58 Creating Demos in XPL0

Boreal, Wed 29 Aug 2007

Creating Demos in XPL0

By Boreal

How often have you been in a bookstore thinking about buying a book, and just wanting to see the example programs on the CD in the back? It would tell you in a glance if the book was worth buying or not.

Well, you're in luck! Not only do you not have to buy anything; but if you go to the XPL0.zip file in the bonus pack, you can quickly and easily run the programs. There's no seal to break and no BS legal mumbo-jumbo to ignore.

You can even modify the code, with your favorite editor, and recompile and run it.

The example programs are essentially Polaris's excellent demos from Hugi 31 that have been translated into XPL0. However in some cases I couldn't resist making a few slight changes.

What's XPL0?

XPL0 is a similar to Pascal. It was initially created by my computer club (the 6502 Group) back in the mid 1970s. Since then, it has been enhanced and implemented on a variety of processors, especially those used in PCs. I'm the current maintainer of the language with a webpage at: http://www.idcomm.com/personal/lorenblaney/

There's lots of information there, including a 127-page manual, but let me give you a brief introduction here. We'll begin with the traditional Hello World program:

```
code Text=12;
Text(0, "Hello World!")
```

"Text" is a built-in routine that outputs strings of characters. The zero (0) tells where to send the string. In this case it's sent to the display screen, but it could just as easily have been sent to the printer or out a serial port by using a different number.

In XPL0 all names must be declared before they can be used. The command word 'code' associates the name "Text" to built-in routine

number 12, which is the one that outputs strings. There are about 80 of these built-in routines.

Is XPL0 worth messing with?

I can easily understand why you wouldn't want to bother learning yet another computer language. But I hope you'll find the following examples intriguing enough to at least consider it.

I'm not trying to sell you on XPL0. I don't make any money from it. It's just that it's the high-level language I use all the time. XPL0 has been tailored to my needs, and consequently I rarely use other high-level languages.

A possible objection to using XPL0 is that it's based on DOS rather than Windows. It doesn't make Windows applications. Fortunately Windows is still able to run most DOS programs.

The best feature of XPL0, besides being easy to learn, is you get all the source code. You can tailor the language to your needs.

Getting your feet wet

I'm assuming that you're at least somewhat familiar with running DOS. Under Windows XP, you can get a DOS window by going to Start -> Accessories -> "C: Command Prompt". I'm also assuming you have some kind of text editor (such as EDIT or Notepad) that makes plain-vanilla ASCII files.

To compile and run the Hello program, you need to extract the files in the accompanying bonus pack into a directory called CXPL. XPL0 programs expect to find certain files in this directory, immediately below your root directory, like this: C:CXPL

The Hello program is in the file called HELLO.XPL, and it can be compiled and run like this (type in the lowercase letters):

```
C:CXPL>x hello
C:CXPL>hello
```

The Matrix has you!

Now let's get into something more interesting - some oldskool demos. For this we'll borrow heavily from Polaris's examples. Here's his Matrix program translated into XPL0:

```
include c:cxplcodesi; standard library 'code' definitions
int     PosX, PosY;     X and Y position (column and row) of text on screen
```

```
begin   Main
while not ChkKey do                 loop until a key is pressed
        begin
        PosX:= Ran(80);             calculate random position to draw text
        Ran(80) gives a range 0..79, which is what we want
        PosY:= Ran(25);             Y position is similar

        Attrib(if Ran(2) then 10 else 2);randomly select light or dark green
        Cursor(PosX, PosY);             where we want our text to show up
        ChOut(6, Ran(2)+^0);            randomly output an ASCII 0 or 1
        end;
end;    Main
```

XPL0 provides built-in (library) routines called "intrinsics". The 'code' declarations for all the intrinsics are in the file CODESI.XPL. This file is usually 'include'd at the top of every XPL0 program, and it gives the standardized names to all of the intrinsics. Here we're using these intrinsics:

ChkKey checks to see if a key has been struck on the keyboard and returns either 'false" (zero) or 'true' (non-zero).

Ran is the random number generator. It returns a number between 0 and the argument minus 1.

