and then the right channel. The values must be taken alternately from the two buffers.
procedure inside_loop_4_stereo;
{
  This is where the actual mixing of the data takes place. The buffer
  is filled with the calculated data. This is the MONO
  version of the routine.
} begin;
calc_size := blocksize;
if Proc then
  destination := pt(buffer1)
else
  destination := pt(buffer2);
fillchar(mixed_data',#000,128);
fillchar(mixed_data_st',#000,128);
asm
  mov cx,1
  @Voices_loop:
    mov mixed_posi,0
    mov current_Voice,cx
    mov si,cx
dec si
    shl si,2
call dword ptr Mixingprocs[si]
  inc cx
cmp cx,Voices
  jbe @Voices_loop
  mov mixed_posi,0
  mov cx,calc_size
  @Mixed_2_blk:
    les di,mixed_data
    add di,mixed_posi
    mov ax,es:[di]
push cx
    mov cx,shiftfactor_stereo
    shr ax,cl
    pop cx
    mov bx,destination.sgm  { write byte to destination   }
    mov es,bx
    mov bx,destination.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc destination.ofs
    les di,mixed_data_st
    add di,mixed_posi
    mov ax,es:[di]
push cx
    mov cx,shiftfactor_stereo
    shr ax,cl
    pop cx
    add mixed_posi,2
    mov bx,destination.sgm  { write byte to destination   }
    mov es,bx
    mov bx,destination.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
Mixing itself occurs in the unit’s assembler module. It’s necessary to swap out/relocate the routines, since 386 code for fixed point values is required. The mixing procedure works as follows: First, the number of the current voice is loaded to the SI register. It serves as an index to the various voice fields. Next the variable calc_size is loaded to the CX register. It gives the amount of data to be calculated.

The loop that loads the data always checks to see if the end of the voice has been reached. If not, loading the sample bytes can proceed directly. Otherwise we must check to see if the sample is to be “looped”. If this is the case, we set the position back to the start of the voice. Otherwise we are at the end, and the mixing procedure is redirected to the dummy procedure Empty_voice, which does nothing except return. The volume for the current voice is also set to 0, and the procedure jumps to the end of the loop.

When we have positioned the sample at a new byte to be processed, the data from the sample is loaded. The sample segment is moved to ES and the high word of the position within the voice is moved to BX. We load the high word because a position variable is a fixed-point variable, whose low word represents only the fractional portion of the number.

Now the sample byte can be loaded into AL via es:[bx]. Bit 7 of the byte must be inverted, since the SB outputs unsigned data (0 to 255), as opposed to the signed data (-127 to 128) of the Amiga and the Gravis UltraSound. Next we adjust the volume of the voice. This is done by multiplying the byte in AL by the appropriate value found in Voice_notvol, followed by a SHR 6 instruction to divide the result by 64.

With the sample byte therefore adjusted, we can move it to the mixer buffer. First, we load the mixer buffer address to es:[di]. Then we add the relative position within the mixer buffer, found in mixed_posi. This variable is always set to 0 before the procedure is called. Now the calculated byte value can be added to the mixer buffer (a mov would overlay the existing data).

All that remains is to increment the pointers. We increment mixed_posi by 2, and the sample position by the fixed-point value in Voice_incval.

```pascal
inc destination.ofs
loop @mixed_2_blk
end;
end;
```

The following procedure is part of the MOD386.PAS file on the companion CD-ROM.
Chapter 13
Sound Support For Your Programs

```
jae @voice_is_looped

;{Voice is at the end and not looped => out}
mov eax,emptyvoice
mov mixingprocs[si],eax
mov notvol_voice[si],0
jmp @end_voice_normal

; {Parameter for voice1 at start of loop }
@voice_is_looped:
    mov bx,w loop_start_voice[si]
    mov word ptr position_voice[si + 2],bx

; {Load byte from sample of voice1 }
@voice_not_at_end:
    mov bx,w segment_voice[si]
    mov es,bx
    mov bx,word ptr position_voice[si + 2]
    mov al,es:[bx]
    sub al,128
    mul b notvol_voice[si]
    shr ax,6

@voice_output:
    les di,mixed_data
    add di,mixed_posi
    add es:[di],ax
    add mixed_posi,2

; {Increment pointer}
    mov ebx,Incval_voice[si]
    add dword ptr position_voice[si],ebx

loop @load_loop

@end_voice_normal:
    popa
    ret
voice_normal endp
```

The procedure responsible for the mixing is calculate_music. It checks to see whether another procedure is running. If this is the case, variable mycli has the value 1, and we jump to procedure end. Also, if music_out is does not equal 0 (<> 0), we jump to the end. Otherwise, Sound_handler is called to perform the actual mixing.

This procedure checks to see whether the next block has already been calculated (Loop_pos is in this case greater than Speed). If not, it does the calculation. These calculations are complete when another check is made to see whether the SB interrupt procedure has set a flag indicating successful output of the old block. Only at this point are procedures nmw Proc and Initialize_mixer called. They advance the MOD one line. If outfading is TRUE, the output volume is decremented. In any case, Note_attack (used for equalizing and other effects) is decremented.
Before mixing can proceed, values must be provided for the necessary variables. In addition to \textit{nmw_proc}, procedure \textit{mix_start_4} provides the values. Here the pitch interval is determined for each voice of the note to be played. Looping parameters are also set. If the program calls for a portamento, the pitch interval is raised or lowered accordingly.

```
procedure mix_start_4;
var rdiff : real;
   dummy : byte;
   var li : integer;
begin;
   for li := 1 to voices do begin;
     if note[li] <> 0 then begin;
       pitch_voice[li] := (Rm_Song[mli,li,1] and $0F)*256+Rm_Song[mli,li,2];
       get_pctune1(pitch_voice[li],Incval_voice[li]);
       end;
     ls[li] := loop_s[In_st[li]];
     ll[li] := loop_l[In_st[li]];
     Loop_Length_voice[li] := ll[li];
   end;
```
Now we come to the procedure that advances our position within the MOD file. It controls line-by-line processing of the data and initializes the variables needed by the mixing procedure. Any effects encountered are handled by `effect_handling`.

As a parameter to this procedure, we pass the number of the voice to be processed. The procedure determines whether there is an effect for the voice. If so, it gets the effect parameters, checks the type of effect and processes it appropriately.

Effect 01 is portamento up and Effect 02 is portamento down. We start both by determining the speed of the portamento. Then we get the start increment and end increment for the current pitch. To determine the incremental (or decremental) value for the voice, we divide by the number of ticks per line (playspeed).

Effect 09 controls the sample offset. The position is calculated from the following formula:

\[
\text{Position} := \text{Effect\_value} \times 256;
\]

Effect 11 allows jumping within the arrangement. The procedure terminates the current pattern and reads the new target number.

Effect 12 is one of the most important and frequently used effects. It sets the channel volumes. The value of the operand is assigned to `Voice\_notvol`.

Effect 13 terminates the current pattern and resumes playback with the next one. This is another frequently used effect. Here we set the current line to the end of the MOD file, and the next pattern loads automatically with the subsequent execution.

Effect 14 includes several sub-effects. The upper four bits of the operand specify the sub-effect, and the lower four bits serve as the actual operand. This player only handles Sub-effect 12, which is the notecut. To achieve this, we simply set all playback parameters to 0. We'll talk later about programming the others.

Effect 15, which sets the speed. If the value of the operand is less than or equal to 15, it indicates ticks per line (Playspeed). Here the value can be assigned directly to the associated variable. If the operand is greater than 15, it refers to beats per minute (BPM). This affects the size of the output sample block, which is computed as

\[
\text{Size} := \text{Speed} \times (\text{SamplingFrequency} \div (\text{BPM} \times 4));
\]

The value for the `Sound\_loops` must be reset accordingly. Make certain this value is within reasonable limits. The player will crash if the new size exceeds the reserved memory. Sometimes MOD files contain invalid data here to make illegal copying or pirating more difficult.
procedure effect_handling(li : integer);
    var idx : word;
        Portamento_Speed : word;
        Startnote,
        endnote : word;
        startinc,
        endinc : longint;
    begin;
        if Rm_Song[mli,li,3] and $0F <= 15 then begin;
            Eff[li] := 0;
            case (Rm_Song[mli,li,3] and $0F) of
                01 : begin;
                    effect_Voice[li] := 1;
                    Portamento_Speed := Rm_Song[mli,li,4];
                    Startnote := Notes_nr(pitch_Voice[li]);
                    Endnote   := Startnote+Portamento_Speed;
                    get_pctunel(Modoctave[Startnote],Startinc);
                    get_pctunel(Modoctave[Endnote],Endinc);
                    Portamento_up_Voice[li] := round((Endinc - Startinc) / playspeed);
                end;
                02 : begin;
                    effect_Voice[li] := 2;
                    Portamento_Speed := Rm_Song[mli,li,4];
                    Startnote := Notes_nr(pitch_Voice[li]);
                    Endnote   := Startnote-Portamento_Speed;
                    get_pctunel(Modoctave[Startnote],Startinc);
                    get_pctunel(Modoctave[Endnote],Endinc);
                    Portamento_do_Voice[li] := round((Endinc - Startinc) / playspeed);
                end;
                9 : begin;  { Sample offset }
                    Position_Voice[li] := longint(Rm_Song[mli,li,4]) shl 24;
                end;
                13 : begin;
                    mli := 64;
                end;
                11 : begin;
                    mli := 64;
                    mlj := Rm_Song[mli,li,4];
                end;
                12 : begin;
                    Notvol_Voice[li] := Notevolumes(li);
                end;
                14 : begin;
                    case (Rm_Song[mli,li,4] shr 4) of
                        12 : begin;
                            inl[li] := 0;
                            Notvol_Voice[li] := 0;
                            inp[li] := 0;
                            Pnk[li] := 0;
                        end;
                    end;
                end;
                15 : begin;
                    idx := Rm_Song[mli,li,4];
                    if idx <= 5 then begin;
                        Playspeed := idx;
                        Speed := Playspeed*105 div 10;
                        blocksize := Speed * Sound_Loops;
                        end else begin;
                            bpm := idx;
                            mod_SetLoop(Sampling_Frequency div (BPM * 4));
Sound Support For Your Programs

Chapter 13

Finally, we come to procedure `nmw_all_4`, which is referenced through pointer `nmw_proc`. It begins with a definition for the stereo procedures. Two voices are assigned to the left channel and two voices to the right. You can easily change these assignments by modifying the mixing procedure according to your own needs.

First, the procedure increments the current line. If this exceeds 64, the pattern is ended, and the variable must be reset to 1. The arrangement position is incremented at the same time. If this exceeds the song length, the MOD file is ended, and playing starts over. Otherwise, we determine the number of the pattern to be played. This pattern is then loaded to the pattern hold area, which the other procedures reference directly.

Once the MOD is correctly positioned and the current pattern loaded, we can start processing the individual voices. The flag for the current effect is initialized to 0 for each voice since effects always refer to individual lines within the pattern. Next, we receive the note from the MOD and insert it in the variable called note. A non-zero value indicates a note to be played, so we continue to process it. The mixing procedure to be used for the voice is obtained from the `Stereoprocs` constant if in stereo mode. Otherwise, we always use the assembler procedure `Voice_normal`.

The variable `Note_attack` is set to 500, because a new note is being struck. This variable can be used to achieve a simple equalizer.

The variable `Vc_In` (for voice instrument) is used to access instrument-related values. It’s assigned the value of `Note` and then used to pass the sample pointer to `inst`. Next the length of the sample is determined and saved in `Voice_length`. The position within the sample is set to 0, the start position. The default sample volume is placed in `Voice_notvol`. Finally, the sample segment for the voice is placed in `Voice_segment`.

This concludes the handling of the voices, however, any effects must still be considered. This task is accomplished by the procedure `effect_handling` which we talked about earlier.

```plaintext
procedure nmw_all_4;
const stereoprocs : array[1..8] of pointer =
(@Voice_normal,@Voice_normal,@Voice_normal_st,@Voice_normal_st,
 @Voice_normal,@Voice_normal,@Voice_normal_st,@Voice_normal_st);
var idx : byte;
li : integer;
begin;
  inc(mli);
  if mli > 64 then mli := 1;
  if mli = 1 then begin;
    inc(mlj);
    if mlj > Songlength then begin;
      if nloop then begin;
        mlj := 1;
        move(rm[song[mlj]] ^,Rm_Song,2048);
      end else begin;
```
We can't do anything with our MOD processing, of course, until we get the MOD data into memory. So, first, we must take care of loading the MOD file. A load procedure called from the main program makes various checks before calling \texttt{init\_Song}. This is the procedure that actually loads the MOD file into memory. The global variable \texttt{Mod\_Name} is used as an index for this process.

We start by setting the variables for the instruments to be played, as well as the complete arrangement (\texttt{Song}), to 0. Then the MOD file is opened. If an error occurs here, the routine aborts and returns FALSE. The routine allows for loading a MOD file that is part of a larger file. This is of interest especially in demos.

To use the feature, you must first supply the size of the MOD file in bytes, in the variable \texttt{modfilesize}. If this is 0, the size is assumed to be that of the file opened. The variable \texttt{msp} (for MOD start position) gives the start position within the MOD file. The default value for this variable is 0.

Once the file has been successfully opened, we must make certain it is in fact a MOD file that we can process. Our player can play four-voice and eight-voice MOD files. This means we must check the header for one of the MOD identifiers 'M.K.', 'FLT4' or '8CHN'. For 31-voice files, the identifier is at position 1080. If we don't find one of the expected values here, the file could be a 15-voice MOD file, a MOD file without an identifier, or some other format entirely.

After determining how many voices are in the MOD file (31 or 15), we read them in one at a time. The loop to accomplish this starts at the corresponding position within the MOD file and reads the two bytes that indicate the size of the instrument sample. This field represents a word count in Amiga word format, which must be converted to a byte count in PC format. To do this, the \texttt{Swap} function reverses the byte order and multiplies by 2.
Now the internal variable `Inststart` is decremented by the instrument size. It is initially set to the file length and allows us to determine the number of patterns. As you may recall, the pattern count must be derived by calculation because it is not included anywhere in the file. After the total file length is reduced by the size of the header and all the instrument data, what remains is the size of the pattern data. Dividing this by either 1024 or 2048 bytes (the size of one pattern) gives us the number of patterns.

After reading the sample length for the current instrument, we get its default volume. This is a one-byte field that does not require conversion. The next two values to be read, however, are two-byte fields in Amiga word format that must be converted as described above. They represent the instrument’s loop length and loop start position within the sample data.

Once the base parameters for the instruments have been obtained, we check to see whether the MOD file has four or eight voices. The test is made by comparing the header `Ident` to the value '8CHN'. A match indicates an eight-voice file, so `Voices` is set to 8, and `Pattsize` is set to 2048. Otherwise, these variables are assigned the values 4 and 1024. The pointer for the mixing procedure is then set to reference either the mono or stereo version of the routine, depending on the output mode desired.

Next the samples are loaded. After reserving the amount of memory needed for all instruments, we read the sample data from the disk into memory.

Finally, we come to loading the patterns. We determine the number of patterns in the file, then enter a loop to load them one at a time. Note that an 8-voice pattern takes twice the space of a 4-voice pattern. We end this procedure by loading the 128-byte arrangement (found at offset position 952) and the song length (found at offset position 950).

Now we have managed to get the MOD data into memory. So everything is absolutely complete, we should set the song and instrument names, too. Remember, a user-friendly program will display song and instrument names simply for informational purposes even if they’re not needed for playing. The song name is at the very beginning of the file. Each instrument name is at the beginning of that instrument’s data. The names are stored as 0-terminated strings, which must be converted to Pascal format. The function `ConvertString` does this. If the process is successful, the function returns the value TRUE. The variables for the current line and pattern position are both initialized at the beginning.

```pascal
FUNCTION ConvertString(Source : Pointer; Size : BYTE):String;
VAR
  WorkStr : String;
BEGIN
  Move(Source^,WorkStr[1],Size);
  WorkStr[0] := CHR(Size);
  ConvertString := WorkStr;
END;
function init_Song : boolean;
const ID1 : string = 'FLT4';
ID2 : string = 'M.K.';
ID3 : string = '8CHN';
var rmod : file;
  ssz : word;  { size of a sample }
  inststart : longint;  { position in file where}
                      { sample data begin }
  filsz : longint;  { size of the MOD file }
  Mdt : array[1..4] of char;  { for Mode type - detection }
```

Chapter 13
Sound Support For Your Programs
helpp : ^byte;
strptr : pointer;
IDch : array[1..4] of char;
IDstr : string;
instance : byte;
idx : integer;

begin;
In_St[1] := 0;
In_St[2] := 0;
In_St[3] := 0;
In_St[4] := 0;
In_St[5] := 0;
In_St[6] := 0;
In_St[7] := 0;
In_St[8] := 0;
for mlj := 0 to 128 do
  Song[mlj] := 0;
{$I-}
assign(rmod,Mod_Name);
reset(rmod,1);
{$I+}
if IOresult <> 0 then begin;
  init_song := FALSE;
  exit;
end;
if moddatsize <> 0 then filsz := moddatsize else
  filsz := filesize(rmod);
inststart := filsz;
seek(rmod,1080);
blockread(rmod,IDch,4);
IDstr := IDch;
if (IDstr <> ID1) and (IDstr <> ID2)
  and (IDstr <> ID3) then begin;
  instance := 15;
end else begin;
  instance := 31;
end;
if instance = 31 then begin;           { 31 Voices detected from ID     }
  for mlj := 1 to 31 do begin;
    idx := mlj;
    seek(rmod,msp+42+(idx-1)*30);
    blockread(rmod,ssz,2);
    ssz := swap(ssz) * 2;
    if ssz <> 0 then inststart := inststart - ssz;
    Sam_l[idx] := ssz;
    seek(rmod,msp+45+(idx-1)*30);
    blockread(rmod,inst_vol[idx],1);
    blockread(rmod,loop_s[idx],2);
    blockread(rmod,loop_l[idx],2);
    loop_s[idx] := swap(loop_s[idx]) * 2;
    loop_l[idx] := swap(loop_l[idx]) * 2;
  end;
  seek(rmod,msp+1080);
  blockread(rmod,Mdt,4);
if pos('8CHN',Mdt) <> 0 then begin;
Pattsize    := 2048;
Voices        := 8;
shiftfactor        := 3;
shiftfactor_stereo := 3;
end else begin;
Chapter 13

Sound Support For Your Programs

{ 4-channel MOD-File }

Pattsize := 1024;
Voices := 4;
shiftfactor := 2;
shiftfactor_stereo := 2;
end;
Mix_Proc := @mix_start_4;
nmw_Proc := @nmw_all_4;
if stereo then
  inside_proc := @inside_loop_4_stereo
else
  inside_proc := @inside_loop_4
end;

seek(rmod,msp+inststart);
for mlj := 1 to 31 do begin;
  idx := mlj;
  getmem(Samp[idx],Sam_l[idx]);
  blockread(rmod,Samp[idx]^,sam_l[idx]);
end;

filsz := inststart - 1083;
pat_num := filsiz div Pattsize;
for mlj := 0 to pat_num-1 do begin;
  getmem(rm[mlj],2048);
  fillchar(rm[mlj]^,2048,0);
  seek(rmod,msp+1084+mlj*Pattsize);
  helpp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
  for mli := 0 to 63 do begin;
    helpp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
    blockread(rmod,helpp^,Pattsize div 64);
  end;
end;

seek(rmod,msp+952);
blockread(rmod, Song,128);

getmem(strptr,25);
for i := 0 to 30 do begin;
  seek(rmod,msp+20+i*30);
  blockread(rmod,strptr^,22);
  instnames[i+1] := convertstring(strptr,22);
end;

seek(rmod,msp+950); { von 470 }
blockread(rmod, Songlength,1);
end else begin;
for mlj := 1 to 15 do begin;
  seek(rmod,msp+42+(mlj-1)*30);
  blockread(rmod,ssz,2);
  sssz := swap(ssz) * 2;
  if sssz <> 0 then inststart := inststart - sssz;
  Sam_l[mlj] := sssz;
  seek(rmod,msp+45+(mlj-1)*30);
  blockread(rmod,inst_vol[mlj],1);
  blockread(rmod,loop_s[mlj],2);
  blockread(rmod,loop_l[mlj],2);
  loop_s[mlj] := swap(loop_s[mlj])*2;
  loop_l[mlj] := swap(loop_l[mlj])*2;
end;
for mlj := 16 to 31 do begin;
  Sam_l[mlj] := 0;
  loop_s[mlj] := 0;
  loop_l[mlj] := 0;
end;
if pos('8CHN',Mdt) <> 0 then begin;
  Pattsize    := 2048;
  Voices        := 8;
  shiftfactor        := 3;
  shiftfactor_stereo := 3;
end else begin;
{ 4-channel MOD-File }
  Pattsize    := 1024;
  Voices        := 4;
  shiftfactor        := 2;
  shiftfactor_stereo := 2;
end;
Mix_Proc := @mix_start_4;
nmw_Proc       := @nmw_all_4;
if stereo then
  inside_proc     := @inside_loop_4_stereo
else
  inside_proc     := @inside_loop_4;
seek(rmod,msp+inststart);
for mlj := 1 to 15 do begin;
  getmem(Samp[mlj],Sam_l[mlj]);
  blockread(rmod,Samp[mlj]^,sam_l[mlj]);
end;
for mlj := 0 to pat_num-1 do begin;
  getmem(rm[mlj],2048);
  fillchar(rm[mlj]^,2048,0);
  seek(rmod,msp+1084+mlj*Pattsize);
  helpp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
  for mli := 0 to 63 do begin;
    helpp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
    blockread(rmod,helpp^,Pattsize div 64);
  end;
end;
seek(rmod,msp+472);
blockread(rmod,Song,128);
for i := 0 to 14 do begin;
  seek(rmod,msp+20+i*30);
  blockread(rmod,strtpr^,22);
  instnames[i+1] := convertstring(strtpr,22);
end;
for i := 15 to 30 do begin;
  instnames[i+1] := '';
end;
seek(rmod,msp);
blockread(rmod,strtpr^,20);
songname := convertstring(strtpr,20);
freemem(strtpr,25);
seek(rmod,msp+470);
Several variables must also be initialized when the MOD file is loaded. This is done by using function\texttt{load\_modfile}. It takes the name of the file as the first parameter (including the complete path, if needed). For the second and third parameters, we simply code AUTO. The last parameter is the output frequency. You’ll get best results with a value of 22, which plays the MOD file at a frequency of 22 KHz. If your computer is a 16 MHz 386SX, however, you’ll have to be satisfied with 16 KHz playback.

The function first validates the file name. The error message -1 appears here if a problem is detected. Then output must be mono (\texttt{force\_mono = TRUE}), Stereo is set to FALSE. If the installed card is the SBPro, the correct output mode must be enabled. After stereo handling is complete, procedure\texttt{init\_data} takes care of initializing some data fields for us.

The MOD file is loaded into memory using\texttt{init\_song}. Next, the desired output frequency and the default output speed are set. Then the proper sampling rate is sent to the Sound Blaster and the speakers are turned on. For a Sound Blaster with stereo capability, the balance is set to mid-range and the output volume is set to the value stored in\texttt{SoundMastervolume}. The procedure ends with a function result of 0.

```pascal
function load_modfile(modname : string;ispeed,iloop : integer;freq : byte) :
var df : file;
sterreq : byte;
fsz : longint;
begin;
Playing_MOD := TRUE;
Playing_VOC := FALSE;
outfading := FALSE;
outvolume := 63;
Mod_Name := modname;
{$I-}
assign(df,Mod_name);
reset(df,1);
{$I+}
if IOResult <> 0 then begin
{$I-}
close(df);
load_modfile := -1;                  { File not found !     }
exit;
end;
{$I-}
fsz := filesize(df);
close(df);
music_played := TRUE;
music_Off := FALSE;
Mod_At_End := FALSE;
if ispeed <> AUTO then Speed3 := ispeed;
if iloop <> AUTO then Loop3  := iloop;
```
if force_mono then stereo := FALSE;
if force_sb then begin;
  if Sb16Detected then stereo := FALSE;
  Sb16Detected := FALSE;
end;

if SbPordetected then begin;
  if stereo then begin;
    sterreg := Read_Mixer($0e);
    write_Mixer($0e, sterreg OR 2);
  end else begin;
    sterreg := Read_Mixer($0e);
    write_Mixer($0e, sterreg AND $FD);
  end;
end;
init_data;
if init_song then begin;
  phase_1 := FALSE;
  phase_2 := TRUE;
  mycli := 0;
  mod_Samplefreq(freq);
  Playspeed := Playspeed*105 div 10;
  bpm := 125;
  mod_SetLoop(Sampling_Frequency div (BPM * 4));
  blocksize := Speed * Sound_Loops;
  if blocksize < 100 then blocksize := 100;
  if blocksize > 4000 then blocksize := 4000;
  asm call [nmw_proc] end;
  set_timeconst_sb16(Sampling_Rate);
  Initialize_Mix;
  Runsec := 0;
  Runmin := 0;
  wr_dsp_sb16($D1);
  if sb16detected or sbpordetected then begin;
    filter_Mid;
    Set_Balance(Balance);
    Set_Volume(Mastervolume);
  end;
  Load_Modfile := 0;
end else begin;
  Load_Modfile := -3;                { Error loading song }
end;
end;

Now we can actually play the MOD file. Procedure periodic_on is called. It initializes the last_output flag (signaling the end of play) to FALSE, and begins playing the file immediately. The data block calculated in the procedure load is output through DMA. Inverting the Boolean variable bsw then switches to the next block of the double buffer. After this, the values for the next line must be loaded, which is accomplished in turn by the procedures nmw_proc and Initialize_Mix.

The final step is most important: The timer interrupt must be diverted to a custom procedure and its frequency reset. For this task, procedure SetTimerOn takes the parameters NewTimer and timer_per_second (whose value is 50). A simple call is all you need to start the player.
Chapter 13

Sound Support For Your Programs

Procedure periodic_on;
Begin
    outvolume := 64;
    last_output := FALSE;
    { for Loop_pos := 1 to Speed do begin; }
        asm call [inside_proc] end;
    { end; }
    dsp_block_sb16(blocksize,blocksize,buffer1,TRUE,FALSE);
    Proc := not Proc;
    Loop_pos := 0;
    asm call [nmw_proc] end;
    Initialize_Mix;
    init_sbperiod(@periodic_off);
    music_played := TRUE;
End;

Congratulations, you’ve started playing the MOD file. You’ll need eventually to stop the MOD file. This is as easy as starting the MOD file. You simply call procedure periodic_off. This sets last_output to TRUE, so no more blocks are sent out by the interrupt procedure. Then the timer interrupt is restored to its normal value by SetTimerOff.

Procedure periodic_off;
Begin
    last_output := true;
    SetTimerOff;
End;

Now that you know how to play MOD files with our player unit, let’s look at playing VOC files. You need only two procedures to play VOC files: Init_Voc to start the output and Voc_Done to end it. We’ll talk about Init_Voc first.

This procedure takes the name of the VOC file to be played as a parameter (including the complete path, if needed). It sets the flags Playing_MOD to FALSE and Playing_VOC to TRUE. This tells the interrupt procedure that a VOC file, not a MOD file, is being played. By default, stereo output is disabled. For a stereo VOC file, it must be re-enabled. Now the program opens the VOC file and reads the header. To be considered valid here, the header identifier must be 'Creative Voice File'+#1A.

Next, the first voice block is read in. Its type identifier is at position 2 of the data fields. Three voice block types are important to us here. The first is type 1, the simplest block type. There the sampling rate can be obtained directly at position 6 and sent to the DSP.

Working with block type 8 is not quite so straightforward. Because Creative Labs expanded the internal sampling constant from 8 to 16 bits, we must calculate the true sampling frequency. We read the new sampling rate, then use it to calculate the actual frequency as follows:

\[ \text{Frequency} := 256000000 \div (65536 - SR) \]

Next we determine whether the sample uses stereo mode. If so, stereo output must be re-enabled and the sampling frequency divided by 2. Then we apply the following formula:

\[ \text{SB_SR} := 256 - (10000000 \div \text{Sampling-Frequency}) \]
to obtain the frequency to be sent to the Sound Blaster.

The last voice block type we'll talk about is type 9. This type was introduced with the SB 16. This type finally contains the true sampling frequency for the file (located in bytes 6 and 7). For a stereo sample, byte 11 has the value 2. In this case, stereo output must be re-enabled. Next we check for the SBpro. The sampling frequency is doubled for this card (and only this card). Otherwise, the sampling frequency is not changed. Now the Sound Blaster frequency is again computed as follows:

\[ \text{SB}_{\text{SR}} := 256 - (10000000 \div \text{Sampling-Frequency}) \]

For a MONO block, the frequency can be computed directly using this formula.

After finally extracting the proper sampling rate from the VOC file, we still must pass it to the Sound Blaster. We'll use the procedure `set_timeconst_sb16` for this; we simply pass the frequency to it as a parameter.

Now we read the first block of sample data. The standard block size is 2500 bytes. The first block is read into `Buffer1`. If the remaining data exceeds the buffer size, the second block is read into `Buffer2`.

Next we must turn on the speaker. This has to be done at the beginning of every VOC file or nothing will play. For sound cards other than the SB16, the mixer chip for mono and stereo output must also be reprogrammed. Then output of the blocks can finally begin through the procedure `dsp_block_sb16`. The SB interrupt takes care of everything else.

To stop VOC file output, call procedure `Voc_Done`. It sets the variable `lastone` to TRUE, telling the interrupt procedure there is nothing more to play. Now the VOC file can be closed and the Sound Blaster card reset. The reset ensures that other Sound Blaster routines will find the card in its original status.

```pascal
procedure Init_Voc(filename : string);
const VOCid : string = 'Creative Voice File'+#$1A;
var ch : char;
  IDstr : string;
  ct : byte;
  h : byte;
  error : integer;
  srlo,srhi : byte;
  SR : word;
  Samplinsz : word;
  stereoreg : byte;
begin;
  Playing_MOD := FALSE;
  Playing_VOC := TRUE;
  VOC_READY := FALSE;
  vocs stereo := stereo;
  stereo := FALSE;
  assign(vocf,filename);
  reset(vocf,1);
  if filesize(vocf) < 5000 then begin;
    VOC_READY := TRUE;
    exit;
end;
  blockread(vocf,voch,$19);
  IDstr := voch.IDstr;
  if IDstr <> VOCid then begin;
```

The procedure `Init_Voc` is part of the MOD_SB.PAS file on the companion CD-ROM.
VOC_READY := TRUE;
exit;

Blockread(vocf,inread,20);
vblock.ID := inread[2];

if vblock.ID = 1 then begin;
vblock.SR := inread[6];
end;

if vblock.ID = 8 then begin;
SR := inread[6]+(inread[7]*256);
Samplinsz := 256000000 div (65536 - SR);
if inread[9] = 1 then begin; {stereo}
  if sb16detected then samplinsz := samplinsz shr 1;
  stereo := TRUE;
end;
vblock.SR := 256 - longint(1000000 DIV samplinsz);
end;

if vblock.ID = 9 then begin;
Samplinsz := inread[6]+(inread[7]*256);
if inread[11] = 2 then begin; {stereo}
  stereo := TRUE;
  if sbprodetected then samplinsz := samplinsz * 2;
  vblock.SR := 256 - longint(1000000 DIV (samplinsz));
end else begin;
  vblock.SR := 256 - longint(1000000 DIV samplinsz);
end;
end;

if vblock.SR < 130 then vblock.SR := 166;
set_timeconst_sb16(vblock.SR);
blocksz := filesize(vocf) - 31;
if blocksz > 2500 then blocksz := 2500;
blockread(vocf,vocb1^,blocksz);
ch := #0;
fsz := filesize(vocf) - 32;
fsz := fsz - blocksz;
Block_active := 1;

if fsz > 1 then begin;
  blockread(vocf,vocb2^,blocksz);
  fsz := fsz - blocksz;
end;
wr_dsp_sb16($D1);
lastone := FALSE;
if not sb16Detected then begin;
  if Stereo then begin;
    stereoreg := Read_Mixer($0E);
    stereoreg := stereoreg OR 2;
    Write_Mixer($0E,stereoreg);
  end else begin;
    stereoreg := Read_Mixer($0E);
    stereoreg := stereoreg AND $FD;
    Write_Mixer($0E,stereoreg);
  end;
end;
pause_voc := FALSE;
dsp_block_sb16(blocksz,blocksz,vocb1,FALSE,TRUE);
Now we arrive at a extremely important procedure: The SB interrupt procedure. Control branches to this procedure every time the Sound Blaster reports that a DMA transfer is finished. The procedure first determines whether this is a Sound Blaster hardware interrupt check, in which case the `IRQDetected` flag is set to TRUE. The detection routine then knows the correct interrupt has been found.

For a normal interrupt, however, we check to see whether the file being played is a MOD file or a VOC file. If it is a MOD file, we first read a byte from the SB data port. This is required to regain access to the Sound Blaster. Next, Boolean variable `dsp_rdy_sb16` is set to TRUE, indicating the entire block has been output. Now we check to see if `last_output` has the value FALSE, meaning that output should continue. If so, output is started for the newly active block using a call to `dsp_block_sb16`. Finally, a flag is set to indicate the timer procedure can advance the MOD file by one line.

The process is a little different if a VOC file is being played. Here also, we start by "reading" the Sound Blaster. Next, however, we check for the end of the VOC file. If there is more data to be played, the active block is output to the Sound Blaster through `dsp_block_sb16`. Then new sample data is loaded from the VOC file into the other buffer and the active and passive indicators are switched. On the other hand, if the end of the file has been reached or play is to be stopped, `dsp_rdy_sb16` is set to TRUE, and the speaker is turned off.

Finally, the Boolean variable `VOC_READY` is set to TRUE. Note that you must write the value 20h to port 20h at the end of the interrupt procedure, or no more interrupts will be executed.

**Tips for programming effects**

Our unit does not implement all the available Protracker effects. About 80% of all MOD files will play with this version. To enhance the unit to handle other effects, or even write an improved player of your own, you'll need to know how the effects are programmed.

Since reliable information in this area is hard to come by, we can only briefly describe the handling of additional effects here. When we mention the X parameter, we mean the upper four bits of the effect operand. The Y parameter is the lower four bits of the operand. XY stands for the entire operand. The effects are programmed as follows:

*Arpeggio*

Arpeggio plays a note using three pitches in rapid succession: The first is the original pitch, then pitch + X and then pitch + Y. You perform voice calculations as normal but activate the three tones successively, one tick at a time.
Sound Support For Your Programs

Chapter 13

Portamento Up/Portamento Down

These portamento effects are also fairly easy to program. For each tick, the interval specified by XY is either added or subtracted. Other calculations are performed normally.

Portamento To Note

Portamento To Note is slightly more involved than Portamento Up/Down. With this effect, the note given in the pattern line is not the note to be played initially, but the target note of a portamento. Based on this note, you must determine whether a Portamento Up or a Portamento Down is required. Then you increase or decrease the interval at each tick by the XY value. XY determines the speed of the portamento. The effect continues until the target note is reached. If this doesn’t happen within the line, it continues into the next line if no other effect is present. If the note field in this line is empty, the last note given is still considered the target. Otherwise, the new note becomes the target.

Vibrato

The vibrato effect is rather complex. It requires three tables representing the three possible wave forms. These forms are a normal sine wave, a rectangular wave and a “ramp down” sawtooth wave. Table entries must range from 255 to -255. One half oscillation (the portion of a wave between two consecutive points of intersection with the Y axis) must encompass 32 entries. The whole wave, then, consists of 64 entries.

You must increment the table index for each tick by the X parameter. The resulting table value is then multiplied by the Y parameter, and this result is divided by 256. The resulting value is added to the increment value for the note. If the XY field is 0, use the values from the previous vibrato. The effect continues until a new note is struck. Be careful not to reset the vibrato when you jump to the next line of the song.

Portamento and Volume Sliding

This combination of two effects can be easily handled as two separate commands. For each tick, you must first apply the appropriate volume reduction to the voice, and then change its pitch as we described earlier.

Vibrato and Volume Sliding

For this effect, proceed as for a simple vibrato, but also adjust the volume as required.

Tremolo

This effect is closely related to the vibrato. The one difference is that here it is the volume of a voice, not its frequency, that is changed. The effect is also called a volume vibrato. In fact, it is often used instead of a normal vibrato. You implement it in much the same way, even using the same tables. Make certain not to exceed the maximum volume.

Volume Sliding

Volume sliding can go either up (using the X parameter to increase the volume) or down (using the Y parameter to decrease the volume). For each tick, change the volume by the amount specified in the parameter. Continue until you reach the end of the line or the end of the volume range.
Position Jump

This effect is easy to handle. You simply jump to the position within the arrangement indicated by XY. To do this, go to the end of the current pattern, and then assign the value of XY to the variable representing the arrangement position.

Set Volume

The meaning of this effect is very simple...just set the current voice volume to the value given in XY.

Pattern Break

Although this effect is also very easy to achieve, not all MOD players handle it correctly (for example, DMP 2.92). First, you must stop playing the current pattern and go to the next one. Everyone usually gets this part right. However, some players ignore the XY value. This parameter is 0 for 90% of all MOD files. However, if a value is specified, it indicates the line within the pattern where playing should resume. In this case, you must jump to the specified line, not to the beginning of the pattern.

