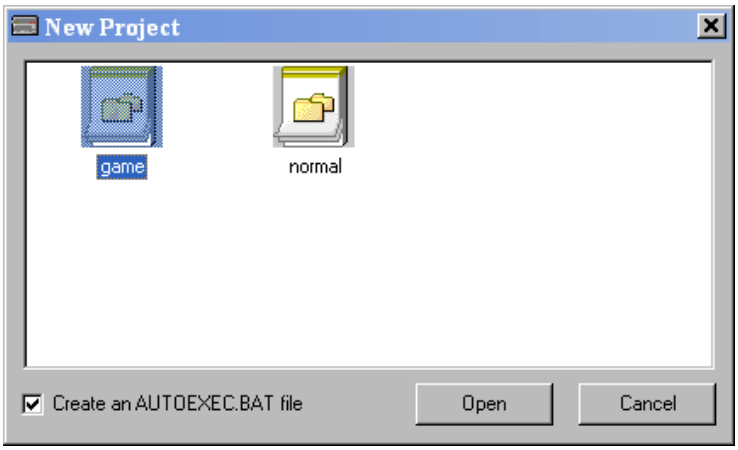
MSXPad HELP Index

MSX Pad is a program by SLotman from MSX Files

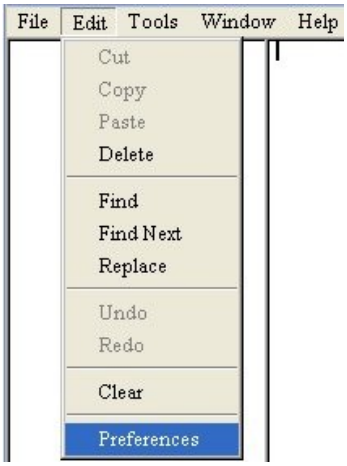
General Use

MSXPad setup

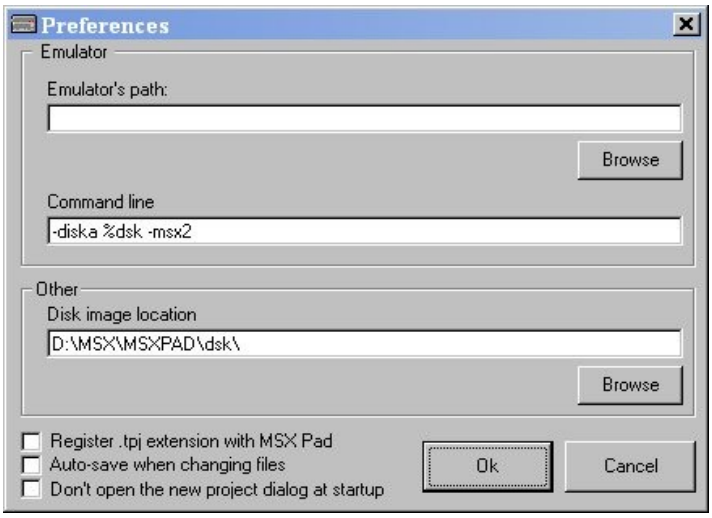
When the program starts for the first time, this screen will show up:



This screen is the projects templates dialog. For now, don't bother about it, just hit cancel. After that, the program will just sit there. You have to configure it, so you can use it correctly. On the menu, click on **EDIT→PREFERENCES**



Then the following screen will appear:



MSXPad works along with emulators, to make the development process easier and faster than on a real MSX. So, for this program to work 100%, you should have a working emulator installed on your computer.

After you have your program ready to compile, MSXPad will automatically generate a DSK file, and will run the emulator using that file. So, let's setup MSXPad so it can work correctly.

First click on the upper **BROWSE** button, which will open a dialog box, asking you where is the emulator you want to use with MSXPad. Any emulator that accepts command line arguments can be used. Select the emulator ".exe" file and the path to the emulator should appear on the textbox above the button.

Next, set the Command line on the textbox just below. The default command line should work on most emulators (on openMSX you should also specify a machine to emulate). The **%dsk** should be present, because that will be changed by the program to the dsk file generated at the specified location.

Now, set the location where the dsk file will be generated.To do this click the other **BROWSE** button and select the path. We are almost done.

MSXPad uses the .tpj extension on the generated projects, so you can associate them with the program. If you wish to do so, then mark the checkbox "**REGISTER .TPJ EXTENSION WITH MSXPAD**", so if you double click on any .tpj file, it will automatically open on MSXPad.

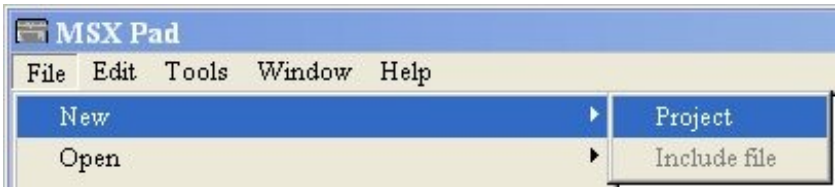
Check "**AUTOSAVE WHEN CHANGING FILES**", or everytime you change from the main pascal file to include files and you dont save first, all changes will be lost.

If you don't want that template screen appearing everytime MSXPad starts up, check the last checkbox "**DON'T OPEN THE NEW PROJECT DIALOG AT STARTUP**".

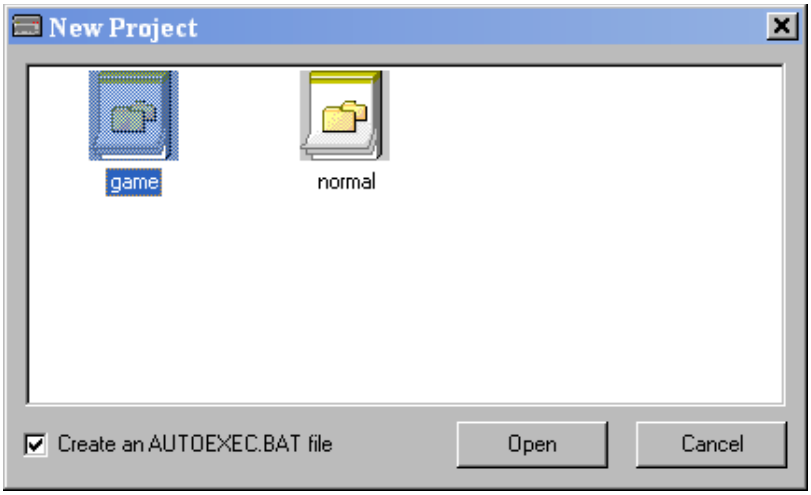
Click OK and the program should be working now.