Attrib specifies the attributes (usually foreground and background colors) used for displaying characters.

Cursor specifies the column and row on the screen where characters will appear. The upper-left corner is 0,0.

ChOut outputs a character. The "6" indicates that it goes to the screen and uses the colors set up by Attrib. Characters can be sent to other output devices - such as printers, serial ports, or files - merely by using different numbers.

Note that names are capitalized. XPL0 requires that all names be capitalized - like the good proper nouns that they are. This avoids any possible conflict with reserved words, which are in lowercase. For example, there's no problem having a variable called "End".

The integer variables used by the program are declared by the command word 'int'. This could just as well have been written out as 'integer', but since only the first three letters of commands are significant, XPL0 honchos typically use abbreviations.

The code above is a fairly straightforward translation of Polaris's C++ program. However the argument to the Attrib intrinsic "if Ran(2) then 10 else 2" is slightly unusual. It's an example of an 'if' expression, which is different than the more common 'if' statement. It's equivalent to the C expression: "rand()%2 ? 10 : 2".

If Ran(2) returns a zero value it's interpreted as being 'false', while any non-zero value (such as 1) is treated as being 'true'. Thus the argument evaluates to either 2 (for false) or 10 (for true). These are the values of the two shades of green for EGA (and greater) video modes.

This Matrix example can be simplified to five lines (greetz! Fable Fox):

```
include c:cxplcodesi;                  standard library 'code' definitions
repeat  Cursor(Ran(80), Ran(25));      randomly select location on screen
        Attrib(if Ran(2) then 10 else 2); randomly select light or dark green
        ChOut(6, Ran(2)+^0);           randomly output an ASCII 0 or 1
until   ChkKey;                        run until a key is struck
```

When either of the examples above are run, you get something like this:

Let it snow

Here's a translation of Polaris's Snow demo. This introduces arrays and procedures. It begins by defining a name for the constant value that specifies the size of the arrays. 'def' is an abbreviation for the command word 'define'.

```
include c:cxplcodesi;              standard library 'code' definitions

Data structures for snow flakes = very simple
def     Total_flakes=900;
def     Total_layers=3;

int     FlakesX(Total_flakes),
        FlakesY(Total_flakes),
        FlakesLayer(Total_flakes);


proc    Initialize_particle_flakes;     Initialize the particle flakes
int     I;
begin
for I:= 0, Total_flakes-1 do
```

```
        begin
        FlakesX(I):= Ran(320);          -319 (x)
        FlakesY(I):= Ran(200);          -199 (y)
        FlakesLayer(I):= Ran(Total_layers); (0-2) (layer)
        end;
end;


proc    Draw_particle_flakes;           Draw all the particle flakes
int     I;
begin
for I:= 0, Total_flakes-1 do
        begin
        Point(FlakesX(I), FlakesY(I), FlakesLayer(I)*4+23); (bright flakes)
        Note - we are drawing according to the default color palette
        end;
end;


proc    Update_particle_flakes;         Update the particle flakes
int     I;
begin
for I:= 0, Total_flakes-1 do
        begin
        Drop the particle down - depending on layer.
        Add one since layer zero would result in no motion
        FlakesY(I):= FlakesY(I) + FlakesLayer(I) + 1;

        Check for wrap around
        if FlakesY(I) > 199 then
                begin
                FlakesX(I):= Ran(320);
                FlakesY(I):= 0;
                FlakesLayer(I):= Ran(Total_layers);
                end;

        New X position
        FlakesX(I):= Rem((FlakesX(I)+2-Ran(5)) / 320);
```

```
        end;
end;


proc    VSync;                  Wait for vertical retrace to begin
begin
while PIn(&#036;3DA,0) & &#036;08 do [];  wait for vertical retrace signal to go away
repeat until PIn(&#036;3DA,0) & &#036;08; wait for vertical retrace
end;    VSync


begin   Main
SetVid(&#036;13);                   set graphics display for 320x200x8-bit color
Initialize_particle_flakes;
while not ChkKey do
        begin
        Update_particle_flakes;
        VSync;                  (slow flakes)
        VSync;
        VSync;
        VSync;
        Clear;
        Draw_particle_flakes;
        end;
SetVid(&#036;03);                   restore normal text mode (for DOS)
end;
```

Notice that the program is divided into procedures, which are specified by the 'proc' abbreviations. The term "procedure" is just another name for "subroutine". Unlike C, XPL0 distinguishes between procedures and functions. Functions are subroutines that return a value, and when they are called they are used as values.