Set Speed

This effect sets the MOD playing speed. Normal speed is 6 ticks or 125 beats per minute (BPM). If the XY value is not greater than 0Fh, it refers to the number of ticks. Otherwise, it refers to the number of beats per minute.

In addition to the normal effects we’ve talked about so far, there are several extended effects that should be supported by a first-class player.

Set Filter

This is the easiest effect to program but it is specific to the Amiga and, therefore, cannot be used for our purposes.

Fine Portamento Up/Down

This effect works like a normal portamento, but the pitch is changed only once initially. It remains the same for subsequent ticks.

Glissando

You program a glissando as follows: Generate a table of all possible whole notes. Then program a normal portamento. However, for each pitch value obtained, round to the nearest value in the table.

Set Vibrato Waveform

This effect allows you to choose one of the three waveform tables for the vibrato. We recommend using a pointer for referencing the current table. Then you can implement this effect by simply setting the pointer to the address of the table requested.

Set Loop

Save the current position (line) of the pattern. Looping will go back to this position.
Jump To Loop

Jump To Loop is the second part of the Loop command. You change to the position marked by Set Loop. Repeat this as often as specified in Y.

Set Tremolo Waveform

This is used similar to Set Vibrato Waveform.

Retrigger Note

When the current tick equals the Y parameter, the sample position must be reset to the start position.

Fine Volume Slide Up/Down

Raise or lower the volume of the voice initially, by the value given in Y. Do not apply additional volume changes with subsequent ticks.

Note Cut

Cut off the playing of a voice. The Y parameter gives the number of ticks after which playing should stop. You achieve the effect by a corresponding change to the voice length or position.

Note Delay

The Y parameter gives the number of ticks by which the start of a voice’s output should be delayed.

Pattern Delay

This effect resembles Note Delay. The delay is applied to all voices, however, instead of one.

Invert Loop

Because we have yet to find a MOD file that uses this command or a player that supports it, we can only speculate on this effect. We assume the pattern plays backwards from the current position to that indicated by Set Loop.

The MOD player MOD386

We’ll use an example called MOD386 to show how to use the MOD_SB unit. It plays both MOD and VOC files. If you start the program without parameters, you get a file selection menu where you can select a sound file from the current directory. If you use the -r parameter, you’ll be in Repeat mode. Here you can choose a new sound file when the current one is finished playing. To stop, press Esc.

The program uses the unit Design which contains general functions for windowing and file selection. We won’t discuss it in detail here, but you can find it on the companion CD-ROM.

The various procedures of the program can be summarized as follows:

Procedure Scale_Box;

This procedure draws the player window. It is fairly self-explanatory.
Scale is responsible for outputting runtime information. It shows the volume bars, the current position within the song and the active instruments. It updates them continually. This procedure should also be self-explanatory.

Procedure Play_the_mod (s: String);

This is the most important procedure for us. It plays a MOD file and responds to user input.

We must first reset the Sound Blaster card since we cannot be sure of its current condition. Next, certain parameters needed for the output are initialized. Unless you have a 386SX-16 or slower processor, Samfreq can be set to 22 (KHz). Then load_modfile loads the MOD file into memory. If problems occur, this function returns a non-zero value, and the program displays an error message and terminates. Otherwise, interrupt-controlled playing of the MOD file begins with periodic_on.

Now the screen can be drawn. The loop checks for the Esc key or certain other predefined keys. If one of them is detected, it responds accordingly.

When the Esc key is pressed, outfading is set to TRUE and playing becomes gradually softer. When outvolume falls below 1, no more sound is heard. To this point, Scale continues to update the screen. Then periodic_off sets the timer interrupt back to its old value. Finally, end_mod releases the memory occupied by the MOD file and the Sound Blaster is reset to its original state.

```
procedure Play_the_Mod(s : string);
var h : byte;
  error : integer;
  li : integer;
begin;
  Reset_Sb16;
  mod_SetSpeed(66);
  mod_Samplefreq(Samfreq);
  dsp_rdy_sb16 := true;
  error := load_modfile(s,AUTO,AUTO,Samfreq);
  if error <> 0 then begin;
    clrscr;
    writeln('Error loading MOD file ! ');
    if error = -1 then writeln('File not found !');
    if error = -2 then writeln('Insufficient memory available !');
    halt(0);
  end;
  periodic_on;   { Turns on periodic playing }
  Scale_window;
  ch := #255;
  while not (ch=#27) and not (upcase(ch)='X')
    and not (upcase(ch)='N') do begin;
    Scale;
    if keypressed then ch := readkey;
    case ch of
    #0 : begin;
      dch := readkey;
      case dch of
      #61 : begin; { F3 }
        if Mastervolume > 0 then dec(Mastervolume);
        Set_Volume(Mastervolume);
        textbackground(black);
      end;
      #62 : begin; { F4 }
        if Mastervolume < 255 then inc(Mastervolume);
        textbackground(black);
      end;
    end;
    #63 : begin; { F5 }
      ifailable;
Sound Support For Your Programs

Chapter 13

PC underground

textcolor(lightblue);
writexy(47,2,'Volume: ');
textcolor(lightcyan);
write(Mastervolume:2);
ch := #255;
end;

#62 : begin; { F4 }
if Mastervolume < 31 then inc(Mastervolume);
Set_Volume(Mastervolume);
textbackground(black);
textcolor(lightblue);
writexy(47,2,'Volume: ');
textcolor(lightcyan);
write(Mastervolume:2);
ch := #255;
end;

#63 : begin; { F5 }
if Balance > 0 then dec(Balance);
Set_Balance(Balance);
textcolor(lightblue);
textbackground(black);
writexy(58,2,'Balance ');
textcolor(14);
writexy(66,2,'•••••••••••••');
textcolor(4);
writexy(78-Balance DIV 2,2,'•')
ch := #255;
end;

else begin;
ch := #255;
end;
end;

'6' : begin;
inc(mli);
ch := #255;
end;

'f' : begin;
filter_activ := not filter_activ;
if filter_activ then begin;
Filter_On;
textcolor(lightblue);
textbackground(black);
writexy(36,2,'Filter: ');
textcolor(lightcyan);
write('ON');
end else begin;
filter_mid;
textcolor(lightblue);
textbackground(black);
The next important procedure is `play_sound`. This procedure determines if the file is a MOD file. If so, it calls procedure `Play_the_Mod`. Otherwise, the file is a VOC file. This requires resetting the Sound Blaster card. Then a short information text is displayed before `Init_Voc` starts the VOC file output. Now we must wait until the variable `Voc_ready` is TRUE or the user presses a key. Playing can be paused with `voc_pause`
Chapter 13
Sound Support For Your Programs

and resumed with `voc_continue`. When playing is finished (the end of the VOC file is reached) or the user quits during play, `Voc_done` is called. This procedure is called to terminate output and close the file.

```pascal
procedure write_vocmessage;
begin;
    clrscr;
    writexy(10,08,'Warning ! VOC looping relentlessly !!!');
    writexy(10,10,'To exit, press >> Q <<');
    writexy(10,14,'If possible, remove Smartdrv, because it is sioooooow !!');
    writexy(10,21,'                            E N J O Y');
end;

procedure play_sound(datname : string);
var li : integer;
    ch : char;
begin;
    for li := 1 to length(datname) do
        datname[li] := upcase(Datname[li]);
    if pos('.MOD',datname) <> 0 then begin
        Play_The_Mod(datname);
        exit;
    end;
    if pos('.VOC',datname) <> 0 then begin
        repeat
            Reset_Sb16;
            write_vocmessage;
            Init_Voc(datname);
            ch := #0;
            repeat
                if keypressed then ch := readkey;
                if ch = 'p' then begin
                    voc_pause;
                    repeat
                        ch := readkey;
                    until ch = 'c';
                    voc_continue;
                end;
            until VOC_READY or (ch = 'n') or (upcase(ch) = 'Q');
            VOC_DONE;
        until upcase(ch) = 'Q';
    end;
end;
```

Make certain to call `Init_the_mod` before trying to use any sound output routines. Otherwise, the Sound Blaster card will not be properly initialized, which will most likely cause an abort.

```pascal
begin;
    Samfreq := 22;
    clrscr;
    test_systemspeed;
    cursor_off;
    interpret_commandline;
    if (Nummods = 0) and not repeatmode then begin
        textcolor(15);
        textbackground(1);
```

396
Sound Support For Your Programs

Chapter 13

PC underground

clrscr;
Nummods := 1;
modd[1] := '+select_file('*.?o?','*.?o?','','Please select MOD file');
if modd[1] = 'xxxx' then begin;
clear;
    writeln('You must already have 'as a MOD file !');
    Cursor_on;
    halt(0);
end;
end;
end;
for i := 1 to Nummods do begin;
    if pos('.',modd[i]) = 0 then modd[i] := modd[i]+'.mod';
end;
Init_The_Mod;
stereo := false;
ext_song := random(Nummods)+1;
textcolor(lightgray);
textbackground(black);
write_SBconfig;
writeLn;
writeLn;
write('                     ENTER  to continue ...');
readln;
repeat
    if repeatmode then begin;
        textcolor(15);
        textbackground(1);
clear;
        modd[1] := '+select_file('*.?o?','*.?o?','');
        if modd[1] = 'xxxx' then next_song := 255
            else Play_Sound(modd[1]);
    end else
    Play_Sound(modd[next_song]);
    if next_song <> 255 then next_song := random(Nummods)+1;
    until next_song = 255;
cursor_on;
textmode(3);
end.

We hope you have a firm understanding about Sound Blaster programming. We wish you luck and fun exploring and improving these examples. There are many ways you can optimize these programs and integrate them into your programs. Maybe you'll use the unit for your next game or demo, or even write your own MOD editor.

Programming A MOD Player For The Gravis UltraSound

The Gravis UltraSound (GUS) card is perfect for playing MOD files. It allows you to load complete samples from the module into the card's own RAM and to play the voices directly from the arrangement in the MOD. No manual mixing is required as this is handled by the GUS.

How can we take advantage of these features? With the control unit called gus_mod, you can easily handle all the basic functions required to play a MOD file, including initializing the card, loading the MOD and outputting the sound. We'll talk about what you need to interface this unit to your own programs in this section. We'll also describe the structures and procedures used for each of the basic functions.
The MOD player structure

Playing a MOD file is a simple process. First, we have to initialize the Gravis UltraSound. We start by calling function \_gus_init_env to get the base port of the GUS. It returns TRUE if the port was successfully determined using the environment variable Ultrasnd. Otherwise, it returns FALSE. If the function was successful, we initialize the card with \_gus_initialize.

Now the card is ready to accept data. We pass the name of the file to be opened to the function called \_gus_modload. If the file opens and loads into memory successfully, it returns TRUE. Otherwise, if it isn't opened or loaded successfully, it returns FALSE. Once the MOD is loaded, we can start play by calling \_gus_modstart. The MOD file automatically loops back when reaching the end. To stop play, we call \_gus_mod_end. This procedure halts output and removes the MOD file from memory. This is all you really need to know to integrate a MOD player for the Gravis UltraSound into your programs.

Key variables of the GUS MOD player

You'll want your MOD player to do more than just play MOD files. It should also provide visual information to the user and be subject to some interaction during play.

The unit provides variables for these purposes. Let's take a closer look at the global variables:

- **Play_Channel**: array[1..14] of byte;
  
The entries in the Play_Channel array have values of 1 (indicates the corresponding channel will play) or 0 (turns the channel off). This is an easy way to mute a channel.

- **Channels**: array[0..31] of PChannelInfo;
  
This array stores information for the individual channels. The array consists of pointers that point to the corresponding channel structures. A channel structure would appear similar to the following:

```plaintext
TChannelInfo = record
    InstNr     : byte;  { Hardware-related variables }
    Mempos     : longint;
    End        : longint;
    Loop_Start : longint;
    Loop_End   : longint;
    Volume     : integer;  { MOD-related variables }
    Frequency  : word;
    Looping    : byte;
    Tone       : integer;
    Start_Tone : integer;
    Targ_Tone  : integer;
    Effect     : byte;
    Operand    : byte;
    Effectx,   { Effect-related variables }
    Effecty    : integer;
    Arpegpos   : integer;
    slidespeed : integer;
    vslide     : integer;
end;
```
The individual fields have the following meanings:

**InstNr**
InstNr contains the number of the instrument to be played, as found in the MOD file. Values are from 1 to 31.

**Mempos**
The start position of the active voice in GUS-RAM is found in Mempos.

**End**
End denotes the end position of the voice in GUS-RAM.

**Loop_Start**
A voice can be looped. If so, the start position of the looping in GUS-RAM is stored here.

**Loop_End**
This variable contains the corresponding loop end position, also relative to GUS-RAM.

**Volume**
The volume at which the voice is played is found here. This is the volume used in the MOD file, which ranges from 0 to 63.

**Frequency**
The frequency of the voice as stored in the MOD file is found here. It is not the actual output frequency, which must still be calculated. This is mostly needed for various effects.

**Looping**
This variable indicates whether the voice is looped. The value 8 indicates looping, 0 indicates no looping. You need this indicator for programming the GUS. If the third bit of the voice start command is set, the voice is looped.

**Tone, Start_Tone, Targ_Tone**
These variables are needed for MOD file calculations. We need more than one because some values have to be stored temporarily.

**Effect**
The current effect from the MOD file is stored here.

**Operand**
This holds the operand for the MOD effect.

**Effectsx, Effecty**
Effectsx contains the upper four bits of the operand, and Effecty contains the lower four bits.

**Arpegpos**
An Arpeggio effect plays three different pitches in succession. This variable indicates which of the three pitches is currently active.
slidespeed
This variable is needed for frequency sliding. It gives the speed with which the slide occurs.

vslide
Similar to slidespeed, this variable gives the speed of volume sliding.

retrig_count
This variable is needed to execute the Retriggering effect. It gives the position for the voice restart.

vibpos
This holds the current position within the Vibrato table.

vibx,viby
These variables hold the X and Y parameters for the Vibrato effect.

Instruments : array[0..31] of PInstrumentInfo

This is used for referencing information about the 31 instruments of a MOD file. The array contains pointers to the following structure:

```pascal
PInstrumentInfo = ^TInstrumentInfo;
TInstrumentInfo = record
  Name       : string[22];
  Mempos     : longint;
  End        : longint;
  l_Start    : longint;
  l_end      : longint;
  Size       : word;
  Loop_Start : word;
  Loop_End   : word;
  Volume     : word;
  Looping    : byte;
end;
```

The fields have the following meanings:

**Name**
This is the name of the corresponding instrument.

**Mempos**
Mempos references the position of the instrument's sample data within the GUS-RAM.

**End**
End references the end position of the instrument in GUS-RAM.

**L_Start**
An instrument can be looped. If so, this variable contains the start position of the instrument looping in GUS-RAM.

**L_End**
The corresponding end position of the instrument looping in GUS-RAM is stored here.

**Size**
The instrument length found in the MOD file is stored here (in bytes).
Loop_Start
Loop_Start references the offset (within the instrument's sample data) where looping begins.

Loop_End
This is the offset (within the instrument's sample data) where looping ends.

Volume
The volume of the instrument is stored here. It can range from 0 to 63.

Looping
This variable indicates whether the instrument is looped. The value 8 indicates looping, 0 indicates no looping. You need this indicator for programming the GUS.

MOD_Voices : word
Depending on how many voices the song has, this variable contains the value 4 or 8.

Mod_Patternsize : word
The size of a pattern is stored here. It is computed as MOD_Voices * 256. A four-voice pattern is therefore 1024 bytes long.

Stop_TheVoice : array[1..8] of boolean;
This array is used to stop certain voices. You stop a voice by setting the corresponding indicator to TRUE.

Modinf : Modinfo;
Modinfo contains global information about the MOD.

Title
The title of the song is stored here.

Patt_cnt
This count refers to the number of patterns defined.

Runinf : Runinfo
This area contains run-time information needed while playing the MOD. The following fields are defined:

Attack [1..8]
This is used to simulate an equalizer for the voices. The value is set to 63 each time a new note is struck. Then it is repeatedly decremented on subsequent ticks.
**Sound Support For Your Programs**

**Line**

References the current line in the pattern that is being played. Values are from 1 to 64.

**Pattnr**

This variable holds the position within the arrangement. When it's used with the **Line** variable, it is effective for synchronizing certain events or animation with the music.

**Volumes [1..8]**

The current volume for each voice is found here.

**Speed, BMP**

The MOD speed is stored here. **Speed** has a value from 1 to 15 and specifies a ratio to the current base speed in BPM (default: 125).

**chpos : array[1..8] of byte**

This array is used for arranging the voice (channel) positions. Values range from 0 for far left to 15 for far right.

**VibratoTable : array[0..63] of integer;**

This table holds entries needed for the Vibrato effect. The 64 entries describe one oscillation, where positions 15 and 47 represent the extreme values, and positions 0 and 32 have the value 0. The same table is used for the Tremolo effect.

### Core routines of the MOD player

Now we're at the more interesting part of the unit: The procedures. What procedures do we need for a MOD player? Well, first there is one for loading the MOD from disk to memory. Then the sample data must be sent to the Gravis UltraSound card. After that, we need some routines to perform the playing.

**Loading the MOD**

Let's look first at how a MOD is loaded. In our unit, function `gus_modload` does this. You pass it the complete name of the MOD file to be loaded. If loading is successful, the function returns TRUE, otherwise it ends immediately and returns FALSE. The function starts by reserving memory for the channel structures and instrument information. Then it initializes two values in `Runinf: Line` is set to 0 and `Pattnr` is set to -1. This ensures the playback routine starts at the beginning of the MOD.

Now the file can be opened. The procedure saves this amount and displays a graphic to indicate the MOD file being loaded using `Save_Screen` and `display_loading`. You can omit these calls or replace them with your own screen processing.
Restlength is then set to the file size minus 1084 (the size of the MOD file without the header). We’ll use this later to determine the number of patterns in the MOD. Next we check the identifier in the header to validate the file type and determine the number of voices. Variables MOD_Voices and MOD_Patternsize are set accordingly. Now the complete header can be read. We start with the name of the MOD file. This is a null-terminated string and cannot be processed directly in Pascal. To convert it to Pascal format, we use function ConvertString, passing it a pointer to the string’s location in memory and its maximum length. The function returns the Pascal string ready for use in our program.

Next we get the song length, that is, the number of validly defined entries in the arrangement. This song length has nothing to do with the number of patterns stored in the MOD. Finally, before reading in the instrument data, we must get the 128-byte arrangement.

The problem of file compatibility must be addressed here. A MOD can currently have up to 31 instruments, but the old format allowed for only 15. Separate code sections could have been written for the two formats. We chose instead to have one routine that is controlled by two variables. One is called vh_Num_Inst. It contains the number of instruments found in the MOD file. The other we call ias (for instrument-dependent start position). For 31 instruments this has the value 0. For 15 instruments it has the value -16*30, because there are 16 fewer instruments, each requiring 30 bytes of information. Since we work internally with the offset for 31 instruments, we must reduce the offset accordingly with fewer instruments.

After we have these two variables, we can use a loop to read in the data for all the instruments. The process is as follows: First, we read the 22-byte long instrument name. This is a 0-terminated string, which we must convert to a Pascal string using function ConvertString. Then we must get the size of the sample in bytes. The size value we read in is not exactly what we need, however. First, it’s in Amiga format, so we must swap the high and low bytes to get PC format. Second, it is a word count, so we must multiply by 2 to get bytes.

The same conversion procedure is used for several other data items expressed as Amiga words. The next field, however, is a single byte that needs no conversion. Its value, from 0 to 63, specifies the instrument’s volume. Now follow the start position (expressed as an offset) and loop length for sample looping. Both are Amiga word values and must be converted. We add them to derive the end position of the sample, which we will need later.

Next the load function checks to see whether the instrument is looped. If the difference between loop start and loop end is greater than or equal to 10, variable Looping for the instrument is assigned the value with_loop (= 8). Otherwise, it’s set to no_loop. Finally, Restlength is decremented by the length of the MOD. As a result, after reading in the data for all the instruments, this variable will reflect the size of the pattern data in bytes. By dividing it by the size of one pattern, we get the number of patterns defined. These are then read in by another loop.

The instruments (samples) are loaded last and in a loop. Here we use the procedure Load_Instrument, which we’ll soon look at closer. After the procedure call, the loop manipulates the screen memory, which is referenced by variable Screen. The result is a continual change to the foreground color of the progress bar. This feature is needed only for the MOD player TCP. You can omit it or replace it with your own graphics handling.

At this point, the entire MOD file has been loaded into memory and can be closed. There are just a few more important variables to be initialized. First, all the GUS voices are properly set. Then the channels are arranged to simulate a semicircle, giving a sort of “concert hall” effect. To do this, we assign values from 0 to 15 to the array of variables chpos[1] through chpos[Mod_Voices].
These variables are distributed to the individual voices by using \_gus_set_channelpos. In conclusion, the interrupt speed is set to the standard value of 125, and the screen is restored by the procedure restore_screen. This screen processing can also be omitted in your own load routine, since it is needed only for TCP.

```pascal
function _gus_modload(name : string) : boolean;
{
    Loads the MOD file passed in name. Requirements are that the specified file in the path exists and is not write protected.
}
var dummya : array[0..30] of byte;{ For string handling }
daptr : pointer;{ Pointer to dummya }
dumw : word;{ Dummy variables for reading/inputting }
dumb : byte;
Restlength1 : longint;{ For determining the pattern number }
li : integer;
ID : array[1..4] of char;{ ID of the MOD file }
ias : integer;{ Instrument dependent start position }
begin;
    U_Ram_freepos := 32;
    for li := 0 to 15 do begin;
        new(Channels[li]);
        channels[li]^\_vibpos := 0;
    end;
    for li := 0 to 31 do begin;
        new(Instruments[li]);
        Instruments[li]^\_Name := '';
    end;
    runinf\_Line := 0;
    runinf\_Pattnr := -1;
    tickcounter := 0;
    ticklimit := 6;
    runinf\_bpm := 125;
    ias := 0;
    daptr := @dummya;
    assign(gusmf,name);{ open file + initialize length }
    reset(gusmf,1);
    save_screen;
    display_loading(name);
    Restlength1 := filesize(gusmf);
    Restlength1 := Restlength1 - 1084;
    seek(gusmf,1080);{ check whether MOD with 15/31 voices }
    Blockread(gusmf,ID,4);{ 15 voices ? }
    if pos(ID,ModIDs) = 0 then begin;
        Blockread(gusmf,ID,4);{ Not a valid .MOD file }
        writeln('Not a valid .MOD file !!!');
        halt(0);
```
end else begin;
    Modinstance := 15;
    ias := -16*30;
end;
end;

if (ID = MODId[1]) or (ID = MODId[2]) or (ID = MODId[3]) then begin;
    MOD_Voices := 4;
    MOD_Patternsize := 4*256;
end else
    if (ID = CHn6) then begin;
        _gus_modload := false;
        exit;
    end else
    if (ID = CHn8) then begin;
        MOD_Voices := 8;
        MOD_Patternsize := 8*256;
    end;

seek(gusmf,0);
Blockread(gusmf, dummya, 20);  { get name of file } 
vh.SongName := ConvertString(datptr,20);
seek(gusmf,950+ias);  { get song length in Pattern } 
Blockread(gusmf, vh.Songlength, 1);
seek(gusmf, 952+ias);  { read in arrangement } 
Blockread(gusmf, vh.Arrang, 128);

vh.Num_Inst := Modinstance;  { read in instruments (15/31) } 
seek(gusmf, 20+ias);

for li := 1 to 32 do Instruments[li].Name := ' '; 

for li := 1 to vh.Num_Inst do begin;
    Blockread(gusmf, dummya, 22);  { Instruments - Name }
    Instruments[li].Name := ConvertString(datptr, 22);
    Blockread(gusmf, dumw, 2);  { Length of sample } 
    Instruments[li].Size := swap(dumw) * 2;
    Blockread(gusmf, dumb, 1);  { read in volume } 
    Blockread(gusmf, dumb, 1);
    Instruments[li].Volume := dumb;
    Blockread(gusmf, dumw, 2);  { read in start of loop } 
    Instruments[li].Loop_Start := swap(dumw) * 2;
    Blockread(gusmf, dumw, 2);
    dumw := swap(dumw) * 2;  { read in loop end from Start+Length } 
    Instruments[li].Loop_End := Instruments[li].Loop_Start+dumw;
    if (Instruments[li].Loop_End - Instruments[li].Loop_Start) >= 10 then begin;
        Instruments[li].Looping := with_loop;
    end else begin;
        Instruments[li].Looping := no_loop;
    end;
Dec(Restlength1, Instruments[li].Size);
end;
Chapter 13

Sound Support For Your Programs

```plaintext
Vh.Num_Patts := Restlength DIV MOD_Patternsize; { Pattern number ? }
seek(gusmf,1084+ias);

for li := 1 to Vh.Num_Patts do begin; { read in pattern }
dos_getmem(Pattern[li],MOD_Patternsize);
Blockread(gusmf,Pattern[li]^,MOD_Patternsize);
end;

for li := 1 to vh.Num_Inst do begin; { read in instruments }
Load_Instrument(li);
screen[16,23+li].a := 5;
end;

close(gusmf);

for i := 1 to 31 do begin; { initialize channel variables }
u_VoiceBalance(i,7);
u_VoiceVolume(i,0);
u_VoiceFreq(i,12000);
U_StartVoice(i,Stop_Voice);
u_Voicedata(0,0,0,i);
end;

runinf.Line := 0;                     { runtime - init. variables }
runinf.Pattnr := -1;
tickcounter := 0;
ticklimit := 6;
runinf.speed := 6;
rinf.bpm := 125;
if MOD_Voices = 4 then begin;  { arrange channels in semicircle }
chpos[1] := 2;
chpos[2] := 5;
chpos[3] := 9;
chpos[4] := 12;
_gus_set_chanelpos;
end;

if MOD_Voices = 8 then begin;
chpos[1] := 1;
chpos[2] := 3;
chpos[3] := 5;
chpos[6] := 9;
chpos[8] := 13;
_gus_set_chanelpos;
end;

new_interrupt_Speed(runinf.bpm);
restore_screen;
Modinf.Voices := MOD_Voices;  { constant MOD info in structure }
Modinf.Title := vh.Songname;  { to be passed }
Modinf.Pattnum := Vh.Songlength;
_gus_modload := true;
end;
```

Now let’s look at the procedure `Load_Instrument` for loading the instruments. You assign it the number of the instrument to be loaded. If greater than 10 bytes, the data is loaded into GUS-RAM (a smaller size indicates there is no valid data for this instrument). The procedure begins by reading the sample into a 64K buffer. Then it searches for the next paragraph address (divisible by 16) in GUS-RAM. This represents the sample start position and is assigned to the instrument variable Mempos. The procedure `Ultra_Mem2Gus`
then transfers the sample to GUS-RAM, after which the reserved buffer is released. Finally, the loop start position and the voice end position in GUS-RAM are saved in the instrument information. The following shows how the coding list appears:

```pascal
procedure Load_Instrument(Nr : byte);
{ Loads the instrument with the number nr into GUS RAM }
var gr : longint;
  samp : pointer;
begin;
  gr := Instrument[nr]^\.Size;
  if gr > 10 then begin; { Load only if > 10, otherwise not worth it! }
    dos_getmem(samp,gr);
    Blockread(gusmf,samp^,gr);
    U_Ram_freepos := U_Ram_freepos + (16-(U_Ram_freepos MOD 16));
    Instruments[nr]^\.Mempos := U_Ram_freepos;
    Ultra_Mem2Gus(samp,Instruments[nr]^\.Mempos,gr);
    dos_freemem(samp);       { initialize variables for Voices }
    if Instruments[nr]^\.Looping = With_loop then begin;
      end else begin;
        Instruments[nr]^\.end := Instruments[nr]^\.Mempos + gr - 25;
      end;
    Inc(U_Ram_Freepos,gr);   { continue setting management pointer }
  end;
end;
```

### Playing the loaded Mod

Now the MOD is loaded, we can play it by using one of many methods. One method is to use a loop to increment a variable, or synchronize with the graphic card’s vertical retrace. There is another method, however, that is not only more precise, but also much more flexible. This method uses the timer interrupt.

We program the interrupt with a new frequency and redirect the handling routine to our own procedure. This procedure increments a counter with every tick (every call). When the counter reaches the value specified as Speed in the MOD file, the MOD advances one line. At the end, we send an EOI (end of interrupt) by writing the value 20h to port 20h.

Our procedure for starting the MOD is **_gus_modstart**. It uses the value of BPM (beats per minute) to calculate a new frequency for the timer. This is programmed over ports 43h and 40h. Before redirecting the timer interrupt handler to our own procedure, we must save the old interrupt so we can restore it when we’re finished. This is done by the procedure **timerint_reset**. It resets the timer to its original frequency and returns the interrupt to the proper handling routine.

By using the timer interrupt, we can also construct a simple Pause function. To suspend play, we divert the timer interrupt handler to a routine that does nothing except to send an EOI. First, however, we set the individual channel volumes to 0. To resume play, we set the interrupt back to the player routine and turn on the voices again.
procedure mytimer; interrupt;
{
  My Timer Interrupt
}
begin;
tick_effects;
inc(tickcounter);
if tickcounter >= ticklimit then begin;
  tickcounter := 0;
  Play_Pattern_gus;
end;
Port[$20] := $20;
end;

procedure do_nothing; interrupt;
{
  Dummy interrupt. Program switches to this interrupt when output is paused/stopped.
}
begin;
port[$20] := $20;
end;

procedure _gus_modstart;
{
  Starts output of MOD file via the timer interrupt. MOD file
  must already be loaded !
}
var counter : word;
loc,hic : byte;
begin;
counter := 1193180 DIV interrupt_speed;
loc := lo(counter);
hic := hi(counter);
asm
  cli
  mov dx,43h
  mov al,36h
  out dx,al
  mov dx,al
  out dx,al
  mov al,loc
  out dx,al
  mov al,hic
  out dx,al
end;
getintvec(8,oldtimer);
setintvec(8,@Mytimer);
asm sti end;

procedure _gus_player_pause;
{
  Pauses output via the timer interrupt
}
var li : integer;
begin;
setintvec(8,@Do_nothing);
for li := 0 to 31 do
  _u_VoiceVolume(li,0) ;
end;

procedure _gus_player_continue;
{ Continues output via the timer interrupt. }
var li: integer;
begin;
  setintvec(8,@Mytimer);
  for li := 1 to 31 do 
  Voice_Rampin(li,Channels[li]^\.volume);
end;

procedure timerint_reset;
{ Resets the timer interrupt to its original value. }
begin;
  asm
    cli
    mov dx,43h
    mov al,36h
    out dx,al
    xor ax,ax
    mov dx,40h
    out dx,al
    out dx,al
  end;
  setintvec(8,oldtimer);
  asm sti end;
end;

Playing the MOD file now advances one line for each call to the procedure \texttt{play\_pattern\_gus}. It interprets the arrangement and patterns to produce the appropriate tones. It begins by incrementing the line number by 1. If the resulting line number exceeds 64, it is reset to 1 for the start of the next pattern. If the next pattern number (the position within the arrangement) exceeds the song length, the pattern number in turn is reset to 1. Then the current line is read from the pattern to an array called \texttt{The\_Line}. This facilitates access to the individual bytes.

Now the line is processed. For each voice, we first check to see if there is something to be played. If so, the corresponding entry in \texttt{play\_channel} will be 1. From the information in the line, we then obtain the required tone, instrument, and effect.

If the tone entry is not 0, a frequency will be given. To handle the frequency properly, we must first check for $\text{Effect} = 3$. In this case, the frequency refers to the target tone of a slide. In all other cases, the frequency refers to the tone that is to be played immediately. The channel variables \texttt{Tone} and \texttt{Start\_Tone} or \texttt{Targ\_Tone} are set accordingly.

If the instrument entry is not 0, we must reload the channel variables for the instrument parameters. We first pass the new instrument number, then use it to set the relevant positions in GUS-RAM, the volume and the looping indicator.

Next the effects can be handled. Since this requires complex special instructions, effect-handling is done in a separate procedure, which we will discuss in detail shortly.

Finally, if \texttt{Tone} doesn’t equal 0 (\texttt{<>} 0), we can actually play the note. First, we calculate the frequency for the channel, dividing the base frequency for the number of voices by the frequency in the MOD file. Then we set the frequency with procedure \texttt{u\_VoiceFreq}. We also set the volume once it has been validated (it
must be in the range 1 to 63). We then stop the playing of the previous voice. Now the new voice can be started with the newly programmed data.

```
procedure play_pattern_gus;
{
  This procedure is called periodically. It plays back a line of the MOD file.
}
var li        : integer;
  dumw      : word;
  The_Line : array[1..8,0..3] of Byte;
  Effect : byte;
  Ton       : word;
  Inst      : byte;
begin;
{
  **************************************************************************
  ***   Move forward in Mod                                                 ***
  **************************************************************************
}
  inc(runinf.Line);              { Move forward one line              }
  if runinf.Line > 64 then runinf.Line := 1;
  if runinf.Line = 1 then begin; { New Pattern ?                   }
    inc(runinf.Pattnr);
    if runinf.Pattnr > vh.Songlength then runinf.Pattnr := 1;
    end;
    { load notes                       }
  move(ptr(seg(pattern[vh.Arrang[runinf.Pattnr]+1]^),
       (runinf.Line-1)*4*Mod_Voices)^,
       Die_Line,4*8);
{
  **************************************************************************
  ***   Process voices                                           ***
  **************************************************************************
}
  for li := 1 to MOD_Voices do begin;
    if play_chanel[li] = 1 then begin;
      stop_Thevoice[li] := false;
      Tone  := ((The_Line[li,0] AND $0f) shl 8)+The_Line[li,1];
      Inst := (The_Line[li,0] AND $f0)+((The_Line[li,2] AND $f0) SHR 4);
      Channels[li]^.Operand := The_Line[li,3];
      Channels[li]^.Start_Tone := oldv[li];
    if Tone <> 0 then begin;  { Tone entered ??? }
      if Channels[li]^.Effect = 3 then begin;
        Channels[li]^.Destination_Tone := Tone;
      end else begin;
        Channels[li]^.Tone := Tone;
        Channels[li]^.Start_Tone := Tone;
      end;
      end;
    if Inst <> 0 then begin; { use new instrument ??? }
      Channels[li]^.InstNr     := Inst;
    end;
```
Sound Support For Your Programs

Chapter 13

PC underground

The interesting part is handling the effects when we must distinguish between a procedure for initializing the effects and a procedure for updating them with every tick. Let’s look first at the initialization procedure. This is called from `play_pattern_gus`.

If no effect is specified, the procedure terminates immediately. Otherwise, a Case instruction checks the effect type. We could have used a jump table with the addresses of individual effect procedures instead. However, since the GUS allows us to use a somewhat slower player, we prefer the case structure for its readability.

According to the effect specified, we make the necessary changes to the parameters for the current channel. We will look at each one in turn:
Chapter 13

Sound Support For Your Programs

Effect 0 - Arpeggio

This effect plays three pitches in quick succession, starting with the pitch in \texttt{Start_Tone}. The X-nibble is the first increment for raising the pitch, the Y-nibble the second. A new frequency must be calculated and set with each tick.

```pascal
procedure Initialize_Effects(nr : byte);
var swaplong : longint;
vibswap : integer;
begin;
if Channels[nr]^\^Effect = 0 then exit;
case Channels[nr]^\^Effect of
  0 : begin; \{ Arpeggio \}
    Channels[nr]^\^Arpegpos := 0;
    Channels[nr]^\^Effectx := Channels[nr]^\^Operand shr 4;
    Channels[nr]^\^Effecty := Channels[nr]^\^Operand and $0f;
    inc(Channels[nr]^\^Arpegpos);
    case (Channels[nr]^\^Arpegpos MOD 3) of
      0 : begin; \{ap = 3 !\}
        Channels[nr]^\^Start_Tone :=
        Channels[nr]^\^Tone + Channels[nr]^\^Effecty;
      end;
      1 : begin; \{ap = 1 !\}
        Channels[nr]^\^Start_Tone := Channels[nr]^\^Tone;
      end;
      2 : begin; \{ap = 2 !\}
        Channels[nr]^\^Start_Tone :=
        Channels[nr]^\^Tone + Channels[nr]^\^Effectx;
      end;
    end;
    if Channels[nr]^\^Start_Tone < 1 then
      Channels[nr]^\^Start_Tone := 1;
    Channels[nr]^\^Frequency :=
    longint(Voice_Base[14] div Channels[nr]^\^Start_Tone);
    u_VoiceFreq(nr,Channels[nr]^\^Frequency);
  end;
1 : begin; \{ Portamento up \}
  dec(Channels[nr]^\^Start_Tone,Channels[nr]^\^Operand);
  if Channels[nr]^\^Start_Tone < 1 then
    Channels[nr]^\^Start_Tone := 1;
    Channels[nr]^\^Frequency :=
    longint(Voice_Base[14] div Channels[nr]^\^Start_Tone);
    u_VoiceFreq(nr,Channels[nr]^\^Frequency);
end;
2 : begin; \{ Portamento down \}
  inc(Channels[nr]^\^Start_Tone,Channels[nr]^\^Operand);
  if Channels[nr]^\^Start_Tone < 1 then
```

Effect 1 - Portamento Up

For Portamento Up, the frequency is incremented by the value in the operand with each tick.