Creating your first program

To create your first program, click on the menu on **FILE→NEW→PROJECT** as indicated below:



This will bring up the New projects template dialog, with 2 options:

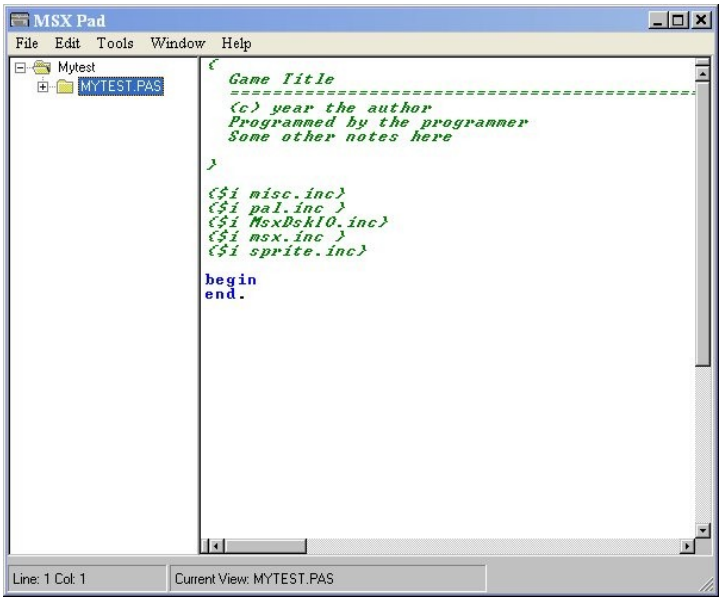


- Game: this will create a project with several include files pre-programmed, to create na MSX2 game.
- Normal: this will create an empty project for you, withou any pre-included file.

Juse select one of the two and hit the **OPEN** button. A dialog will open asking the name of the project file.

It is recommended that you create a directory for every new project you start. This directory should never have more than 720kb of files in it, otherwise the program will not be able to generate correctly a dsk file and run it on an emulator.

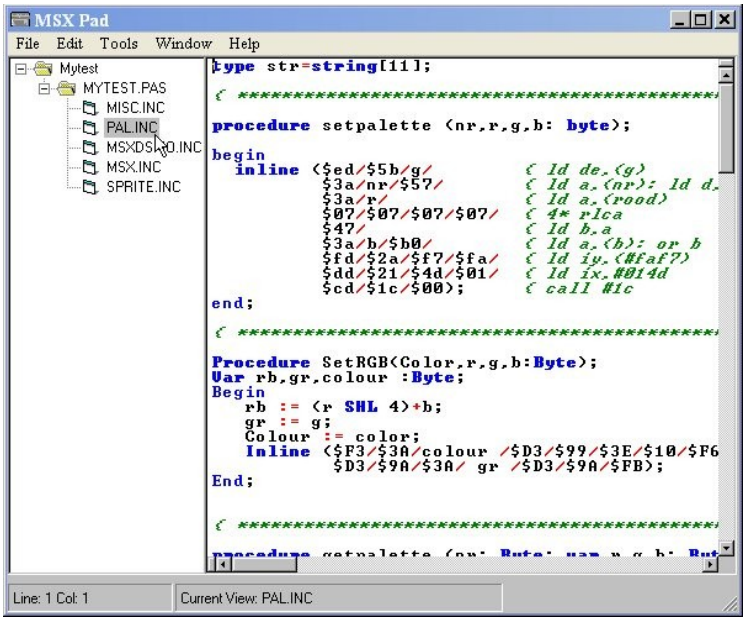
Click on **SAVE** and the basic program will be generated. If you choose a **GAME** project, you should get na screen like this:



This screen is divided in two parts. The left part shows the project structure. The right part shows the code of the file selected on the left part.

{ $i file.inc } means that at that point, an include pascal file will be inserted. Any include file inserted will show up on the left part of the screen, on the project structure (if they don't then click on **FILE→REFRESH PROJECT TREE**). Click the + sign at the side of MYTEST.PAS and all the include files will show up.

If you click on any of those include files, they will show up on the right side of the screen.



Now let's try it out, with a simple hello world. Click again on the MYTEST.PAS file on the left side of the screen, this will take you back to the main code. Insert the following code where you see the "begin… end" part:

```
Begin
   Screen(0);
   Writeln('Hellow world');
   Writeln('SLotman rox!');
End.
```

The program will look like this:

```
{
   Game Title
   ==============================
   (c) year the author
   Programmed by the programmer
   Some other notes here

}

{$i misc.inc}
{$i pal.inc }
{$i MsxDskIO.inc}
{$i msx.inc }
{$i sprite.inc}

begin
   Screen(0);
   Writeln('Hello world');
   writeln('SLotman rox!');
end.
```

That's because MSXPad recognizes the Pascal commands and colors them. Now press F5 or click on the menu on **FILE→COMPILE AND RUN** to execute your program. The chosen emulator should start, and you will be seeing the DOS prompt, something like A:>

Then to execute your program type **MYTEST** and hit **ENTER** (If you left the **CREATE AN AUTOEXEC.BAT FILE** option checked, the program will auto-execute (new option implemented on MSXPad 1.6). You should see an screen like this:

If you don't, then check your setup again.

That's the basic working of MSXPad. Now you can start creating your own programs for MSX! Study the given include files, they have a lot of functions pre-built to make your work easier.

Your own templates

Projects templates are the most versatile thing you can use on MSXPad. You can create templates for specific tasks, like for text mode programs, for MSX1 games, and so on.

To create a new template is very simple. Where you installed MSXPad you should find a directory called "Projects". Inside this directory there are two folders, GAME and NORMAL.

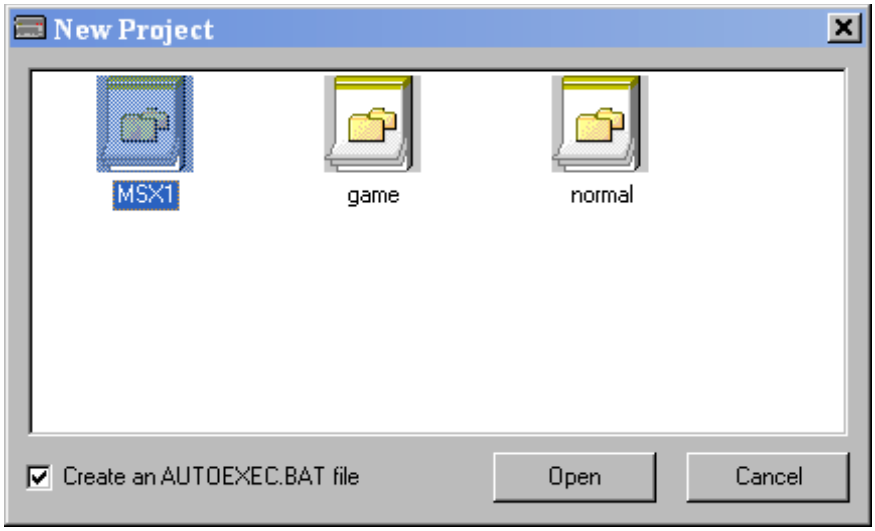
Now, let's imagine you have a file called "MSX1.inc" where you have all your MSX1 routines, and you want to make a MSX1 template. Go to the projects directory and create another folder, named MSX1 (the folder name is what will show up on the templates dialog). Then, copy your "MSX1.inc" file to this folder.