Since in XPL0 all named things must be declared before they are used, procedures (and functions) are written before they're called. This tends to make programs look upside down. Generally you want to read an XPL0 program starting with the Main routine, which is at the bottom, and work your way upward to get the details.

Five new intrinsics are used by this program.

SetVid is used to change the video display mode. By passing hex 13, it changes to 320x200 graphics with 256 colors. By passing 3, it

changes back to text mode. (When running under Windows, it's not necessary to restore text mode if the program exits back to the GUI; however, it might be running in a DOS box where it will return to a window that expects to be in text mode.)

Clear erases the entire screen.

Point is a subroutine that draws pixels on the graphic screen.

Rem returns the remainder of an integer division. It's equivalent to C's modulo (%) operator.

PIn reads from an input port.

SetVid, Clear and Point are examples of intrinsics that are used as subroutines, while Rem, Ran and ChkKey are examples where they're used as functions because they return a value.

This Snow program contains a procedure that is not shown in Polaris's code. It's the VSync procedure. In Polaris's code, Allegro provides the VSync routine, like the way XPL0 provides intrinsic routines. Since VSync is not provided, we must write it ourselves. Fortunately it's simple. The PIn intrinsic is used to access the hardware register that contains video status information. Bit 3 of this port indicates if vertical retrace is occurring. This typically occurs 60 times per second, so we use it to regulate the speed of our program.

XPL0 only has three data types: int, real and char. It does not have structures (or records). Instead of writing (as in Polaris's C++ code):

```
struct particle
{  int x, y;
   int layer;
};
particle flakes[total_flakes];
```

you must declare names explicitly, by doing something like this:

```
int     FlakesX(Total_flakes),
        FlakesY(Total_flakes),
        FlakesLayer(Total_flakes);
```

Another possibility for faking structures is to use arrays. For example, PointX, PointY, PointZ might be implemented as: Point(X), Point(Y), Point(Z).

When you run SNOW.COM, you should see something like this:

Wormhole

Here's another little demo. It shows lots of action by merely manipulating the color registers. Don't get sucked in!

```
include c:cxplcodesi;        standard library 'code' definitions

int     Fade;                color intensity (0..128)
char    Palette(3*256);      copy of color registers


proc    Rotate;              Rotate Palette and adjust Fade intensity
int     I, J, K,             indexes
        R, G, B;             red, green, blue
begin
for I:= 0, 240*3-1 do        shift down 16*3 bytes
        Palette(I):= Palette(I+16*3);
for I:= 0, 16*3-1 do         wrap 16*3 bytes to the end
        Palette(I+240*3):= Palette(I);
K:= 0;                       rotate 16*3 bytes in each of 16 groups
for I:= 0, 16-1 do
        begin
        R:= Palette(K);
        G:= Palette(K+1);
        B:= Palette(K+2);
        for J:= 0, 15*3-1 do
                begin
                Palette(K):= Palette(K+3);
                K:= K+1;
                end;
        Palette(K):= R;
        Palette(K+1):= G;
        Palette(K+2):= B;
        K:= K+3;
```

```
        end;
VSync;                          regulate speed
POut(16, &#036;3C8, 0);             copy Palette to color registers
for I:= 16*3, 256*3-1 do        but don't alter the border color (0)
        POut(Palette(I)*Fade/128, &#036;3C9, 0);
end;    Rotate


proc    LoadBmp(Name);          Load a 320x200x8 .bmp file but don't display it
char    Name;                   file name
int     Hand, I, T,
        X, Y;                   coordinates
begin
Hand:= FOpen(Name, 0);                  open file for input
FSet(Hand, ^I);                         set device 3 to handle
OpenI(3);

for I:= 0, 53 do T:= ChIn(3);           skip unused header info

for I:= 0, 255 do
        begin
        Palette(I*3+2):= ChIn(3)>>2;    blue
        Palette(I*3+1):= ChIn(3)>>2;    green
        Palette(I*3+0):= ChIn(3)>>2;    red
        T:= ChIn(3);                    unused
        end;

for Y:= -(200-1), 0 do                  .bmp files are upside down!
    for X:= 0, 320-1 do
        Point(X, -Y, ChIn(3));

FClose(Hand);
end;    LoadBmp


begin   Main
SetVid(&#036;13);                       320x200x8-bit color
Fade:= 0;                               set all color registers to black
```

```
Rotate;
LoadBmp("WORMHOLE.BMP");

repeat  if Fade         Rotate;
until   ChkKey;                     loop until keystroke

repeat  Fade:= Fade-1;              fade out
        Rotate;
until   Fade
SetVid(&#036;03);                     restore normal text mode
end;    Main
```