```
Effect 2 - Portamento Down

This effect is similar to Portamento Up but the frequency is decremented instead of incremented.

```

The program listings on pages 412-421 are from \texttt{GUS.MOD.PAS} file on the companion CD-ROM
Effect 3 - Tone Portamento

We must first determine whether the target pitch for this effect is lower or higher than the current pitch. Then we increment or decrement the frequency accordingly, at a speed determined by the operand. The frequency is changed and reset on the GUS with each tick. Here the initialization is implemented in the case query by a separate procedure.

```
3 : begin; { Tone Portamento }
    EI_toneportamento(nr);
end;

procedure EI_toneportamento(nr : byte);
{ Init for effect-handling Tone Portamento procedure }
begin;
{ Determine Inc factor }
    if Channels[nr]^.Operand <> 0 then
    begin
        if Channels[nr]^.Start_Tone > Channels[nr]^.Targ_Tone then
            begin
                Channels[nr]^.slidespeed := -(Channels[nr]^.Operand);
            end else begin;
                Channels[nr]^.slidespeed := (Channels[nr]^.Operand);
            end;
        end;
    end;
    if Channels[nr]^.Start_Tone < 1 then
        Channels[nr]^.Start_Tone := 1;
    Channels[nr]^.Frequency :=
        longint(Voice_Base[14] div Channels[nr]^.Start_Tone);
    u_VoiceFreq(nr,Channels[nr]^.Frequency);
    oldv[nr] := Channels[nr]^.Start_Tone;
end;
```

Effect 4 - Vibrato

For a vibrato, we first get the speed and depth from the X and Y nibbles of the operand. Then the Vibrato procedure is called. It increments a pointer in the Vibrato table. If this becomes greater than 64, it is decremented by 64 to bring it back into range. Using the pointer, we get a value from the Vibrato table, which we then multiply by the depth and divide by 256. We then increment the frequency by the value obtained. This procedure is called and the frequency reset with every tick.

```
4 : begin; { Vibrato *new* }
    Channels[nr]^.vibx := Channels[nr]^.Operand shr 4;
    Channels[nr]^.viby := Channels[nr]^.Operand and $0f;
    effect_vibrato(nr);
end;

procedure effect_vibrato(nr : byte);
```
Chapter 13
Sound Support For Your Programs

```pascal
{ From the effect-handling Vibrato procedure }
var vibswap : integer;
begin;
  inc(Channels[nr]^\.vibpos,Channels[nr]^\.vibx);
  if Channels[nr]^\.vibpos > 64 then
    dec(Channels[nr]^\.vibpos,64);
  vibswap :=
    (VibratoTable[Channels[nr]^\.vibpos] * Channels[nr]^\.viby) div 256;
  inc(Channels[nr]^\.Start_Tone,vibswap);
  if Channels[nr]^\.Start_Tone < 1 then
    Channels[nr]^\.Start_Tone := 1;
  Channels[nr]^\.Frequency :=
    longint(Voice_Base[14] div Channels[nr]^\.Start_Tone);
  u_VoiceFreq(nr,Channels[nr]^\.Frequency);
end;
{------------------------------------------------------------------------}
```

**Effect 5 - Note & Volume sliding**

This effect can be divided into two parts: Volume sliding and Portamento. We first determine whether the operand is greater than 0fh. If not, the effect is a Sliding down and we compute the speed by the formula speed := Operand AND $0f. Otherwise, the effect is a Sliding up and we determine the speed by a 4-bit right shift of the operand. We then change the voice volume and verify the new volume is within range. The frequency is also validated. Volume and frequency are then set on the GUS.

```pascal
5 : begin; { NOTE SLIDE + VOLUME SLIDE: *new* }
  { init }
  if Channels[nr]^\.Operand <= $0f then
    begin;
      Channels[nr]^\.vslide := -(Channels[nr]^\.Operand AND $0f);
      Channels[nr]^\.slidespeed := -(Channels[nr]^\.Operand AND $0f);
    end else begin;
      Channels[nr]^\.vslide := (Channels[nr]^\.Operand shr 4);
      Channels[nr]^\.slidespeed := (Channels[nr]^\.Operand shr 4);
    end;
  { volume slide }
  inc(Channels[nr]^\.volume,Channels[nr]^\.vslide);
  if Channels[nr]^\.volume < 0 then Channels[nr]^\.volume := 0;
  if Channels[nr]^\.volume > 63 then Channels[nr]^\.volume := 63;
  u_VoiceVolume(Nr,Channels[nr]^\.volume);
  { Note slide }
  inc(Channels[nr]^\.Start_Tone,Channels[nr]^\.slidespeed);
  if Channels[nr]^\.Start_Tone < 1 then
    Channels[nr]^\.Start_Tone := 1;
  Channels[nr]^\.Frequency :=
    longint(Voice_Base[14] div Channels[nr]^\.Start_Tone);
  u_VoiceFreq(nr,Channels[nr]^\.Frequency);
end;
```

**Effect 6 - Vibrato & Volume sliding.**

This effect also has two parts. See Effect 5 for Volume sliding and Effect 4 for the Vibrato routine.

```pascal
6 : begin; { Vibrato & Volume slide *new* }
  { init }
  Channels[nr]^\.vibx := Channels[nr]^\.Operand shr 4;
  Channels[nr]^\.viby := Channels[nr]^\.Operand and $0f;
```
if Channels[nr]^.Operand <= $0f then
  begin;
    Channels[nr]^.vslide := -(Channels[nr]^.Operand AND $0f);
    Channels[nr]^.vslide := (Channels[nr]^.Operand shr 4);
  end;
{ volume slide }
inc(Channels[nr]^.volume,Channels[nr]^.vslide);
if Channels[nr]^.volume < 0 then Channels[nr]^.volume := 0;
if Channels[nr]^.volume > 63 then Channels[nr]^.volume := 63;
u_VoiceVolume(Nr,Channels[nr]^.volume);
{ vibrato }
effect_vibrato(nr);
end;

Effect 7 - Tremolo

The Tremolo effect works like the Vibrato. Here, however, the volume is manipulated instead of the frequency.

7 : begin; { tremolo 'new' }
  Channels[nr]^.vibx := Channels[nr]^.Operand shr 4;
  Channels[nr]^.viby := Channels[nr]^.Operand and $0f;
  inc(Channels[nr]^.vibpos,Channels[nr]^.vibx);
  if Channels[nr]^.vibpos > 64 then
    dec(Channels[nr]^.vibpos);
  vibswap :=
    (VibratoTable[Channels[nr]^.vibpos] * Channels[nr]^.viby) div 256;
  inc(Channels[nr]^.Volume,vibswap);
  if Channels[nr]^.Volume < 0 then Channels[nr]^.Volume := 0;
u_VoiceVolume(nr,Channels[nr]^.volume);
end;

Effect 8 - Not used

This effect is not used. You can use it for your own effects or graphic synchronizations.

8 : begin; { not used !!! }
{ Not officially used. Can be used to synchronize certain events in a demo ... }
end;

Effect 9 - Sample-Offset

In the Sample-Offset command, the operand represents the high byte of an offset within the sample. Multiply the operand by 256 to obtain the offset. We then add the result to the position of the sample in GUS-RAM and set the channel to this position.

9 : begin; { Sample - Offset 'new' }
  swaplong := longint((Channels[nr]^.Operand+1)) * 256;
  u_Voicedata(Channels[nr]^.Mempos,Channels[nr]^.Loop_Start,
    Channels[nr]^.Ende,nr);
  U_StartVoice(nr,Play_Voice+Bit8+Channels[nr]^.Looping+Unidirect);
end;
Effect 0ah - Volume sliding

This is one of the easiest effects to achieve. If the operand is less than 16, we must subtract its lower four bits from the current volume with each tick. Otherwise, we shift it right by 4 bits and add the result to the volume with each tick. Again, make certain the new volume is within range.

```pascal
$s : begin; { Volume sliding *new* }
  if Channels[nr]^.Operand <= $0f then
    begin;
      Channels[nr]^.vslide := -(Channels[nr]^.Operand AND $0f);
    end
  else begin;
    Channels[nr]^.vslide := (Channels[nr]^.Operand shr 4);
  end;
  inc(Channels[nr]^.volume,Channels[nr]^.vslide);
  if Channels[nr]^.volume < 0 then Channels[nr]^.volume := 0;
  if Channels[nr]^.volume > 63 then Channels[nr]^.volume := 63;
  u_VoiceVolume(Nr,Channels[nr]^.volume);
end;
```

Effect 0bh - Position Jump

Here the operand specifies the next desired position within the arrangement. We set the line number to 64 (the end of the current pattern) so the new pattern will start automatically with the next execution.

```pascal
$\underline{b} : begin; { Position Jump *ok* }
  runinf.Line := 64;
  runinf.Pattnr := Channels[nr]^.Operand;
end;
```

Effect 0ch - Set Note Volume

This effect sets the volume of a voice. We simply assign the operand value to the voice volume and make sure it is within range.

```pascal
$c : begin; { Set Note Volume *ok* }
  if Channels[nr]^.Operand > 63 then Channels[nr]^.Operand := 63;
  if Channels[nr]^.Operand < 1 then
    begin
      Channels[nr]^.volume := 0;
      u_VoiceVolume(nr,0);
      U_StartVoice(nr,Stop_Voice);
      stop_Thevoice[nr] := true;
    end
  else begin
    Channels[nr]^.volume := Channels[nr]^.Operand;
    u_VoiceVolume(Nr,Channels[nr]^.volume);
    Runinf.Volumes[nr] := 63;
  end;
end;
```

Effect 0dh - Pattern Break

This effect jumps to the next pattern of the MOD. Strictly speaking, to program it correctly we would set the line number to the value of the operand, and set a Boolean variable to TRUE to trigger the jump. It's easier, however, to set the line to the end of the current pattern for the jump to be triggered automatically.

```pascal
$d : begin; { Pattern Break *ok* }
  runinf.Line := 64;
end;
```
Effect 0eh - Extended effect commands

Effect 0eh defines several subeffects. The subeffect is specified by the upper four bits of the operand. The following are of interest:

**Subeffect 1 - Fine Sliding Up**

This effect works like Portamento-Up. Instead of updating with every tick, however, we change the frequency only once initially.

```
$e : begin;  { Extended effect command }
  case (Channels[nr]^.Operand shr 4) of
    1 : begin; { Fine slide up }
      inc(Channels[nr]^.Start_Tone,Channels[nr]^.Operand and $0f);
      if Channels[nr]^.Start_Tone < 1 then Channels[nr]^.Start_Tone := 1;
      Channels[nr]^.Frequency :=
        longint(Voice_Base[14] div Channels[nr]^.Start_Tone);
      u_VoiceFreq(nr,Channels[nr]^.Frequency);
    end;
```

**Subeffect 2 - Fine Sliding Down**

This is like Subeffect 1, but corresponds to Portamento down.

```
2 : begin; { Fine slide down }
  dec(Channels[nr]^.Start_Tone,Channels[nr]^.Operand and $0f);
  if Channels[nr]^.Start_Tone < 1 then Channels[nr]^.Start_Tone := 1;
  Channels[nr]^.Frequency :=
    longint(Voice_Base[14] div Channels[nr]^.Start_Tone);
  u_VoiceFreq(nr,Channels[nr]^.Frequency);
  end;
```

**Subeffect 9 - Retriggering Note**

This effect plays the note again after the number of ticks specified in the operand. This simply means the sample starts over from the beginning.

```
9 : begin; { Retriggering !!! *new* }
  Channels[nr]^.Retrig_count :=
    Channels[nr]^.Operand and $0f;
  end;
```

**Subeffect 0ah - Fine Volume Slide Up**

The volume of the channel is increased by the amount specified in the lower four bits of the operand. The volume does not change with subsequent ticks.

```
$a : begin; { Fine slide down }
  Channels[nr]^.vslide := (Channels[nr]^.Operand AND $0f);
  inc(Channels[nr]^.volume,Channels[nr]^.vslide);
  if Channels[nr]^.volume < 0 then Channels[nr]^.volume := 0;
  if Channels[nr]^.volume > 63 then Channels[nr]^.volume := 63;
  u_VoiceVolume(Nr,Channels[nr]^.volume);
  end;
```
Subeffect 0bh - Fine Volume Slide Down

The volume of the channel is decreased by the amount specified in the lower four bits of the operand. The volume does not change with subsequent ticks.

```pascal
$b : begin;  { fine volume slide down }
Channels[nr]^.vslide := (Channels[nr]^.Operand AND $0f);
dec(Channels[nr]^.volume,Channels[nr]^.vslide);
if Channels[nr]^.volume < 0 then Channels[nr]^.volume := 0;
if Channels[nr]^.volume > 63 then Channels[nr]^.volume := 63;
u_VoiceVolume(Nr,Channels[nr]^.volume);
end;
```

Subeffect 0ch - Cut Voice

This effect stops the playing of a channel. In our player, we simply set Stop_TheVoice to TRUE.

```pascal
$\textit{c} : \begin{array}{l}
\text{begin;  \{ Cut Voice \textit{*ok*} \}} \\
\text{stop_Thevoice[nr] := true;}
\text{end;}
\text{end;}
\end{array}
```

Effect 0fh - Set Speed

This effect sets the MOD speed. If the operand value is less than 16, it represents the speed directly and is assigned to the variable ticklimit. Otherwise, it is a BPM (beats per minute) value, and procedure `new_interrupt_speed` must be called.

```pascal
$f : begin;  { Set Speed \textit{*ok*} \}
\text{if Channels[nr]^.Operand <= $f then begin;}
\text{ticklimit := Channels[nr]^.Operand;}
\text{runinf.speed := ticklimit;}
\text{end else begin;}
\text{runinf.bpm := Channels[nr]^.Operand;}
\text{new_interrupt_Speed(Channels[nr]^.Operand);} \\
\text{end;}
\text{end;}
\text{end;}
```

Remember the effects must not only be initialized, but updated with each tick. The algorithms required here are essentially the same as we have just seen, but without the initializing of most of the variables. The procedure is as follows:

```pascal
procedure tick_effects;
\text{var li : integer;}
\text{vibswap : integer;}
\text{begin;}
\text{for li := 1 to MOD_Voices do begin;}
\text{if runinf.volumes[li] > 0 then}
\text{dec(runinf.volumes[li]);}
\text{case Channels[li]^.Effect of ( Process runtime effects )}
0 : begin;
\text{inc(Channels[li]^.Arpegpos);} \\
\text{case (Channels[li]^.Arpegpos MOD 3) of}
0 : begin; \{ ap = 3 \}
```

418
1: begin; { ap = 1 ! }
   Channels[*li*]^.Start_Tone :=
   Channels[*li*]^.Tone;
end;

2: begin; { ap = 2 ! }
   Channels[*li*]^.Start_Tone :=
   Channels[*li*]^.Tone + Channels[*li*]^.Effectx;
end;
end;

1: begin;
{ new }
   Channels[*li*]^.Operand := Channels[*li*]^.Operand and $0F;
{ new end }
   dec(Channels[*li*]^.Start_Tone, Channels[*li*]^.Operand);
if Channels[*li*]^.Start_Tone < 1 then
   Channels[*li*]^.Start_Tone := 1;
   Channels[*li*]^.Frequency :=
   longint(Voice_Base[14] div Channels[*li*]^.Start_Tone);
   u_VoiceFreq(li, Channels[*li*]^.Frequency);
end;
2: begin;
{ new }
   Channels[*li*]^.Operand := Channels[*li*]^.Operand and $0F;
{ new end }
   inc(Channels[*li*]^.Start_Tone, Channels[*li*]^.Operand);
if Channels[*li*]^.Start_Tone < 1 then
   Channels[*li*]^.Start_Tone := 1;
   Channels[*li*]^.Frequency :=
   longint(Voice_Base[14] div Channels[*li*]^.Start_Tone);
   u_VoiceFreq(li, Channels[*li*]^.Frequency);
end;
3: begin; { Tone Portamento }
   K_toneportamento(li);
end;

-------------------------------
procedure K_toneportamento(nr : byte);
{ From the effect-handling TonePortamento procedure }
begin;
  if Channels[nr]^.slidespeed < 0 then
  begin
    inc(Channels[nr]^.Start_Tone, Channels[nr]^.slidespeed);
    if Channels[nr]^.Start_Tone < Channels[nr]^.Targ_Tone then
      Channels[nr]^.Start_Tone := Channels[nr]^.Targ_Tone;
  end else begin
    inc(Channels[nr]^.Start_Tone, Channels[nr]^.slidespeed);
    if Channels[nr]^.Start_Tone > Channels[nr]^.Targ_Tone then
      Channels[nr]^.Start_Tone := Channels[nr]^.Targ_Tone;
  end;
  if Channels[nr]^.Start_Tone < 1 then
    Channels[nr]^.Start_Tone := 1;
  Channels[nr]^.Frequency :=
  longint(Voice_Base[14] div Channels[nr]^.Start_Tone);
  u_VoiceFreq(nr, Channels[nr]^.Frequency);
  oldv[nr] := Channels[nr]^.Start_Tone;
end;
-------------------------------
4 : begin;  (* vibrato  *new* *)
   effect_vibrato(li);
end;
5 : begin;
   ( volume slide )
   inc(Channels[li]^.volume,Channels[li]^.vslide);
   if Channels[li]^.volume < 0 then Channels[li]^.volume := 0;
   if Channels[li]^.volume > 63 then Channels[li]^.volume := 63;
   u_VoiceVolume(li,Channels[li]^.volume);
   ( Note slide )
   inc(Channels[li]^.Start_Tone,Channels[li]^.slidespeed);
   if Channels[li]^.Start_Tone < 1 then
      Channels[li]^.Start_Tone := 1;
    Channels[li]^.Frequency :=
      longint(Voice_Base[14] div Channels[li]^.Start_Tone);
    u_VoiceFreq(li,Channels[li]^.Frequency);
end;
6 : begin;
   ( volume slide )
   inc(Channels[li]^.volume,Channels[li]^.vslide);
   if Channels[li]^.volume < 0 then Channels[li]^.volume := 0;
   if Channels[li]^.volume > 63 then Channels[li]^.volume := 63;
   u_VoiceVolume(li,Channels[li]^.volume);
   ( vibrato )
   inc(Channels[li]^.vibpos,Channels[li]^.vibx);
   if Channels[li]^.vibpos > 64 then
      dec(Channels[li]^.vibpos);
    vibswap :=
      (VibratoTable[Channels[li]^.vibpos] * Channels[li]^.viby) div 256;
    inc(Channels[li]^.Volume,vibswap);
    if Channels[li]^.Volume < 0 then Channels[li]^.Volume := 0;
    u_VoiceVolume(li,Channels[li]^.volume);
end;
7 : begin;  (* tremolo  *new* *)
   inc(Channels[li]^.vibpos,Channels[li]^.vibx);
   if Channels[li]^.vibpos > 64 then
      dec(Channels[li]^.vibpos);
    vibswap :=
      (VibratoTable[Channels[li]^.vibpos] * Channels[li]^.viby) div 256;
    inc(Channels[li]^.Volume,vibswap);
    if Channels[li]^.Volume < 0 then Channels[li]^.Volume := 0;
    u_VoiceVolume(li,Channels[li]^.volume);
end;
8 : begin;  (* not used !!! *)
end;
$a : begin;  (* Volume sliding  **new* *)
   inc(Channels[li]^.volume,Channels[li]^.vslide);
   if Channels[li]^.volume < 0 then Channels[li]^.volume := 0;
   if Channels[li]^.volume > 63 then Channels[li]^.volume := 63;
   u_VoiceVolume(li,Channels[li]^.volume);
end;
$e : begin;  (* Extended effect command *)
case (Channels[li]^.Operand shr 4) of
9 : begin;  (* Retriggering !!! *)
   if Channels[li]^.Operand and $0f <> 0 then begin
      dec(Channels[li]^.Retrig_count);
    if Channels[li]^.Retrig_count = 0 then begin
end;
These are the essential procedures of the `GUS_MOD` unit. You have seen how to load and play a MOD. When playing is finished, we use procedure `_gus_mod_end` to remove the MOD from memory. This first calls procedure `timerint_reset`, which we have already discussed. The memory is then released by procedure `dispose_mod`. This procedure first stops all GUS channels. Then it loops to free the memory occupied by the patterns, channel information, and instrument data.

```pascal
procedure dispose_mod;
{
Removes a loaded MOD from main memory. The samples are NOT deleted from the GUS.
}
begin;
  for i := 0 to 31 do begin;
    U_StartVoice(i,Stop_Voice);
  end;
  for i := 1 to Vh.Num_Patts do begin;
    dos_freemem(Pattern[i]);
  end;
  for i := 0 to 15 do begin;
    dispose(Channels[i]);
  end;
  for i := 0 to 31 do begin;
    dispose(Instruments[i]);
  end;
end;
procedure _gus_mod_end;
{
Ends MOD output
}
begin;
  timerint_reset;
  dispose_mod;
end;
```

We hope we've covered everything you need to know for programming a MOD player for the GUS. You can add your own enhancements as required. A track for sound effects or a graphic-mode player are two ideas you might have fun developing.

**The GUS TCP Player from CoExistence**

We start here with a GUS player in text mode. To fit as much information on the screen as possible, we use a 50-line screen mode.

The player works rather simply. First, `__gu_init_env` initializes the GUS. If no GUS is found, or if errors occur during initialization, an error message is displayed. This message is created using "The Draw" and
sound support for your programs

Chapter 13

PC underground

saved in PASCAL object format, which is essentially a snapshot of video memory. You can link the object into your own Pascal program and reference the screen by its procedure name. For example, if you have saved the text under the procedure name helptxt, you can display the ANSI screen with the following instructions:

```
move (@helptxt^, ptr($B800, 0)^, 80*50*2)
```

If initialization was successful, memory is reserved for the file-handling structure. This is a component of the fselect unit, located on the companion CD-ROM. Function check_commandline now checks to see whether a file name was given on the command line. If so, it returns the value TRUE. Otherwise, the fselect unit can be used to select one or more files.

Next, _gus_initialize performs additional setup work for the MOD unit, and the ANSI code for the main screen is displayed. Then procedure write_phunliners is called. This procedure reads three lines from the PHUN.TXT file and places them at the bottom of the screen. The lines in the PHUN.TXT file are from BBS-On-Line. The file is a standard editable file.

After outputting the Phunlines, the program loads the MOD file using _gus_modload. You have already learned about this function in detail. Once the MOD is loaded, the file name and other important information (instrument names) are displayed on the screen. When the graphic operations are completed, MOD output begins through _gus_modstart.

Next procedure User is called. This procedure periodically updates the volume bars on the screen and handles any input from the user. When the procedure exits, the current MOD is removed from memory. If the user initially selected the file from the selection menu, the menu is redisplayed. Otherwise, the player stops.

The coding of the player itself is simple. It consists basically of the GUS_MOD unit. The following is the source code listing:

```pascal
{$A+,B-,D+,E+,F+,G+,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+,Y+}
{$M 16384,0,250000}
program gusdemo;

uses crt, dos, design, fselect, gus_mod;

type
  Pphun = ^TPhun;
  TPhun = array[0..799] of string[80];

const
  mod_path = '';
  Program_end : boolean = false;

var
  phun : Pphun;
  phuncount : integer;
  modify_voice : integer;
  i : integer;
  The_files : Pfileselect_struct;
  curr_modnr : integer;

{ Linking of Ansi screens

You can find TCP.PAS on the companion CD-ROM
```
function file_exists(fname : string) : boolean;
{
    Checks to see if requested file exists
}
var dumf : file;
begin;
{${-}
    assign(dumf,fname);
    reset(dumf,1);
{${+}
    if IOResult <> 0 then
        file_exists := false
    else begin;
        file_exists := true;
        close(dumf);
    end;
end;

procedure color_writeln(s : string);
{
    Outputs a string in the TC colors
}
var colpos,li : integer;
begin;
    colpos := 1;
    for li := 1 to length(s) do begin;
        if s[li] = ' ' then colpos := 0;
        inc(colpos);
        case colpos of
            1..2 : begin;
            textcolor(8);
            end;
            3..4 : begin;
            textcolor(2);
            end;
            5.$ff : begin;
            textcolor(10);
            end;
        end;
        write(s[li]);
    end;
end;

procedure write_filename(s : string);
{
    Writes song file name, centered
}
var li,slen : integer;
Chapter 13

Sound Support For Your Programs

begin;
gotoxy(33,13);
while pos("\",s) <> 0 do begin;
delete(s,1,pos("\",s));
end;
slen := length(s);
slen := (15 - slen) div 2;
for li := 1 to slen do s := ' '+s;
write(s);
end;

procedure write_phunliners;
{ Reads three lines from the file "Phun.txt" and displays them.
"Phun.txt" is a normal text file that you can edit as you wish.
}
var tf : text;
begin;
randomize;
if not file_exists('phun.txt') then exit;
assign(tf,'phun.txt');
reset(tf);
phuncount := 0;
{$I+}
if ioresult = 0 then begin;
while not eof(tf) do begin;
readln(tf,phun^[phuncount]);
inc(phuncount);
end;
close(tf);
gotoxy(3,43);
color_writeln(phun^[random(phuncount)]);
gotoxy(3,44);
color_writeln(phun^[random(phuncount)]);
gotoxy(3,45);
color_writeln(phun^[random(phuncount)]);
end;
{$I-}
end;

procedure display_modinfo;
{ Displays instrument names of current module
}
var li : integer;
begin;
textcolor(14);
textbackground(black);
for li := 1 to 16 do begin;
gotoxy(6,17+li);
color_writeln(Instruments[li]^..name);
gotoxy(50,17+li);
color_writeln(Instruments[li+16]^..name);
end;
end;

procedure exit_program;
{ Before leaving the program, a plug for TC WHQ,
the Farpoint Station (04202 76145) ... }
begin;
display_ansi(@call,co80+font8x8);
cursor_off;
repeat until keypressed;
while keypressed do readkey;
cursor_on;
asm mov ax,03; int 10h; end;
halt;
end;

procedure next_mod;
{
Starts output of next selected MOD
}
begin;
_gus_mod_end;
inc(curr_modnr);
if curr_modnr > The_Files^.nofiles then
  curr_modnr := 1;
if not _gus_modload(The_Files^.fn[curr_modnr]) then begin;
  clrscr;
gotoxy(10,10);
  write('Sorry, dude, can\'t handle this MOD file\');
delay(1200);
  exit_program;
end;
display_ansi(@tcpans,co80+font8x8);
cursor_off;
write_phunliners;
write_Filenames(The_Files^.fn[curr_modnr]);
display_modinfo;
fillchar(Play_Channel,14,1);
_gus_modstart;
end;

procedure display_we_are;
{
ANSI output with info about the group THE COEXISTENCE
}
begin;
  display_ansi(@we_are,co80+font8x8);
cursor_off;
repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
cursor_off;
  write_phunliners;
  write_Filenames(The_Files^.fn[curr_modnr]);
display_modinfo;
  end;
procedure display_buy_it;
{
Advertises the book PC Underground
}
begin;
  display_ansi(@buy_it,co80+font8x8);
cursor_off;
repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
cursor_off;
  write_phunliners;
write_filenames(The_Files^\,,fn[curr_modnr]);
display_modinfo;
end;

procedure handle_keys(key1,key2 : char);
{ Reacts to keyboard input from user }
var pchan : byte;
begin;
case key1 of
  #00 : begin;
  case key2 of
    #45 : begin;
      Program_end := true;
    end;
    #72 : begin;
      if modify_voice > 1 then
        dec(modify_voice);
    end;
    #80 : begin;
      if modify_voice < Modinf.Voices then
        inc(modify_voice);
    end;
    #75 : begin; { cursor left }
      runinf.Line := 64;
      dec(runinf.Pattnr,2);
      if runinf.Pattnr < -1 then runinf.Pattnr := -1;
    end;
    #77 : begin; { cursor right }
      runinf.Line := 64;
      inc(runinf.Pattnr);
    end;
  end;
  #27 : begin;
    Program_end := true;
  end;
  #32, 'W', 'w', 'I', 'i' : begin;
    display_we_are;
  end;
  'D', 'd', 'b', 'B' : begin;
    display_buy_it;
  end;
  'L', 'l' : begin;
    chpos[modify_voice] := 1;
    _gus_set_channelpos;
  end;
  'R', 'r' : begin;
    chpos[modify_voice] := 15;
    _gus_set_channelpos;
  end;
  'M',
Sound Support For Your Programs

Chapter 13

PC underground

'm' : begin;
    chpos[modify_voice] := 7;
    _gus_set_channelpos;
end;

'u',

'u' : begin;
    if Modinf.Voices = 4 then
        begin
            chpos[1] := 2;
            chpos[2] := 5;
            chpos[3] := 9;
            chpos[4] := 12;
        end;
    if Modinf.Voices = 8 then
        begin
            chpos[1] := 1;
            chpos[2] := 3;
            chpos[3] := 5;
            chpos[6] := 9;
            chpos[8] := 13;
        end;
    _gus_set_channelpos;
end;

'U',

'U' : begin;
    if chpos[modify_voice] > 1 then
        dec(chpos[modify_voice]);
    _gus_set_channelpos;
end;

',' : begin; { to left }
    if chpos[modify_voice] > 1 then
        dec(chpos[modify_voice]);
    _gus_set_channelpos;
end;

'.' : begin; { to right }
    if chpos[modify_voice] < 15 then
        inc(chpos[modify_voice]);
    _gus_set_channelpos;
end;

'1'..'8' : begin;
    pchan := ord(key1)-48;
    if Play_Channel[pchan] = 1 then begin;
        Play_Channel[pchan] := 0;
        gotoxy(77,2+pchan);
        write('M');
        textcolor(2);
        write('UTE');
    end else begin;
        Play_Channel[pchan] := 1;
        gotoxy(77,2+pchan);
        write('C');
        textcolor(2);
        write('H');
    end;
end;

'n',

'N' : begin;
    next_mod;
end;
end;

procedure screen_update;
const colvals : array[1..35] of byte =
    (08,08,08,08,08,08,02,02,02,02,02,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,05,05,05,05,05,05);
```pascal
var volstr : string[66];
  li : integer;
  outs : integer;
begin;
  { Update volume bars }
  for li := 1 to Modinf.Voices do begin;
    for outs := 1 to round(Runinf.Volumes[li] / 1.78) do begin;
      screen[li+2,37+outs].a := colvals[outs];
    end;
    for outs := round(Runinf.Volumes[li] / 1.78) to 36 do begin;
      screen[li+2,38+outs].a := 7;
    end;
  end;

  { Sets proper background color for arrow }
  for li := 1 to 8 do begin;
    if li = modify_voice then begin;
      screen[2+li,34].a := 05;
      screen[2+li,35].a := 05;
      screen[2+li,36].a := 05;
      screen[2+li,37].a := 05;
    end else begin;
      screen[2+li,34].a := 07;
      screen[2+li,35].a := 07;
      screen[2+li,36].a := 07;
      screen[2+li,37].a := 07;
    end;
  end;

  { Shows runtime MOD information }
  gotoxy(18,14);
  color writeln(Modinf.Title);
  textcolor(7);
  gotoxy(18,16);
  write(runinf.pattnr:3);
  gotoxy(64,16);
  write(runinf.line:3);
  gotoxy(64,15);
  write(64:3);
  gotoxy(18,15);
  write(modinf.Patt_cnt:3);
  gotoxy(60,14);
  write(runinf.speed,' / ',runinf.bpm);
end;