Now, go over the game folder and copy the files "command.com" and "msxdos.sys" to your msx1 folder. Those files are required if you want to run your compiled program on any MSX.

To finish up the template, create a new file called MSX1.pas (this file should have the same name as the folder it is, with the .pas extension). To have your MSX1.inc included automatically in the project, MSX1.pas should be like this:

```
{$i msx1.inc }

begin
end.
```

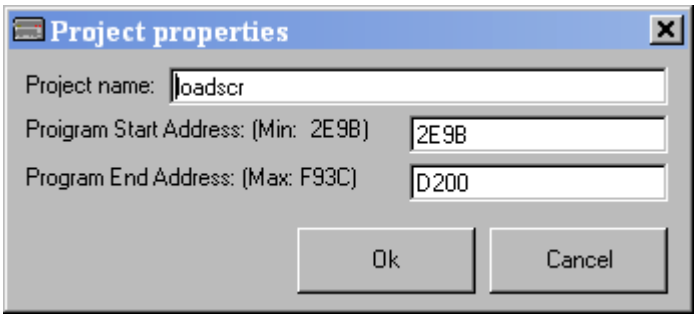
You should add na {$I } directive to every file you want to include on the template.
Save this file. Now let's test our first simple template! Open MSXPad, and if the template dialog does not show, click on **FILE→NEW→PROJECT**.



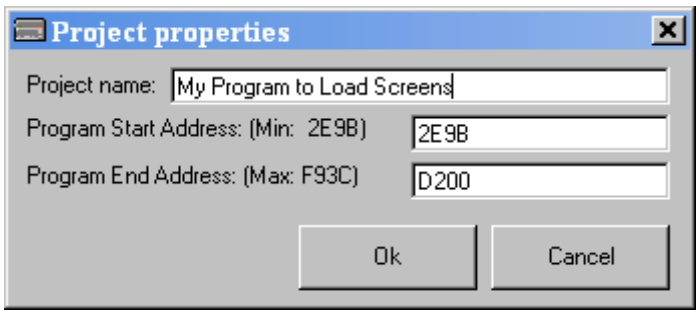
And there it is your MSX1 template on the New Project dialog! Click on **OPEN** and you will see your project on MSXPad. Easy isn't it?

Project properties

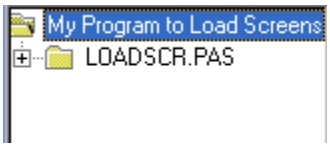
To show the Project properties dialog, click on **FILE→PROJECT PROPERTIES**. The dialog below will show up:



The first textbox asks for a name for your project, which can be anything, it's **not** related to the compiled program filename, so it can have any size, spaces, etc. See an example below:



After changing the Project name, it will appear on the left side of the main screen:



In the Project properties dialog, you also have two other fields, Program Start Adress and Program End Adress, which respectively represents where you program will start on memory and where it will end. The start address cannot be lower then 2E9Bh (it's where the Pascal runtime stays). The end address can go up to F93Ch, but it's not recommended to use further than D200h, or your program may not work correctly on IDE, MSX-DOS 2 or even under MSX-DOS 1.

Click **OK** and the settings will be saved.

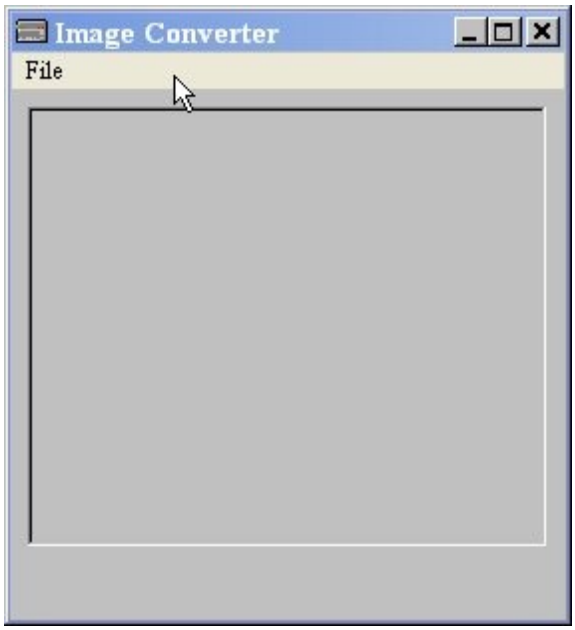
Graphics

Converting pictures to MSX

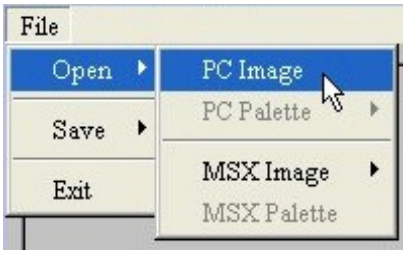
Let's see here how easy it is to convert PC pictures to MSX with MSXPad.

MSXPad can convert BMP files to screens 5 and 8. To convert BMP files to screen 5 you should have a 16 color BMP (since screen 5 uses only 16 colors). To convert to screen 8, the image should have 256 colors. In both cases the images should have 256 points in width and 212 points in height.

To open the image converter, click on **TOOLS→IMAGE CONVERTER**. A new window will popup.



Now click on **FILE→OPEN→PC IMAGE**. The program will ask you to point to a BMP File



The BMP file will then be loaded and showed up on the screen. Now you can save it as CC5 file (so you can load the image on BASIC with the COPY command) or you can save it as .GFX, which is a raw file format that can be loaded into your Pascal program. So, click on **FILE→SAVE→TURBO PASCAL SCREEN (GFX)** and the program will prompt you for a filename.

Choose a name and click on **SAVE**. Your .GFX image will be generated. Now let's create a small program to load it and see the result as it will appear on MSX.

Go to **FILE→NEW→PROJECT** and select **GAME**. On the "begin…end" part put the following code: (this program assumes you converted a picture to Screen 5, and name it SCRTEST.GFX)

```
{$i misc.inc}
{$i pal.inc }
{$i MsxDskIO.inc}
{$i msx.inc }
{$i sprite.inc}

begin
   Screen(5);
   LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
   Repeat until keypressed;
   Screen(0);
end.
```

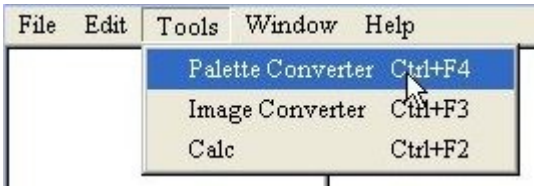
This program should load your image on screen. The LoadGraphic command works like this:

1 - The filename of the image to be loaded.
2 - The image width.
3 - The image height.
4 and 5 - The x and y coordinates on screen where the graphic will be drawn
6 - In which page the graphic will be loaded
7 - This one is the logical operation performed (OR, AND, XOR, etc)
8 - The screen mode (5 since we are loading the graphic on screen 5).