The 'char' declaration sets up memory space for characters, or any kind of bytes. Here we're using it in two places: one to set up an array of bytes specifying the colors in the Palette; and another (in LoadBmp) to receive the string of characters that specifies the name of the file to load.

The Rotate procedure uses the POut intrinsic to write to the output port. The repeated writes to port $3C9 might seem strange. The PC has a hardware counter that increments each time this port is written. That way successive color registers are written. There are 256 groups each containing a red, green and blue register. Each register can have a value from 0 to 63. Thus you can get 63 shades of red (plus one shade of black), or 64*64*64 = 262144 total colors. However only 256 different colors can be displayed at one time.

The LoadBmp procedure uses several intrinsics to read in an image file. FOpen, FSet and OpenI are used to set up a file to be read in, one character at a time by ChIn. The .bmp file format is used here because it's simple (it doesn't use compression), and it's a standard Windows format that's universally supported.

If this program doesn't work (you might see ERROR 3), it probably means that you tried to run it directly from Windows without extracting the files to a folder. The accompanying file, Wormhole.bmp, must be in the same directory (or folder) as Wormhole.exe. Incidentally, any jitteriness you see is entirely due to Windows. I prefer to run these kinds of programs under pure DOS.

The Wormhole should look like this, but with lots of action:

32-bit XPL0

Now we're going to shift into high gear and use a much more powerful version of the XPL0 compiler.

Up until now we've been using the interpreted version. It has the advantage of being complete and self-contained (and is included in the bonus pack). However it's slow, it only has 16-bit integers, and it only provides about 60K of memory for variables and arrays. Actually, it does have double-precision floating point, and it will handle arrays that are much larger than 60K by using segment addressing; but for simplicity we're going to use the 32-bit version of XPL0 from now on.

Unfortunately this version is not included in the bonus pack because it requires Borland's TASM assembler. However you can download everything you need from the 'Net. The 32-bit compiler is on the XPL0 website at: http://www.idcomm.com/personal/lorenblaney/ and a free version of TASM is in TasmIDE.zip at: http://www.soem.ecu.edu.au/units/ens1242/workshops/tasm.htm

When you have TASM and TLINK set up on your computer, you can compile and run the Hello program like this:

```
C:CXPL>xpx hello
C:CXPL>hello
```

Note that the batch file xpx.bat is used instead of x.bat. You'll need to type "hello.exe" instead of just plain "hello" to run the newly compiled version if hello.com is in the current directory. That's because DOS gives first priority to running .com files.