procedure user;
{ Checks keyboard input and updates screen }
var ch1,ch2 : char;
begin;
  repeat
    ch1 := #255;
    ch2 := #255;
    if keypressed then begin;
      ch1 := readkey;
      if keypressed then ch2 := readkey;
```
handle_keys(ch1,ch2);
end;
screen_update;
until Program_end;
end;

procedure display_help;
{
Displays help text. Also shown if no GUS found
}
begin;
display_ansi(@helptxt,co80+font8x8);
cursor_off;
repeat until keypressed;
while keypressed do readkey;
exit_program;
end;

function check_commandline : boolean;
{
Returns true if a module name was given
}
var pst : string;
is_mod : boolean;
li : integer;
retval : boolean;
begin;
retval := false;
for li := 1 to 9 do begin;
pst := paramstr(li);
is_mod := true;
if (pos('-h',pst) <> 0) or (pos('-H',pst) <> 0) or
(pos('-?',pst) <> 0) then
begin;
is_mod := false;
display_help;
end;
if (pst <> '') and is_mod then begin;
if pos('.','pst') = 0 then pst := pst + '.mod';
if file_exists(pst) then  { valid mod }
begin;
inc(The_Files^.nofiles);
The_Files^.fn[The_Files^.nofiles] := pst;
retval := true;
end;
end;
end;
check_commandline := retval;
end;

begin;
cursor_off;
clrscr;
if not _gus_init_env then display_help;
new(The_Files);
new(phun);
The_Files^.path := mod_path;
Chapter 13
Sound Support For Your Programs

The_Files^.Mask := '*.mod';
The_Files^.sx := 24;
The_Files^.sy := 10;
The_Files^.nofiles := 0;
The_Files^.Title := 'Select MOD file !!!';
modify_voice := 1;
for i := 1 to 30 do 
The_Files^.fn[i] := '---';
save_screen;
if not check_commandline then begin;
select_packfiles(The_Files);
repeat
  restore_screen;
  if The_Files^.fn[1] = '---' then exit_program;
  _gus_initialize;
  display_ansi(@tcpans,co80+font8x8);
cursor_off;
write_phunliners;
curr_modnr := 1;
if not _gus_modload(The_Files^.fn[1]) then begin;
  clrscr;
gotoxy(10,10);
  write('Sorry, dude, can't handle this MOD file');
delay(1200);
  exit_program;
end;
write_filenames(The_Files^.fn[1]);
display_modinfo;
fillchar(Play_Channel,14,1);
_gus_modstart;
user;
_gus_mod_end;
dispose(The_Files);
new(The_Files);
The_Files^.path := Mod_path;
The_Files^.Mask := '*.mod';
The_Files^.sx := 24;
The_Files^.sy := 10;
The_Files^.nofiles := 0;
for i := 1 to 30 do 
The_Files^.fn[i] := '---';
Program_end := false;
select_packfiles(The_Files);
until The_Files^.fn[1] = '---';

dispose(The_Files);
dispose(phun);
exit_program;
end else begin;
restore_screen;
if The_Files^.fn[1] = '---' then exit_program;

_gus_initialize;
XM - The sound format of the new generation

Because of its features, XM format is rather complex in structure.

It is divided into the following parts:

The XM header

<table>
<thead>
<tr>
<th>Bytes 0 - 16</th>
<th>XM-ID</th>
</tr>
</thead>
</table>

These bytes contain ID 'Extended module: '. You can identify an XM by this ID.

<table>
<thead>
<tr>
<th>Bytes 17 - 36</th>
<th>Song name</th>
</tr>
</thead>
</table>

These bytes contain the name of the song. It will be filled up with 0 bytes if it doesn't exceed the full 20 characters or isn't even specified.

<table>
<thead>
<tr>
<th>Byte 37</th>
<th>ID 1Ah</th>
</tr>
</thead>
</table>

Byte 37 serves as another ID and always has the value 1Ah.

<table>
<thead>
<tr>
<th>Bytes 38 - 57</th>
<th>Name of tracker</th>
</tr>
</thead>
</table>

The name of the tracker with which the XM was generated is located here. Like the song name, the name of the tracker is also filled with 0 bytes.

<table>
<thead>
<tr>
<th>Bytes 58 - 59</th>
<th>Bytes 58 - 59</th>
</tr>
</thead>
</table>

Byte 58 contains the main version number, while the sub version number of the XM is stored in byte 59. The current XM format is 1.3, so you will find the value 0103h here.
Chapter 13

Sound Support For Your Programs

<table>
<thead>
<tr>
<th>Bytes 60 - 63</th>
<th>Size of the header</th>
</tr>
</thead>
</table>

The 4 bytes are to be interpreted as a DWord. They indicate the size of the header in bytes. This number of bytes must be read when reading(in) the header.

The following position specifications apply to the size of the header relative to the position of the value. This structure pervades the entire XM format and makes a higher flexibility possible.

<table>
<thead>
<tr>
<th>+4 - Word</th>
<th>Length of the song</th>
</tr>
</thead>
</table>

This is where the number of the defined pattern in the arrangement (pattern table) is specified.

<table>
<thead>
<tr>
<th>+6 - Word</th>
<th>Restart Position</th>
</tr>
</thead>
</table>

The restart position indicates the place in the arrangement to jump to when the song reaches the end.

<table>
<thead>
<tr>
<th>+8 - Word</th>
<th>Number of Channels</th>
</tr>
</thead>
</table>

The number of channels of the song is defined in this word. Between 2 and 32 channels are possible.

<table>
<thead>
<tr>
<th>+10 - Word</th>
<th>Number of Patterns</th>
</tr>
</thead>
</table>

This word contains the number of defined patterns. This number is very important, because a corresponding number of patterns must be read subsequently. A maximum of 256 patterns can be defined.

<table>
<thead>
<tr>
<th>+12 - Word</th>
<th>Number of Instruments</th>
</tr>
</thead>
</table>

This word is also quite important. It contains the number of defined instruments. We’ll go into the structure of these instruments a bit later. They must also be read from the XM.

<table>
<thead>
<tr>
<th>+14 - Word</th>
<th>Flags</th>
</tr>
</thead>
</table>

At present only bit 0 is defined in the flags. If this bit has the value 0, it means only Amiga frequencies are to be used. Otherwise a linear frequency structure is used.

<table>
<thead>
<tr>
<th>+16 - Word</th>
<th>Default Speed</th>
</tr>
</thead>
</table>

This word contains the starting speed of the song.

<table>
<thead>
<tr>
<th>+18 - Word</th>
<th>Default BPM</th>
</tr>
</thead>
</table>

This word contains the BPM starting value of the XM.

<table>
<thead>
<tr>
<th>+20 - 256 Bytes</th>
<th>Arrangement</th>
</tr>
</thead>
</table>

The sequence in which the patterns are to be played is stored in the arrangement, or pattern table. The maximum length of an XM song amounts to 256 patterns.
Individual patterns

The individual patterns follow the header. They don’t have a fixed length since the information is compressed by a simple, but effective compression system, and the number of lines in the pattern are specified per pattern. A pattern always consists of a pattern header with the necessary head information and the subsequent compressed data. The following shows the structure of a pattern:

<table>
<thead>
<tr>
<th>+0 - DWord</th>
<th>Pattern size</th>
</tr>
</thead>
</table>

The size of the pattern header in bytes.

<table>
<thead>
<tr>
<th>+4 - Byte</th>
<th>Compression Type</th>
</tr>
</thead>
</table>

At present Compression Type always has the value 0. If you find a different value, it means that the gurus at Triton have integrated a new compression type or else you’ve got an error in the read procedure.

<table>
<thead>
<tr>
<th>+5 - Word</th>
<th>Number of Lines</th>
</tr>
</thead>
</table>

This word contains the number of lines in this pattern. This is important during runtime of the player. Between 1 and 256 lines are possible.

<table>
<thead>
<tr>
<th>+7 - Word</th>
<th>Size of compressed data</th>
</tr>
</thead>
</table>

This word contains the size of the compressed data in bytes. These must be read after the header.

<table>
<thead>
<tr>
<th>+9 - ?? Byte</th>
<th>Compressed data</th>
</tr>
</thead>
</table>

You will find the pattern in compressed form here. We recommend that you always wait until runtime before unpacking it, since this will save you some disk space.

Structure of the compressed data

If the highest bit (MSB) is set in the read value byte, interpret the byte as compression information. In this case, the following applies:

<table>
<thead>
<tr>
<th>Bit 0 is set</th>
<th>Note follows</th>
<th>Bit 3 is set</th>
<th>Effect follows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 1 is set</td>
<td>Instrument follows</td>
<td>Bit 4 is set</td>
<td>Effect parameter follows</td>
</tr>
<tr>
<td>Bit 2 is set</td>
<td>Byte for volume column follows</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Otherwise the read byte is a note, and the four values from Instrument to Effect parameter follow. The individual bytes can have the following values:

<table>
<thead>
<tr>
<th>Note:</th>
<th>Effect:</th>
<th>Volume column:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 71, 0 stands for C-0</td>
<td>Effect Parameter</td>
<td>You’ll find between 0 and FFh an exact breakdown in the description of the effects.</td>
</tr>
<tr>
<td>Effect Parameter</td>
<td>Between 0 and “T”, see below</td>
<td>Between 0 and FFh</td>
</tr>
</tbody>
</table>
The Instruments

The patterns in XM are followed by the instruments. An instrument consists of a header (general header data) and, if any samples are available, of general instrument data and sample data.

Here is the structure of the instrument header:

+0 - DWord  Instrument Size
This DWord contains the size of the instrument in bytes. It applies to the pure instrument without the sample data.

+4 - 22 Char  Instrument Name
Contains the name of the instrument. If the name doesn't take up the entire length, the 22 bytes are filled with 0 bytes.

+26 - Byte  Instrument Type
At present the instrument type always has a value of 0. It is conceivable that other instrument types will be added, especially in view of Midi or parallel operated sound cards.

+27 - Word  Number of Samples
This word contains the number of samples in the instrument. If it is greater than 0, general instrument header data follows, and then one or more samples. Only then do the following bytes get read, otherwise you have to continue with the next instrument.

+29 - DWord  Size of Additional Instrument Info
The value gives the size of additional info on the instrument in bytes.

+33 - 96 Bytes  Sample Table
This table contains the sample numbers for each pitch. For example, each octave can have its own assigned sample. Perhaps you are already familiar with this principle from the patches of the GRAVIS ULTRASOUND.

+129 - 12 EnvPoints  Volume Envelope
Definition of the volume envelopes. An EnvPoint is made up of two words. The first word stores the x-position of the point, while the second word stores its (Y) value.

+177 - 12 EnvPoints  Panning Envelope
Here are the points for the panning envelope. While the values of 0 to 64 for the volume envelope are to be understood as the volume, the panning values are to be seen in their relation to the middle (32). That is, a value of 0 means a sound all the way to the left, while 64 is a sound all the way to the right.
Number of Volume Points

Only in rare cases are all of the volume points used. You will find the number of points that were used here.

Number of Panning Points

Contains the number of panning points.

Volume Sustain Point

This byte specifies the point at which processing of the envelope stops until a Key Off note is struck for the instrument.

Volume Loop Starting Point

It's possible to have a loop within the envelope. This point specifies the starting point within the loop. The value is the number of the envelope point to use, and does not specify the x-position.

Volume Loop End Point

This point specifies the end of the loop within the envelope.

Panning Sustain Point

The same as what we said for Volume Sustain Point, only for the Panning Sustain Point.

Panning Loop Starting Point

This value behaves similar to the way the Volume Loop Starting Point acts.

Panning Loop End Point

This value is similar to the Volume Loop End Point.

Volume Type

This byte defines whether a volume envelope is to be used and whether it contains a sustain point or looping. The following applies:

| Bit 0 set | Use envelope |
| Bit 1 set | Use sustain point |
| Bit 2 set | Use envelope looping |

Panning Type

This byte defines whether a panning envelope is to be used, as well as whether it contains a sustain point or looping. The following applies:
Chapter 13

Sound Support For Your Programs

<table>
<thead>
<tr>
<th>Bit 0 set</th>
<th>Use envelope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 1 set</td>
<td>Use sustain point</td>
</tr>
<tr>
<td>Bit 2 set</td>
<td>Use envelope looping</td>
</tr>
</tbody>
</table>

+235 - Byte  
**Vibrato Type**

You can set a general vibrato type here.

+236 - Byte  
**Vibrato sweep**

Sweeping of general vibrato

+237 - Byte  
**Vibrato depth**

This byte specifies the intensity of the general vibrato.

+238 - Byte  
**Vibrato Rate**

The speed of the general vibrato

+239 - Word  
**Volume fadeout**

The speed at which the struck instrument fades out is stored here.

+241 - Word  
**Reserved**

Sample header

After the additional info about the instruments comes the sample data. The sample data has the same structure - first the header, then the data. If you have several samples, you must first load all the sample headers and then the sample data.

The sample headers have the following structure:

+0 - DWord  
**Sample Length**

You will find the length of the sample data in bytes here. This specification also applies to 16 bit data, which aren’t stored in word lengths.

+4 - DWord  
**Loop Start**

If the sample is looped the start position of the loop will be located in this DWord. Keep in mind that this is a DWord value, since the samples can be even larger than 64K.

+8 - DWord  
**Loop Length**

This value specifies the length of the loop. The end position of the loop is calculated from the Loop Start and Loop Length.
Sound Support For Your Programs

### +12 - Byte  Volume

The default volume of the instrument is stored here. Values can range between 0 and 64.

### +13 - Byte  Fine Tune

You can set the fine tuning of the instrument here. Values range between -16 and +15.

### +14 - Byte  Sample Type

The sample type indicates whether the sample is looped and whether the data is 8-bit or 16-bit.

<table>
<thead>
<tr>
<th>Bit 0-1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No looping</td>
</tr>
<tr>
<td>1</td>
<td>Normal forward looping</td>
</tr>
<tr>
<td>2</td>
<td>Ping Pong looping (forward and back)</td>
</tr>
<tr>
<td>4</td>
<td>16 bit data, otherwise 8 bit data</td>
</tr>
</tbody>
</table>

### +15 - Byte  Panning

The panning value specifies the default panning position. Possible values range between 0 and 255.

### +16 - Byte  Relative Note

Specifies which pitch the sample applies to. The value is used to adapt samples to each other.

### +17 - Byte  Reserved

### +18 - 22 Char  Sample Name

Name of the sample, if necessary, filled with 0 bytes.

---

**Sample data**

After the sample headers comes the actual sample data. You can’t immediately play back this data, instead, it is Delta compressed. That means that you have an initial value, and then the differences to the next value. This has the advantage of generating more redundant data that can be compressed better.

---

**SFX Pro**

SFX Pro is one of the first XM players published. One very important concept during its development was that SFX Pro was not to be just another player. Instead, it was to serve as the basis of a sound system for games and for demos. That’s also why the timing module works with a constantly running timer and not with variable BPMs. This greatly simplifies handling, and there won’t be any speed differences between slow and high BPM rates.
Chapter 13

Sound Support For Your Programs

However, that means that synchronization by means of a constantly incremented Sync Counter would no longer be possible. And it is precisely synchronization that is most important for a demo, since a good demo must run on a slow computer at the same speed as on a Pentium 90MHz.

The programming language was another important point. Until now we've concentrated in Pascal and assembly. However, SFX Pro is not written in either Pascal or assembly for many reasons. The most important reason may be that with Pascal, you can only manage 64K segments. However, XMs have samples that can be larger than 64K.

If you write a true GUS player, you may still be able to get around this problem. SFX Pro is designed in such a way that you can integrate support for other sound cards with a minimum of time and effort. For example, for the SoundBlaster all you need to do is integrate a mixing procedure and fine tune the frequency and volume handling of the GUS procedures accordingly. If you're working with 64K segments, you'll encounter problems since the management overhead is tremendous.

Another point was the question "which language are all the serious game developers now using?". A frequency answer is Watcom C because it's probably the best compiler for DOS systems, both with regard to optimization as well as memory management. The P-MODE interface from TRAN is also good and saves you the trouble of using the Rational Systems DOS Extender. Although Watcom C may not enjoy the extensive distribution of a language like Borland C, the compiler is very affordable (under $300).

Determining frequency in the XM Module

Enough of the introduction, now let's take a closer look at what makes SFX Pro tick. First on the agenda is the question: How are the frequencies determined? We'll use Triton's formula for determining the current period as a basis. The function getlinearperiod calculates the period to be used.

```c
unsigned int getlinearperiod(char note,unsigned int fine)
{
    return((10L*12*16*4)-((unsigned int)note*16*4)-(fine/2)+128);
}
```

When you have determined the period to use (and if necessary, changed it with effects), you can define the frequency to be played. We'll use GetFreq2 to determine the frequency we actually need. This solution, devised by MikMak, is somewhat faster than calculation according to the original Triton formula, which is based on powers.

```c
int GetFreq2(int period)
{
    int okt;
    long frequency;
    period=7680-period;
    okt=period/768;
    frequency=lintab[period%768];
    frequency<<=2;
    return(frequency>>(7-okt));
}
```

In the player itself, the get_freq function is used mainly. First the function determines the note to be used. The reason this is so important is that the note can be 0 in XM (since we're in the next line) and we must then
resort to the last used note. Then `getlinearperiod` determines the period and `GetFreq2` gets the frequency. Then the function returns the frequency.

```c
unsigned int get_freq(unsigned short int channel)
{
    unsigned short int note;
    int RealNote,period,FineTune,Frequency;

    if (Channel[channel].note == 0) {
        RealNote = Channel[channel].last_note +
        insts[Channel[channel].inst_no].sheader[Channel[channel].smpno]->relative_note;
    } else {
        RealNote = Channel[channel].note +
        insts[Channel[channel].inst_no].sheader[Channel[channel].smpno]->relative_note;
    }

    FineTune = Channel[channel].finetune;
    period = getlinearperiod(RealNote,FineTune);
    Frequency = GetFreq2(period);
    return(Frequency);
}
```

**Converting XM samples**

Another basic routine is `convert_sample` or `convert_sample16`. This routine translates the Delta compressed sample data (wave information) into the "genuine" sample data required by sound cards. To do this, the routine uses the following procedure: it gets the first value, which is the base value for further calculations. To get the genuine data, add the next value from the read sample data to this base value. The value resulting from the addition becomes the new base value and can be written back.

```c
void convert_sample(signed char *s,unsigned int length)
{
    unsigned int n;
    int new,ltmp;
    signed char old,tmp;
    signed char *tmpptr;

    tmpptr = s;
    old = *tmpptr;
    tmpptr++;
    for (n = 1; n <= length; n++) {
        tmp = *tmpptr;
        ltmp = tmp;
        new = ltmp + old;
        tmp = new;
        *tmpptr = tmp;
        old = tmp;
        tmpptr++;
    }
}
```
Chapter 13

Sound Support For Your Programs

**XM load routine**

Now let's get to the interesting part, namely the load routine for XMs. If an error occurs during loading, SFX Pro displays an error message. The texts that are output are defined in the `sfxerror` function.

The function first opens the XM file and then reads in the XM header. Next it reads in the pattern. In so doing, the routine allocates memory for the compressed pattern data and reads the data, without converting it. The data is unpacked at runtime to save memory.

The instruments can be loaded once the patterns are in memory. There's one point to remember: First, the instrument header, 29 bytes in size, is read. The remaining data must be read if the number of the available samples is greater than zero. Otherwise, read in the rest of the header and continue with the next instrument. If you have sample data, read the complete header and then complete the samples.

First, read the sample headers in a loop. It's possible that not every sample will be allocated. For example, may be only the 2nd, 4th and 5th samples will be allocated. These are the ones you need to intercept. Load the sample data in a loop. After reading the actual wave data, the load procedure gets the loop type of the sample (none, normal, ping pong ?) and the type of sample data (8/16 bit ?). `convert_sample` (if the data is 8-bit) or `convert_sample16` (if the data is 16-bit) transforms the data into the data format required by the sound card. Then the data is transferred to the GUS RAM.

Finally, the routine initializes the song position and sets the pattern to be played to the first position in the song. The following is the source code for the routine:

```c
void load_module(char *modulename)
{
    int slength;
    char far *tmpptr;
    char buffer[80];

    GusDramStart = 0;
    if((file = fopen(modulename,"rb")) == NULL)
    {
        sfxerror(1);
    }

    fread(&xmh,sizeof(xmh),1,file);
    xmh.xm_id[16] = 0;
    xmh.module_name[19] = 0;
    xmh.tracker_name[19] = 0;

    //  ======================================================================
    //    Read in Pattern
    //  ======================================================================
    for (i = 0; i < xmh.num_patterns; i++) {
        fread(&patts[i],9,1,file);
        if (patts[i].packed_size > 0) {
            if(( patts[i].packed_data = malloc(patts[i].packed_size+64)) == NULL)
            {
                sfxerror(2);
            }
            fread(patts[i].packed_data,patts[i].packed_size,1,file);
        }
        memset(buffer,0,80);
        sprintf(buffer,"[SFXPRO] Pattern %d loaded ... ",i);
    }
}
```
myprint(buffer,7,22,40);
}

// Load Instruments
for (i = 0; i < xmh.num_instruments; i++) {
    sprintf(buffer,"[SFXPRO] Loading Instrument %d ...",i);
    myprint(buffer,7,22,40);
    fread(insts[i],29,1,file);
    if (insts[i].num_samples == 0) {
        memset(buffer,0,80);
        sprintf(buffer,"[SFXPRO] no instrument data");
        myprint(buffer,7,22,40);
        if ((insts[i].iheader = malloc(insts[i].inst_size-29+64)) == NULL) {
            sfxerror(3);
        }
        fread(insts[i].iheader,1,insts[i].inst_size-29,file);
    } else {
        // Allocate & read instrument header
        if ((insts[i].iheader = malloc(insts[i].inst_size-29+64)) == NULL) {
            sfxerror(3);
        }
        fread(insts[i].iheader,1,insts[i].inst_size-29,file);
        for (j = 0; j < insts[i].num_samples; j++) {
            // Sample - allocate & read instrument header
            if ((insts[i].sheader[j] = malloc(sheadersize+64)) == NULL) {
                sfxerror(3);
            }
            fread(insts[i].sheader[j],insts[i].iheader->inst_hsize,1,file);
        } // End For loop
        for (j = 0; j < insts[i].num_samples; j++) {
            // Allocate & read sample instrument data
            slength = insts[i].sheader[j]->sample_length + 64;
            if ((sampdata[i][j] = malloc(slength)) == NULL) {
                sfxerror(4);
            }
        }
    }
} // End loop
Chapter 13

Sound Support For Your Programs

```
};

fread(sampdata[i][j],insts[i].sheader[j]->sample_length,1,file);

insts[i].sheader[j]->Loop_Mode = 0;

// Get loop type
switch (insts[i].sheader[j]->type & 3) {
  case 0: break;
  case 1: insts[i].sheader[j]->Loop_Mode += 8;
        break;
  case 2: insts[i].sheader[j]->Loop_Mode += 8;
        insts[i].sheader[j]->Loop_Mode += 16;
        break;
}

if ((insts[i].sheader[j]->type & 16) == 16) {
  insts[i].sheader[j]->Loop_Mode += 4;
  convert_sample16(sampdata[i][j],insts[i].sheader[j]->sample_length);
  insts[i].sheader[j]->Is_16Bit = 1;
} else {
  convert_sample(sampdata[i][j],insts[i].sheader[j]->sample_length);
  insts[i].sheader[j]->Is_16Bit = 0;
}

insts[i].sheader[j]->Gus_Start = GusDramStart;

if (insts[i].sheader[j]->Loop_Mode > 0) {  // looped
  mem_2_gus(i,j,insts[i].sheader[j]->sample_length,insts[i].sheader[j]->sample_loop_start);
} else {
  mem_2_gus(i,j,insts[i].sheader[j]->sample_length,0);
}

// End reading data
// end for
// end load instrument

Songpos = 0;
NewPattern(xmh.pat_table[Songpos]);
```

**Playing (back ) XM patterns**

First, we need a pointer that points to the packed data. Move this pointer forward during playback. Now process the pattern, row by row by using a loop. It should run for all the defined channels (2 to 32). Read the first byte and determine if there is packed data or if you can read 5 bytes in a row. The `process_patternrow` routine determines which bytes are to be read (note, instrument, volume, effect, parameter) and sets the appropriate info flags (p_note to p_para). If the data is not compressed, simply set all the flags and the position pointer remains at its initial value.

Depending on the flags, the bytes to be used are determined and the pointer is moved up accordingly. In this case, the handling of the note values is the only important item. A keyoff note is a value greater than or equal to 95. The keyoff flag for the channel must be modified accordingly.
Also, check whether effect 3 or effect 5 is active (Portamento to Note). In this case, don’t play the note directly. Instead, interpret it as a destination frequency. The `replay_note` and `replay_freq` flags govern if a note is to be played and if its frequency is to be changed. If these flags have a value of 1, the voice parameters are reset or the frequency is changed.

The next step after specifying the bytes for the current channel is to handle the values of the volume column and the effects by using `process_volume_effects` and `process_effects`. We’ll explain these two functions later. After calling these two functions, if necessary you can reset the parameters for the voice and change the frequency.

This completes the actual loop. Now all you have to do is move the row forward. Load the next pattern (or end the song) if the row lands at the end of the pattern or if you have a pattern break.

```c
void process_patternrow()
{
    char Note;
    char dummy;
    int  tmp;
    unsigned long begin; /* start location in ultra DRAM */
    unsigned long start; /* start loop location in ultra DRAM */
    unsigned long end;  /* end location in ultra DRAM */
    unsigned char mode; /* mode to run the voice (loop etc) */
    char s_note;
    char s_inst;
    char s_vol;
    char s_eff;
    char s_para;

    for (i = 0; i < xmh.num_channels; i++) {
        Channel[i].note = 0;
        Channel[i].param = 0;
        Channel[i].veffect = 0;
        Channel[i].effect = 255;
        p_note = 0;           // Set Defaults
        p_inst = 0;
        p_vol = 0;
        p_eff = 0;
        p_para = 0;
        replay_note = 0;
        replay_freq = 0;

        Note = *pos_patt;
        pos_patt ++;

        if (Note >= 128) {    // Pack - Info
            if ((Note & 1) == 1) { // Note follows
                p_note = 1;
            }
            if ((Note & 2) == 2) { // Instrument follows
                p_inst = 1;
            }
            if ((Note & 4) == 4) { // Volume follows
                p_vol = 1;
            }
            if ((Note & 8) == 8) { // Effect follows
                p_eff = 1;
            }
            if ((Note & 16) == 16) { // Parameter follows
                p_para = 1;
            }
        }
    }
}
```
\begin{verbatim}
else
{
    pos_patt --;
    p_note = 1;
    p_inst = 1;
    p_vol = 1;
    p_eff = 1;
    p_para = 1;
}

if (p_note == 1)
{
    s_note = *pos_patt;
    pos_patt ++;
}

if (p_inst == 1)
{
    if (*pos_patt <= xmh.num_instruments)
    {
        Channel[i].inst_no = *pos_patt -1;
        pos_patt ++;
        if (insts[Channel[i].inst_no].num_samples > 0)
        {
            Channel[i].smpno = insts[Channel[i].inst_no].iheader->samp_numbers[s_note-1];
            Channel[i].volume = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->volume;
            Channel[i].finetune = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->finetune;
            Channel[i].vibpos = 0;
            Channel[i].Wait_Keyoff = 0;
            Channel[i].is_pan_env = (insts[Channel[i].inst_no].iheader->pan_type & 1);
            if (Channel[i].is_pan_env == 1)
            {
                Channel[i].is_pan_sus = (insts[Channel[i].inst_no].iheader->pan_type & 2) - 1;
                Channel[i].is_pan_loop = (insts[Channel[i].inst_no].iheader->pan_type & 4) - 3;
                Channel[i].pan_pt1 = 0;
                Channel[i].pan_pt2 = 1;
                Channel[i].pan_pos = 0;
                Channel[i].pan_val = insts[Channel[i].inst_no].iheader->pan_envelope[1];
            }
            else
            {
                Channel[i].pan_val = 32;
            }
            Channel[i].is_vol_env = (insts[Channel[i].inst_no].iheader->vol_type & 1);
            if (Channel[i].is_vol_env == 1)
            {
                Channel[i].is_vol_sus = (insts[Channel[i].inst_no].iheader->vol_type & 2) - 1;
                Channel[i].is_vol_loop = (insts[Channel[i].inst_no].iheader->vol_type & 4) - 3;
                Channel[i].vol_pt1 = 0;
                Channel[i].vol_pt2 = 1;
                Channel[i].vol_pos = 0;
                Channel[i].vol_val = insts[Channel[i].inst_no].iheader->vol_envelope[1];
            }
            else
            {
                Channel[i].vol_val = 64;
            }
        }
        else // no sample data
        {
            Channel[i].smpno = 0;
            Channel[i].volume = 0;
            Channel[i].finetune = 0;
            Channel[i].vibpos = 0;
        }
    }
}
\end{verbatim}
Channel[i].Wait_Keyoff = 0;
Channel[i].is_pan_env = 0;
Channel[i].pan_val = 32;
Channel[i].is_vol_env = 0;
UltraSetDirectVolume(i,0);
replay_note = 0;
replay_freq = 0;
}
else {
    pos_patt ++;
}
}
if (p_vol == 1) {
    Channel[i].veffect = *pos_patt;
    pos_patt ++;
}
if (p_eff == 1) {
    Channel[i].effect = *pos_patt;
    pos_patt ++;
}
if (p_para == 1) {
    Channel[i].param = *pos_patt;
    pos_patt ++;
}
if (p_note == 1) {
    Channel[i].dest_note = Channel[i].last_note;
    Channel[i].note = s_note-1;
    if (Channel[i].note >= 95) {
        if (Channel[i].is_vol_sus == 1) {   //  Keyoff active ?
            Channel[i].is_vol_sus = 0;
            Channel[i].Wait_Keyoff = 0;
        }
        else {
            Channel[i].vol_val = 0;
            UltraSetDirectVolume(i,0);
            replay_note = 0;
            replay_freq = 0;
        }
    }
    else {
        if ((Channel[i].effect == 3) || (Channel[i].effect == 5)) {
            tmpp = Channel[i].period;
            Channel[i].dest_period = get_freq(i);
            Channel[i].note = Channel[i].dest_note;
            Channel[i].period = tmpp;
        }
        else {
            Channel[i].period = get_freq(i);
        }
    }
}
Channel[i].last_note = Channel[i].note;
Channel[i].ret_count = Channel[i].param & 0x0f;
replay_note = 1;
replay_freq = 1;
}
process_volume_effects(i);
process_effects(i);

if ((Channel[i].note < 120) && (Channel[i].note > 0)) {
    if (replay_note == 1) {
        begin = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start;
        start = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_start;
        if (insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length > 0) {
            end = start + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length;
        } else {
            end = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_length;
        }
        mode = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Loop_Mode;
        UltraPrimeVoice(i, begin,start,end,mode,Channel[i].volume * Channel[i].vol_val / 64, insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Is_16Bit);
    } else {
        UltraPrimeVoice(i, begin,start,end,mode,Channel[i].volume * Channel[i].vol_val / 64, insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Is_16Bit);
        UltraSetFrequency(i,GetFreq2(Channel[i].period));
    }
} // end for

Row ++;

if (Break_The_Pattern == 1) {
    if (Songpos > xmh.song_length-1) {
        Songpos = xmh.restart_position;
    }
    NewPattern(xmh.pat_table[Songpos]);
}

if (Row > (maxrows -1)) {
    Songpos ++;
    if (Songpos > xmh.song_length-1) {
        Songpos = xmh.restart_position;
    }
    NewPattern(xmh.pat_table[Songpos]);
}

Break_The_Pattern = 0;
row_changed = 1;
}
Volume effects

One special feature of XM format is that it doesn’t just offer the “normal” effects, you can also enter effects in the Volume Column. These effects are handled in the player by the `process_volume_effects` function. Its effect is determined by the size of the value.

Now the routine checks which value range the entry is in and executes the corresponding effect. Since the internal handling of the effects runs the way it does with other players, we’ll only discuss some special features. First, let’s look at what each of the values means:

<table>
<thead>
<tr>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>10h - 50h</td>
<td>Set Volume</td>
</tr>
<tr>
<td>60h - 6Fh</td>
<td>Volume slide down</td>
</tr>
<tr>
<td>70h - 7Fh</td>
<td>Volume slide up</td>
</tr>
<tr>
<td>80h - 8Fh</td>
<td>Fine volume slide down</td>
</tr>
<tr>
<td>90h - 9Fh</td>
<td>Fine volume slide up</td>
</tr>
<tr>
<td>A0h - AFh</td>
<td>Set vibrato speed</td>
</tr>
<tr>
<td>B0h - BFh</td>
<td>Vibrato</td>
</tr>
<tr>
<td>C0h - CFh</td>
<td>Panning</td>
</tr>
</tbody>
</table>

- A value between 10h and 50h is not an “effect” in the true sense of the word. Instead, the value indicates the volume at which the voice is to be played.
- Normal volume sliding is called at each tick. The volume is decreased by the value in the range.
- Similar to volume slide down, here the volume is increased with each tick by the specified value. Remember not to exceed the maximum value for the volume (64).
- Unlike normal volume sliding, fine volume sliding isn’t called with each tick but only once.
- Increases the volume once by the specified value. Remember not to exceed the maximum value.
- Use this value to set the vibrato speed for the channel. Assign a value of A0h through AFh to the speed.
- This effect executes a vibrato with the specified depth. The depth is calculated according to the value in this range.
- Panning sets the current panning position. Since the player works internally with values from 0 to 64, you have to multiply the value for C0h - CFh times 4 to get the correct panning position.
Panning slide left is well suited for stereo effects. It causes the panning position to shift to the left by the specified value - D0h. For a panning slide to the left you have to subtract the passed value from the current panning value.

Like panning slide left, this effect causes the current panning position to shift to the right.

In addition to normal portamento, you can also execute a portamento to note using this effect. Use the passed value for the range F0h - FFh to calculate the speed of the portamento.

```c
def process_volume_effects(unsigned short int i)
{
    char param;
    // Set volume
    if ((Channel[i].veffect >= 0x10) && (Channel[i].veffect <= 0x50)) {
        Channel[i].volume = Channel[i].veffect - 0x10;
    }
    // Volume slide down
    if ((Channel[i].veffect >= 0x60) && (Channel[i].veffect <= 0x6f)) {
        param = Channel[i].veffect - 0x60;
        Channel[i].volume -= param;
        if (Channel[i].volume < 0) {
            Channel[i].volume = 0;
        }
        UltraSetDirectVolume(i,Channel[i].volume * Channel[i].vol_val / 64);
    }
    // Volume slide up
    if ((Channel[i].veffect >= 0x70) && (Channel[i].veffect <= 0x7f)) {
        param = Channel[i].veffect - 0x70;
        Channel[i].volume += param;
        if (Channel[i].volume > 0) {
            Channel[i].volume = 64;
        }
        UltraSetDirectVolume(i,Channel[i].volume * Channel[i].vol_val / 64);
    }
    // FINE Volume slide down
    if ((Channel[i].veffect >= 0x80) && (Channel[i].veffect <= 0x8f)) {
        param = Channel[i].veffect - 0x80;
        Channel[i].volume -= param;
        if (Channel[i].volume < 0) {
            Channel[i].volume = 0;
        }
        UltraSetDirectVolume(i,Channel[i].volume * Channel[i].vol_val / 64);
    }
    // FINE Volume slide up
```
if ((Channel[i].veffect >= 0x90) && (Channel[i].veffect <= 0x9f)) {
    param = Channel[i].veffect - 0x90;
    Channel[i].volume += param;
    if (Channel[i].volume > 64) {
        Channel[i].volume = 64;
    }
    UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
    Channel[i].veffect = 0;
}

// Set vibrato speed
if ((Channel[i].veffect >= 0xa0) && (Channel[i].veffect <= 0xaf)) {

}

// vibrato
if ((Channel[i].veffect >= 0xb0) && (Channel[i].veffect <= 0xbf)) {

}

// panning
if ((Channel[i].veffect >= 0xc0) && (Channel[i].veffect <= 0xcf)) {
    param = Channel[i].veffect - 0xc0;
    Channel[i].pan_val = param << 2;
    UltraSetPanning(i, Channel[i].pan_val);
    Channel[i].veffect = 0;
}

// panning slide left
if ((Channel[i].veffect >= 0xd0) && (Channel[i].veffect <= 0xdf)) {
    param = Channel[i].veffect - 0xd0;
    if ((Channel[i].pan_val - param) > 0) {
        Channel[i].pan_val -= param;
    } else {
        Channel[i].pan_val = 0;
    }
    UltraSetPanning(i, Channel[i].pan_val);
    Channel[i].veffect = 0;
}

// panning slide right
if ((Channel[i].veffect >= 0xe0) && (Channel[i].veffect <= 0xef)) {
    param = Channel[i].veffect - 0xe0;
    if ((Channel[i].pan_val + param) < 64) {
        Channel[i].pan_val += param;
    } else {
        Channel[i].pan_val = 64;
    }
    UltraSetPanning(i, Channel[i].pan_val);
    Channel[i].veffect = 0;
}

// tone portamento
if ((Channel[i].veffect >= 0xf0) && (Channel[i].veffect <= 0xff)) {

}
XM Module effects

Next we'll talk about the default effects which are handled by `process_effects` in the player.

**Effect 01h**  
*Portamento up*

Increases the pitch during portamento, thus reducing the value for the current period. The value specified in the parameter is the intensity of this change.

**Effect 02h**  
*Portamento down*

This effect is like portamento up, only with the opposite sign.

**Effect 03h**  
*Tone Portamento*

Executes a change in the pitch which starts from the current note to the note specified with the effect. The `DoTonePortamento` function controls this effect.