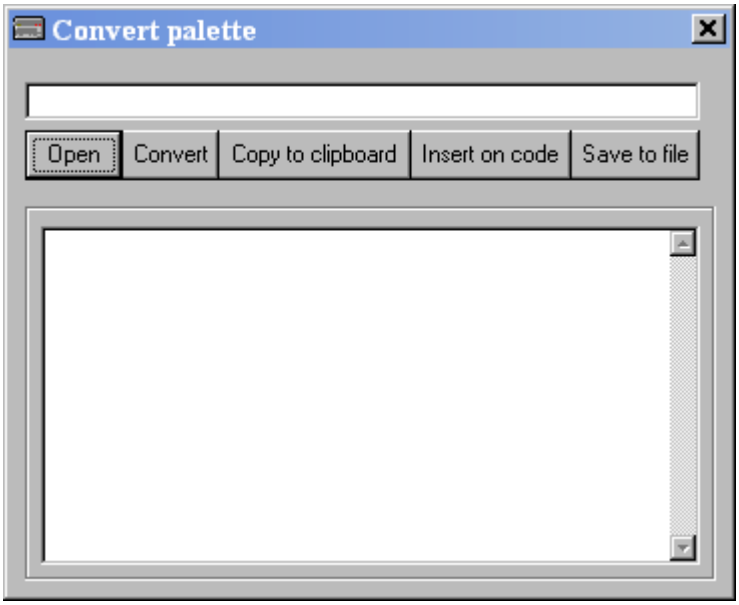
The rest of the program just waits for any key to be pressed, and then goes back to text mode. Let's see now the picture loaded on MSX:

Wait! That's all wrong! And I didn't took any pills!! What happened to all the colors?!
The color problem is normal. It's because you didn't set the palette on MSX, so it uses it's default one, and so, the colors appears all wrong. If your image was for screen 8, everything would be allright, since screen 8 has a fixed palette, and MSXPad convert your image from any palette to this fixed one… but on screen 5 it will be a problem. Don't worry, let's see how to fix this now.

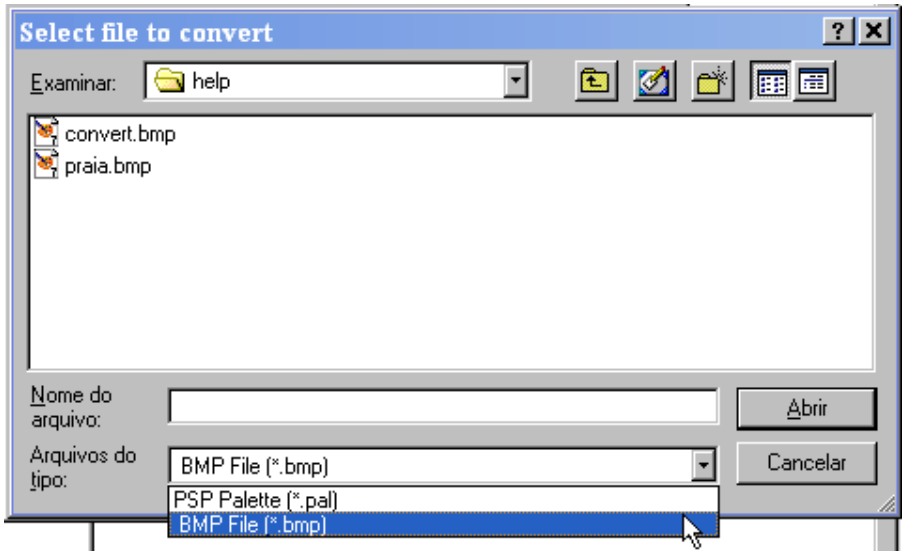
First go back to MSXPad menu, and click **TOOLS→PALETTE CONVERTER**



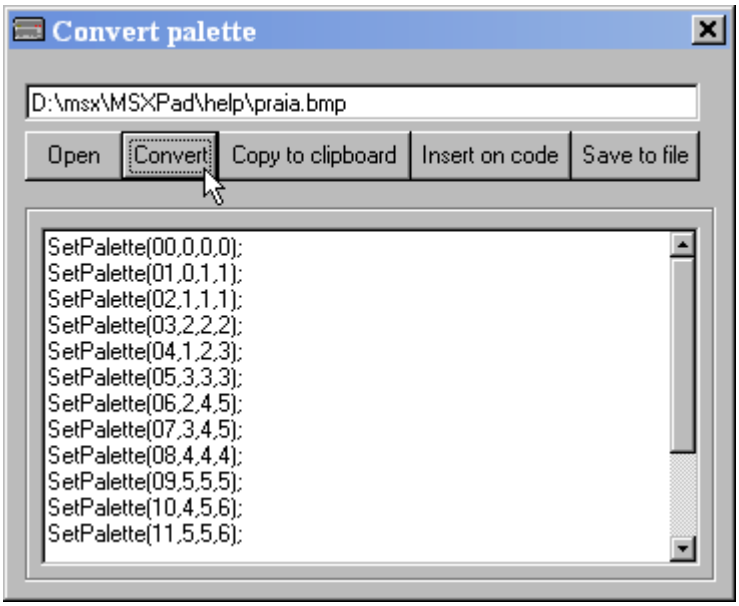
And a new window will show up. What this program does is to take the color of any BMP file (or from a Paint Shop Pro .PAL file) and convert it to MSX.



So, first we click on the **OPEN** button to load a BMP file (dont forget to change the file type to be loaded on the bottom of the open dialog)



After the image is loaded, click the **CONVERT** button. The converted palette will show up on the textbox just below. You have now two options, Click "**COPY TO CLIPBOARD**" button, and the code will be copied to the clipboard area of windows, or to Click "**INSERT ON CODE**" and the palette will be inserted directly into your code *WHERE YOUR CURSOR IS AT THE MOMENT*.