XFade

Now let's start taking advantage of this new compiler. Here's Polaris's cross-fade demo.

```
include c:cxplcodes;          intrinsic 'code' declarations for 32-bit XPL0

int     CpuReg,               array containing CPU hardware registers
        DataSeg,              our data segment address
        I;                    index for Main
real    XFade;                cross-fade; 0.=image1, 1.=image2
char    Image1(320*200),      image from .bmp file
        Pal1(256*3),          palette from the .bmp file
        Image2(320*200),
        Pal2(256*3),
        ImageCombo(320*200),  combined images
        PalCombo(256*3);
```

```
proc    VSync;                  Wait for vertical retrace to begin
begin
while port(&#036;3DA) & &#036;08 do [];   wait for vertical retrace signal to go away
repeat until port(&#036;3DA) & &#036;08;  wait for vertical retrace
end;    VSync


proc    LoadBmp(Name, Image, Pal);      Load 320x200x8 .bmp file
char    Name,           file name
        Image,          array to store image into
        Pal;            palette array
int     Hand, I, T,
        X, Y;           coordinates
begin
Hand:= FOpen(Name, 0);                  open file for input
FSet(Hand, ^I);                         set device 3 to handle
OpenI(3);
for I:= 0, 53 do T:= ChIn(3);           skip unused header info
for I:= 0, 255 do
        begin
        Pal(I*3+2):= ChIn(3)>>2;        blue
        Pal(I*3+1):= ChIn(3)>>2;        green
        Pal(I*3+0):= ChIn(3)>>2;        red
        T:= ChIn(3);                    unused
        end;
for Y:= -(200-1), 0 do                  .bmp files are upside down!
    for X:= 0, 320-1 do
        Image(X-Y*320):= ChIn(3);
FClose(Hand);
end;    LoadBmp


proc    GenPal(Degree); Vary PalCombo by interpolating between Pal1 and Pal2
real    Degree;         varies from 0. to 1.
int     I, J;
begin
When Degree = 0. then PalCombo is at full intensity for Pal1.
When Degree = 1. then PalCombo is at full intensity for Pal2.
```

```
for I:= 0, 256-1 do                  for all the color registers...
   for J:= 0, 3-1 do                 for red, green, blue...
      PalCombo(I*3+J):= Pal1((I>>4)*3+J) +
           fix(Degree*float( Pal2((I&amp;#036;0F)*3+J) - Pal1((I>>4)*3+J) ));
end;   GenPal


begin   Main
CpuReg:= GetReg;                     access copy of CPU (H/W) registers
DataSeg:= CpuReg(12);                our data segment address is in DS reg.

LoadBmp("IMAGE1.BMP", Image1, Pal1);
LoadBmp("IMAGE2.BMP", Image2, Pal2);

Create a combined bitmap from the two 16-color bitmaps
Image1 gets the upper nibble, and Image2 gets the lower nibble
for I:= 0, 320*200-1 do
      ImageCombo(I):= Image1(I)
SetVid(&amp;#036;13);                          320x200x8-bit color graphics
Blit(DataSeg, ImageCombo, &amp;#036;A000, 0, 320*200);   copy bitmap to screen

XFade:= 0.0;                         start by showing Image2
repeat  XFade:= XFade + 0.005;
      GenPal(abs(1.0 - Mod(XFade, 2.0)) );
           varies between 1 and 0, then from 0 back to 1 (sawtooth)
      VSync;                         regulate speed
      port(&amp;#036;3C8):= 0;                write color registers
      for I:= 0, 256*3-1 do
           port(&amp;#036;3C9):= PalCombo(I);
until   ChkKey;

SetVid(&amp;#036;03);                          restore normal text mode
end;   Main
```

XFade is not only the name of this program, but it's also declared as the name of a 'real' variable. In 32-bit XPL0 integer variables have a range of -2147483648 to 2147483647, whereas 'real' variables range between +/-2.23E-308 and +/-1.79E+308 with 16 decimal digits (53 bits) of precision. This is plenty for what we're doing here.

Notice that the array declarations are taking up much more than 60K. In fact three of them take 64000 bytes apiece. 32-bit XPL0 uses extended memory, thus arrays can be megabytes in size.

Another little nicety available with this version of XPL0 is the 'port' command. It's used in the VSync procedure. This is a more efficient and intuitive way to access I/O ports. Contrary to what you might expect, directly accessing the hardware is a more consistent and reliable interface than calling BIOS routines to do these kinds of operations.

Similarly, there are three other command words used in this 32-bit code that are not available in the 16-bit, interpreted version of XPL0 (there you must use less efficient intrinsic calls to do the same operations):

'fix' converts (rounds) a real value to its closest integer.

'float' converts an integer to a real.