```c
void DoTonePortamento(char i) {
    int dtp;
    char para;

    if (Channel[i].param != 0) {
        para = Channel[i].param << 2;
        Channel[i].last_param = Channel[i].param;
    } else {
        para = Channel[i].last_param << 2;
    }

    dtp = Channel[i].period - Channel[i].dest_period;

    if ((dtp == 0) || (para > abs(dtp))) {
        Channel[i].period = Channel[i].dest_period;
    } else {
        if (dtp > 0) {
            Channel[i].period -= para;
        } else {
            Channel[i].period += para;
        }
    }
}
```

**Effect 04h**  
*Vibrato*

Executes a vibrato for the current voice. The speed at which the vibrato is to be executed is located in the upper 4 bits of the parameter. The lower 4 bits contain the depth (amplitude) with which the vibrato is to be sounded.

If one of the two values is zero, the last specified value will be used. The `DoVibrato` function executes the actual vibrato. This function changes the period of the voice accordingly.
void DoVibrato(char i)
{
    char q;
    int tmp;

    q = (Channel[i].vibpos >> 2) & 64;
    tmp = VibratoTable[q];

    tmp = tmp * Channel[i].vib_depth;
    tmp >>= 7;
    tmp <<= 2;

    Channel[i].period += tmp;
    Channel[i].vibpos += Channel[i].vib_rate;
}

Effect 05h  Tone Portamento & Volume Slide

Combines effects 3 and 0Ah. The parameter specifies the speed and direction of volume sliding. The effect uses the DoTonePortamento function and adopts the function of effect 0Ah.

Effect 06h  Vibrato & Volume Slide

This is also a combined effect. The parameters also apply to volume sliding. The vibrato runs at its set values with the help of the DoVibrato function.

Effect 07h  Tremolo

Basically, a vibrato effect except it doesn't apply to the pitch, but rather to the volume. The DoTremolo function, whose structure is similar to the DoVibrato function, produces a tremolo.

void DoTremolo(char i)
{
    char q;
    int tmp;

    q = (Channel[i].vibpos >> 2) & 64;
    tmp = VibratoTable[q];

    tmp = tmp * Channel[i].vib_depth;
    tmp >>= 6;

    Channel[i].volume += tmp;
    if (Channel[i].volume < 0) {
        Channel[i].volume = 0;
    }
    if (Channel[i].volume > 64) {
        Channel[i].volume = 64;
    }
    Channel[i].vibpos += Channel[i].vib_rate;
}

Effect 08h  Set Panning

Sets the current panning position in the channel. Since we're working internally with 64 panning positions, we have to divide the specified parameter by 4.
Effect 09h  Play sample at offset

Effect 9 doesn't play the sample from its normal starting position, but instead, plays it from a passed offset. The offset results from the formula: \[ \text{Parameter} \times 100h \]. Remember, this effect cannot be repeated with every tick and the sample cannot be started after the effect procedure.

Effect 0Ah  Volume sliding

The direction of volume sliding depends on the passed parameter. If it has a value \( \leq 0Fh \), the volume is reduced by the specified value, otherwise it is increased by the parameter \( \text{DIV 16} \).

Effect 0Bh  Position Jump

Jumps to the position in the pattern table passed in the parameter.

Effect 0Ch  Set Volume

Sets the volume of the channel. More than anything else, this effect was integrated into the XM format for compatibility reasons, since, if at all possible, the volume should be set using the volume column.

Effect 0Dh  Pattern Break

Ends the current pattern so the following pattern is played. Simply set the break flag here.

Effect 0Eh  Extended effects

The player handles the extended effects using the \text{do\_e\_effects} function. We'll describe this function later.

Effect 0Fh  Set speed

If this effect has a parameter greater than or equal to 20h, the specification applies to the BPM, otherwise the speed is changed. Only execute this effect once and not with every tick.

The following lists the entire effect handling function:

```c
void process_effects(unsigned short int i) {
    char para;
    unsigned int begin;
    unsigned int start;
    unsigned int end;
    unsigned char mode;

    switch (Channel[i].effect) {
        case 0x01 : /// Portamento UP
            if (Channel[i].param != 0) {
                Channel[i].period -= Channel[i].param;
                Channel[i].last_param = Channel[i].param;
            } else {
                Channel[i].period -= Channel[i].last_param;
            }
```
Sound Support For Your Programs

Chapter 13

UltraSetFrequency(i, GetFreq2(Channel[i].period));
replay_freq = 0;
break;
case 0x02 :  /// Portamento DOWN
    if (Channel[i].param != 0) {
        Channel[i].period += Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    }
    else {
        Channel[i].period += Channel[i].last_param;
    }
    UltraSetFrequency(i, GetFreq2(Channel[i].period));
    replay_freq = 0;
    break;
case 0x03 :  // Tone Portamento
    DoTonePortamento(i);
        UltraSetFrequency(i, GetFreq2(Channel[i].period));
    replay_freq = 0;
    break;
case 0x04 :  // Vibrato
    if (Channel[i].param != 0) {
        para = Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    }
    else {
        para = Channel[i].last_param;
    }
    if (para & 0x0f) Channel[i].vib_depth = para & 0x0f;
    if (para & 0xf0) Channel[i].vib_rate = (para & 0xf0) >> 4;
    DoVibrato(i);
    UltraSetFrequency(i, GetFreq2(Channel[i].period));
    replay_freq = 0;
    break;
case 0x05 :  // Tone Portamento & Volume Slide
    if (Channel[i].param != 0) {
        para = Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    }
    else {
        para = Channel[i].last_param;
    }
    DoTonePortamento(i);
    if (para <= 0x0f) {
        if (Channel[i].volume - para < 0) {
            Channel[i].volume = 0;
        } else {
            Channel[i].volume -= para;
        }
        UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
    } else {
        Channel[i].volume += (para >> 4);
        if (Channel[i].volume > 64) {
            Channel[i].volume = 64;
        }
UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
}
UltraSetFrequency(i, GetFreq2(Channel[i].period));
replay_freq = 0;
break;

case 0x06: // Vibrato & Volume Slide
    if (Channel[i].param != 0) {
        para = Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    } else {
        para = Channel[i].last_param;
    }
    if (para <= 0x0f) {
        if ((Channel[i].volume - para) < 0) {
            Channel[i].volume = 0;
        } else {
            Channel[i].volume -= para;
        }
        UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
    } else {
        Channel[i].volume += (para >> 4);
        if (Channel[i].volume > 64) {
            Channel[i].volume = 64;
        }
        UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
    }
    DoVibrato(i);
    UltraSetFrequency(i, GetFreq2(Channel[i].period));
    replay_freq = 0;
    break;

case 0x07: // Tremolo
    if (Channel[i].param != 0) {
        para = Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    } else {
        para = Channel[i].last_param;
    }
    if (para & 0x0f) Channel[i].vib_depth = para & 0x0f;
    if (para & 0xf0) Channel[i].vib_rate = (para & 0xf0) >> 4;
    DoTremolo(i);
    UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
    break;

case 0x08: // Set Panning
    if (Channel[i].param != 0) {
        para = Channel[i].param;
        Channel[i].last_param = Channel[i].param;
    } else {
        para = Channel[i].last_param;
    }
    Channel[i].pan_val = para >> 2;
    UltraSetPanning(i, Channel[i].pan_val);
    break;

case 0x09: // Play with Offset
```c
begin = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start;
start = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start
  + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_start;
if (insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length > 0) {
  end = start
  + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length;
} else {
  end = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start
  + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_length;
}
begin += Channel[i].param * 0x100;
mode = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Loop_Mode;
UltraPrimeVoice(i, begin, start, end, mode, Channel[i].volume * Channel[i].vol_val / 64,
  insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Is_16Bit);
UltraSetFrequency(i, GetFreq2(Channel[i].period));
replay_note = 0;
replay_freq = 0;
Channel[i].effect = 255;
break;
case 0x0a : /// Volume Slide
  if (Channel[i].param != 0) {
    para = Channel[i].param;
    Channel[i].last_param = Channel[i].param;
  } else {
    para = Channel[i].last_param;
  }
  if (para <= 0x0f) {
    if ((Channel[i].volume - para) < 0) {
      Channel[i].volume = 0;
    } else {
      Channel[i].volume -= para;
    }
    UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
  } else {
    Channel[i].volume += (para >> 4);
    if (Channel[i].volume > 64) {
      Channel[i].volume = 64;
    }
    UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
  }
break;
case 0x0b : // Position Jump
  Break_The_Pattern = 1;
  Songpos = Channel[i].param;
  break;
case 0x0c : /// Set Volume
  if (Channel[i].param <= 0x40) {
    Channel[i].volume = Channel[i].param;
    UltraSetDirectVolume(i, Channel[i].volume * Channel[i].vol_val / 64);
  } else {
    Channel[i].effect = 255;
  }
break;
case 0x0d : // Pattern Break
  Break_The_Pattern = 1;
  Songpos ++;
```
Sound Support For Your Programs

Extended effects

Not all the effects were implemented in the player here. We’ll only cover the incorporated effects here, they are the same ones as with the default MOD format:

Extended Effect 01h  Fine Portamento up
The effect functions like the normal portamento except the function is only called once, not with every tick.

Extended Effect 02h  Fine Portamento down
The same as fine portamento up, only down.

Extended Effect 09h  Retriggering
Retriggering is the multiple striking of a note. This is especially interesting for percussion instruments. You have to run a counter that you initialize with the value passed in the parameter and decrement each tick. If the counter reaches the value 0, start the sample over again.

Extended Effect 0Ah  Fine volume slide up
This effect runs similar to normal volume sliding. The only difference is the volume is updated only once instead of with each tick.

Extended Effect 0Bh  Fine volume sliding down
Similar to volume sliding up, only for volume sliding down.
switch (Channel[i].param >> 4) {
    case 0x00 : //
      break;
    case 0x01 : /// FINE Portamento UP
      if ((Channel[i].param & 0x0f) != 0) {
        Channel[i].period -= (Channel[i].param & 0x0f)<<2;
        Channel[i].last_param = Channel[i].param;
      } else {
        Channel[i].period -= (Channel[i].last_param & 0x0f)<<2;
      }
      UltraSetFrequency(i,GetFreq2(Channel[i].period));
      replay_freq = 0;
      Channel[i].effect = 255;
      break;
    case 0x02 : /// FINE Portamento DOWN
      if ((Channel[i].param & 0x0f) != 0) {
        Channel[i].period += (Channel[i].param & 0x0f)<<2;
        Channel[i].last_param = Channel[i].param;
      } else {
        Channel[i].period += (Channel[i].last_param & 0x0f)<<2;
      }
      UltraSetFrequency(i,GetFreq2(Channel[i].period));
      replay_freq = 0;
      Channel[i].effect = 255;
      break;
    case 0x03 : //
      break;
    case 0x04 : //
      break;
    case 0x05 : //
      break;
    case 0x06 : //
      break;
    case 0x07 : //
      break;
    case 0x08 : //
      break;
    case 0x09 : // retriggering
      if ((Channel[i].param & 0x0f) != 0) {
        Channel[i].ret_count --;
        if (Channel[i].ret_count == 0) {
          Channel[i].ret_count = Channel[i].param & 0x0f;
          begin = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start;
          start = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start
            + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_start;
          if (insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length > 0) {
            end = start
              + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_loop_length;
          } else {
            end = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Gus_Start
              + insts[Channel[i].inst_no].sheader[Channel[i].smpno]->sample_length;
          }
          mode = insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Loop_Mode;
          UltraRestartVoice(i, begin,start,end,mode,Channel[i].volume *
                        Channel[i].vol_val / 64, insts[Channel[i].inst_no].sheader[Channel[i].smpno]->Is_16Bit);
        }
      }
  }
Sound Support For Your Programs

Chapter 13

Envelopes

The handling routine for envelopes is called by the timer consistently and does not depend on the set speed or the BPMs. This is how the routine works: The current position within the envelope is incremented with each call. In doing so, it’s necessary to check the following:

- Whether the function is waiting for a keyoff (do not shift any further).
- The last point has been reached.
- If a loop point has been reached so that the position has to be set at the loop start.

If the current point is found, its date value (y-position) is obtained from an interpolation between the two known envelope points (linear equation). Finally, set the determined volume or the panning position. By looking at the procedure, you’ll discover that you will be able to process quite complex handling quickly, provided you work in a structured fashion.
void Process_Vol_Env()
{
    int pxa;
    int pxb;
    int pva;
    int pvb;
    int dx;
    int dv;
    int dt;
    int li;
    short int li;

    for (li = 0; li < xmh.num_channels; li++) {
        if ((Channel[li].is_vol_env == 1) && (Channel[li].Wait_Keyoff != 1)) {
            if ((Channel[li].vol_pos < 324) && (insts[Channel[li].inst_no].iheader->num_vol_points > 1)) {
                Channel[li].vol_pos++;
                // Volume point reached
                if (Channel[li].vol_pos == insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt2*2]) {
                    Channel[li].vol_pt1++;
                    if (Channel[li].is_vol_sus == 1) { // Wait for Keyoff?
                        if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->vol_sustain_p) {
                            if (Channel[li].is_vol_loop == 1) { // If at loop point then loop
                                if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->vol_loop_endp) {
                                    Channel[li].vol_pt2++;
                                } else {
                                    Channel[li].vol_pt1 = insts[Channel[li].inst_no].iheader->vol_loop_startp;
                                    Channel[li].vol_pt2 = Channel[li].vol_pt1 + 1;
                                    Channel[li].vol_pos = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2];
                                }
                            } else {
                                Channel[li].vol_pt1 = insts[Channel[li].inst_no].iheader->vol_loop_startp;
                                Channel[li].vol_pt2 = Channel[li].vol_pt1 + 1;
                                Channel[li].vol_pos = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2];
                            }
                        } else { // no looping
                            if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->num_vol_points) {
                                Channel[li].vol_pt2++;
                            }
                        }
                    } else { // sustain-p reached
                        if (Channel[li].is_vol_sus == 1) { // Wait for Keyoff?
                            if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->vol_sustain_p) {
                                if (Channel[li].is_vol_loop == 1) { // If at loop point then loop
                                    if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->vol_loop_endp) {
                                        Channel[li].vol_pt2++;
                                    } else {
                                        Channel[li].vol_pt1 = insts[Channel[li].inst_no].iheader->vol_loop_startp;
                                        Channel[li].vol_pt2 = Channel[li].vol_pt1 + 1;
                                        Channel[li].vol_pos = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2];
                                    }
                                } else {
                                    if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->num_vol_points) {
                                        Channel[li].vol_pt2++;
                                    }
                                }
                            }
                        } else { // no sustain
                        }
                    }
                } else { // no sustain
                }
            } else { // no sustain
            }
        } else { // no sustain
        }
    }
}
if (Channel[li].is_vol_loop == 1) {  // If at loop point then loop
  if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->vol_loop_endp) {
    Channel[li].vol_pt2++;
  } else {
    Channel[li].vol_pt1 = insts[Channel[li].inst_no].iheader->vol_loop_startp;
    Channel[li].vol_pt2 = Channel[li].vol_pt1 + 1;
    Channel[li].vol_pos = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2];
  }
} // end looping
else {  // no looping
  if (Channel[li].vol_pt2 < insts[Channel[li].inst_no].iheader->num_vol_points) {
    Channel[li].vol_pt2++;
  }
}
} // end no sustain

pva = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2+1];
pvb = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt2*2+1];
if (pva == pvb) {
  Channel[li].vol_val = pva;
  UltraSetDirectVolume(li,Channel[li].volume * Channel[li].vol_val / 64);
} else {
  pxa = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt1*2];
pxb = insts[Channel[li].inst_no].iheader->vol_envelope[Channel[li].vol_pt2*2];
dtx = pxb - pxa;
dtv = pvb - pva;
dtist = Channel[li].vol_pos - pxa;
if (dtx > 0) {
  Channel[li].vol_val = pva + ((dtist * dtv) / dtx);
  UltraSetDirectVolume(li,Channel[li].volume * Channel[li].vol_val / 64);
}
} // end new point
} // end channel loop

SFX Pro main program

In this section we’ll look at the main program. First, the program checks whether the name of an XM module was specified. If not, the program terminates with an error message. Then the sound card is initialized and the necessary preallocations of the data structures follow.

The program then displays the main ANSI and executes the load procedure. If the module loaded successfully, the timer can be initialized and the program lands in its main loop. Each time the player is moved forward one row in the main loop, the screen is updated and the program checks for user input. The following keys are available:
Sound Support For Your Programs

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Mastervolume</td>
<td>Increases the master volume of the system.</td>
</tr>
<tr>
<td>6</td>
<td>Mastervolume</td>
<td>Lowers the master volume of the system.</td>
</tr>
<tr>
<td>8</td>
<td>Infopage</td>
<td>Displays an information page with some info about the sound system and greetings from the author.</td>
</tr>
<tr>
<td>2</td>
<td>DOS-Shell</td>
<td>If COMMAND.COM is in the same directory as the player, you can start a DOS SHELL from here. Then the player will still run when you start other (Watcom C) Protected Mode programs that don’t change the GUS or the timer.</td>
</tr>
<tr>
<td>5</td>
<td>Forward</td>
<td>Moves forward in the piece.</td>
</tr>
<tr>
<td>7</td>
<td>Back</td>
<td>Moves back in the piece.</td>
</tr>
</tbody>
</table>

```c
void main(int argc, char *argv[]) {
    char chchar = 'ä';

    if (argc < 2) {
        sfxerror(0);
    }

    Init_Gus(0x0220);
    speaker_off();
    Break_The_Pattern = 0;
    Init_Instruments();
    Init_Channels();

    memcpy(screen, &SFXMAIN, 80*25*2);
    load_module(argv[1]);
    row_changed = 0;

    Set_Timer_Defaults();
    SetTimer(xmh.default_bpm-1, xmh.default_tempo-1);

    for (i = 0; i < 32; i++) {
        UltraSetPanning(i, 32);
    }

    init_timer();
    speaker_on();

    cls();
    show_mainscreen();

    while (chchar != 27) {
        if (kbhit()) {
            chchar = getch();
        } // process_patternrow();

        switch(chchar) {
            case '*': if (Mastervolume < 245) {
                Mastervolume += 10;
                chchar = 'ä';
            } break;
```
Chapter 13

Sound Support For Your Programs

```c
case '-' : if (Mastervolume > 10) {
    Mastervolume -= 10;
    chchar = 'ä';
}
break;
case 'i' : show_info();
break;
case 'o' : spawnl(P_WAIT,"command.com",NULL);
break;
case 'v' : Songpos ++;
    Break_The_Pattern = 1;
    chchar = 'ä';
    break;

    case 'z' : Songpos -= 2;
    Break_The_Pattern = 1;
    chchar = 'ä';
    break;
}
if (row_changed == 1) {
    row_changed = 0;
    screen_update();
}
reset_timer();
stop_gus();
```
Most computer games are written to perform either fast or realistic. Two games have recently broken this mold and have generated much excitement with their fast, realistic 3-D graphics and sound. These are the DOOM and DOOM II games. Even if you’re not a DOOM fanatic, you’ll be impressed with the programming techniques behind the game.

Due to the huge success of DOOM, an entire wave of games based on these techniques have followed.

In DOOM, you wander through a labyrinth of rooms and defend yourself against a host of enemies. The graphic arrangement of the "enemies", the floor, the walls and ceilings and the other objects is drawn in detailed, complex resolution, produced by the 3-D display.

Before you can begin programming your own DOOM-like games or demos, you must first master 3-D graphics programming. This is what we’ll talk about in this chapter. We’ll include some ready-to-use routines that you can borrow for your own programs. You can then use the routines to develop DOOM-like games and demos quickly and easily.

Although we’ve already done some work with three-dimensional computer graphics, in this chapter we’ll show you how to calculate and display vector objects that are even more impressive. The faces of these objects won’t be filled with a single color. Instead, we’ll introduce two methods called Gouraud shading and texture mapping that allow a more realistic display.

Quick Math: Fixed Point Arithmetic

You may think a math coprocessor is necessary to perform all the necessary calculations involved in displaying vector graphics. Although most PCs manufactured today include a math coprocessor, earlier models (386DX-40, 486SX-25 or 33) lack a math coprocessor. In addition, the coprocessor for most 386 computers is comparatively slow since it is not integrated in the chip.

However, don’t despair if your system lacks a math coprocessor. In fact, we don’t always recommend using the coprocessor, even if you do have a 486DX. The reason is because you often need to convert floating point numbers to integers which still requires time for the 486 CPU. So, one question is “do you need to buy a Pentium to run games?”. The answer is “NO” because you can get around floating point operations with a technique that uses fixed point numbers.

Although the technique is rather simple, it increases the execution speed of your computer many times. An integer (whole number) is used, a portion of which is treated as a decimal. To achieve this, you multiply the floating point number by a constant factor and then round it off. This makes an integer out of a floating point number.
As you can easily verify, addition and subtraction operations can be executed immediately. However, you need to make some corrections with division and multiplication. Here are some sample calculations:

- **Addition:** \(1.2\times100 + 1.5\times100 = 2.7\times100\)
- **Subtraction:** \(1.2\times100 - 1.5\times100 = 0.3\times100\)
- **Multiplication:** \(1.2\times100 \times 1.5\times100 = 1.8\times100\times100\)

So when you multiply, you still have to divide the result by 100.

- **Division:** \(1.2\times100 / 1.5\times100 = 0.8\)

When you divide, you still have to multiply the result by 100.

Since it doesn’t really matter what kind of a value you multiply/divide by (only the accuracy depends on it), it is natural to use a power of 2, not a power of 10. The multiplication/division can be replaced by several speedier shift operations.

256 (8 bit decimal) is ideal for this purpose. However, in the following example, we use the number 512 (9 bit decimal) to get a higher degree of accuracy. In addition, we use 32 bit integer = (Longint) variables. Naturally, you could also put the same principle into practice with 16 bit (faster, but smaller range of numbers). Thus, converting a floating point number to a fixed point number is performed according to the following formula:

\[
\text{fixed := round(_real * }$200)\]

and a corresponding way to reconver the number:

\[
\text{_real := fixed / }$200;\]

Since these conversions require quite a bit of time on a computer that doesn’t have a coprocessor, you can perform the calculations ahead of time so they don’t need to be calculated at runtime.

You can easily perform division and multiplication in assembly language using the 32-bit instructions of the 386. Both routines were written for Turbo Pascal but can also be used under any 16-bit C compiler. Please note that we used TASM in IDEAL mode syntax; you’ll have to make the necessary syntax corrections if you’re using another assembler.

```
;-------------------------------------------------------------
; function FixMul(A,B :longInt) :LongInt; near; external;
;-------------------------------------------------------------
PROC FixMul NEAR
    pop    cx
    pop    ebx
    pop    eax
    imul   ebx
    shrd   eax,edx,9
    rol    eax,16
    mov    dx,ax
    rol    eax,16
    push   cx
    ret
ENDP
```
The Secrets Behind DOOM

Chapter 14

PC underground

To avoid having to set up a stack frame, the return address is removed from the stack first, and then the two factors. Next, the multiplication is carried out and the result is shifted to the right by 9 bits (= division by 512). Since Turbo Pascal, as a 16 bit compiler, doesn’t support 32 bit registers, the return value must be passed in dx:ax. Remember, the procedure is NEAR declared. So, it cannot be exported by a UNIT. That’s why the procedure headers are declared in an INCLUDE file.

The routine `FixSQR` calculates the square of the value that is passed to it:

```assembly
;---------------------------------------------------------------------
function FixSQR(A :Fixed)
:Fixed; near; external;
;---------------------------------------------------------------------

PROC FixSQR NEAR

    pop    cx
    pop    eax
    imul   eax
    shrd   eax,edx,9
    rol    eax,16
    mov    dx,ax
    rol    eax,16
    push   cx
    ret

ENDP
```

The routine for division is similar:

```assembly
;-------------------------------------------------------------
; function FixDiv(A,B :LongInt) :LongInt; near; external;
;-------------------------------------------------------------

PROC FixDiv NEAR

    pop    cx
    pop    ebx
    pop    eax
    cdq
    shld   edx,eax,9
    shl    eax,9
    idiv   ebx
    rol    eax,16
    mov    dx,ax
    rol    eax,16
    push   cx
    ret

ENDP
```

The dividend in eax is extended to edx:eax. After that, everything is shifted to the left by 9 bits and the division is executed.

Besides division and multiplication, the trigonometric functions such as sine and cosine (important for rotation of points) are also very important. Because the calculation of sine and cosine values using rows is very processor intensive, you need to use tables. Remember, 360° corresponds to the second power. Then an AND operator can be used, instead of division, to manage multiple rotations (0°=360°=720°).

```assembly
;-------------------------------------------------------------
; function FixSin(W :Integer) :LongInt; near; external;
;-------------------------------------------------------------

PROC FixSin NEAR

    pop    cx
    pop    bx

```
The Secrets Behind DOOM

Chapter 14

PC

PC underground

To prevent the sine table from becoming too large, it contains only values for 0°-90°. The value 360°
corresponds to 512 "fixed". Also, no LongInts were saved, but rather Words which were converted to a
LongInt in dx:ax with cwd.

The cosine routine simply adds 128 (90) and then jumps to the sine routine. Refer to the listing for more
information.

Vector Arithmetic: The Math Behind DOOM

Because of its 3-D display, DOOM is based on fastvector arithmetic. We're going to show you how to program
fast vector arithmetic. After reading the chapter on 3-D graphic programming, you know what vectors are.
If the term matrix doesn't mean anything to you, don't worry about that either. In the following, we will
compile a complete vector/matrix library. This vector/matrix
library is defined in _MATH.ASM and _MATH.PAS.

A 3-dimensional vector is defined there as an array:
type
P3-D = T3-D;
T3-D = array[x..z] of LongInt;

All the vector routines work with this type. The procedure VecZero sets the vector
passed in C to zero.

\[
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]

The \textbf{Vec} procedure loads the vector in C with the components \(v_x, v_y\) and \(v_z\).

\[
\begin{bmatrix}
v_x \\
v_y \\
v_z
\end{bmatrix}
\]

UnVec is the counterpart of Vec.

\[
\begin{align*}
v_x &= a_x \\
v_y &= a_y \\
v_z &= a_z
\end{align*}
\]
The Secrets Behind DOOM

Chapter 14

VecAdd adds two vectors, A and B. The result is returned in C.

\[ \vec{C} = \vec{A} + \vec{B} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix} \]

VecSub subtracts B from A. The result is returned in C.

\[ \vec{C} = \vec{A} - \vec{B} = \begin{pmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{pmatrix} \]
The Secrets Behind DOOM

Chapter 14

PC underground

VecDot calculates the scalar product of \( \vec{A} \) and \( \vec{B} \). The result is returned as a function value.

\[
\vec{A} \cdot \vec{B} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z
\]

VecCross calculates the cross-product of \( \vec{A} \) and \( \vec{B} \). The result is returned in \( \vec{C} \).
Chapter 14

The Secrets Behind DOOM

\[ \vec{C} = \vec{A} \times \vec{B} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix} \]

; procedure VecCross(A,B :T3-D; var C :T3-D);

PROC VecCross FAR
    ARG C:DWord, B:DWord, A:DWord = @@return
    enter 0,0
    push ds
    lds si,[A]
    les di,[B]
    lgs bx,[C]
    mov eax,[si+4]
    imul [DWORD PTR es:di+8]
    shrd eax,edx,9
    mov ecx,eax
    mov eax,[si+8]
    imul [DWORD PTR es:di+4]
    shrd eax,edx,9
    mov ecx,eax
    mov eax,[si]
    imul [DWORD PTR es:di+4]
    shrd eax,edx,9
    sub ecx,eax
    mov [gs:bx],ecx
    mov eax,[si+8]
    imul [DWORD PTR es:di]
    shrd eax,edx,9
    mov ecx,eax
    mov eax,[si]
    imul [DWORD PTR es:di+8]
    shrd eax,edx,9
    sub ecx,eax
    mov [gs:bx+4],ecx
    mov eax,[si]
    imul [DWORD PTR es:di+4]
    shrd eax,edx,9
    mov ecx,eax
    mov eax,[si+4]
    imul [DWORD PTR es:di]
    shrd eax,edx,9
    sub ecx,eax
    mov [gs:bx+8],ecx
    pop ds
    leave
    ret @@return
ENDP

VecDistance returns the square of the amount of \( \mathbf{A} \) as a function value.

\[ d = \mathbf{A}^2 = \sqrt{a_x^2 + a_y^2 + a_z^2} = a_x^2 + a_y^2 + a_z^2 \]
The Secrets Behind DOOM

Chapter 14

PC underground

; function VecDistance(A :T3-D) :LongInt;

PROC VecDistance FAR
    ARG A:DWord = @@return
    enter 0,0
    push ds
    lds si,[A]
    lodsd or eax,eax
    jns @@11
    neg eax
    @@11:
        mul eax
        shrd eax,edx,9
        mov ecx,eax
        lodsd or eax,eax
        jns @@12
        neg eax
    @@12:
        mul eax
        shrd eax,edx,9
        add ecx,eax
        lodsd or eax,eax
        jns @@13
        neg eax
    @@13:
        mul eax
        shrd eax,edx,9
        add ecx,eax
        mov ax,ecx
        rol ecx,16
        mov dx,ecx
        pop ds
    leave
    ret @@return
ENDP

VecScalMul multiplies vector A by scalar k and returns the result in C.

; procedure VecScalMul(k :LongInt; A :T3-D; var C :T3-D);

PROC VecScalMul FAR
    ARG C:DWord, A:DWord, k:DWord = @@return
    enter 0,0
    push ds
    lds si,[A]
    les di,[C]
    lodsd imul [k]
    shrd eax edx,9
    stosd lodsd imul [k]
    shrd eax edx,9
    stosd lodsd
In computer graphics, vector objects are described by points (as a point quantity in three-dimensional Euclidean point space). Thus, to move an object, it is necessary to change these points.

We can describe the position of a point mathematically by linear transformations: Translation (=movement in space), rotation and scaling. If you transfer all the points of an object in the same way, the object changes its position in space. It's very easy to describe these linear transformations using matrices. This has the advantage of letting you combine the various matrices into a single matrix. This makes the task much easier and less cluttered.

Another important advantage is that you always use the same amount of time to calculate each transformation (9 multiplications and 16 additions). This is true regardless of how complicated the transformation is.

We'll delve into math more in the next section. You can skip it if you only want to use the library functions.

Perhaps you still remember linear equation systems from school.

The y quantities of y1, y2, and y3 are linearly dependent on the x variables of x1, x2, and x3. If you view the y and x quantities as 3-dimensional vectors, a relationship between x and y is achieved through the linear equation system mentioned above. For each vector \( \mathbf{x} \) there is a corresponding vector \( \mathbf{y} \) (see small illustration on the right).

You may notice this is exactly what we need. A vector (which describes a point of our object) can be assigned to another point through an assignment. The factors a11..a33 define the type of assignment. So, you combine the factors a11..a33 to a variable named MATRIX.

\[
\begin{align*}
y_1 &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\
y_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\
y_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3
\end{align*}
\]

A rectangular number array of x rows and y columns is called an x-y matrix

Now you're probably wondering what the translation matrix (movement in space) looks like. There is no combination of factors a11..a33 that brings about a translation. So, there is no 3x3 translation matrix for 3-dimensional vectors.

By introducing what we could call "homogenous coordinates", it becomes possible to describe even the translation of a single matrix. In so doing, our point is no longer
The Secrets Behind DOOM

Chapter 14

PC underground

described by a three-column vector, but rather, by a four-column vector. Therefore, the linear transformations are described by a 4x4 matrix. The components of the fourth dimension (w) are set to 1, so our vector object corresponds to a projection at the level w=1 of 4-dimensional space.

A movement of vector A can now be described by the following translation matrix (see illustration, below left). The illustration below right shows how the scaling matrix reads:

\[
\begin{pmatrix}
\alpha_x + t_x & 0 & 0 & 0 \\
0 & \alpha_y + t_y & 0 & 0 \\
0 & 0 & \alpha_z + t_z & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix} = \begin{pmatrix}
\alpha_x & 0 & 0 & 0 \\
0 & \alpha_y & 0 & 0 \\
0 & 0 & \alpha_z & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

For rotation, there are three different rotation matrices (one for each axis: x,y,z):

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

You should now be familiar with these matrices from the last section when we introduced them as an equation system.

Now we're coming to the most important advantage of matrices. All translation matrices can be combined into a single matrix. We've already saved you some work by defining this total matrix. What is important here is the sequence in which the matrices are multiplied; the individual transformations will be executed in this precise sequence. The scaling is executed last, which makes it possible to adapt the vector object to different screen resolutions easily. The following illustration shows the result of the computation:

\[
M = \begin{pmatrix}
\cos \alpha & \sin \alpha & 0 & d_x \\
-\sin \alpha & \cos \alpha & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

This may at first seem like a great deal of work for the PC. However, remember, these calculations only have to be performed once per image.

A 4x4 matrix in the _MATH_ unit is declared in the following manner:

```pascal
type
  TMatrix = array[0..3, 0..3] of LongInt;

procedure Zero(var C :TMatrix);

PROC Zero FAR
```

The procedure Zero sets the total matrix, C, to zero.
The Secrets Behind DOOM

Chapter 14

The procedure **Cross** calculates the product of the A and B matrices:

\[
\begin{align*}
\sigma &= \sum_{k=1}^{n} c_k k_y \\
&= \sum_{k=1}^{n} a_k b_k
\end{align*}
\]

The procedure **Transform** multiplies Point A, described by vector A, by Matrix M, achieving a transformation of Point A. The program simplified certain dimensions here (4th dimension of A always 1; 4th dimension of C ignored, since it's unimportant).
The Secrets Behind DOOM

Chapter 14

PC underground

; procedure Transform(A :T3-D; M :TMatrix; var C :T3-D);
PROC Transform FAR
ARG C:DWord, M:DWord, A:DWord = @@return
enter 0,0
lfs si,[A]
lgs bx,[M]
les di,[C]
mov cx,3
@@11:
push cx
mov eax,[fs:si]
imul [DWORD PTR gs:bx]
shrd eax,edx,9
mov ecx,eax
mov eax,[fs:si+4]
imul [DWORD PTR gs:bx+4]
shrd eax,edx,9
add ecx,eax
mov eax,[fs:si+8]
imul [DWORD PTR gs:bx+8]
shrd eax,edx,9
add ecx,eax
add ecx,[DWORD PTR gs:bx+12]
mov [es:di],ecx
add di,4
add bx,16
pop cx
loop @@11
leave
ret @@return
ENDP

Rotate generates a rotation matrix. This procedure rotates around the x-axis, y-axis and finally the z-axis.

; procedure Rotate(rx,ry,rz :Integer; var R :TMatrix);
PROC Rotate FAR
ARG R:DWord, rz:Word, ry:Word, rx:Word = @@return
enter @@local,0
les di,[R]
push di
mov cx,(4*4*4)/4
xor eax,eax
repe stosd
pop di
mov bx,[rx]
call FixSin
mov [SinX],eax
mov bx,[rx]
call FixCos
mov [CosX],eax
```assembly
mov    bx,[ry]
call   FixSin
mov    [SinY],eax
mov    bx,[ry]
call   FixCos
mov    [CosY],eax
mov    bx,[rz]
call   FixSin
mov    [SinZ],eax
mov    bx,[rz]
call   FixCos
mov    [CosZ],eax
imul   [SinY]
shrd   eax,edx,9
mov    [A],eax
mov    eax,[SinZ]
imul   [SinY]
shrd   eax,edx,9
mov    [B],eax
mov    eax,[CosZ]
imul   [CosY]
shrd   eax,edx,9
mov    [es:di],eax
mov    eax,[A]
imul   [SinX]
shrd   eax,edx,9
mov    ebx,eax
mov    eax,[SinZ]
imul   [CosX]
shrd   eax,edx,9
sub    ebx,eax
mov    [es:di+4],ebx
mov    eax,[A]
imul   [CosX]
shrd   eax,edx,9
mov    ebx,eax
mov    eax,[SinZ]
imul   [SinX]
shrd   eax,edx,9
add    eax,ebx
mov    [es:di+8],eax
mov    eax,[SinZ]
imul   [CosY]
shrd   eax,edx,9
mov    [es:di+16],eax
mov    eax,[B]
imul   [SinX]
shrd   eax,edx,9
mov    ebx,eax
mov    eax,[CosZ]
imul   [CosX]
shrd   eax,edx,9
add    eax,ebx
mov    [es:di+20],eax
mov    eax,[B]
imul   [CosX]
shrd   eax,edx,9
mov    ebx,eax
mov    eax,[CosZ]
imul   [SinX]
shrd   eax,edx,9
sub    ebx,eax
```
The Secrets Behind DOOM

Chapter 14

PC underground

Create creates a procedure according to Gl. 19. The rotations are executed first (as with Rotate). Next, the translation is executed. The scaling is executed last.

Create (tx,ty,tz,sx,sy,sz:LongInt;rx,ry,rz:Integer; var M :TMatrix);
The Secrets Behind DOOM

Chapter 14

The PC underground

```assembly
mov [B], eax
mov eax, [CosZ]
imul [CosY]
shrd eax, edx, 9
imul [ax]
shrd eax, edx, 9
mov [es:di], eax
mov eax, [SinZ]
imul [CosX]
shrd eax, edx, 9
mov ebx, eax
mov eax, [A]
imul [SinX]
shrd eax, edx, 9
sub eax, ebx
imul [sx]
shrd eax, edx, 9
mov [es:di+4], eax
mov eax, [A]
imul [CosX]
shrd eax, edx, 9
mov ebx, eax
mov eax, [SinZ]
imul [SinX]
shrd eax, edx, 9
add eax, ebx
imul [ax]
shrd eax, edx, 9
mov [es:di+8], eax
mov eax, [A]
imul [SinX]
shrd eax, edx, 9
mov ebx, eax
mov eax, [CosZ]
imul [CosX]
shrd eax, edx, 9
add eax, ebx
imul [sx]
shrd eax, edx, 9
mov [es:di+16], eax
mov eax, [B]
imul [SinX]
shrd eax, edx, 9
mov ebx, eax
mov eax, [CosZ]
imul [CosX]
shrd eax, edx, 9
add eax, ebx
imul [sy]
shrd eax, edx, 9
mov [es:di+20], eax
mov eax, [CosZ]
imul [SinX]
shrd eax, edx, 9
mov ebx, eax
mov eax, [B]
imul [CosX]
shrd eax, edx, 9
sub eax, ebx
imul [sy]
shrd eax, edx, 9
mov [es:di+24], eax
mov eax, [SinY]
neg eax
imul [sz]
shrd eax, edx, 9
mov [es:di+32], eax
```
You should be familiar with the conversion of 3-D coordinates to 2-D coordinates (projection) from Chapter 7 ("The Third Dimension: 3-D Graphics Programming"). The eye-screen distance was set to one:

```
procedure Project(A :T3-D; var C:T2D);
begin
  if (A[z] div 200) <> 0 then begin
  end;
end;
```

We'll also need the following two procedures for adding and subtracting 2-dimensional vectors.