So, do not close this window, move it over to any side, and let's go back to our code.

```
begin
  Screen(5);
  LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
  Repeat until keypressed;
  Screen(0);
end.
```

Let's insert a PROCEDURE (a routine that executes some code) on top of this program. The program should look now like this:

```
Procedure SETPAL;
begin
 |
end;
begin
   Screen(5);
```

```
   LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
   Repeat until keypressed;
   Screen(0);
end.
```

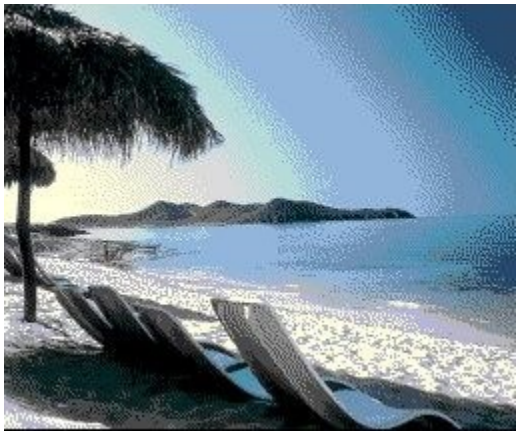
Let the cursor between the begin..end; of SETPAL. Now, go back to the palette converter screen and hit the **INSERT ON CODE** button. The program will look like this:

```
Procedure SETPAL;
begin
SetPalette(00,7,0,7);
SetPalette(01,0,0,3);
SetPalette(02,1,3,1);
SetPalette(03,0,0,5);
SetPalette(04,0,4,0);
SetPalette(05,0,6,0);
SetPalette(06,1,4,7);
SetPalette(07,1,7,7);
SetPalette(08,3,4,4);
SetPalette(09,6,0,0);
SetPalette(10,6,3,3);
SetPalette(11,4,4,1);
SetPalette(12,7,7,1);
SetPalette(13,7,6,3);
SetPalette(14,0,0,0);
SetPalette(15,6,7,7);
end;

begin
   Screen(5);
   LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
   Repeat until keypressed;
   Screen(0);
end.
```

Now, just call this procedure from the main code:

```
begin
   Screen(5);
   SETPAL;
   LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
   Repeat until keypressed;
   Screen(0);
end.
```

And the screen will now be perfect!



But… there is another way: inserting the palette *hardcoded* is easy, but consumes memory. If you want, you can also save the palette to file (using the SAVE TO FILE button - New in MSXPad 1.6 - to be

released). This will generate a .PL5 file which is a PALETTE file, in the same format used by Graph Saurus (an MSX2 graphic editor) and that can be loaded on BASIC with bload"",s. This same PL5 file can be loaded in Pascal with the following procedure:

LoadPal('filename.pl5',1);

This procedure is pre-defined in PAL.INC, an include file that works with palettes. The first parameter should always be the palette filename, the second one defines if the palette will be loaded directly on VRAM (and so the palette is set on screen) or if the palette should stay on RAM, waiting to be "activated". Any value different from zero will load the palette file and activate it. The current palette is always stored in the CurPal variable, declared on PAL.INC. Check the include for more info on this.

So your program should get like this now:

```
begin
   Screen(5);
   LoadPal('SCRTEST.PL5',1);
   LoadGraphic('SCRTEST.GFX', 256, 212, 0,0,0,0,5);
   Repeat until keypressed;
   Screen(0);
end.
```

Without the SETPAL procedure. The program gets smaller and much cleaner.

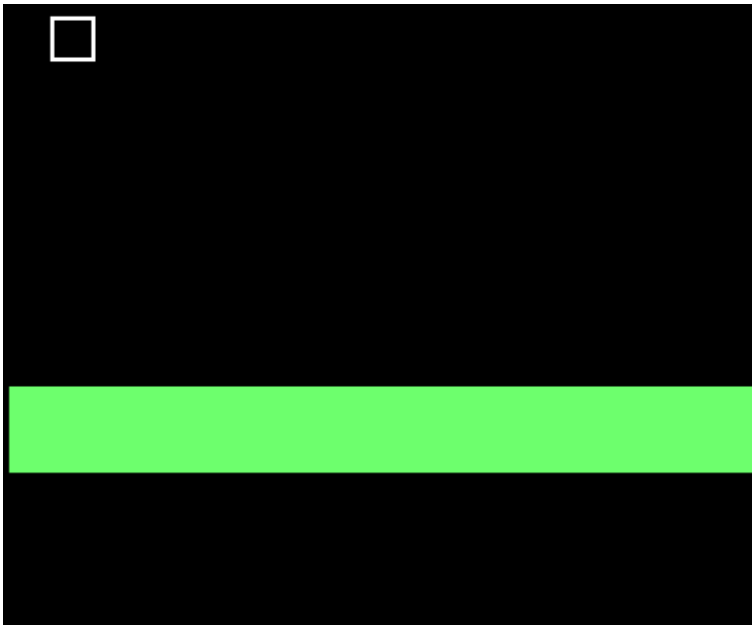
Using the COPY function

Let's see how you can make copies from a graphic from one VRAM part to another. One thing you should know (if you don't know it already) is that the VRAM is divided in PAGES. Each screen has an ammount of available pages, for example, screen 5 has 4 pages (0,1,2 and 3); screen 8 has only 2 pages (pages 0 and 1). When you first enter any screen, page 0 is the one “visible” by default.

Let's start with a basic program to draw something on the screen. Go to **FILE→NEW→PROJECT** and select **GAME**. Make sure the code is like the one below:

```
{$i MsxDskIO.inc}
{$i msx.inc }
{$i pal.inc }
{$i sprite.inc}
{$i misc.inc}

begin

 screen(5);               { enter screen 5 }
 setpage(0,0);            { visible page=0, active page=0 }
 color(15,0,0);          { same as basic command }
 cls2;                     { clears the active page }
 line_b(10,10,20,20,15,0); { draw an empty rectlangle,  just like line(),b on basic   }
 line_bf(0,100,255,120,3,0);{ draw an filled rectlangle }

 repeat until keypressed; { wait for a key }
 screen(0); { go back to screen 0 }
end.
```

This program will render an empty rectangle and a green bar to the screen, just as shown below:



The first thing you will notice is the **SETPAGE** command. This receives two parameters, the visible page and the active page. The visible page is the page that will be appearing on the screen (duh!) and the active page is the page where you will draw or work on.

The both **LINE_B** and **LINE_BF** commands are pretty similar to their BASIC counterparts, so there is no need for further explanations.

Now that the basic program is explained, let's move on to the good stuff. On MSX BASIC, you have the COPY command, which has the following syntax: COPY (x1,y1)-(x2,y2),page TO (x3,y3),page,LogOp.

On MSXPad, we have two copy commands, the first one is:

Copy (x1,y1,x2,y2,SrcPage,DestX,DestY,DestPage);

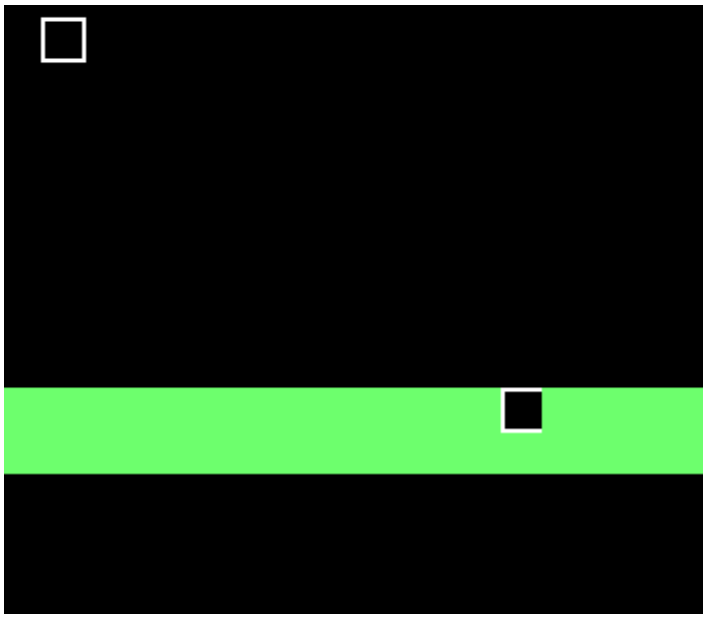
x1,y1 e x2,y2 are the top-left and bottom-right coordinates of the rectangle to be copied.
SrcPage is the page where the graphic will be copied from.
DestX, DestY are the X and Y location where the graphic should be copied to.
DestPage is the page where the graphic will be copied to.