'abs' takes the absolute value of a real or an integer.

The Blit intrinsic is used to quickly copy a block of memory from one place to another. In this case the target location is video memory (segment address $A000). Since DOS (or Windows) decides where to put our data in memory, we must ask it where that is. GetReg provides access to an array that contains copies of the CPU's hardware registers. Among these is the data segment (DS) register which tells us where our data is located. In the program this information is stored into DataSeg, which is then passed on to Blit.

The Mod intrinsic takes the modulo of real numbers. It's similar to the remainder (Rem) intrinsic that is used for integers. For example, Mod(11.3, 2.0) = 1.3.

Here's a screen shot of what this program looks like, but you really need to see it run to appreciate the effect.

Lens

Here's Polaris's Lens example. This version takes advantage of another feature of 32-bit XPL0 by using 640x480 VESA graphics instead of the original 320x200 VGA.

```
include c:cxplcodes;          run-time library routines (intrinsics)

def     Radius = 150,         lens radius (pixels)
        Radius2 = Radius*Radius,
```

```
      Zoom = 3;              magnification factor

int    CpuReg;                address of CPU register array (from GetReg)
char   Image(640*480);        image from .bmp file


func   CallInt(Int, AX, BX, CX, DX, BP, DS, ES); Call software interrupt
int    Int, AX, BX, CX, DX, BP, DS, ES; (unused arguments need not be passed)
begin
CpuReg:= GetReg;
CpuReg(0):= AX;
CpuReg(1):= BX;
CpuReg(2):= CX;
CpuReg(3):= DX;
CpuReg(6):= BP;
CpuReg(9):= DS;
CpuReg(11):= ES;
SoftInt(Int);
return CpuReg(0) & &#036;FFFF;      return contents of AX register
end;   CallInt


func   OpenMouse;             Initializes mouse; returns 'false' if it fails
begin                      Pointer is at center of screen but hidden
CallInt(&#036;21, &#036;3533);          Make sure vector (&#036;33) points to something
if ((CpuReg(1) ! CpuReg(11)) & &#036;FFFF) = 0 then return false;
return if CallInt(&#036;33, &#036;0000) then true else false; reset mouse & return status
end;   OpenMouse


func   GetMousePosition(N); Return position of specified mouse coordinate
int    N;      = X coordinate; 1 = Y coordinate
For video modes &#036;0-&#036;E and &#036;13 the maximum coordinates are 639x199, minus
 the size of the pointer. For modes &#036;F-&#036;12 the coordinates are the same as
 the pixels. For 80-column text modes divide the mouse coordinates by 8 to
 get the character cursor position.
begin
CallInt(&#036;33, &#036;0003);
```

```
return (if N then CpuReg(3) else CpuReg(2)) & &#036;FFFF;
end;    GetMousePosition


proc    LoadBmp(Name);  Load 640x480x8 .bmp file into Image and set color regs
char    Name;           file name
int     Hand, I, T,
        X, Y,           coordinates
        R, G, B;        red, green, blue
begin
Hand:= FOpen(Name, 0);                  open file for input
FSet(Hand, ^I);                     set device 3 to handle
OpenI(3);

for I:= 0, 53 do T:= ChIn(3);           skip unused header info
port(&#036;3C8):= 0;                    set color registers
for I:= 0, 255 do
        begin
        B:= ChIn(3)>>2;
        G:= ChIn(3)>>2;
        R:= ChIn(3)>>2;
        T:= ChIn(3);
        port(&#036;3C9):= R;
        port(&#036;3C9):= G;
        port(&#036;3C9):= B;
        end;
for Y:= -(480-1), 0 do                  .bmp files are upside down!
    for X:= 0, 640-1 do
        Image(X-Y*640):= ChIn(3);

FClose(Hand);
end;    LoadBmp


proc    ExamineImage;   Run magnifier until a key is struck
int     MX, MY,         mouse coordinates
        X, Y,           screen coordinates
        C;              color of pixel to plot
```

```
begin
repeat  MX:= GetMousePosition(0);
        MY:= GetMousePosition(1);
        for Y:= 0, 480-1 do             constantly scan entire Image
           for X:= 0, 640-1 do
              begin
              if (MX-X)*(MX-X) + (MY-Y)*(MY-Y)                  C:= Image((X+(Zoom-1)*MX)/Zoom + (Y+(Zoom-1)*MY)/Zoom*640)
              else C:= Image(X+Y*640);
              Point(X, Y, C);
              end;
until   ChkKey;                     loop until a key is struck
end;    ExamineImage


begin   Main
if not OpenMouse then [Text(0, "A mouse is required");   exit];

SetVid(&#036;12);                       fool old mouse drivers
SetVid(&#036;101);                       640x480x8-bit color

LoadBmp("MRESCHER.BMP");
ExamineImage;

SetVid(&#036;03);                       restore normal text mode
end;    Main
```