```
; procedure VecAdd2D(A,B :T2D; var C :T2D);
;
PROC VecAdd2D FAR
  ARG C:DWord, B:DWord, A:DWord = @@return
enter 0,0
les di,[C]
mov ax,[WORD PTR A]
add ax,[WORD PTR B]
stosw
mov ax,[WORD PTR A+2]
add ax,[WORD PTR B+2]
stosw
leave
```
Chapter 14
The Secrets Behind DOOM

Cast An Impressive Shadow With Gouraud Shading

Now that we've described and moved objects in space and converted a three-dimensional object to the two-dimensional point coordinates of the screen, we're at another interesting part: Filling the polygons.

We don't want to fill the polygons with a uniform color because that would result in a color gap between the various surfaces called flatshading. Instead, we want to assign each point a normal vector/color value. Then the polygons can be displayed using the Phong shading or Gouraud shading technique.

Both techniques are interpolation procedures. The normal vector of the pixel to be drawn is determined in Phong shading using interpolation (and then the color value is calculated). With Gouraud shading, only the color values are interpolated.

The Gouraud shading method uses relatively little computation. The greatest disadvantage of Gouraud shading is that no highlights can be displayed inside the polygon.

For Gouraud shading, each point must be assigned a color value. For this purpose, each point must have a normal vector (this normal vector represents the mean value of the adjacent surface normal vectors). We'll explain the calculation of the color value later.

First, the color values for the edges of the polygon are interpolated and then the color values for each individual point. Here, for the first time, is the definition of some important structures:

```asm
STRUC TLine
  x     dw ?
  ddy   dw ?
  ddx   dw ?
  icx   dw ?
  error dw ?
  yTo   dw ?
  xTo   dw ?
  Col   dd ?
  icC   dd ?
ENDS
```

This section's program listings are from CXYGPOLY.ASM and CXYTPOLY.ASM on the companion CD-ROM.
The Secrets Behind DOOM

Chapter 14

The next listing uses two variables. The first is \texttt{VirtualVSeg}, which contains the segment address of the video RAM. This can amount to 0A000h as well as containing the segment address of a single video buffer. The second variable is \texttt{xSize}, which specifies the x-width of the screen in pixels (with Mode 13h it would be \texttt{= 320}).

EXTRN \_VirtualVSeg :Word
EXTRN xSize :Word
yi dw ?
LL TLine ?
RL TLine ?
yMin dw ?
yMax dw ?
min dw ?
L dw ?
R dw ?
K dw ?
k1 TEntry ?
k2 TEntry ?
DeltaX dw ?
Clc dd ?

The screen coordinates of the polygon are passed in \texttt{P} to the routine. The coordinates are passed as "array [0..Count-1,x..y] of Integer". The color values are passed in \texttt{C} as "array [0..Count-1] of Byte". \texttt{Count} specifies the number of corners in the polygon.

; _______________________________________________________________
; \texttt{CxyShadedPoly(P :PPoints; C :PBytes; Count :Word);} 
; _______________________________________________________________
PROC CxyShadedPoly FAR
ARG Count:Word, C:DWord, P:DWord = @@return
enter 0,0

First, the highest and lowest value for y are determined and saved in \texttt{yMin} and \texttt{yMax}. At the same time, the position of \texttt{yMin} in the coordinate array is written in \texttt{min}.

cld
mov [min],-1
mov [yMin],200
mov [yMax],-1
xor cx,cx
les si,[P]
mov bx,si
inc si
inc si
@@11:
mov ax,[es:si]
cmp ax,[yMin]
jnl @@12
mov [yMin],ax
mov [min],cx
@@12:
mov ax,[es:si]
cmp ax,[yMax]
jnge @@13
The Secrets Behind DOOM

Chapter 14

PC underground

mov  [yMax],ax
@@13:
add   si,4
inc   cx
cmp   cx,[Count]
jb    @@11

If \textit{yMin} is greater than 199 or \textit{yMax} is less than 0, it means the polygon is not visible on the screen.

cmp   [yMin],199
jg    @@71
cmp   [yMax],0
jl    @@71

\textit{yi} specifies the current y position of the polygon routine. The polygon routine then goes through all the rows from \textit{yMin} to \textit{yMax} and draws a corresponding horizontal line. The routine calculates a right and left line for this purpose. This line determine the starting point and the end point of the horizontal line.

The polygons to be drawn must be convex. A polygon is convex when you can insert a straight line through the polygon without the line intersecting the polygon more than two times. A triangle and quadrangles, for example, are convex. Remember this when defining the polygons of your vector object. It’s faster to draw two convex polygons than one concave polygon.

The following code segment loads the initial values for the polygon routine.

mov   ax,[yMin]
mov   [yi],ax

\textbf{L} specifies the position of the left line in the coordinate array and \textbf{R} specifies the position of the right line. Both are loaded with the starting point. \textbf{LL} contains the data of the "L"eft "L"ine and \textbf{RL} contains the data of the "R"ight "L"ine. The x values of both lines are loaded with the initial value for x.

mov   ax,[min]
mov   [L],ax
mov   [R],ax
shl   ax,2
add   bx,ax
mov   ax,[es:bx]
mov   [LL.x],ax
mov   [RL.x],ax

\texttt{InitLeft} and \texttt{InitRight} load \textbf{LL} and \textbf{RL} with the values for the next line segment.

call   InitLeft
call   InitRight

We’ll talk about \texttt{InitLeft} and \texttt{InitRight} before discussing \texttt{CxyShadedPoly}.

First, the initial value for x of the line is read. The \textbf{L} is placed one position further. For this purpose, \textbf{L} is decremented by one. If \textbf{L} equals ZERO then \textbf{L} is assigned to \textbf{Count} and decremented by one.

482
If \( \text{L} \) is moved by one, the \( y \) end value (=\( y_{\text{To}} \)) of the line is loaded and the difference between \( y_{\text{To}} \) and \( y_i \) (=current \( y \) position) is saved in \( ddy \). After that, the \( x \) end value is loaded and saved in \( x_{\text{To}} \).

First, the initial color value is loaded, converted to a 16 bit fixed point number and saved in \( \text{Col} \).

This is where the calculation for the interpolation of the edge color value takes place. The initial and end color values are loaded and the difference between the two is obtained. Then the whole thing is divided by \( ddy \). In \( \text{Cic} \) you get the value that you have to add to \( \text{Col} \) per \( y \)-row to get the color value for the current edge point in \( \text{Col} \).
Chapter 14

The Secrets Behind DOOM

```
sub    eax,[LL.Col]
cdq
movzx  ebx,[LL.ddy]
idiv   ebx
mov    [LL.icC],eax
@@15:
ret
ENDP

InitRight has a similar structure. The only difference is that R is not decremented but is incremented instead. The following is the complete source for the procedure:

`---
PROC InitRight NEAR
`---
les    si,[P]
@@11:
    mov    bx,[R]
    mov    [K],bx
    shl    bx,2
    mov    ax,[es:si+bx]
    mov    [RL.x],ax
    mov    bx,[R]
    mov    ax,[Count]
    dec    ax
    cmp    bx,ax
    jne    @@12
    mov    bx,-1
@@12:
    inc    bx
    mov    [R],bx
    shl    bx,2
    mov    ax,[es:si+bx+2]
    mov    [RL.yTo],ax
    sub    ax,[yi]
    jns    @@13
    neg    ax
@@13:
    inc    ax
    mov    [RL.ddy],ax
    mov    ax,[es:si+bx]
    mov    [RL.xTo],ax
    les    si,[C]
    mov    bx,[K]
    movzx   ax,[BYTE PTR es:si+bx]
    shl    eax,16
    mov    [RL.Col],eax
    mov    [RL.icx],i
    mov    ax,[RL.xTo]
    sub    ax,[RL.x]
    jns    @@14
    neg    [RL.icx]
    neg    ax
@@14:
    mov    [RL.ddx],ax
    mov    [RL.error],0
    mov    bx,[R]
    movzx   ax,[BYTE PTR es:si+bx]
    shl    eax,16
    sub    eax,[RL.Col]
cdq
---`
"
Now we’ll continue with CxyShadedPoly. If yMax is greater than 199, yMax is set to 199.

The main loop begins here. A modified version of the Bresenham algorithm is used to draw the left or right line. If error is greater than or equal to ddx, the routine jumps to label @@33. Otherwise, the routine compares to determine whether the x end value has been reached. If this is not the case, error is incremented by ddy and icx is added to x. If error is greater than or equal to ddx or x is equal to xTo, then the loop is ended. Otherwise, the routine jumps to label @@31 and starts over from the beginning.

At the end, the modified values from the registers are written back to the variable.
The Secrets Behind DOOM

Chapter 14

The routine tests here whether the line to be drawn is outside the screen. If this is the case, the line doesn't get drawn.

```
cmp [yi], 0
jl @@60
```

Now copies of variables L.x/R.x and L.Col/R.Col are generated. GouraudVLine, which draws the horizontal line in the screen/video buffer, is then called. The routine is declared following this routine.

```
mov ax, [LL.x]
mov [k1.x], ax
mov eax, [LL.Col]
mov [k1.Col], eax
mov ax, [RL.x]
mov [k2.x], ax
mov eax, [RL.Col]
mov [k2.Col], eax
call GouraudVLine
```

If yi is equal to L.yTo or R.yTo, then InitLeft or InitRight is called.

```
@@60:
mov ax, [yi]
cmp [LL.yTo], ax
jne @@61
call InitLeft
mov ax, [yi]
@@61:
cmp [RL.yTo], ax
jne @@62
call InitRight
@@62:
```

Then yi is incremented by one. As long as yi is less than yMax, the routine jumps to the beginning (label @@21).

```
inc [yi]
mov ax, [yMax]
cmp [yi], ax
jle @@21
@@71:
leave
ret @@return
```
Next comes the routine which draws the horizontal line.

```assembly
;--------------------------------------------------------------
PROC GouraudVLine NEAR
;--------------------------------------------------------------
push ds
```

First, the difference from \( k_1.x \) and \( k_2.x \) is obtained and saved in \( \Delta x \).

```assembly
movzx ebx, [k2.x]
sub bx, [k1.x]
jns @@01
neg bx
@@01:
inc bx
mov [DeltaX], bx
```

Then the color increment is determined.

```assembly
xor edx, edx
mov eax, [k2.Col]
sub eax, [k1.Col]
jns @@02
neg eax
@@02:
div ebx
mov [Cic], eax
```

Then the x coordinates are clipped. The routine determines whether \( k_1.x \) is greater than \( k_2.x \). The color values must also be determined with the coordinates.

```assembly
movzx ebx, [k1.x]
movzx ecx, [k2.x]
cmp bx, cx
jnl @@13
cmp bx, 0
jnl @@11
neg bx
mov eax, [Cic]
imul ebx
add [k1.Col], eax
xor bx, bx
@@11:
cmp cx, 319
jng @@12
sub cx, 319
mov eax, [Cic]
imul ecx
add [k2.Col], eax
mov cx, 319
@@12:
cmp bx, cx
jg @@40
jmp @@16
@@13:
cmp cx, 0
jnl @@14
neg cx
mov eax, [Cic]
imul ecx
add [k1.Col], eax
xor cx, cx
```
Chapter 14

The Secrets Behind DOOM

@@14:

```
cmp    bx,319
jng    @@15
sub    bx,319
mov    eax,[Cic]
imul   ebx
add    [k2.Col],eax
mov    bx,319
@@15:
cmp    cx,bx
jg     @@40
@@16:
```

DeltaX is recalculated here, since it may have changed on account of clipping.

```
mov    [k1.x],bx
mov    [k2.x],cx
mov    bx,[k2.x]
sub    bx,[k1.x]
jns    @@03
neg    bx
@@03:
inc    bx
mov    [DeltaX],bx
```

This is where the low byte of the hi word of DWord Cic is loaded into the cl register and the low word of Cic is loaded into the hi word of ebx (we'll explain why this occurs when we describe the main loop).

```
mov    eax,[Cic]
mov    bx,ax
rol    ebx,16
rol    eax,16
mov    cl,al
```

The bx register is loaded with the increment by which the offset in the video buffer is increased.

```
mov    bx,1
mov    ax,[k1.x]
cmp    ax,[k2.x]
jng    @@10
neg    bx
@@10:
```

Another check is make here to see if k1.Col is greater than k2.Col.

```
mov    eax,[k1.Col]
cmp    eax,[k2.Col]
jne    @@21
```

Correspondingly, either k1.Col/k1.x or k2.Col/k2.x is taken as the color/x initial value. The low word of Col is loaded into the hi word of edi. The low byte of the hi word of Col is loaded into the al register. The offset in the video RAM/video buffer is loaded into the di register.

```
mov    di,[WORD PTR k1.Col]
rol    edi,16
mov    ax,[xSize]
mul    [yi]
mov    di,[k1.x]
add    di,ax
mov    al,[BYTE PTR 2+k1.Col]
```
The low byte of the hi word of Cic which was copied to the cl register is now transferred to the ah register.

It's better if we work our way through the main loop step by step.

The color value of the pixel [ds:di] is located in al.

Two things happen simultaneously during the subsequent 32 bit addition. Two 16-bit additions are combined into one here. di is "increased" by 1 and -1, and at the same time, the decimal portions of Col and Cic are added (in the hi word). If there is an overflow, the carry flag is set, otherwise it is cleared.

Instead of loop, the dec/jnz instruction sequence is used here. This is faster on 486s than the loop instruction. However, we're actually just fooling around, since there are far better places to optimize the code.

Using Realistic 3-D Objects: Polygons And Textures

We can improve the display even more by projecting onto a graphic on the vector object. As a result, the vector object is no longer uniformly shaded but contains a complex structure. By using this process, called texture mapping, it's possible to create very realistic objects, such as stones, without having to generate very many surfaces. While it's true that you need more processing time for texture mapping, the objects look remarkably more realistic.
The polygon routine we'll introduce is similar to the Gouraud shading routine. There are various approaches for producing texture mapping. The algorithm we'll introduce uses a very fast procedure but may generate errors in the display. The error results from the perspective, which is not taken into account in this algorithm.

For each horizontal line that is drawn on the screen, a corresponding line is drawn by the graphic (=texture map). Each point on the line drawn on the screen is assigned a point in the texture map.

\[
\text{Screen } \quad \text{Texture map}
\]

The texture line is mapped on the screen

Now we need to know how to determine the coordinates (x,y) in the texture map. The vertices of the polygon are assigned coordinates in the texture map. Similar to Gouraud shading, a twofold interpolation process is used to determine the intermediate coordinates. The only difference here is that instead of interpolating one value (the color value with Gouraud shading), the x and y coordinates are also interpolated.

You should now understand why this algorithm cannot function correctly. The interpolation which we're using here is linear but the perspective is not linear. Therefore, a distortion results that is noticeable as a bump in the texture map. Look at this distortion closely in the texture mapping parts. You won't be able to make out the "bend" so easily with the objects used here since the surfaces are relatively small. Therefore, the flaw isn't as noticeable.

```asm
STRUC TLine
  x     dw ?
  Tx    dd ?
  Ty    dd ?
  Ticx  dd ?
  Ticy  dd ?
  ddy   dw ?
  ddx   dw ?
  icx   dw ?
  error dw ?
  yTo   dw ?
  xTo   dw ?
ENDS
STRUC TEntry
  x     dw ?
  Tx    dd ?
  Ty    dd ?
ENDS
```

\(\text{Tx}\) and \(\text{Ty}\) contain the x and y coordinates in the texture map. \(\text{Ticx}\) and \(\text{Ticy}\) are the increment values for the texture coordinates.
Also, two variables (\texttt{TexXSize} and \texttt{TexYSize}) containing the height and width of the texture map are new.

\begin{verbatim}
EXTRN _VirtualVSeg :Word
EXTRN xSize :Word
EXTRN TexXSize :Word
EXTRN TexYSize :Word
LL        TLine ?
RL        TLine ?
yMin      dw ?
yMax      dw ?
min       dw ?
L         dw ?
R         dw ?
yi        dw ?
K         dw ?
k1       TEntry ?
k2       TEntry ?
DeltaX   dw ?
uicx     dd ?
uicy     dd ?
TextureSeg dw ?
\end{verbatim}

As we mentioned, the routine is similar in structure to the Gouraud shading routine. Therefore, we'll only describe the new material and changes.

\begin{verbatim}
;_____________________________________________________________________
; procedure CxyTexturedPoly(P,T :Pointer; Count :Word;
;                           Texture :Pointer);
;_____________________________________________________________________
PROC CxyTexturedPoly FAR
ARG Texture:DWord, Count:Word, T:DWord, P:DWord = @@return
enter  0,0
cld
The texture must be located at an address divisible by 16 so the offset value of the address can be ignored. In allocating memory with \texttt{Getmem}, only those addresses are given which are PARA aligned. Therefore, you won't need to worry when you allocate the texture map on the heap. Next, the highest point is determined.

\begin{verbatim}
mov    ax,[WORD PTR 2+Texture]
mov    [TextureSeg],ax
mov    [min],-1
mov    [yMin],200
mov    [yMax],-1
xor    cx,cx
les    si,[P]
mov    bx,si
inc    si
inc    si
@011:
mov    ax,[es:si]
cmp    ax,[yMin]
jnl    @012
mov    [yMin],ax
mov    [min],cx
@012:
mov    ax,[es:si]
cmp    ax,[yMax]
jnge   @013
\end{verbatim}
Chapter 14

The Secrets Behind DOOM

492

```assembly
mov    [yMax], ax
@@013:
    add    si, 4
    inc    cx
    cmp    cx, [Count]
    jb     @@011
    cmp    [yMin], 199
    jg     @@71
    cmp    [yMax], 0
    jl     @@71
    mov    ax, 199
    cmp    [yMax], ax
    jle    @@14
    mov    [yMax], ax
@@14:
    mov    ax, [yMin]
    mov    [yi], ax

LL and RL are then loaded with the corresponding values similar to the Gouraud shading routine.

mov    ax, [min]
mov    [L], ax
mov    [R], ax
shr    ax, 2
add    bx, ax
mov    ax, [es:bx]
mov    [LL.x], ax
mov    [RL.x], ax
call   _InitLeft
call   _InitRight
mov    ax, [yMin]
inc    ax
mov    ax, [yMax]
jg     @@71
@@21:

This is where the loop, which searches for the individual horizontal lines, begins.

mov    [yi], ax
@@021:

Determining the x value is the same as with Gouraud shading.

mov    ax, [LL.x]
mov    dx, [LL.dxy]
mov    bx, [LL.dxy]
cmp    bx, [LL.ddx]
jge    @@33
cmp    ax, [LL.xTo]
jle    @@33
@@31:
    add    bx, dx
    add    ax, [LL.icx]
mov    [LL.error], bx
mov    [LL.x], ax
mov    bx, [LL.ddx]
jge    @@32
cmp    ax, [LL.xTo]
jne    @@31
@@32:
    mov    [LL.x], ax
@@33:
```
The first step of texture map interpolation occurs here.

```
sub    bx,[LL.ddx]
mov    [LL.error],bx
```

This is also true for the right line (which doesn't have to be on the right side).

```
mov    eax,[LL.Ticx]
add    [LL.Tx],eax
mov    eax,[LL.Ticy]
add    [LL.Ty],eax
```

```
mov    ax,[RL.x]
mov    dx,[RL.ddy]
mov    bx,[RL.error]
cmp    bx,[RL.ddx]
jge    @@43
```  

```
cmp    ax,[RL.xTo]
je     @@43
@@41:
add    bx,dx
add    ax,[RL.icx]
mov    [RL.error],bx
mov    [RL.x],ax
```  

```
cmp    bx,[RL.ddx]
jge    @@42
```  

```
cmp    ax,[RL.xTo]
jne    @@41
@@42:
mov    [RL.x],ax
```

```
jmp    @@43:
sub    bx,[RL.ddx]
mov    [RL.error],bx
mov    [RL.Ticx],eax
mov    eax,[RL.Ticy]
add    [RL.Ty],eax
```

`TextureVLine` is called after copies of `x`, `Tx` and `Ty` are made. It then controls drawing the horizontal line.

```
cmp    [yi],0
jl     @@60
push   ds
pop    es
mov    si,OFFSET LL
mov    di,OFFSET k1
movsd
movsd
movsw
mov    si,OFFSET RL
mov    di,OFFSET k2
movsd
movsd
movsw
```

call   TextureVLine

Check to see if an end point has been reached.

```
@@60:
mov    ax,[yi]
cmp    [LL.yTo],ax
```
The Secrets Behind DOOM

Chapter 14

If yi is less than or equal to yMax, then loop.

If yi is less than or equal to yMax, then loop.

Here is the beginning, which is familiar.

Tx and Ty are loaded with their corresponding values.
The Secrets Behind DOOM

Chapter 14

PC underground

les si,[T]
mov bx,[k]
shl bx,2
mov ax,[es:si+bx]
shl eax,16
mov [LL.Tx],eax
mov ax,[es:si+bx+2]
shl eax,16
mov [LL.Ty],eax
mov [LL.icx],1
mov ax,[LL.xTo]
sub ax,[LL.x]
jns @@15
neg [LL.icx]
eg
@@15:
mov [LL.ddx],ax
mov [LL.error],0

Ticx and Ticy are determined by dividing the difference between the old and new Ticx/Ticy by the number of y rows (ddy) between them.

mov bx,[L]
shl bx,2
mov ax,[es:si+bx]
shl eax,16
sub eax,[LL.Tx]
movzx ecx,[LL.ddy]
idiv ecx
mov [LL.Ticx],eax
mov ax,[es:si+bx+2]
shl eax,16
sub eax,[LL.Ty]
movzx ecx,[LL.ddy]
idiv ecx
mov [LL.Ticy],eax
@@16:
ret

The same is true for the right line.

;---------------------------------------------------------------------
; procedure InitRight;
;---------------------------------------------------------------------
PROC _InitRight NEAR
les si,[P]
@@11:
mov dx,[K]
mov bx,[K]
mov [K],bx
shl bx,2
mov ax,[es:si+bx]
mov [RL.x],ax
mov bx,[R]
mov ax,[Count]
dec ax
cmp bx,ax
jne @@12

495
The Secrets Behind DOOM

Chapter 14

The most interesting part of the routine is next: The part that draws the horizontal line.

```
mov    bx,-1  
@@012:
  inc    bx
  mov    cx,bx
  mov    [R],bx
  shl    bx,2
  mov    ax,[es:si+bx+2]
  mov    [RL.yTo],ax
  sub    ax,[yi]
  jns    @@013
  neg    ax
@@013:
  mov    [RL.ddy],ax
  cmp    ax,0
  jne    @@014
  cmp    cx,[min]
  je     @@16
  jmp    @@11
@@014:
  mov    ax,[es:si+bx]
  mov    [RL.xTo],ax
  les    si,[T]
  mov    bx,[k]
  shl    bx,2
  mov    ax,[es:si+bx]
  shl    eax,16
  mov    [RL.Tx],eax
  mov    ax,[es:si+bx+2]
  shl    eax,16
  mov    [RL.Ty],eax
  mov    [RL.icx],i
  mov    ax,[RL.xTo]
  sub    ax,[RL.x]
  jns    @@015
  neg    [RL.icy]
  neg    ax
@@015:
  mov    [RL.ddx],ax
  mov    [RL.error],0
  mov    bx,[k]
  shl    bx,2
  mov    ax,[es:si+bx]
  shl    eax,16
  sub    eax,[RL.Tx]
  cdq
  movzx  ecx,[RL.ddy]
  idiv   ecx
  mov    [RL.Ticx],eax
  mov    ax,[es:si+bx+2]
  shl    eax,16
  sub    eax,[RL.Ty]
  cdq
  movzx  ecx,[RL.ddy]
  idiv   ecx
  mov    [RL.Ticy],eax
@@016:
  ret
ENDP
```
First, the difference is obtained from \( k1.x \) and \( k2.x \) and saved in \( \text{deltaX} \).

\[
\begin{align*}
\text{movsx} & \quad \text{ebx,}[k2.x] \\
\text{sub} & \quad \text{bx,}[k1.x] \\
\text{jns} & \quad @@01 \\
\text{neg} & \quad \text{bx} \\
@@01: & \quad \text{inc} \quad \text{bx} \\
\text{mov} & \quad [\text{DeltaX}],bx
\end{align*}
\]

\( \text{uicx} \) and \( \text{uicy} \) receive the increment values for \( Tx \) and \( Ty \).

\[
\begin{align*}
\text{mov} & \quad \text{eax,}[k2.Tx] \\
\text{sub} & \quad \text{eax,}[k1.Tx] \\
\text{cdq} & \\
\text{idiv} & \quad \text{ebx} \\
\text{mov} & \quad [\text{uicx}],\text{eax} \\
\text{mov} & \quad \text{eax,}[k2.Ty] \\
\text{sub} & \quad \text{eax,}[k1.Ty] \\
\text{cdq} & \\
\text{idiv} & \quad \text{ebx} \\
\text{mov} & \quad [\text{uicy}],\text{eax}
\end{align*}
\]

Lines outside of the screen get clipped.

\[
\begin{align*}
\text{movsx} & \quad \text{eax,}[k1.x] \\
\text{cmp} & \quad \text{ax,}[k2.x] \\
\text{jnl} & \quad @@c3 \\
\text{movsx} & \quad \text{eax,}[k1.x] \\
\text{or} & \quad \text{ax,ax} \\
\text{jnl} & \quad @@c1 \\
\text{push} & \quad \text{eax} \\
\text{imul} & \quad [\text{uicx}],\text{eax} \\
\text{sub} & \quad [\text{k1.Tx}],\text{eax} \\
\text{pop} & \quad \text{eax} \\
\text{imul} & \quad [\text{uicy}],\text{eax} \\
\text{sub} & \quad [\text{k1.Ty}],\text{eax} \\
\text{mov} & \quad [\text{k1.x}],0 \\
@@c1: & \\
\text{movsx} & \quad \text{eax,}[k2.x] \\
\text{cmp} & \quad \text{ax,319} \\
\text{jng} & \quad @@c2 \\
\text{sub} & \quad \text{eax,319} \\
\text{push} & \quad \text{eax} \\
\text{imul} & \quad [\text{uicx}],\text{eax} \\
\text{sub} & \quad [\text{k2.Tx}],\text{eax} \\
\text{pop} & \quad \text{eax} \\
\text{imul} & \quad [\text{uicy}],\text{eax} \\
\text{sub} & \quad [\text{k2.Ty}],\text{eax} \\
\text{mov} & \quad \text{ax,319} \\
\text{mov} & \quad [\text{k2.x}],\text{ax} \\
@@c2: & \\
\text{cmp} & \quad [\text{k1.x}],\text{ax} \\
\text{jnie} & \quad @@40
\end{align*}
\]
The Secrets Behind DOOM

Chapter 14

PC underground

jmp @@c6
@@c3:
  movsx eax, [k1.x]
  cmp ax, 319
  jng @@c4
  sub eax, 319
  push eax
  imul [uicx]
  add [k1.Tx], eax
  pop eax
  imul [uicy]
  add [k1.Ty], eax
  mov [k1.x], 319
@@c4:
  movsx eax, [k2.x]
  or ax, ax
  jnl @@c5
  push eax
  imul [uicx]
  add [k2.Tx], eax
  pop eax
  imul [uicy]
  add [k2.Ty], eax
  xor ax, ax
  mov [k2.x], ax
@@c5:
  cmp ax, [k1.x]
  jnle @@40
@@c6:

DeltaX is recalculated here since it may have changed due to clipping.

mov bx, [k1.x]
sub bx, [k2.x]
jns @@11
neg bx
@@11:
  inc bx
  mov [DeltaX], bx

Now the main loop is initialized.

mov eax, [k2.Tx]
cmp eax, [k1.Tx]
jng @@21
movzx edi, [k1.x]
mov ax, [xSize]
mul [yi]
add di, ax
mov ax, [WORD PTR 2+k1.Ty]
mul [TexXSize]
add ax, [WORD PTR 2+k1.Tx]
add ax, [WORD PTR Texture]
movzx esi, ax
mov eax, [k2.Ty]
cmp eax, [k1.Ty]
jng @@12
mov ax, [TexXSize]
mul [WORD PTR 2+uicy]
mov bp, [WORD PTR 2+uicx]
add bp, ax
mov dx, [TexXSize]
The Secrets Behind DOOM

Chapter 14

PC underground

jmp @@13
@@12:
  neg [uicy]
  mov ax,[TexXSize]
  mul [WORD PTR 2+uicy]
  mov bp,[WORD PTR 2+uicx]
  sub bp,ax
  mov dx,[TexXSize]
  neg dx
@@13:
  push 1
  jmp @@24
@@21:
  neg [uicx]
  movzx edi,[k2.x]
  mov ax,[xSize]
  mul [y1]
  add di,ax
  mov ax,[WORD PTR 2+k2.Ty]
  mul [TexXSize]
  add ax,[WORD PTR 2+k2.Tx]
  add ax,[WORD PTR Texture]
  movzx esi,ax
  mov eax,[k2.Ty]
  cmp eax,[k1.Ty]
  jng @@22
  mov ax,[TexXSize]
  mul [WORD PTR 2+uicy]
  mov bp,[WORD PTR 2+uicx]
  sub bp,ax
  mov dx,[TexXSize]
  neg dx
  jmp @@23
@@22:
  neg [uicy]
  mov ax,[TexXSize]
  mul [WORD PTR 2+uicy]
  mov bp,[WORD PTR 2+uicx]
  add bp,ax
  mov dx,[TexXSize]
@@23:
  push -1
@@24:
  movzx ebx,[WORD PTR uicx]
  rol ebx,16
  rol ebp,16
  mov bp,[WORD PTR uicy]
  rol ebp,16
  add edi,ebx
  adc esi,ebp
  jnc @@30
  add si,dx
@@30:
  pop bx
  mov ax,[k1.x]
  cmp ax,[k2.x]
  jng @@31
  neg bx
@@31:
  mov cx,[DeltaX]
  mov es,[_VirtualVSeg]
  mov ds,[TextureSeg]
Chapter 14

The Secrets Behind DOOM

You probably weren't expecting the main loop to be so small. If you understood the carry trick in Gouraud shading, then you shouldn't have any problems understanding what happens here.

```assembly
@@32:
    mov    al, [si]
    mov    [es:di], al
    add    edi, ebx
    adc    esi, ebp
    jnc    @@33
    add    si, dx
@@33:
    dec    cx
    jnz    @@32
```

The si register contains the offset in the texture map, while di contains the offset in video RAM or in the video buffer. The hi word of edi contains the decimal part of the x coordinate in the texture. The hi word of ebx contains the decimal part of the x increment of the texture. Bx is either one or minus one, depending on whether the line is drawn from right to left or from left to right. In case edi overflows, esi, whose hi word contains the decimal part of the y coordinate in the texture, is incremented by one. Ebp contains the decimal part of the y increment in the hi word and the integral part of the x and y increment of the texture coordinates in the lo word. If esi overflows, dx, which contains the width of the texture, is added to si.

```assembly
@@40:
    pop    bp
    pop    ds
    ret
ENDP
```

DOOM Has Great 3-D Vector Graphics

In this section we'll see you how to combine all the routines we've introduced so far. You may be surprised to learn the main program which performs the complete display is written in Pascal (except for the vector math, Gouraud and texture routines).

You may have expected that it was written entirely in assembly language. However, routines to load 3-D data or texture maps don't have to be fast. Also, many places in the program use so little time that it makes no difference whether they are written in assembly language or in Pascal.

Furthermore, the code is much easier to read with a high-level language as a basis. This readability helps you in two ways:

1. Because the code is easier to read, you can find areas in the code faster and possibly rework the slow portions after you've identified them with a profiler. It's harder for errors to sneak into the program, or if they do, you can detect them relatively quickly.

2. High-level languages normally include an extensive selection of standard routines (Disk IO, Command Line parsing, etc.). If you don't use a high-level language, you'll have to program the assembly language routines yourself.
**The Secrets Behind DOOM**

Chapter 14

PC underground

### Type Declarations

```pascal
type
  PPointList = ^TPointList;
  TPointList = array[0..2447] of T3D;
  PProjectList = ^TProjectList;
  TProjectList = array[0..2447] of T2D;
  PMappingList = ^TMappingList;
  TMappingList = array[0..2447] of T2D;
  PColorList = ^TColorList;
  TColorList = array[0..2447] of Byte;
```

**TFace** is only a prototype. **GetMem** always requests only as much memory as **TFace** needs. For example, for a rectangle this would be 2+4*2. In this way only as much memory is consumed as is required. You have to do without Turbo Pascal’s range checking (you don’t need it anyway).

**Count** specifies the number of corner coordinates. **List** specifies the index under which the corner point (vertex) can be found in the point list (**TPointList**).

```pascal
TFace = ^TFace;
TFace = RECORD
  Count :Integer;
  List  :array[0..255] of Word;
end;
```

So **FaceList** is an array of pointers to a **TFace**.

```pascal
PFaceList = ^TFaceList;
TFaceList = array[0..2447] of PFace;
TZEntry = ^TZEntry;
TZEntry = RECORD
  z :LongInt;
  P :PFace;
end;
TZList = ^TZList;
TZList = array[0..2047] of TZEntry;
```

**PCount** contains the points of the vector object. **VL** contains the points that haven’t yet been transformed and **TL** contains the transformed points. **PL** receives the projected 3-D coordinates. The corresponding normal vector in **NL** represents each point (we’ll explain this later). **ML** contains a corresponding point in the texture for each point. The color values are saved in **CL** to prevent the corner color values from being calculated twice.

```pascal
var
  PCount  :Integer;
  VL      :PPointList;
  TL      :PPointList;
  PL      :PProjectList;
  NL      :PPointList;
  ML      :PMappingList;
  CL      :PColorList;
ZL      :PZList;
```

**ZL** is a list containing a distance and a pointer to the object of the distance. The depth-sort algorithm is used as a hidden surface algorithm. This algorithm sorts the surfaces by their distance from the viewer and then displays the surfaces. The z values are recalculated for each frame. The list is then sorted with a Q-Sort. After that, the surfaces can be displayed in the correct sequence by simply running through the list.
**FCount** contains the number of faces of the object and **FL** contains pointers to them.

```pascal
FCount : Integer;
FL : PFaceList;
```

**Tx, ty, tz** contain the current position of the vector object. The angles of rotation are contained in **rx, ry, rz** contains the angles of rotation.

```pascal
tx : LongInt;
ty : LongInt;
tz : LongInt;
rx : Integer;
ry : Integer;
rz : Integer;
Texture : Pointer;
DAC : TDACBlock;
OldExit : Pointer;
```

One very important technique which will increase the execution speed is to remove rear sides that aren’t visible. If you cannot see into a vector object, you can remove those faces of which the rear side is visible.

Using the screen coordinates, you can determine whether the front or rear side of an object is visible. It is, however, necessary to have the points defined in the correct sequence for all faces (e.g., rotating to the right for the front and rotating to the left for the rear side). The formula for this is:

\[ S = (y_1 - y_0)(x_2 - x_0) - (y_2 - y_0)(x_1 - x_0) \]

This corresponds to the \( z \) components of the vector cross product. Screen coordinates and color values are copied in two arrays and passed to **CxyShadedPoly** with the number of corner points.

```pascal
procedure ShowShadedFace(F : PFace);
var
  i : Integer;
  w : Word;
  Buffer : array[0..31] of T2D;
  ColBuf : array[0..31] of Byte;
  P2,P1 : T2D;
  A,B : T3-D;
begin
  for i := 0 to F^.Count-1 do begin
    w:=F^.List[i];
    Buffer[i] := PL^[w];
    ColBuf[i] := Cl^[w];
  end;
  VecSub2D(Buffer[1],Buffer[0],P1);
  VecSub2D(Buffer[2],Buffer[0],P2);
  if LongInt(P1[y])*P2[x]-LongInt(P1[x])*P2[y] <= 0 then begin
    CxyShadedPoly(@Buffer,@ColBuf, 3);
  end;
end;
```

In the corresponding texture routine, you also have to check whether the texture extends beyond the border.

```pascal
procedure ShowTextureFace(F : PFace);
var
  i : Integer;
  w : Word;
  Buffer : array[0..31] of T2D;
  TexBuf : array[0..31] of T2D;
end;
```
The Secrets Behind DOOM

Chapter 14

PC underground

P2,P1 :T2D;
A,B :T3-D;
g :T2D;
h1 :Boolean;
h2 :Boolean;
begin
h1:=false;
h2:=false;
for i := 0 to F^.Count-1 do begin
w:=F^.List[i];
Buffer[w] := PL^[w];
g:=ML^[w];
TexBuf[w] := g;
if g[0]>128 then h1:=true;
if g[1]>128 then h2:=true;
end;
if h1 then for i:= 0 to F^.Count-1 do
  if TexBuf[i,0]<128 then Inc(TexBuf[i,0],256);
if h2 then for i:= 0 to F^.Count-1 do
  if TexBuf[i,1]<128 then Inc(TexBuf[i,1],256);
VecSub2D(Buffer[1],Buffer[0],P1);
VecSub2D(Buffer[2],Buffer[0],P2);
if LongInt(P1[y])*P2[x]-LongInt(P1[x])*P2[y] <= 0 then begin
  CxyTexturedPoly(@Buffer,@TexBuf, 3, Texture);
end;
end;

Project controls transformation and projection.

procedure Project(M :TMatrix);
var
  i :Integer;
  V :T3-D;
begin
  for i:= 0 to PCount-1 do begin
    Transform(VL^[i],M,V);
    TL^[i]:=V;
    PL^[i,0] := 160+ (LongInt(V[x])*$200) div V[z];
    PL^[i,1] := 100- (LongInt(V[y])*$200) div V[z];
  end;
end;

The distance list is loaded with the pointers to the faces. This is only done once at the beginning.

procedure InitZList;
var
  i :Integer;
begin
  for i := 0 to FCount-1 do ZL^[i].P := FL^[i]
end;

For the distance, UpdateZList takes the mean value of the z components of the points of the face. Although this is not entirely correct (because to do it properly, you would have to take the mean value of the quantities) this is acceptable for most objects.

procedure UpdateZList;
var
  i,j :Integer;
  _z :LongInt;
  F :PFace;
begin
  for i := 0 to FCount-1 do begin
    ...
The Secrets Behind DOOM
Chapter 14

PC
PC underground

begin
for i := 0 to FCount-1 do begin
   with ZL^[i].P do begin
      _z:=List[0];
      for j:=1 to Count-1 do Inc(_z,TL^[List[j],2]);
      _z:= _z div Count;
   end;
   ZL^[i].z:=_z;
end;
end;

Perhaps you are familiar with the next procedure. It's Borland's Quicksort sample procedure for this program.

procedure SortZList(l, r: Integer);
var
   i, j :Integer;
   x :LongInt;
   S :TZEntry;
begin
   i := l;
   j := r;
   x := ZL^[l+r shr 1].z;
repeat
   while ZL^[i].z < x do Inc(i);
   while ZL^[j].z > x do Dec(j);
   if i <= j then begin
      S := ZL^[i];
      ZL^[i] := ZL^[j];
      ZL^[j] := S;
      Inc(i);
      Dec(j);
   end;
until i > j;
   if i < j then SortZList(l, j);
   if i < r then SortZList(i, r);
end;

A normal vector of a face is always vertical to the face. At this point you may be wondering how a normal vector of a point appears. We simply took the mean value of the normal vectors of the faces that contain the point. We'll perform the color calculation here slightly easier than we did in the preceding section. We don't use the cosine between the viewer vector and the normal vector. Instead, the viewer vector is assumed to be (0,0,1). Although not correct, it looks good because the light falls parallel on the object.

The color value corresponds to the z components of the rotated normal vector. Since 9 bit decimal parts are being used, the z components must be shifted to the right by one to obtain a color value between 0 and 255. For this purpose, it's important the quantity, i.e., the length of the normal vector, is 1. Since the normal vectors are only being rotated and not translated or scaled, this is always guaranteed providing you calculate the normal vectors correctly.

procedure UpdateColorList(R :TMatrix);
var
   i :Integer;
   N :T3-D;
   w :Word;
begin
   for i := 0 to PCount-1 do begin
The procedure `ShowShadedObject` performs all the operations necessary for displaying an object.

First, the procedure clears the screen. The screen here is not the actual screen, but rather a 64K buffer in RAM. Memory accesses in RAM are twice as fast as access to VGA RAM (at least with the ISA bus).

`Create` generates the transformation matrix `M`. `Rotate` builds the rotation matrix `R`. `M` is required for the transformation of the points. `R` is required for the rotation of the normal vectors (they aren’t recalculated every time, but simply rotated accordingly). The procedure `Project` performs these operations. After that, `XL` is updated and sorted. The same thing happens to `CL`.