(Note: on newer versions of MSXPad, this function was renamed to COPYF, and x2, y2 are now the width and height of the graphic to be copied. This was changed to gain more speed on the command)

So let's copy that white rectangle over the green bar. After the two line commands, put a copy like below:

```
line_b(10,10,20,20,15,0); { draw an empty rectlangle,  just like line(),b on basic   }
line_bf(0,100,255,120,3,0);{ draw an filled rectlangle }

copy(10,10,20,20,0,123,100,0);

repeat until keypressed; { wait for a key }
```

Then, run your program and you should see this:



Oh no! The rectangle was copied, but the black interior was copied too! How can I avoid this??
Well, to be able to copy only the white lines, you should make a logical operation. In this case, it should be TPSET, a logical operation that will not draw all the points with palette color 0 when the image is copied.

But the COPY command shown does not do any logical operation - and by not doing it, it's a little faster than the following command, CopyL, that does a logical operation when copying graphics. Let's take a look at it:

CopyL(x1,y1,x2,y2,SrcPage,DestX,DestY,DestPage,LogOp);
(Note: on newer versions of MSXPad, this function was renamed to COPYFL, and x2, y2 are now the width and height of the graphic to be copied. This was changed to gain more speed on the command)

You can see that it's exactly like the COPY command, but has an extra parameter LogOp, which defines the logical operator. The values below defines which operation will be performed:

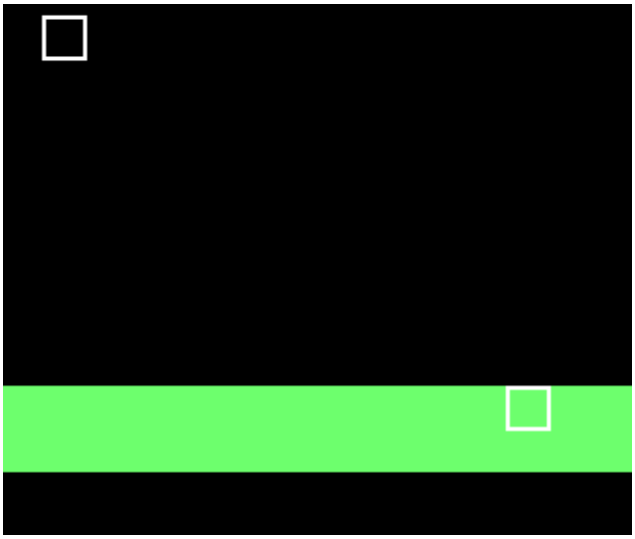
0 - Overwrite
1 - AND operation
2 - OR

3 - XOR
4 - NOT
8 - TPSET (already defined in MSX.INC, so you can write TPSET in the command without any problem)
9 - TAND
10 - TOR
11 - TXOR
12 - TNOT

So now change your program to this:

```
{$i MsxDskIO.inc}
{$i msx.inc }
{$i pal.inc }
{$i sprite.inc}
{$i misc.inc}

begin

 screen(5); { enter screen 5 }
 setpage(0,0); { visible page=0, active page=0 }
 color(15,0,0); { same as basic command }
 cls2; { clears the screen }
 line_b(10,10,20,20,15,0);{ draw an empty rectlangle, just like line(),b on basic }
 line_bf(0,100,255,120,3,0);{ draw an filled rectlangle }

 copyL(10,10,20,20,0,123,100,0,TPSET);

 repeat until keypressed; { wait for a key }
 screen(0); { go back to screen 0 }
end.
```

And as a result, you will see the following screen:



Now the rectangle is copied correctly. Try changing TPSET for other values on the Logical Operation table to see how each one changes the way the COPYL command works.

Let's have some more fun! Change the code to the one below (insert whatever is in bold in your code):

```
{$i MsxDskIO.inc}
{$i msx.inc }
{$i pal.inc }
{$i sprite.inc}
{$i misc.inc}
```

```
var c:char;

begin

 screen(5); { enter screen 5 }
 setpage(0,0); { visible page=0, active page=0 }
 color(15,0,0); { same as basic command }
 cls2; { clears the screen }
 line_b(10,10,20,20,15,0);{ draw an empty rectlangle, just like line(),b on basic }
 line_bf(0,100,255,120,3,0);{ draw an filled rectlangle }

 copyL(10,10,20,20,0,123,100,0,TPSET);

 setpage(0,1); { make page 1 active }
 cls2; { clear page 1 }
 Print(10,10,1,'You pressed ENTER!'); { writes 'You pressed ENTER' }

 repeat
   c:=inkey;
 until c=chr(13); { wait for enter key }

 { wait until keyboard buffer is empty }
 while keypressed do c:=inkey;

 repeat until keypressed; { wait for a key }

 screen(0); { go back to screen 0 }
end;
```

Press F5 to compile and…. No changes?! Wait, let's first understand what is being done here.

Setpage(0,1) tells MSX to draw or write anything on page 1, while still displaying page 0. Then CLS2 clears page 1, and Print will write the string "You pressed ENTER!" on position 10,10, color 1 on page 1. (remember page 0 is still visible, so you don't see any of this happening!)

Then comes a loop waiting for the enter key to be pressed. Another loop follows, this one waits until no key is pressed, and then the third loop waits for any key to be pressed, so the program can return to MSX-DOS.

Once again, add one more command to the code, this time just below the "until c=chr(13);" line, as shown below:

```
repeat
   c:=inkey;
 until c=chr(13); { wait for enter key }

 CopyL(10,10,154,20,1,10,100,0,TPSET);

{ wait until keyboard buffer is empty }
 while keypressed do c:=inkey;
```

And now run the program again… you should see the same screen! But, now press ENTER, and the screen will turn into this:

Hey it worked! What happened is that the text drawn on page 1 was copied to page 0, so you can see it.You can do a lot of cool tricks copying graphics from the other non-visible pages to the visible one… even animations!

Using Sprites

Sprites can be a lot of things in a program. In a application it can be the mouse cursor, in a game it can be the player , bullets, etc. Sprites in general are everything that moves on the screen, whitout erasing the contents below it.