The CallInt function uses the SoftInt intrinsic to access the hundreds of DOS and BIOS interrupt routines available in every PC. Note that the command 'func' is used instead of 'proc' to indicate that this subroutine returns a value. Also note the command 'return' at the end that specifies the value that is to be returned.

One thing to keep in mind when calling DOS and BIOS routines is that they live in a (humble) 16-bit world. The "& $FFFF" is used to mask off any junk that might be lurking in the high 16 bits of our 32-bit world.

An unusual characteristic of CallInt is that it allows a variable number of arguments to be passed to it. Only the register values (and place holders) that are actually used need to be passed.

Among the many BIOS interrupt routines are routines for using the mouse. These are made a little more programmer-friendly by the functions OpenMouse and GetMousePosition.

The meat of this program is in the procedure called ExamineImage. Everything else is either support or initialization code.

The Main routine at the bottom contains a couple new items. Note the 'exit' command. This exits the program, in this case, when a mouse is not detected. A value can optionally follow 'exit' and be used to return an error code to DOS. This works similar to the 'return' command, but 'exit' terminates the program.

Also note that brackets [] are another way to write 'begin' and 'end'. (XPL0 honchos prefer to use these sparingly. C programmers are welcome to go hog-wild.)

Here's what the screen looks like when Lens.exe is run, but you really need to move the lens around with the mouse to appreciate it.