After that, all the data is calculated and the faces can be displayed. The video buffer is copied to VGA RAM following the display.

```pascal
procedure ShowShadedObject;
var
  M :TMatrix;
  R :TMatrix;
  i :Integer;
begin
  ClearScreen(0);
  Create(tx,ty,tz,$200,$1B0,$200,rx,ry,rz,M);
  Rotate(rx,ry,rz,R);
  Project(M);
  UpdateZList;
  SortZList(0,FCount-1);
  UpdateColorList(R);
  for i := FCount-1 downto 0 do ShowShadedFace(ZL^[i].P);
  ShowVBuffer;
  CheckKey;
end;
```

`ShowTexturedObject` is identical except it doesn’t call `ShowShadedFace`. Instead, it calls `ShowTexturedFace`.

```pascal
procedure ShowTexturedObject;
var
  M :TMatrix;
  i :Integer;
begin
  ClearScreen(0);
  Create(tx,ty,tz,$200,$1B0,$200,rx,ry,rz,M);
  Project(M);
  UpdateZList;
  SortZList(0,FCount-1);
  for i := FCount-1 downto 0 do ShowTextureFace(ZL^[i].P);
  ShowVBuffer;
  CheckKey;
end;
```

The FCE format used here is one that we created. Since it is an ASCII format, no further explanation should be necessary.
procedure InitVectorData(FCE: String; MM: Word);
var
  i: Integer;
  f: Text;
  s: String;
  r: Real;
  c: Word;
begin
  PCount := -1;
  New(VL);
  New(TL);
  New(PL);
  New(ML);
  FillChar(ML^, SizeOf(ML^), 0);
  New(ZL);
  New(NL);
  New(CL);
  FillChar(NL^, SizeOf(NL^), 0);
  FCount := 0;
  New(FL);
  Assign(f, FCE);
  Reset(f);
  if IOResult <> 0 then Error('Opening vector data file...');
  while 1 = 1 do begin
    readln(f, s);
    if s = '[END]' then
      break
    else
      if s = '[POINT]' then begin
        Inc(PCount);
        for i := 0 to 2 do begin
          readln(f, r);
          if IOResult <> 0 then Error('Reading vector data...');
          VL^[PCount, i] := round(r * $200);
        end;
      end
    else
      if s = '[NORMAL]' then begin
        for i := 0 to 2 do begin
          readln(f, r);
          if IOResult <> 0 then Error('Reading vector data...');
          NL^[PCount, i] := round(r * $200);
        end;
      end
    else
      if s = '[MAPPING]' then begin
        readln(f, r);
        if IOResult <> 0 then Error('Reading vector data...');
        ML^[PCount, 0] := round(r * (TexYSize - 1) * MM);
        readln(f, r);
        if IOResult <> 0 then Error('Reading vector data...');
        ML^[PCount, 1] := round(r * (TexYSize - 1) * MM);
      end
    else
      if s = '[FACE]' then begin
        readln(f, c);
        if IOResult <> 0 then Error('Reading vector data...');
        GetMem(FL^[PCount], c * 2 + 2);
        FL^[PCount].Count := c;
        for i := 0 to c - 1 do begin
          readln(f, c);
          if IOResult <> 0 then Error('Reading vector data...');
        end;
      end;
  end;
end;
A size of 256x256 was selected for the textures. This has an important advantage: We can choose texture coordinates that are greater than 256. Since the texture routine uses only a 16 bit register as an offset, the texture can repeat itself automatically. To save disk space, the texture loaded here is only 64x64.

```pascal
procedure InitTEX(TEX :String; var P :Pointer; var DAC :TDACBlock);
var
  f   :file;
  i,j :Word;
begin
  TexXSize:=256;
  TexYSize:=256;
  GetMem(P,$FFFF);
  Assign(f,TEX+'.TEX');
  Reset(f,1);
  for i := 0 to 63 do begin
    BlockRead(f,Mem[Seg(P^):i*256],64);
    for j:= 1 to 3 do
      Move(Mem[Seg(P^):i*256],Mem[Seg(P^):i*256+j*64],64);
  end;
  for j:= 1 to 3 do Move(Mem[Seg(P^):0],Mem[Seg(P^):j*64*256],64*256);
  Close(f);
  Assign(f,TEX+'.PAL');
  Reset(f,1);
  BlockRead(f,DAC,3*256);
  Close(f);
end;
```
FreeMem(P,$FFFF);
end;

procedure LoadPAL(PAL :String; var DAC);
var
  f :file;
begin
  Assign(f,PAL);
  Reset(f,1);
  BlockRead(f,DAC,3*256);
  Close(f);
end;

The position and rotation of the vector object are calculated as a function of the variable \( FC \). \( FC \) is started by a routine that has added itself to the timer interrupt. For detailed descriptions of the timer chip and interrupts, see the chapter on programming add-on chips.

procedure SHADED_DUCK;
begins
  FillChar(Mem[$A000:0],64000,0);
  LoadPAL('NDUCK.PAL',DAC);
  SetDACBlock(0,256,DAC);
  InitVectorData('NDUCK.FCE',1);
  InitZList;
  FC:=0;
  tz:= 400*$200;
  rx:=-128;
  repeat
    tx:=-200*$200+LongInt(FC) shl 6;
    ry := FC shr 2;
    rz := FC shr 2;
    ShowShadedObject;
  until tx >=0*$200;
  while FC < 4096 do begin
    ry := FC shr 2;
    rz := FC shr 2;
    ShowShadedObject;
  end;
  Dec(FC,4096);
  repeat
    tx:= LongInt(FC) shl 6;
    ry := FC shr 2;
    rz := FC shr 2;
    ShowShadedObject;
    until tx >=200*$200;
  DecVectorData;
end;

procedure SHADED_FACE;
begins
  FillChar(Mem[$A000:0],64000,0);
  LoadPAL('JFACE.PAL',DAC);
  SetDACBlock(0,256,DAC);
  InitVectorData('JFACE.FCE',1);
  InitZList;
  FC:=0;
  tz:= 300*$200;
  rx:=-128;
  repeat
    tx:=-200*$200+LongInt(FC) shl 6;
    rx := -128+FC shr 2;
    ry := 256+FC shr 2;
    ShowShadedObject;
  until tx >=200*$200;
  DecVectorData;
end;
until tx >=0;
while FC < 4096 do begin
  rx := -128+FC shr 2;
  ry := 256+FC shr 2;
  ShowShadedObject;
end;
Dec(FC,4096);
while FC < 2048 do begin
  if tz > 170*$200 then tz:= 300*$200-LongInt(FC) shl 7;
  rx := -128+FC shr 2;
  ry := 256+FC shr 2;
  ShowShadedObject;
end;
Dec(FC,2048);
repeat
  ty:= -(LongInt(FC) shl 6);
  ShowShadedObject;
  until ty<150*$200;
DoneVectorData;
end;
procedure SHADED_CHOPPER;
begin
  FillChar(Mem[$A000:0],64000,0);
  LoadPAL('HELI.PAL',DAC);
  SetDACBlock(0,256,DAC);
  InitVectorData('HELI.FCE',1);
  InitZList;
  FC:=0;
  tz:= 600*$200;
  rx:=-108;
  repeat
    tx:= -200*$200+LongInt(FC) shl 6;
    ry := FC shr 3;
    ShowShadedObject;
    until tx >=0*$200;
  while FC < 4096 do begin
    ry := FC shr 3;
    ShowShadedObject;
  end;
  Dec(FC,4096);
  repeat
    tz:= 600*$200+LongInt(FC) shl 9;
    tx:= LongInt(FC) shl 8;
    ry := FC shr 4;
    ShowShadedObject;
    until tx >=700*$200;
  DoneVectorData;
end;
procedure TEXTURE_SPHERE;
begin
  FillChar(Mem[$A000:0],64000,0);
  InitTEX('MSPHERE',Texture,DAC);
  SetDACBlock(0,256,DAC);
  InitVectorData('MSPHERE.FCE',1);
  InitZList;
  FC:=0;
  tz:= 600*$200;
  repeat
    tx := (FC shr 3-300)*$200;
    ty := -30*6200+round((1+Sin(FC/100)))*50*$200;
    rx := -FC shr 2;
    rz := -FC shr 2;
  until ty<-150*$200;
DoneVectorData;
end;
The Secrets Behind DOOM

Chapter 14

An **Exit** procedure is necessary here to make certain the installed timer interrupt is removed.

```pascal
procedure NewExit; far;
begin
  ExitProc:=OldExit;
  DoneFrameHandler;
end;
```

**Init13h** sets mode 13h. **InitVBuffer** sets up a 64K video buffer. The segment address of this buffer is saved in **_VirtualVSeg**. **InitFrameHandler** installs the timer handler.

```pascal
var
  i :Integer;
begin
  Init13h;
  InitVBuffer;
  InitFrameHandler;
  OldExit:=ExitProc;
  ExitProc:=@NewExit;
  SHADED_DUCK;
```
TEXTURE_SPHERE;
SHADED_FACE;
TEXTURE_TORUS;
SHADED_CHOPPER;
Done13h;
end.

The following is the source code for the timer unit _FRAME.PAS.

```pascal
var
 FC :LongInt;
const
 Active :Boolean = false;
procedure OldHandler; assembler; asm db 0; db 0; db 0 end;
procedure FrameHandler; far; external; {$L _FRAME.OBJ}
procedure InitFrameHandler;
begin
 GetIntVec($8,Pointer(#OldHandler"));
 SetIntVec($8,@FrameHandler);
 Port[$43]:=$34;
 Port[$40]:=$34;
 Port[$40]:=$12;
 FC := 0;
 Active := true;
end;
procedure DoneFrameHandler;
begin
 if Active then begin
  Port[$43]:=$34;
  Port[$40]:=0;
  Port[$40]:=0;
  SetIntVec($8,Pointer(#OldHandler"));
  Active := false;
 end;
end;
```

The timer handler is located in a TASM module.

```asm
j--------------------------; procedure FrameHandler;
far;
j----------------------------- PROC FrameHandler
push ds
push ax
mov ax,@data
mov ds,ax
inc [PC]
mov al,20h
out 20h,al
pop ax
pop ds
iret
ENDP
```

For those readers interested in performance comparisons, we've performed a few tests. For the UNEATABLE demo, we switched from X mode to mode 13h using a video RAM buffer. This doubled the speed.

The doubling of the speed with the same routines has to do with the ISA bus (with a VESA local or PCI bus you won't notice any great differences in speed). Unfortunately, ISA is extremely slow. Thus it is faster to
keep a buffer in RAM and always copy it to video RAM, which runs rather smoothly because of the REPNE MOVSD instruction.

As you can imagine, it can go even faster. These routines, developed for the 32-bit WATCOM C (CONTAGION demo), are about three times as fast. For example, the Gouraud shading routine creates 6510 overlapping "threads" on a 486 DX 66 (and there is no limit to the number of times the objects can intersect each other), the Gouraud shaded texture routine makes more than 4000. 32 bit protected mode offers a great advantage in this connection. This type of object requires enormous amounts of memory that no normal MS-DOS program can provide in a simple, fast manner. You would have to use EMS or Real mode FlatMem (both will be explained in one of the following chapters). However, to run your programs in 32 bit protected mode you would have to either program completely in assembly language (PMODE Extender from Tran) or else purchase a 32 bit compiler such as WATCOM C/C++.

One advantage of these routines that cannot be overlooked is that they are easy to understand. The data structures that are used are very easy to read and "hacker code" such as self-modifying protected mode code and the development of loops isn’t used. As you can see, we haven’t reached the end yet, not by a long shot. Consider this an opportunity to write your own routines. Invest a little time and you’ll be able to do it. In real time you can even achieve such effects as Phong shading, environment mapping, bump mapping and many other effects. So there’s a lot to be done.

If you attempt to use the routines we’ve just introduced to you to fly through an object, you probably won’t make it. The program will crash with a "Division by zero" error.

If you take a look at the Project procedure, the problem is obvious. Since you are dividing by the z coordinates of the point, they can never be zero. You will also have problems if the z coordinates are negative.

You can solve this problem easily by clipping all the points whose z coordinates are less than or equal to zero.

![Clipping a line (x,z) view](image)

The gradients can be equated:
This can be transformed to:

\[ x_i = x_1 + \frac{x_2 - x_1}{z_2 - z_1} \left( z_i - z_1 \right) \]

If you make the same considerations for \( y \), you get:

\[ y_i = y_1 + \frac{y_2 - y_1}{y_2 - y_1} \left( z_i - z_1 \right) \]

So why wasn’t \( z_c \) simply set to zero? It’s better to set \( z_c \) to a value that is greater than zero. In the following, a clip removal of 5 will be used.

By clipping, we are now in a position to create a three-dimensional world and move through it.

```pascal
const
  zClipPlane = $5*$200;

procedure GetCrossing(A,B:T3-D; var C:T3-D);
var
  L1,L2 : LongInt;
begin
  L1 := zClipPlane - A[2];
  if L2 = 0 then begin
    C := A;
    exit;
  end;
  C[0] := FixDiv(FixMul(L1, B[0] - A[0]), L2) + A[0];
end;
```

Besides the projected coordinates, information about whether the point is visible is also saved.
type
  TProject = RECORD
    P :T2D;
    Visible :Boolean;
    _res :array[1..3] of Byte;
  end;
PPointList = ^TPointList;
TPointList = array[0..2447] of T3-D;
PProjectList = ^TProjectList;
TProjectList = array[0..2447] of TProject;
PMappingList = ^TMappingList;
TMappingList = array[0..2447] of T2D;
PColorList = ^TColorList;
TColorList = array[0..2447] of Byte;

PFace = ^TFace;
TFace = RECORD
  Count   :Integer;
  texture :Integer;
  List    :array[0..255] of Word;
end;
PFaceList = ^TFaceList;
TFaceList = array[0..2447] of PFace;

PZEntry = ^TZEntry;
TZEntry = RECORD
  z  :LongInt;
  P  :PFace;
end;
PZList = ^TZList;
TZList = array[0..2047] of TZEntry;

This time, the texture coordinates of a surface are equal for each face.

const
  PointDef :array[0..3] of T2D = (
    (0,0),
    (319,0),
    (319,199),
    (0,199)
  );
var
  M   :TMatrix;
  rx,ry,rz :Integer;
  Textures :array[0..2] of Pointer;
  DAC    :TDACBlock;
  t     :T3-D;
PCount :Integer;
VL    :PPointList;
TL    :PPointList;
PL    :PProjectList;
PCount :Integer;
FL    :PFaceList;
ZL    :PZList;

The procedure that draws the surfaces is expanded to handle the clipping. To guarantee the fastest possible processing, the procedure checks each possible alternative. How the procedure proceeds depends on whether the first point is visible.

procedure DrawFace(F :PFace);
var
Buffer : array[0..15] of T2D;
i, j,
p, q, l, h : Integer;
P2, P1 : T2D;
V : T3-D;
Clip : Boolean;
procedure Calc;
begin
  with F^ do begin
    GetCrossing(TL^[List[i]], TL^[List[h]], V);
    Project(V, Buffer[j]);
  end;
end;
begin
  Clip := false;
  with F^ do begin
    j := 0;
    i := 0;
    h := Count-1;
    if PL^[List[0]].Visible then begin
      Copy all visible points.
      while PL^[List[i]].Visible do begin
        Buffer[j] := PL^[List[i]].P;
        Inc(j);
        Inc(h);
        if h >= Count then h := 0;
        Inc(i);
        if i >= Count then break;
      end;
      if i < Count then begin
        If there are still more points, then calculate the point of intersection.
        Clip := true;
        Calc;
        Inc(j);
      end
    end;
    Skip all points that aren't visible.
    while not PL^[List[i]].Visible do begin
      Inc(i);
      if i >= Count then i := 0;
      Inc(h);
      if h >= Count then h := 0;
    end;
    Calculate point of intersection.
    Calc;
    Inc(j);
  end;
  Still points? If yes, then copy.
  if i <> 0 then begin
    while PL^[List[i]].Visible do begin
      Buffer[j] := PL^[List[i]].P;
      Inc(j);
      Inc(h);
      if h >= Count then h := 0;
    end;
  end;
Inc(i);
  if i >= Count then break;
end;
end;
end;

Skip all points that are not visible.

while not PL[List[i]].Visible do begin
  Inc(i);
  if i >= Count then break;
end;

Face not visible at all?

if i < Count then begin

If visible, calculate point of intersection.

  h := i-1;
  Calc;
  Inc(j);

And then copy all visible points.

  while PL[List[i]].Visible do begin
    Buffer[j] := PL[List[i]].P;
    Inc(j);
    Inc(h);
    if h >= Count then h := 0;
    Inc(i);
    if i >= Count then i := 0;
    end;

And finally, calculate the point of intersection one more time.

  Calc;
  Inc(j);
end;
end;

Rear side?

  VecSub2D(Buffer[1],Buffer[0],P1);
  VecSub2D(Buffer[2],Buffer[0],P2);
  if LongInt(P1[y])*P2[x]-LongInt(P1[x])*P2[y] <= 0 then begin
    if not Clip then begin
      CxyTexturedPoly(@Buffer,@PointDef,j,Textures[Texture])
    end;
  end;
end;

TransformPoints transforms and projects the coordinates of the points. In addition, the routine stores information as to whether the point is visible in Visible.
The Secrets Behind DOOM

Chapter 14

PC underground

procedure TransformPoints(M :TMatrix);
  var
    i :Integer;
    V :T3-D;
  begin
    for i := 0 to FC-1 do begin
      Transform(VL^[i],M,TL^[i]);
      if TL^[i][2]<zClipPlane then PL^[i].Visible:=false
      else begin
        PL^[i].Visible:=true;
        Project(TL^[i],PL^[i].P);
      end;
    end;
  end;
end;

The ShowWorld procedure draws the vector world. MakeMatrix is defined in _MATRIX.PAS. For this vector world, it is more advantageous if the translation is executed first, then the y, x and z rotation, and finally the scaling.

procedure ShowWorld;
  var
    i :Integer;
  begin
    if KeyPressed then begin
      while KeyPressed do ReadKey;
      halt(0);
    end;
    ShowVBuffer;
    UpdateZList;
    SortZList(0, FC-1);
    MakeMatrix(t[0],t[1],t[2], $200,$1B0,$200, rx,ry,rz, M);
    TransformPoints(M);
    for i := FC downto 0 do DrawFace(ZL^[i].P);
  end;
end;

This procedure follows Gouraud shading/texture mapping. Position and rotation are determined by FC. Remember, a different fixed point arithmetic module is used. This new fixed point arithmetic module has a larger sine table. That means the rotation can be executed in more precise steps.

procedure DOOM_PART;
  var
    i :Word;
  begin
    t[0] := 0;
    t[1] := -6*$200;
    t[2] := -30*$200;
    rx := 0;
    ry := 0;
    rz := 0;
    FC:= 0;
    i := 0;
    while FC < 512 do begin
      ry := FC shr 2;
      ShowWorld;
    end;
    Dec(FC,512);
    while FC < 1024 do begin
      ry := 512 shr 2-FC shr 2;
      ShowWorld;
    end;
end;
The Secrets Behind DOOM

Chapter 14

Dec(FC,1024);
while FC < 512 do begin
  rz := (FC shr 1);
  ry := -128+FC shr 1;
  rx := FC shr 1;
  ShowWorld;
end;
Dec(FC,512);
rx := 256;
while FC < 512 do begin
  rz := 256-FC shr 1;
  ry := 128-FC shr 1;
  ShowWorld;
end;
Dec(FC,512);
rz := 0;
ry := -128;
while FC < 512 do begin
  rz := -FC;
  rx := 256-FC shr 1;
  ShowWorld;
end;
Dec(FC,512);
rz := 0;
while FC < 4096-512+256 do begin
  rz := -512-FC;
  t[0] := FC;
  t[2] := -$3C00+FC*$14;
  ShowWorld;
end;
Dec(FC,4096-512+256);
t[0] := 4096-256;
while FC < 896 do begin
  t[2] := -$3C00+(4096-256)*$14+FC*17;
  ShowWorld;
end;
t[2] := -$3C00+(4096-256)*$14+896*17;
Dec(FC,896);
ry := -1024;
while FC < 512 do begin
  rz := 192-FC;
  ShowWorld;
end;
Dec(FC,512);
while FC < 1048 do begin
  t[0] := 4096-256+FC*$15*2;
  rz := 192-FC;
  ShowWorld;
end;
Dec(FC,1048);
end;

320x320 images are used here as textures.

procedure LoadTexture(TEX :String; var P :Pointer);
var
  f :file;
begin
  GetMem(P,64000);
  Assign(f,TEX);
  Reset(f,1);
  if IOResult <> 0 then begin
    writeln('['ERROR']: File not found...');
    halt(1);
  end;
  BlockRead(f,P^,64000);
  Close(f);
end;

As we said, the format is self-explanatory.

procedure LoadVectorData;
var
  f :Text;
  s :String;
  i :Integer;
  r :Real;
  v :Integer;
begin
  PCount:=0;
  FCount:=0;
  New(VL);
  New(TL);
  New(PL);
  New(FL);
  New(ZL);
  Assign(f,'DUNGEON.FCE');
  Reset(f);
  if IOResult <> 0 then begin
    writeln('['ERROR']: Opening vector data failed...');
    halt(1);
  end;
  while not eof(f) do begin
    readln(f,s);
    if s='[POINT]' then begin
      for i:=0 to 2 do begin
        readln(f,r);
        VL^[PCount][i]:=round(r*$200);
      end;
      Inc(PCount);
    end
    else
      if s='[FACE]' then begin
        readln(f,v);
        GetMem(FL^[FCount],4+v*2);
        FL^[FCount].Count:=v;
        for i:=0 to v-1 do begin
          readln(f,FL^[FCount].List[i]);
        end;
      end
    else
      if s='[TRIANGLE]' then begin
        readln(f,t);
        FL^[FCount].Count:=3;
        for i:=0 to 2 do begin
          readln(f,FL^[FCount].List[i]);
        end;
      end
    else
      if s='[TEXT]' then begin
        readln(f,s);
        Assign(f,s);
The Secrets Behind DOOM

Chapter 14

You can find _MATRIX.PAS on the companion CD-ROM

Here is the source code for _MATRIX.PAS:

Procedure **Translate** generates a translation matrix.

```pascal
procedure Translate(tx,ty,tz :LongInt; var C :TMatrix);
var
    i : Integer;
begin
    Zero(C);
    for i := 0 to 3 do C[i,i] := $200;
    C[0,3] := -tx;
    C[1,3] := -ty;
    C[2,3] := -tz
end;
```

Procedure **Scale** generates a scaling matrix.

```pascal
procedure Scale(sx,sy,sz :LongInt; var C :TMatrix);
begin
    Zero(C);
    C[0,0] := sx;
    C[1,1] := sy;
    C[2,2] := sz;
end;
```
Rotate generates a rotation matrix. It generates one around the x-axis for m=1, a second around the y-axis for m=2 and the third rotation matrix around the z-axis for m=3.

```pascal
procedure Rotate3-D(m : Integer; Theta : Integer; Var A : TMatrix);
var
  m1, m2 : Integer;
  c, s   : LongInt;
begin
  Zero(A);
  A[m-1,m-1] := $200;
  m1 := (m Mod 3) + 1;
  m2 := (m1 Mod 3);
  m1 := m1 - 1;
  c := FixCos(-Theta);
  s := FixSin(-Theta);
  A[m1,m1] :=  c;
  A[m1,m2] :=  s;
  A[m2,m2] :=  c;
  A[m2,m1] := -s
end;
```

Multiply the two matrices so they are linked. Remember, the second parameter (not the first parameter) is executed first.

```pascal
procedure MakeMatrix(Tx,Ty,Tz,Sx,Sy,Sz :LongInt; Rx,Ry,Rz :Integer;
  var M :TMatrix);
var
  S,Mx,My,Mz,T,M1,M2,M3 :TMatrix;
begin
  Scale(Sx,Sy,Sz,S);
  Rotate3-D(1,Rx,Mx);
  Rotate3-D(2,Ry,My);
  Rotate3-D(3,Rz,Mz);
  Translate(Tx,Ty,Tz,T);
  Cross(Mx,My,M1);
  Cross(Mz,M1,M2);
  Cross(M2,T,M3);
  Cross(S,M3,M);
end;
```
Windows 95 is finally shipping and millions of users are already using environment. Windows 95 is a genuine operating system, unlike its predecessors Windows 3.x or Windows for Workgroups 3.11. Now instead of loading DOS before running Windows 3.x, Windows 95 starts up immediately. Although this new way of starting has many advantages, it also has a few disadvantages.

Using DOS Programs On A Windows 95 System

If you’re planning to use many of your older DOS programs on your Windows 95 PC, you may have to make adjustments before they can run. Basically, you have three ways to run these older programs.

Starting DOS programs without Windows 95

One way is to press $4$ while starting your computer. Then MS-DOS starts instead of Windows 95. Remember, you cannot press $4$ until the message “Starting Windows 95” appears on the screen. Then you have less than three seconds to press $4$. Your earlier version of MS-DOS starts as if Windows 95 was not installed on your computer.

Both your CONFIG.SYS and AUTOEXEC.BAT files are loaded and executed like before, which makes it possible to use your old MS-DOS programs, even those ancient programs that run using MS-DOS 3.31.

Of course, you can do this only if you installed Windows 95 on top of your older MS-DOS system. You can also run Windows 3.1/3.11 if Windows 95 was installed in a separate directory. Refer to your Win 95 manual under "Installing Win 95 Options."

You can also reboot your PC by choosing Shut Down in the Task Bar. The following screen will appear:
Select "Restart the computer in MS-DOS mode" and your PC will restart to the C:> prompt.

**Starting the Windows 95 DOS compatibility box**

A second way of running older DOS programs is to press F8 while rebooting the PC. When the computer is finished with the BIOS power on self test (POST), has found, detected and initialized your hardware and informed you of this with one beep, press or hold down F8.

At this point, the Windows 95 Startup Menu will appear:

```
Microsoft Windows 95 Startup Menu
=================================
1. Normal
2. Logged (BOOTLOG.TXT)
3. Safe mode
4. Safe mode with network support
5. Step-by-step confirmation
6. Command prompt only
7. Safe mode command prompt only
8. Previous version of MS-DOS

Enter a choice:  1
F5=Safe mode Shift+F5=Command prompt
Shift+F8=Step-by-step confirmation [N]
```

Now select "6. Command prompt only" by pressing Enter or by selecting it with the cursor. Confirm your selection by pressing Enter. The computer immediately starts up with the DOS prompt. You are now in the DOS compatibility box of Windows 95. Both CONFIG.SYS and AUTOEXEC.BAT system configuration files are executed. However, this time the two files are not exact copies of your originals. Instead, all the
Windows 95 From The Underground

Chapter 15

MS-DOS driver files required by Windows 95, such as MSCDEX 2.25 or the MODE command, are loaded from the Windows 95 directory. Windows 95 also changes the path to suit its own requirements.

Now you can install, start, delete or rename your MS-DOS programs and perform all the other familiar tasks in MS-DOS. You can also change your AUTOEXEC.BAT and CONFIG.SYS system configuration files.

Starting the DOS box without CONFIG.SYS and AUTOEXEC.BAT

A third way is to start the DOS compatibility box without executing CONFIG.SYS and AUTOEXEC.BAT, select “7. Safe mode command prompt only”. You’ll return to the DOS prompt when you press Enter to confirm your choice.

Use this method to boot the DOS compatibility box whenever your DOS program that you want to run or install does not work with a Windows 95 driver that is ordinarily loaded.

Installing And Starting DOS Programs Under Windows 95

Instead of using the solutions we described above, you can make your own settings under Windows 95. However, be careful because this can result in so many opened dialog boxes that the Windows 95 desktop will become too cluttered.

Installing DOOM II

DOOM is one of the most popular games to come out in recent years. We’ll show how to install DOOM since it is very typical of programs requiring that you set many options so it will run under Windows 95.

To install DOOM, you’ll need to adjust a lengthy series of Windows 95 settings. Without these changes, DOOM may run using only a very limited number of features or may not even run at all under Windows 95.

First, copy all the DOOM files to a working directory on your hard drive using the Windows 95 Explorer or a DOS box. Then start up the installation program and copy DOOM to the desired directory.

Configuring DOOM under Windows 95

Next open the Windows 95 Explorer and change to the DOOM directory. Select the startup file and press the right mouse button. A small popup menu appears with different options.
The PopUp menu

Select Properties. As you can see in the illustration, the dialog box is divided into six individual sections, or pages. The pages are: General, Program, Font, Memory, Screen and Misc.

General information

The first page, General, specifies additional information about the type, the location, the size, the attributes, the date of creation, date of last change and date the file was last accessed or the object associated with a file. This information is only displayed; you cannot make any changes on this page.
Program Information

On the Program page you can change the title of the program or the association. You can also change the Command Line which contains the search path for DOOM and command to start the program. The Working line gives you the option of changing the directory that DOOM considers its working directory.

We recommend you don’t use this method of specifying the working directory for DOS programs. It’s possible the program may not be able to find the necessary data files. If that happens, the program would not be able to run.

On the other hand, you should definitely select item Close on exit, so that Windows 95 automatically closes the DOS window when you exit DOOM. This will free up all the resources used by DOS. This will benefit any other programs running under Windows 95, since more memory and computing cycles will be available when the program ends.

Click on the [Advanced...] button to proceed to the advanced program settings. These program settings change the behavior of Windows 95, thus you should only make changes after careful consideration.
Some MS-DOS programs will not run under Windows. Either the program immediately terminates or an error message is displayed. To eliminate this problem, select the option ‘Prevent MS-DOS-based programs from detecting Windows’ check box. This guarantees that any program that is interfered with by having Windows in the background will no longer be disturbed.

You can also choose ‘Suggest MS-DOS mode as necessary’. This results in the computer booting and starting the program directly after booting. Normally, you won’t choose this option. Otherwise, by doing so, you would be booting DOOM for example.

If, on the other hand, you choose MS-DOS Mode, the program automatically starts directly in MS-DOS mode and there won’t be any prompt. If you select MS-DOS Mode, you can choose from three additional options. The first is to have the computer warn you by choosing Warning before entering MS-DOS mode, since the computer will then boot.

In addition, you then have the option of choosing between two different MS-DOS configurations. You can use your current system files (Use current MS-DOS configuration) or create a new MS-DOS configuration Specify a new MS-DOS configuration.
You won’t need to make any changes to the advanced program settings with DOOM since the program is quite compatible and doesn’t have any problems running under Windows 95.

Press the [Configuration...] button to display other MS-DOS configuration options: Expanded memory (EMS), Mouse, Disk Cache, MS-DOS Command Line Editor (Doskey) as well as Direct Disk Access.

These options make it possible to run virtually any program under Windows 95, even ancient ones or those which use EMS memory.

It’s not necessary to use a different MS-DOS configuration for DOOM.
Chapter 15
Windows 95 From The Underground

Changing the font size for a DOS program

Sometimes you might want a different font size for a DOS program. Under MS-DOS this wasn’t an easy task to do.

![Image of font size selection]

Choose any font size you wish

However, with Windows 95 using a different font size is simple, since you can choose from font sizes between 4 * 6 pixels to 12 * 16 pixels. By default, this option is set to Auto. Choose the font size from the item Font Size. Selecting “TrueType only”, “Bitmap only” or “Both font types” is a factor in how quickly or slowly the program executes.

Choosing “Bitmap only” as your font and “8 x 12” as your font size has at least two advantages:

- Your DOS program loads faster because Windows 95 doesn't have to provide additional memory for converting a TrueType font to a bitmap.
- The DOS program will appear in the normal size and not reduced on the monitor. This happens if you use a "4 x 6" font size instead of "8 x 12". As a result, the window would appear only one quarter of its original size.

Customizing memory requirements

An MS-DOS PC has different types of memory. This leads to an incredibly high number of possible configurations.
Windows 95 From The Underground

Chapter 15

If you’re not familiar with memory requirements and memory types used by a particular DOS program, refer to that program’s documentation.

If you select a wrong settings, in the best-case, the program will simply run a more slowly; in the worst-case, which is more likely, the program won’t run at all.

In a PC, memory is divided into three different areas: conventional memory, expanded memory (EMS) and extended memory (XTS). You can also specify how much RAM your program gets when it runs in DPMI mode.

Always start out using the setting Automatic for all memory types. Make changes to the individual memory allocations only when you notice that the performance of your computer suffers so greatly that the other programs seem to crawl.

Here are the settings for DOOM:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional Memory</td>
<td>Automatic and Protected</td>
</tr>
<tr>
<td>Expanded memory (EMS)</td>
<td>not required</td>
</tr>
<tr>
<td>Extended memory (XMS)</td>
<td>Automatic</td>
</tr>
<tr>
<td>RAM for MS-DOS protected mode (DPMI)</td>
<td>Automatic</td>
</tr>
</tbody>
</table>

The value “Automatic” for memory allocation lets Windows 95 dynamically allocate or deallocate the memory. Deallocating memory is especially critical when the execution speed of other concurrent programs is important. When your DOS program releases the memory, these other programs can then use
the freed memory. This happens automatically under Windows 95, guaranteeing that each task always runs at maximum performance.

On the other hand, by using some other value for the size for a type of memory, it means you are changing the way in which Windows 95 manages memory. Windows 95 now has to consider these requests and can no longer manage memory with total flexibility. This may mean a gain in speed for your DOS program in exchange for a loss in performance of the other programs. The degree of loss in performance depends on the memory type

However, with some DOS programs, this is the only way to get them to run.

If you also select the item Protected, then Windows 95 cannot make the memory used by the DOS program available to other programs. While this does speed up your DOS program, it does so at the expense of your other Windows programs.

**Using EMS memory**

Not very many of today’s new programs require EMS memory. You’ll have difficulties using this type of memory only when you use older programs. Keep in mind that you won’t be able to change the settings for EMS memory if the CONFIG.SYS file has the word NOEMS in the EMM386 command line.

When you use EMS memory, Windows 95 is forced to deal with this type of memory. Windows 95 takes a long time to access EMS memory since only 64K at a time can be moved back and forth. Naturally, this also means the loss of direct access to a memory area, since the data must first be taken from the EMS memory.

Avoid using EMS memory if possible. You may be able to upgrade to a newer version of your program or not starting such programs under Windows 95. Instead, when you boot the computer, select item 6. DOS Prompt Only. Then change your CONFIG.SYS file in such a way that your program gets enough EMS memory. Before doing this, be sure to copy the CONFIG.SYS and AUTOEXEC.BAT files to CONFIG.AAA and AUTOEXEC.AAA, for example, so that you have copies of the original files.

This is the best way to deal with the problem.

**Customizing screen display**

To change the screen display of a DOS program under Windows 95, use the Screen page.
Customizing the screen display

In the Usage field, you can choose between Window and Full-screen. The Initial size item gives you a choice between/among four different values for the screen lines: Default, 25 Lines, 43 Lines and 50 Lines.

By choosing Default, Windows 95 can use the exact number of screen lines that your DOS program requires.

In the Window field, you can specify whether the toolbar is displayed in the window. To display the toolbar, enable the Display toolbar check box.

To use the exact same settings for the screen the next time you start up the program, select the Restore settings on startup check box.

You can change the speed of the screen display using the Performance field. Enable the Fast ROM emulation check box to insert the ROM of the video card into an area of normal conventional memory. This enables significantly faster access to the video card because it is now located in faster RAM.

Running The Demo Programs In Windows 95

To install the demo programs from DEMOS directory of the Windows 95 CD-ROM, you'll need to pay special attention to the DOS Properties Box.

Periscopes-Intro

After copying the Periscopes intro to your hard drive, run PERISCOPE to unpack it.

Next, open the Windows 95 Explorer and select the file named INTRO.EXE. Click the right mouse button. A menu appears. Select the last item in the menu, Properties. Next, choose the Program page, click on the
[Advanced...] button and then select MS-DOS mode. Finally, click the [OK] button to confirm the settings you have made. You are now finished making the necessary settings for the Periscope intro.

To start the intro, run the file named Intro. You can skip the different sections of the intro by pressing the [Spacebar]. Have fun!

**Incompatible Programs**

Despite the complex configuration options Windows 95 offers, you may not be able to start certain programs. In fact, you even run the risk of destroying the Windows task of the program. In the worst-case scenario, your whole Windows system might no longer run, or else the PC will simply reboot, thus also destroying all the data from any other Windows tasks you have running.

If a DOS program requires special entries in your AUTOEXEC.BAT and CONFIG.SYS files under older versions of MS-DOS, perhaps in connection with a DOS device driver, you can assume that the program probably either won't run under Windows 95, or else will need a great deal of configuring before it runs.

If you manage to get such a program running despite this, while the configuration will be useful for this special DOS program, it's possible that the very next DOS program you run will require a completely different configuration.

As you can see, it's much better to get a new Windows 95 version of such troublesome programs. Only programs written specially for Windows 95 will be able to take full advantage of Windows 95.
Besides the demo programs in the DEMOS directory, you’ll also find some demos in the BONUS directory. The programs are meant to be a stimulus for you and give a small overview of the many possibilities of sound and graphics programming.

All the graphics programming examples require a 100% compatible VGA card.

The sound programming examples run either on a TRULY 100% compatible SoundBlaster card or on a Gravis Ultrasound card.