Let's see how you can use sprites in your program now. Please note that those sprite functions will only work correctly on MSX2 or superior. First create a new **GAME** code:

```
{$i MsxDskIO.inc}
{$i msx.inc }
{$i pal.inc }
{$i sprite.inc}
{$i misc.inc}

begin
  Screen(5);
  color(15,1,1);
  cls2;

  repeat
  until keypressed;

  Screen(0);
end.
```

As usual, the program goes to screen 5, clears the screen and waits for a key to be pressed. Now let's start to implement our sprite.

```
{$i MsxDskIO.inc}
{$i msx.inc }
{$i pal.inc }
{$i sprite.inc}
{$i misc.inc}

{ sprite data }
Const mysprite: SprData=($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF);

begin
  Screen(5); color(15,1,3); cls2;
  InitSprites; { call this once always }
  EnableSprites;

  repeat
   UpdateSprites; { copy sprite buffer from RAM to VRAM }
  until keypressed;

  Screen(0);
end.
```

Once again, insert what's on bold into your code. Let's understand what's happening here:

```
Const mysprite: SprData=($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF);
```

This line is defining the sprite graphic, just like the Sprite$ command in BASIC. Every byte means one line of the 8x8 sprite. $FF = &B11111111, so this means a full line of 8 dots (each 1 means one dot, 0 means nothing drawn). Since all lines are $FF, this makes our sprite pattern to be a small 8x8 square. Then we have:

```
  InitSprites; { call this once always }
  EnableSprites;
```

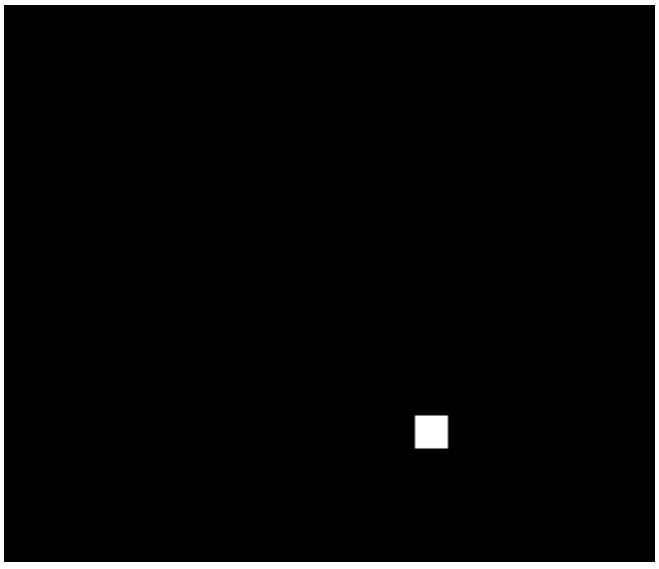
InitSprites will initialize some internal variables of the sprite engine, like the address of the sprite buffer, etc. EnableSprites is to make sure your MSX will have sprites enabled (they can be disabled by other programs, so the screen is updated a little faster - you can also do that by using the **DisableSprites** call)

```
  UpdateSprites; { copy sprite buffer from RAM to VRAM }
```

And for the last, as the comment says, this command copy all the sprite data to VRAM, making the sprites appear on screen. But wait! The program still does not show any sprites! Now comes the good stuff:

```
{ sprite data }
Const mysprite: SprData=($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF);

begin
  Screen(5); color(15,1,3); cls2;
  InitSprites; { call this once always }
  EnableSprites;

  WritePattern(0,mysprite); { writes pattern on pattern buffer on RAM }
  DrawPatterns; { copy all patterns from RAM to VRAM }
  ColorSprite(0,15,false); { Colorize sprite 0 }
  putsprite(0,100,100,0);  { put sprite 0 at 100,100 with pattern 0 on
                             sprite buffer on RAM }
  repeat
   UpdateSprites; { copy sprite buffer from RAM to VRAM }
  until keypressed;

  Screen(0);
end.
```

Now run your program…



…And the square sprite will appear on the screen! Now let's understand what have been done:

```
WritePattern(0,mysprite);
```

This line is actually setting on the pattern buffer at sprite pattern number 0, the sprite pattern you defined in mysprite,. You can have up to 32 sprites patterns at the same time on memory, from pattern 0 to pattern 31. The pattern buffer is a public variable that can be accessed directly by your program:

PtrBuf : Array[0..511] of byte;

So, if you want you can save all your sprites patterns in a file and read it directly to this array of bytes, instead of declaring each sprite on your code and setting one at a time.

```
DrawPatterns; { copy all patterns from RAM to VRAM }
```

This pretty much says it all. This is what actually copy all the 32 patterns from the buffer on RAM to the VRAM. If you are animating a sprite changing it's pattern on the fly (and this routine is fast enough to do it!), you should call this after updating all the patterns on the pattern buffer.

```
ColorSprite(0,15,false); { Colorize sprite 0 }
```

This line colorize the sprite with color 15. The parameter to this funcition are:

ColorSprite (sprite number, sprite color, enable ccBit);

The sprite number, is the number of the sprite to be colorized. The sprite color is a color number from 0 to 15. The "enable ccBit" is something special.

MSX has a feature which is called the ccBit. It allows you to mix the color of two sprites, creating a third color! So you can actually have three colors, using only two sprites. To achieve this effect, there are a few rules you must follow:

1 - Set the firt sprite (with the lower number) with ccBit set to FALSE.
2 - The next sprite (with the next number from the one you used on step #1) with the ccBit set to TRUE.

An example:

```
ColorSprite(0,1,false);
ColorSprite(1,2, true);

  putsprite(0,100,100,0);
  putsprite(1,100,100,1);
```

And back to the main code, the line that specifies where the sprite goes on screen:

```
putsprite(0,100,100,0);
```

This stores on the sprite buffer that the sprite 0 should be on position (100,100), using the pattern 0. The sprites are actually drawn on screen when the program reaches the **UpdateSprites** command.

Now let's see why sprites are so cool. Let's move the sprite around the screen! Change the code, as shown below:

```
Const mysprite: SprData=($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF);

var x:byte;

begin
  Screen(5); color(15,1,3); cls2;
  InitSprites; { call this once always }
  EnableSprites;

  x:=0;

  WritePattern(0,mysprite); { writes pattern on pattern buffer on RAM }
  DrawPatterns; { copy all patterns from RAM to VRAM }
  ColorSprite(0,15,false); { Colorize sprite 0 }

  repeat
   x:=x+1;
   putsprite(0,x,100,0);
   UpdateSprites; { copy sprite buffer from RAM to VRAM }
  until keypressed;

  Screen(0);
```

```
end.
```

Run the program, and you should see the sprite moving from left to right on the screen. You don't have to redraw the screen every time the sprite moves, because MSX does that for you.