Scroller

Here's Polaris's Scroller example.

```
string 0;use the zero-terminated string convention
include c:cxplcodes;          run-time library routines (intrinsics)

def     ScrWidth = 320,         screen dimensions (pixels)
        ScrHeight = 200;
def     FontWidth = 32,         font character dimensions (pixels)
        FontHeight = 64;
def     FontImageWidth = 1536,
        FontImageHeight = 64;

int     CpuReg,                 array containing copy of CPU hardware registers
        DataSeg,                segment address where our data got loaded
        FrameCount;

char    Message,                message string to be displayed
        Background(ScrWidth*ScrHeight), background screen buffer
        FontImage(FontImageWidth*FontImageHeight),      image from the .bmp file
        ScrBuffer(ScrWidth*ScrHeight);  screen buffer
```

```
func    StrLen(Str);            Returns the number of characters in a string
char    Str;
int     I;
begin
I:= 0;
while Str(I) #0 do I:= I+1;
return I;
end;StrLen


proc    LoadBmp(Name, Image);
Load .bmp file into Image and set up color registers
char    Name,           file name
        Image;          array to store image into
int     Hand, I, T,
        X, Y,           coordinates
        W, H,           width and height
        R, G, B;        red, green, blue
begin
Hand:= FOpen(Name, 0);                  open file for input
FSet(Hand, ^I);                         set device 3 to handle
OpenI(3);
for I:= 0, 17 do T:= ChIn(3);           skip unused header info
W:= ChIn(3) + ChIn(3)T:= ChIn(3) + ChIn(3)H:= ChIn(3) + ChIn(3)for I:= 24, 53 do T:= ChIn(3);          skip unused header info

port(&#036;3C8):= 0;                    set up color registers
for I:= 0, 255 do
        begin
        B:= ChIn(3)>>2;                 order is backwards
        G:= ChIn(3)>>2;
        R:= ChIn(3)>>2;
        T:= ChIn(3);
        port(&#036;3C9):= R;
        port(&#036;3C9):= G;
        port(&#036;3C9):= B;
        end;

for Y:= -(H-1), 0 do                    .bmp files are upside down!
```

```
    for X:= 0, W-1 do
        Image(X-Y*W):= ChIn(3);
FClose(Hand);
end;    LoadBmp


proc    MaskedBlit(ImSrc, ImDst, Xs, Ys, Xd, Yd, W, H);
Copy rectangular area of image
char    ImSrc, ImDst;   image array source and destination
int     Xs, Ys, Xd, Yd, coordinates in source and destination
        W, H;           width and height of area to copy
int     X, Y, C, Is, Id;
begin
for Y:= 0, H-1 do
        begin
        Is:= (Y+Ys)*FontImageWidth + Xs;
        Id:= (Y+Yd)*ScrWidth + Xd;
        for X:= 0, W-1 do
                begin
                C:= ImSrc(X+Is);
                if C then
                   if X+Xd>=0 & X+Xd=^A & Ch
Do a linear search
loop    begin
        for I:= 0, StrLen(ChMap)-1 do
                if ChMap(I) = Ch then quit; (I = index into ChMap)
        quit;                           in case Ch isn't found (I = StrLen)
        end;

Copy character from FontImage to ScrBuffer (if we found a valid one)
if I    if X+FontWidth>0 & X        MaskedBlit(FontImage, ScrBuffer,
                I*FontWidth, 0,         coordinates of letter in FontImage
                X, Y,                   coordinates to put letter in ScrBuffer
                FontWidth, FontHeight); font character dimensions
end;    DrawChar


proc    DrawString(X, Y, Msg);
```

```
int     X, Y;
char    Msg;
int     I;
begin
for I:= 0, StrLen(Msg)-1 do
        DrawChar(X+I*FontWidth, Y, Msg(I)&&#036;7F);
end;    DrawString


begin   Main
CpuReg:= GetReg;
DataSeg:= CpuReg(12);

SetVid(&#036;13);                       320x200x8-bit color
LoadBmp("BIGFONT.BMP", FontImage);
LoadBmp("BACKGND.BMP", Background);             use palette from Backgnd

Message:= "Greetz to all you XPL0 coders! ... Have a great time!! - Boreal";
FrameCount:= 0;
repeat  Blit(DataSeg, Background, DataSeg, ScrBuffer, ScrWidth*ScrHeight);
        DrawString(ScrWidth-FrameCount, 68, Message);
        if StrLen(Message)*FontWidth - FrameCount + ScrWidth                FrameCount:= 0;
        FrameCount:= FrameCount+1;
        VSync;
        Blit(DataSeg, ScrBuffer, &#036;A000, 0, ScrWidth*ScrHeight);
until   ChkKey;

SetVid(&#036;03);                       restore normal text mode
end;    Main
```

XPL0 normally terminates strings by setting the most significant bit on the last character. However this convention can be changed with the "string 0" command so that strings are terminated with a zero byte. This makes XPL0 strings compatible with the rest of the world.

About the only other new item in this program is the 'loop' command, used in the DrawChar procedure. XPL0 has four kinds of looping constructs: 'for', 'while', 'repeat' and 'loop'.

The 'loop' command works with the 'quit' command to provide a general way of handling loops. In this instance the 'for' loop scans the

character map (ChMap) for a matching character (Ch); and if one is found, the 'quit' command terminates both the 'for' loop and the 'loop' loop. (XPL0 does not have the much-abused 'goto' command.)

Here's a screen shot of the scroller in action:

Wrap-up

That pretty well wraps it up for this crash course in writing demos with XPL0. In Hugi 31 Polaris also provided a 3D wireframe demo. I'll save that one to launch a separate article.

Please visit my webpage for more examples and to download the 32-bit version of XPL0. It's probably a good idea to also download the 16-bit version, which contains the 127-page manual and smoothes the way toward understanding the 32-bit version.

Thanks for reading all of this, and thanks for your interest in XPL0! (It needs all the users it can get.)

-Boreal (aka: Loren Blaney)