To use all the programs and source texts, first copy them to a directory on your hard drive, then use the MS-DOS ATTRIB command to clear the Read-Only file attribute, for example:

```
Attrib -r *. *
```

**Companion CD-ROM files and directories**

The directories included on the companion CD-ROM are in the file called CD_DIR.TXT.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONUS</td>
<td>Contains cool examples of programming techniques</td>
</tr>
<tr>
<td>DEMOS</td>
<td>More great examples of what you can do</td>
</tr>
<tr>
<td>DOOM</td>
<td>Directory contains sub-directories which pertain to programming the Doom material</td>
</tr>
<tr>
<td>DOOM\EXE_DATA</td>
<td>Executable of the programming examples for the Doom material.</td>
</tr>
<tr>
<td>DOOM\SOURCE</td>
<td>Source code for programming examples for Doom</td>
</tr>
<tr>
<td>DOOM\TOOLS</td>
<td>Tools to help you create Doom examples</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Contains graphic programming examples discussed in the book</td>
</tr>
<tr>
<td>MAGN</td>
<td>Example of programming a magnifying glass on you PC</td>
</tr>
<tr>
<td>MATH</td>
<td>Contains math programming examples discussed in the book</td>
</tr>
<tr>
<td>MEMORY</td>
<td>Directory contains sub-directories which pertain to programming Memory</td>
</tr>
<tr>
<td>MEMORY\DMA</td>
<td>Programming example for DMA Memory</td>
</tr>
<tr>
<td>MEMORY\FLAT</td>
<td>Programming example for Flat Memory</td>
</tr>
</tbody>
</table>
Chapter 16

Look What’s On The Companion CD-ROM

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY\XMS</td>
<td>Programming example for XMS Memory</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>Program which shows how to prevent people from using a debugger to 'debug' your programs.</td>
</tr>
<tr>
<td>NORESET</td>
<td>Program example which shows how to disable the [Ctrl] + [C], [Ctrl] + [Break], Etc.</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Contains password programming examples discussed in the book.</td>
</tr>
<tr>
<td>PORTS</td>
<td>Contains examples for direct port programming discussed in the book.</td>
</tr>
<tr>
<td>RAIDER</td>
<td>Game example that you can train with the included Trainer.</td>
</tr>
<tr>
<td>RTCLOCK</td>
<td>Programming you Real Time Clock (RTC)</td>
</tr>
<tr>
<td>SHARE</td>
<td>Shareware</td>
</tr>
<tr>
<td>SOUND</td>
<td>Directory contains sub-directories which pertain to programming Sound cards</td>
</tr>
<tr>
<td>SOUND\GUSMOD</td>
<td>Programs relating to the MOD Player for Gravis Ultra Sound cards</td>
</tr>
<tr>
<td>SOUND\SBMOD</td>
<td>Programs related to the MOD Player for Sound Blaster cards</td>
</tr>
<tr>
<td>SOUND\SFXPRO</td>
<td>‘C’ source for the new XM sound format player</td>
</tr>
<tr>
<td>SOUND\VOC</td>
<td>Programs related to the VOC player for Sound Blaster cards.</td>
</tr>
<tr>
<td>SPEAKER</td>
<td>Program related to programming your PC speaker</td>
</tr>
<tr>
<td>TRAINER</td>
<td>Source code for a trainer for the included RAIDER game.</td>
</tr>
</tbody>
</table>

We have also added a Windows program called ACROREAD.EXE. Run this program and it will install Adobe's Acrobat Reader program. You will then be able to see all the files in the ABACUS directory with the .PDF file extensions including the complete PC UNDERGROUND manuscript.

Installing Acrobat Reader

Follow these steps to install Acrobat Reader 2.0 on your hard drive (Installation requires approximately 2 Meg of free hard drive space). Insert the CD-ROM in your drive and load Windows. From the Windows Program Manager, choose Run from the File menu. Next, type the following:

```
[drive]:\acroread.exe
```

and press Enter. Then simply follow the instructions and prompts which appear on your screen.

Double-click the Acrobat Reader icon to load it. After the Acrobat Reader is loaded, go to File/Open... and select PCUNDER.PDF to view/read the book.

A note about shareware and public domain software

There are several shareware and public domain software in the DEMO and BONUS directories. The shareware concept allows small software companies and program authors to introduce the application programs they have developed to a wider audience. The programs can be freely distributed and tested for
a specific time period before you have to register them. Registration involves paying registration fees, which make you a licensed user of the program. Check the documentation or the program itself for the amount of registration fee and the address where you send the registration form.

After registration you will frequently get the current full version of the program without restrictions and shareware notes as well as the option of purchasing upgraded versions later for a reduced price. As a rule, larger applications include a users manual. Shareware programs usually feature the complete performance range of the full versions. Some programs use special messages to indicate they’re shareware versions. The message usually appears immediately after program startup or can be called from the Help menu.
Index

Companion CD-ROM ........................................... 535–537
3D_GLASS.PAS ................................................ 181–183
3D_LIGHT.PAS .................................................. 196–197
3D_SOLI.D.PAS ................................................ 195
3D_WIRE.PAS .................................................... 169–170
3DASM.ASM ..................................................... 171–179
Acrobat Reader (installing) ............................ 536–537
BASARITH.PAS .................................................. 4–6
BRES.ASM ......................................................... 167–168
COPPER.ASM .................................................. 112–114
COPPER.EXE .................................................... 111
Directory structure .......................................... 535–536
FADE.ASM ......................................................... 129–130
FADE.PAS ......................................................... 127–128
FADE_IN.PAS ................................................... 119–120
FADE_OUT.PAS ................................................ 119
FADE_OVE.PAS ................................................ 126–127
FADE_TO.PAS .................................................. 121
FADE_TXT.PAS ................................................ 131
FLAMES.PAS ................................................... 136–139
FLOW.PAS ......................................................... 109
GIF.ASM ........................................................... 40–45
GIF.PAS ............................................................ 39–40
GRABBER.PAS ................................................... 49–54
GUSASM.ASM ..................................................... 258–259
Hexcalibur hex editor ..................................... 238–240
LINEFCT.PAS ................................................... 7–8
MEMORY.PAS ..................................................... 263–268
MEMTEST.PAS .................................................. 273–276
MOD_SB unit ....................................................... 357, 358–365
MOD_SB.PAS .................................................... 312–314, 317–319, 324–325, 332–335
MODEXLIB.ASM ............................................ 86–87, 89–90, 95, 100, 104–105, 115–118, 134–135
NO_RST.ASM ..................................................... 20–22
PALROT.PAS ....................................................... 135–136
PAR_TEST.PAS .................................................. 29
PASSWD1.PAS ................................................... 215–218
PASSWD3.PAS ................................................... 218–220
PASSWGEN.PAS ................................................. 213–214
POLY.ASM ......................................................... 180, 183–192
PWMODUL.ASM .................................................. 226–228
QUERY.PAS ......................................................... 225–226
RMEM.PAS ......................................................... 273–276, 279–282
RMEMASM.ASM .................................................. 277–279
ROOT.ASM ......................................................... 11–12
ROOTTEST.PAS .................................................. 12–13
RTC.PAS ........................................................... 305–309
SCAL_TST.PAS .................................................. 157–159
SCROLL4.PAS ..................................................... 101–102
SCROLLT.PAS .................................................... 106
SFXPROC ......................................................... 438–462
SINTEST.PAS .................................................... 10
SPLIT.PAS ......................................................... 97
SPRITES.PAS ................................................... 150–153
SPRT_TST.PAS .................................................. 155–156
SQUEEZE.PAS ................................................... 105, 109–111, 118
SRL_SPT.PAS .................................................... 103
STAR.PAS ......................................................... 33–34
STARX.PAS ....................................................... 88
TIMEF.PAS ......................................................... 289–291
TOOLS.PAS ....................................................... 9
VAR_3D.PAS ..................................................... 169
VOXEL.ASM ...................................................... 141–144
VOXEL.PAS ....................................................... 140–141
WAIT.PAS ........................................................ 223–224
Windows 95 demo programs .......................... 533–534
WOBBLE.ASM ................................................... 116
WOBBLER.PAS ................................................... 116–117
Compression/decompression programs 230, 245
Conventional memory ...................................... 257–259
See also Memory Management
Segment registers ............................................ 257–259
Copper bars (programming) .......................... 109–114
Copy protection ................................................. 211–228
Machine language programming ..................... 220–224
Pascal programming ........................................ 211–220
Password queries ............................................. 211–220
Programming structure .................................. 220–224
CTRL-C
Intercepting ...................................................... 20–23
Custom mathematical functions ...................... 9–13

D

Debug interrupts .............................................. 240–242
Changing ....................................................... 240
Hiding data in interrupt vectors ...................... 240–242
Masking interrupts .......................................... 240
Debugger
"Fooling" the debugger ................................. 242–243
Debugging programs .................................. 230–240
Digital Signal Processor (DSP) ...................... 311–335
Commands .................................................. 315–316
Registers ...................................................... 311
See also Sound Blaster cards
Index

Digital to Analog Converter (DAC) ................. 80–82
See also VGA register bits
Division
Assembly language ........................................ 2, 4
DMA controller .............................................. 293–299
Adjusting size of DMA transfer .................. 298–299
All Mask register ........................................ 295
Autoinitialization on/off .......................... 296
Channel selection ....................................... 297
DMA flip-flop .............................................. 297
Masking a DMA channel ..................... 294–295
Mode selection ........................................ 296
Mode selection (transfer modes) .............. 296
Real Time Clock (RTC) .......................... 300–309
RTC clock functions .................................. 301
RTC configuration bytes ...................... 303–305
RTC status registers ............................. 302–303
Single Mask register .............................. 294–295
Specifying data block address .................. 297
Transfer modes ....................................... 295–297
Transfer selection .................................. 297
DMA controllers .................................. 293–309
Types of controllers ................................ 293–294
DMA flip-flop .............................................. 297
Specifying data block address .................. 297
DMA transfer
Adjusting size ........................................ 298–299
Dongle ....................................................... 212
DOOM .......................................................... 463–521
3-D vector graphics .................................. 500–521
Arithmetic/formulas used in DOOM ... 466–480
Gouraud shading ..................................... 480–489
Secrets of DOOM ..................................... 463–521
Vector arithmetic .................................. 466–480
DOOM II
Windows 95 configuration .................. 525–533
Windows 95 installation ....................... 525–533
Double-scan ........................................... 32

E

EMM
See Expanded Memory Manager (EMM)
EMS
See Expanded Memory Specification (EMS)
Encryption algorithms .............................. 244
Expanded Memory Manager (EMM) .......... 259–268
Function 40h ........................................... 260
Function 41h ........................................... 260
Function 42h ........................................... 261
Function 43h ........................................... 261
Function 44h ........................................... 261
Function 45h ........................................... 261–262
Function 46h ........................................... 262
Function 47h ........................................... 262
Function 48h ........................................... 262
Function 49h ........................................... 262
Function 4Ah ........................................... 263
Function 4Ch ........................................... 263
Function 4Dh ........................................... 263
Functions ........................................... 260–263
See Expanded Memory Specification (EMS)
Expanded Memory Specification (EMS) .... 259–268
EMM functions ........................................ 259–263
Using EMS in an example .................... 263–268
Extended Memory Specification (XMS) .... 268–276
Error codes ........................................... 269
Function 00h ........................................... 270
Function 03h ........................................... 270
Function 04h ........................................... 270
Function 05h ........................................... 270
Function 06h ........................................... 270
Function 07h ........................................... 271
Function 08h ........................................... 271
Function 09h ........................................... 271
Function 0Ah ........................................... 271–272
Function 0Bh ........................................... 272
Function 0Ch ........................................... 272
Function 0Dh ........................................... 272
Function 0Eh ........................................... 273
Function 0Fh ........................................... 273–277
Functions ........................................... 270–273

F

Factoring according to the distributive law ........ 2
Fire (programming) ................................... 136–139
Fixed point arithmetic ................................ 2–8, 463–466
Flat memory model .................................. 276–282
Technical information ............................ 276–277
Floating point arithmetic ........................ 2–3
Flowing images (programming) ............... 108–109
Index

G

Game trainers ............................................. 246–256
GIF file format ............................................. 34–48
Format ......................................................... 35–37
GIF loader .................................................. 89–90, 100–102
LZW compression ....................................... 37–39
Using GIF in an example ............................ 39–40
GIF loader .................................................. 100–102
Glass figures
  3-D Graphics programming ....................... 181–192
Gouraud shading .......................................... 480–489
Graphic effects ............................................ 93–146, 161–209
  3-D Graphics programming ....................... 161–209
Basics ......................................................... 93–94
Copper bars .............................................. 109–114
Fade-from effect ........................................ 123–132
Fade-in effect ........................................... 119–120
Fade-out effect ......................................... 118–119
Fade-to effect .......................................... 120–123
Fire on the screen .................................... 136–139
Flowing images ......................................... 108–109
Palette effects .......................................... 118–136
Palette rotation ....................................... 132–136, 141–144
Programming ............................................. 93–146
Scrolling in 4 directions ......................... 98–102
Smooth scrolling (text mode) .................. 105–108
Split screen programming ...................... 94–98
Split-screen with scrolling ...................... 103–104
Sprites (programming) ......................... 147–159
Squeezing images .................................. 104–105
Voxel spacing ......................................... 139–144
Wobbler .................................................. 115–118
Graphic formats
  GIF ........................................................... 34–48
  PCX ......................................................... 48–55
  VGA ......................................................... 56–82
Graphics Data Controller (GDC) ............... 71–76
See also VGA register bits
Graphic effects ........................................... 93–146
Graphics programming ............................. 31–82, 161–209
  3-D Graphics programming ...................... 161–209
Mode X ..................................................... 83–92
Gravis UltraSound ..................................... 397–437
Loading the MOD player ......................... 402–407
MOD player core routines ...................... 402–421
MOD player structure ............................ 398
MOD player variables .............................. 398–402, 442
Playing MOD files ................................... 407–421
TCP Player ................................................ 421–437
GUS
  See Gravis UltraSound
H

Hardware interrupts ................................. 283–286, 293
Hardware key (dongle) ............................ 212
Hex editor .................................................. 237–240
Hexcalibur hex editor ............................... 238–240
When to use ............................................... 237
Hidden lines
  3-D Graphics programming ...................... 193–195
IDEAL mode ................................................. 464
IMUL .......................................................... 25
INTERLNK.EXE driver .................................. 29
Interrupt controllers ................................. 18–23
Interrupt vector table ............................... 283–286
Interrupts .................................................... 17–23, 283–286
  Changing vectors .................................... 17–18
  Disabling interrupts .................................. 18–19
  Hardware interrupts ................................ 283
  Interrupt controllers ............................... 18–23
  Interrupt vector table .............................. 283
  NMI (NonMaskable Interrupt) ................. 283
  Re-entering DOS ....................................... 19
  Software interrupts ................................ 283
Light source shading
  3-D Graphics programming ...................... 195–200
LIM EMS
  See Expanded Memory Specification (EMS)
Loops
  16/32 bit access ........................................ 23
  Nesting ..................................................... 23
  Programming tips .................................... 23
LZW compression process
  Using with GIF ......................................... 37–39

542
# Index

### M

**Machine language**
- Password protection .......................... 220–224
- Maskable interrupts .......................... 292–293
- Masking interrupts .......................... 240
- Math tables .................................................. 9–13

**Mathematical functions**
- 3-D Graphics programming ................... 161–209
- Approximation ........................................ 11–13
- Custom functions .................................... 9–13

**Memory management** .......................... 257
- Conventional DOS memory .................. 257–259
- Conventional memory ......................... 257–259
- EMS (Expanded Memory Specification) .. 259–268
- Flat memory model ............................... 276–282
- Protected mode ........................................ 276
- Segment registers .................................. 257–259
- XMS (Extended Memory Specification) .. 268–276

**MOD file format** ............................... 337–354
- 669 format .............................................. 343–345
- 669 header ............................................... 343–344
- 669 pattern .............................................. 344–345
- Effects ..................................................... 340–343
- MOD header ............................................ 337–338
- MOD patterns ......................................... 339
- MOD sample files ................................... 339–340

**MOD players**
- Building a mixing procedure ............... 356–357
- Example of using ................................... 392–397
- Gravis UltraSound ................................ 398–437
- Handling MOD files ............................ 369–389
- How they work ....................................... 355–356
- MOD files ................................................. 369–389
- Polling ..................................................... 357
- Programming tips .................................. 389–392
- Routines for Gravis UltraSound .......... 402–421
- SFX Pro .................................................. 437–462
- Sound Blaster card ............................. 354–397
- Sound Blaster MOD player ............... 357–397, 403–421
- Sound routines ...................................... 367–369
- Timer interrupts .................................... 357
- Timer routines ........................................ 365–367
- Variables for Gravis UltraSound ....... 398–402
- Variables for Sound Blaster card ....... 358–365, 403–421
- Mode 13h .................................................. 33–34, 83
- Structure ............................................... 34

**Mode X** .................................................. 83–92
- DRAW_FON.PAS ...................................... 89
- GIF loader .............................................. 89–90
- Higher resolutions ................................ 86–87
- Initializing ............................................ 84
- Setting pixels ....................................... 88
- Structure .............................................. 84–86
- Switching pages .................................... 88–90
- Text scroller ......................................... 90–92
- Text scroller (fonts) ............................ 91–92
- Using Mode X ......................................... 87–89
- MOVX ................................................... 24
- MOVZX .................................................. 24

**Multiplication**
- Assembly language ............................. 2, 4
- IMUL command ....................................... 25

### N

**Nesting** ............................................... 23

**NMI (NonMaskable Interrupt)** ........... 283–286, 293

### O

**OR comparison** ..................................... 13

### P

**Palette**
- Defined ................................................ 31
- Palette effects (programming) .............. 118–136
- Fade-from ............................................. 123–132
- Fade-in ............................................... 119–120
- Fade-out .............................................. 118–119
- Fading to the target palette ............... 120–123
- Palette rotation .................................... 132–136
- Panning ................................................. 106

See also **Smooth scrolling (programming)**

**Parallel port**
- Control register .................................... 28–29
- Data register ........................................ 27–29
- INTERLNK.EXE ...................................... 29
- Other uses for the parallel port ........... 29–30
- Outputting characters .......................... 29
- Programming in Assembly language ...... 27–29
- Registers .............................................. 27–29
- Status register ...................................... 27–29

---

543
### Index

**Parallel projection** .................................................. 164

See also **3-D Graphics programming**

**Pascal**

- Password protection ........................................... 211–220
- Password protection ........................................... 211–220
- PCX graphic format ............................................ 48–55
  - Structure .................................................................. 48–55
  - Using PCX in an example ................................ 49–55
  - PIQ technique .................................................. 245
  - Pixel ..................................................................... 31
  - Polling ................................................................... 357
  - Polygons .................................................................. 489
  - Programming .................................................... 489–500
- Printer ports

See **Parallel port**

- Programmable Interrupt Controller (PIC) ............. 291–293
- Hardware interrupts ........................................... 293
- Maskable interrupts .......................................... 292–293
- Nonmaskable interrupts (NMI) ............................ 293
- Software interrupts ............................................ 293
- Programmable Interval Timer (PIT) ....................... 286–291
  - Control register .................................................. 287
  - Counter register ................................................. 288
  - Timer chip ......................................................... 286–287
  - Using in an example .......................................... 289–291
- Protected mode .................................................. 276–277

**Protecting programs**

- Machine language ............................................. 220–224
- Pascal ................................................................. 211–220

**Protecting your programs** ........................................ 229–256

- Checksums .......................................................... 244
- Compression/decompression programs .............. 230, 245
- Debug interrupts ................................................. 240–242
- Encryption algorithms ..................................... 244
- Fooling the debugger ......................................... 242–243
- Game trainers ..................................................... 246–256
- Hex editors ......................................................... 237–240
- Interrupt vectors ............................................... 240–242
- PIQ technique ...................................................... 245
- Self-modifying programs .................................. 244–245
- Turbo Debugger .................................................. 230
- Pythagorean theorem .......................................... 162

**R**

- Real Time Clock (RTC) ......................................... 300–309
  - Accessing RTC RAM ......................................... 300–309
  - Clock functions ............................................... 301
  - Configuration bytes ........................................... 303–305
  - Register A .......................................................... 302
  - Register B .......................................................... 302
  - Register C .......................................................... 302
  - Register D .......................................................... 303
  - Status registers ............................................... 302–303
  - Using in an example ........................................ 305–309
- Reset
  - Intercepting ...................................................... 20–23
  - Retrace
    - Defined .......................................................... 32

**Rotating objects**

- 3-D Graphics programming ................................ 165–166
  - Considerations ................................................ 165–166

**S**

- S3M file format .................................................. 348–354
- S3m header ....................................................... 348–351
- S3m instruments .............................................. 352–354
- S3m patterns ..................................................... 351
- SB16 (ASP) mixer chip ..................................... 327–331
  - Register 48 ....................................................... 327
  - Register 49 ....................................................... 327
  - Register 50 ....................................................... 328
  - Register 51 ....................................................... 328
  - Register 52 ....................................................... 328
  - Register 53 ....................................................... 328
  - Register 54 ....................................................... 328
  - Register 55 ....................................................... 328
  - Register 56 ....................................................... 328
  - Register 57 ....................................................... 329
  - Register 58 ....................................................... 329
  - Register 59 ....................................................... 329
  - Register 60 ....................................................... 329
  - Register 61 ....................................................... 329
  - Register 62 ....................................................... 330
  - Register 63 ....................................................... 330
  - Register 64 ....................................................... 330
  - Register 65 ....................................................... 330
  - Register 66 ....................................................... 330
  - Register 67 ....................................................... 330
  - Register 68 ....................................................... 330
  - Register 69 ....................................................... 330
  - Register 70 ....................................................... 330
  - Register 71 ....................................................... 330
  - Register 54 ....................................................... 328
  - Register 55 ....................................................... 328
  - Register 56 ....................................................... 328
  - Register 57 ....................................................... 329
  - Register 58 ....................................................... 329
  - Register 59 ....................................................... 329
  - Register 60 ....................................................... 329
  - Register 61 ....................................................... 329
  - Register 62 ....................................................... 330
  - Register 63 ....................................................... 330
  - Register 64 ....................................................... 330
  - Register 65 ....................................................... 330
  - Register 66 ....................................................... 330
  - Register 67 ....................................................... 330
  - Register 68 ....................................................... 330
  - Register 69 ....................................................... 330
  - Register 70 ....................................................... 330
  - Register 71 ....................................................... 331
- SBPro mixer chip ............................................. 325–327
  - Register 00h ...................................................... 325

544
### Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components of the Sound Blaster cards</td>
<td>311–331</td>
</tr>
<tr>
<td>Detecting</td>
<td>323–325</td>
</tr>
<tr>
<td>Determining base port</td>
<td>323–325</td>
</tr>
<tr>
<td>DSP commands</td>
<td>315–316</td>
</tr>
<tr>
<td>DSP registers</td>
<td>311</td>
</tr>
<tr>
<td>Handling MOD files</td>
<td>369–389</td>
</tr>
<tr>
<td>Mixer chip</td>
<td>325–327</td>
</tr>
<tr>
<td>MOD player</td>
<td>354–397</td>
</tr>
<tr>
<td>MOD player programming tips</td>
<td>389–392</td>
</tr>
<tr>
<td>MOD player variables</td>
<td>358–365</td>
</tr>
<tr>
<td>Playing VOC files</td>
<td>331–335</td>
</tr>
<tr>
<td>Programming the DSP</td>
<td>311–323</td>
</tr>
<tr>
<td>SB16 (ASP) mixer chip registers</td>
<td>327–331</td>
</tr>
<tr>
<td>SBPro mixer chip</td>
<td>325–327</td>
</tr>
<tr>
<td>Sound routines</td>
<td>367–369</td>
</tr>
<tr>
<td>Timer routines</td>
<td>365–367</td>
</tr>
<tr>
<td>Sound cards</td>
<td></td>
</tr>
<tr>
<td>See specific card name</td>
<td></td>
</tr>
<tr>
<td>Split screen (programming)</td>
<td>94–98</td>
</tr>
<tr>
<td>Split-screen with scrolling (programming)</td>
<td>103–104</td>
</tr>
<tr>
<td>Sprites</td>
<td>147–159</td>
</tr>
<tr>
<td>Clipping</td>
<td>149</td>
</tr>
<tr>
<td>Defined</td>
<td>31</td>
</tr>
<tr>
<td>Reading</td>
<td>148–149</td>
</tr>
<tr>
<td>Scrolling</td>
<td>156–159</td>
</tr>
<tr>
<td>Structure of sprites</td>
<td>147–148</td>
</tr>
<tr>
<td>Using in an example</td>
<td>150–156</td>
</tr>
<tr>
<td>Writing</td>
<td>148–149</td>
</tr>
<tr>
<td>Squeezing images (programming)</td>
<td>104–105</td>
</tr>
<tr>
<td>STM (Scream Tracker)</td>
<td></td>
</tr>
<tr>
<td>See Scream Tracker file format</td>
<td></td>
</tr>
<tr>
<td>String comparisons</td>
<td>14</td>
</tr>
<tr>
<td>Structure</td>
<td>15</td>
</tr>
<tr>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>Assembly language</td>
<td>4</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td></td>
</tr>
<tr>
<td>TASM</td>
<td>464</td>
</tr>
<tr>
<td>Terminology used in this book</td>
<td>31–33</td>
</tr>
<tr>
<td>Texture mapping</td>
<td>489–500</td>
</tr>
<tr>
<td>Textures</td>
<td>489</td>
</tr>
<tr>
<td>3-D Graphics programming</td>
<td>489–500</td>
</tr>
<tr>
<td>Textures</td>
<td>489–500</td>
</tr>
<tr>
<td>Timer interrupts</td>
<td></td>
</tr>
<tr>
<td>MOD player</td>
<td>357</td>
</tr>
<tr>
<td>Timing sequencer (TS)</td>
<td>68–71</td>
</tr>
<tr>
<td>See also VGA register bits</td>
<td></td>
</tr>
<tr>
<td><strong>Register 02h</strong></td>
<td>325</td>
</tr>
<tr>
<td><strong>Register 0AH</strong></td>
<td>325–326</td>
</tr>
<tr>
<td><strong>Register 0CH</strong></td>
<td>326</td>
</tr>
<tr>
<td><strong>Register 0EH</strong></td>
<td>326</td>
</tr>
<tr>
<td><strong>Register 26h</strong></td>
<td>326–327</td>
</tr>
<tr>
<td><strong>Register 28h</strong></td>
<td>327</td>
</tr>
<tr>
<td><strong>Register 2EH</strong></td>
<td>327</td>
</tr>
<tr>
<td>Scream Tracker file format</td>
<td>345–348</td>
</tr>
<tr>
<td>STM header</td>
<td>346–348</td>
</tr>
<tr>
<td>Scrolling in 4 directions (programming)</td>
<td>98–102</td>
</tr>
<tr>
<td>Segment registers</td>
<td></td>
</tr>
<tr>
<td>Conventional memory</td>
<td>257–259</td>
</tr>
<tr>
<td>Self-modifying programs</td>
<td>244–245</td>
</tr>
<tr>
<td>Checksums</td>
<td>244</td>
</tr>
<tr>
<td>Encryption algorithms</td>
<td>244</td>
</tr>
<tr>
<td>PIQ technique</td>
<td>245</td>
</tr>
<tr>
<td>SET commands</td>
<td>24</td>
</tr>
<tr>
<td>SFX Pro</td>
<td>437–462</td>
</tr>
<tr>
<td>Envelopes</td>
<td>458–460</td>
</tr>
<tr>
<td>Frequencies</td>
<td>438–439</td>
</tr>
<tr>
<td>Sample conversion</td>
<td>439</td>
</tr>
<tr>
<td>Volume effects</td>
<td>447–449</td>
</tr>
<tr>
<td>XM load routines</td>
<td>440–442</td>
</tr>
<tr>
<td>XM module effects</td>
<td>450–458</td>
</tr>
<tr>
<td>XM patterns (playing back)</td>
<td>442–446</td>
</tr>
<tr>
<td>Shading</td>
<td></td>
</tr>
<tr>
<td>3-D Graphics programming</td>
<td>195–200</td>
</tr>
<tr>
<td>SHRD</td>
<td>24</td>
</tr>
<tr>
<td>SHRL</td>
<td>24</td>
</tr>
<tr>
<td>Single Mask register</td>
<td>294–295</td>
</tr>
<tr>
<td>Smooth scrolling (programming)</td>
<td>105–108</td>
</tr>
<tr>
<td>Software interrupts</td>
<td>283–286, 293</td>
</tr>
<tr>
<td>Sound</td>
<td>337–462</td>
</tr>
<tr>
<td>Accompanying your programs</td>
<td>337–462</td>
</tr>
<tr>
<td>Experiment placement</td>
<td>347</td>
</tr>
<tr>
<td>Gravis UltraSound</td>
<td>397–437</td>
</tr>
<tr>
<td>MOD file format</td>
<td>337–354</td>
</tr>
<tr>
<td>MOD player for Gravis UltraSound</td>
<td>398–437</td>
</tr>
<tr>
<td>MOD player for Sound Blaster cards</td>
<td>354–397</td>
</tr>
<tr>
<td>S3M file format</td>
<td>348–354</td>
</tr>
<tr>
<td>Scream Tracker file format</td>
<td>345–348</td>
</tr>
<tr>
<td>Sound Blaster cards</td>
<td>311–335, 354–397</td>
</tr>
<tr>
<td>SFX Pro sound card</td>
<td>442–462</td>
</tr>
<tr>
<td>VOC files</td>
<td>331–335</td>
</tr>
<tr>
<td><strong>See also MOD players</strong></td>
<td></td>
</tr>
<tr>
<td>Sound Blaster cards</td>
<td>311–335</td>
</tr>
<tr>
<td>BLASTER variable</td>
<td>323</td>
</tr>
<tr>
<td>Chip register 22h</td>
<td>326</td>
</tr>
</tbody>
</table>

545
Index

Transfer modes (DMA controller) .......... 295–297
Autoinitialization on/off .................. 296
Channel selection .................................... 297
Incrementing/decrementing addresses ..... 296
Mode selection ........................................ 296
Transfer selection .................................... 297
Turbo Debugger ........................................ 230–240
Command line parameters .................. 231–232
Target searches ....................................... 232–237
Using .............................................. 230–232

V

Variables
Assembly language ............................... 14–17
Vector
3-D Graphics programming .................. 161–164
Calculating with vectors ....................... 162–164
Multiplication ........................................ 162–163
Subtraction ............................................ 162
Vector arithmetic in DOOM .................. 466–480
Vector graphics
3-D Graphics programming .................. 500–521
VGA ......................................................... 56–82
Attribute Controller (ATC) .................. 76–80
Cathode Ray Tube Controller (CRTC) ..... 58–68
Digital to Analog Converter (DAC) ........ 80–82
Graphics Data Controller (GDC) .......... 71–76
Register bits ........................................... 56–82
Timing sequencer (TS) ......................... 68–71
VGA register bits ................................... 56–82
ATC-Register 10h .................................. 78–79
ATC-Register 11h .................................. 79
ATC-Register 12h .................................. 79
ATC-Register 13h .................................. 79–80
ATC-Register 14h .................................. 80
Attribute Controller (ATC) .................. 76–80
Cathode Ray Tube Controller (CRTC) ..... 58–68
CRTC-Register 0 .................................... 58–59
CRTC-Register 0ah ................................. 62
CRTC-Register 0bh .................................. 63
CRTC-Register 0ch .................................. 63
CRTC-Register 0dh .................................. 63
CRTC-Register 0eh .................................. 63
CRTC-Register 0fh .................................. 64
CRTC-Register 1 ..................................... 59
CRTC-Register 10h .................................. 64
CRTC-Register 11h .................................. 64–65
CRTC-Register 12h .................................. 65
CRTC-Register 13h .................................. 65
CRTC-Register 14h .................................. 65–66
CRTC-Register 15h .................................. 66
CRTC-Register 16h .................................. 66
CRTC-Register 17h .................................. 67
CRTC-Register 18h .................................. 68
CRTC-Register 2 ..................................... 59
CRTC-Register 3 ..................................... 59
CRTC-Register 4 ..................................... 60
CRTC-Register 5 ..................................... 60
CRTC-Register 6 ..................................... 60
CRTC-Register 7 ..................................... 61
CRTC-Register 8 ..................................... 61
CRTC-Register 9 ..................................... 62
Digital to Analog Converter (DAC) ........ 80–82
GDC-Register 0 ....................................... 72
GDC-Register 1 ....................................... 72
GDC-Register 2 ....................................... 72–73
GDC-Register 3 ....................................... 73
GDC-Register 4 ....................................... 73
GDC-Register 5 ....................................... 74–75
GDC-Register 6 ....................................... 75
GDC-Register 7 ....................................... 76
GDC-Register 8 ....................................... 76
Graphics Data Controller (GDC) .......... 71–76
Input Status Register 1 ......................... 58
Miscellaneous Output Register .......... 56–76
Timing sequencer (TS) ......................... 68–71
TS-Register 0 ......................................... 68
TS-Register 1 ......................................... 69
TS-Register 2 ......................................... 69–70
TS-Register 3 ......................................... 70
TS-Register 4 ......................................... 71
VOC files .............................................. 331–335
Voxel spacing (programming) .............. 139–144

W

Windows 95 ........................................... 523–524
Demo programs ....................................... 533–534
Incompatible programs ......................... 534
Installing/starting DOS programs ........ 525–533
Starting DOS programs ......................... 523–525
Using DOS programs .............................. 523–525
Windows 95 DOS compatibility box ....... 524–525
Windows 95 DOS compatibility box ....... 524–525
Wireframe modeling

546
3-D Graphics programming ...................... 166–181
Adding surfaces ........................................... 181–192
Bresenham algorithm ................................. 167–181
Wobbler (programming) ......................... 115–118

X

XM players ................................................. 437–462
Effects of XM module .............................. 450–458
Envelopes .................................................. 458–460
Frequency .................................................. 438–439
Sample conversion ........................................ 439
SFX Pro .................................................... 437–462
Volume effects ......................................... 447–449
XM load routines ....................................... 440–442
XM patterns (playing back) ................. 442–446

XMS
See Extended Memory Specification (XMS)
Information About PC Underground CD-ROM

Besides the demo programs in the DEMOS directory, you'll also find some demos in the BONUS directory. The programs are meant to be a stimulus for you and give a small overview of the many possibilities of sound and graphics programming. All the graphics programming examples require a 100% compatible VGA card. The sound programming examples run either on a TRULY 100% compatible SoundBlaster card or on a Gravis Ultrasound card.

To use all the programs and source text, copy them to a directory on your hard drive and use the MS-DOS ATTRIB command to clear the Read-Only file attribute, for example: ATTRIB -r *. *

Files And Directories On The Companion CD-ROM

The directories included on the companion CD-ROM are in the file called CD_DIR.TXT.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONUS</td>
<td>Contains cool examples of programming techniques</td>
</tr>
<tr>
<td>DEMOS</td>
<td>More great examples of what you can do</td>
</tr>
<tr>
<td>DOOM</td>
<td>Directory contains sub-directories which pertain to programming the Doom material</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Contains graphic programming examples discussed in the book</td>
</tr>
<tr>
<td>MAGN</td>
<td>Example of programming a magnifying glass on you PC</td>
</tr>
<tr>
<td>MATH</td>
<td>Contains math programming examples discussed in the book.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>Directory contains sub-directories which pertain to programming Memory</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>Program which shows how to prevent people from using a debugger to ‘debug’ your programs.</td>
</tr>
<tr>
<td>NORESET</td>
<td>Program example which shows how to disable the [Ctrl] + [C], [Ctrl] + [Break], Etc..</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Contains password programming examples discussed in the book.</td>
</tr>
<tr>
<td>PORTS</td>
<td>Contains examples for direct port programming discussed in the book.</td>
</tr>
<tr>
<td>RAIDER</td>
<td>Game example that you can train with the included Trainer.</td>
</tr>
<tr>
<td>RTCLOCK</td>
<td>Programming your Real Time Clock (RTC)</td>
</tr>
<tr>
<td>SHARE</td>
<td>Shareware</td>
</tr>
<tr>
<td>SPEAKER</td>
<td>Program related to programming your PC speaker</td>
</tr>
<tr>
<td>SOUND</td>
<td>Directory contains sub-directories which pertain to programming Sound cards</td>
</tr>
<tr>
<td>TRAINER</td>
<td>Source code for a trainer for the included RAIDER game.</td>
</tr>
</tbody>
</table>

To view the book on the CD-ROM you must install Adobe’s Acrobat Reader 2.0 for Windows on your computer. The Adobe Acrobat Reader 2.0 for Windows software gives you instant access to documents in their original form, independent of computer platforms. By using the Acrobat Reader, you can view, navigate, print selected files and present Portable Document Format (PDF) files.

Installing Acrobat Reader

Follow these easy steps to install Acrobat Reader 2.0. Insert the CD-ROM in your drive and load Windows. From the Windows Program Manager, choose Run from the File menu. Next, type the following:

[drive]:\acroread.exe and press Enter

Then simply follow the instructions and prompts which appear on your screen. Double click the Acrobat Reader icon to load it. After the Acrobat Reader is loaded, go to File/Open... and select MAIN.PDF to view and read the PC UNDERGROUND book.