Until now you saw how to create na 8x8 sprite (which is the default size for sprites). But what if you want a 16x16 sprite? Simple, just set this on your code:

```
begin
  Screen(5); color(15,1,3); cls2;
  InitSprites; { call this once always }
  EnableSprites;
  SpriteSize16;
```

And now each sprite will have a size of 16x16, made with four 8x8 patterns. The first 4 defined patterns defines the first 16x16 pattern, the next 4 the second pattern and so on. To set the sprite size to 8x8 again, use **SpriteSize8**.

And one last trick before finishing this lesson: let's show how to amplify the sprites. MSX have two kind of sprites, normal and amplified. You can have only one type on screen at a time, so you can't mix them both. To use the magnified sprites, just add this on the beginning of the code:

```
begin
  Screen(5); color(15,1,3); cls2;
  InitSprites; { call this once always }
  EnableSprites;
  SpriteSizeMagnified;
```

Now your tiny sprite will appear twice it's size! And you can also combine the **SpriteSize16** command and the **SpriteSizeMagnified** command, getting your 16x16 sprite to appear doubled as an 32x32 sprite! If you want to set manually sprites to normal, use **SpriteSizeNormal**

Input

Keyboard

Reading the keyboard is fairly easy with the included routines. All you have to do is call the inkey procedure, which is included on MSX.INC:

```
{ $I msx.inc }

Var ch: char;
Begin
    ch:=0;
    Repeat
        If keypressed then begin { keypressed is a boolean var that returns
                                   True if any key was pressed }
          ch:=inkey;
          Writeln('you pressed this char:',ch);
        End;
    Until ord(ch)=27; { ord returns the ascii code from any char, in this case it
                        Waits the user to press ESC to exit the main loop }
End.
```

You can also read the cursor keys using the stick and strig functions, which will be show just below. There's also a function called **EmptyKBDBuffer**, and as the name tries to explain, it clears the keyboard buffer.

Joystick

To read the Joystick just use the **STICK** function, which is also on MSX.INC. It works exactly like the BASIC command, which return a byte indicating the direction the joystick is pointing at (0 = center, 1 = up, 2 = up-right, 3 = right, 4 = down-right and so on). If you call **STICK(0)** you will read the keyboard cursor keys. Use **STICK(1)** for joystick 1 and **STICK(2)** for joystick 2.

```
{$I msx.inc }

begin
   repeat
      clrscr;
      writeln('keyboard:',stick(0));
      writeln('joystick 1:',stick(1));
      writeln('joystick 2:',stick(2));
   until (strig(0) or strig(1) or strig(2) or strig(3) or strig(4)); {wait any button or space key}
end.
```

You can read the buttons by using the **STRIG** function. **STRIG(0)** will return true if the spacebar is pressed, or false if not. **STRIG(1)** returns if the joystick 1 button A is pressed. **STRIG(2)** will return if joystcik 2 button A is pressed. **STRIG(3)** checks joystick 1 button B, and **STRIG(4)** will check joystick 2 button B.

Music and Sound

PSG

There are two ways to access the PSG on MSXPad, both much like it's done on BASIC.

The First method is to use the **SOUND** function implemented on MSX.INC, so let's start a small program with it.

```
{$i MSX.inc }

begin
end.
```

Now, we can use the **SOUND(register, value)** function to play some notes or sound effects in our program:

```
{$i MSX.inc }

var c:char;

begin
    Sound(8,0);   { mutes channel A }
    Sound(9,0);   { mutes channel B }
    Sound(10,0); { mutes channel C }

    Writeln('Press:');
    Writeln('[1] for playing a note on channel A');
    Writeln('[2] for playing a note on channel B');
    Writeln('[3] for playing a note on channel C');
    Writeln('[4] mutes all channels');
    Writeln('[ESC] to exit');
    Repeat
      If keypressed then begin
        c:=inkey;
        Case c of
            '1': begin
                    Sound(8,15);   { puts channel A with volume 15 }
                    Sound(0,100); { emits a sound on channel A }
                end;
            '2': begin
                    Sound(9,15);  { puts channel B with volume 15 }
                    Sound(2,50);  { emits a sound on channel B }
                end;
            '3': begin
                    Sound(10,15);  { puts channel B with volume 15 }
                    Sound(4,50);    { emits a sound on channel B }
                end;
            '4', chr(27): begin
                    { mutes all channels }
                    Sound(8,0);
                    Sound(9,0);
                    Sound(10,0);
                end;
      end;
    Until c=chr(27);
end.
```

As you can see, the **SOUND** function behaves just like the BASIC command.

The second option you have, is kind of a cheat. By using the BIOS, you can access the PLAY function from BASIC, and play music just like it. See the program below:

```
{$I MSX.inc }

Procedure WriteMem(endereco:integer; texto:tpstring);
{ this will write some bytes to na specific address in memory }
var indice:integer;
begin
   for indice:=1 TO length(texto) DO begin
      mem[Endereco]:=ORD(texto[indice]);
      Endereco:=Endereco+1;
   end;
   Mem[Endereco]:=0
end;

Procedure Play(MUSIC:tpstring);
{
  music is loaded at F41Fh and the code at F5EEh
  this is a direct call to PLAY basic command
}
var slot:byte;

begin
  WriteMem($F41F,MUSIC);
  if port[168]=$ff then slot:=$f0 else slot:=$a0;
  WriteMem($F5EE,
              #$DB+#$A8+#$F5+#$3E+chr(slot)+#$D3+#$A8+
              #$21+
              #$1F+#$F4+
              #$CD+#$E5+#$73+
              #$F1+#$D3+#$A8+#$C9);
  INLINE($C3/$EE/$F5);
end;

begin
   PLAY('CDEFGAB');
end.
```

As you can see, it works just like the PLAY command in BASIC.

MSX Music / MSX Audio

To play FM songs on Pascal, you have two alternatives.

The first one is to write directly to FM ports, using **PORT[$A0]:=register** and **PORT[$A1]:=value** – something very unpleasant.

The second option is to use the already made routines on MSXDOS2.INC, MAPPER.INC and MBPLAYER.INC libraries, which support not only MSX-Music, but also every music made on MOONBLASTER. The disadvantage of using those libraries, is that they need MSX-DOS2, and at least two mapper pages, one for the player and another for the music.

To use those libraries, you first must include them in your main program:

```
   {$i MSXDOS2.inc }
   {$i MAPPER.inc }
   {$i MBPLAYER.inc }
```

```
begin
end.
```

Then in the program start you should first check if the system has a mapper installed:

```
{$i MSXDOS2.inc }
{$i MAPPER.inc }
{$i MBPLAYER.inc }

begin
    { check if there is a mapper }
    if (MapperSupport(MapperCount)=false) then begin
       writeln('No mapper found.');
       halt;
    end else writeln('This system has ', MapperCount, ' mapper(s) installed');

end.
```

After making sure the system has a mapper, then we can proceed on checking if there is a MSX Audio or MSX Music compatible hardware, and then load the Moonblaster player and initialize it. Don't forget to put the file MBPLAYER.MPC on the same path your program is.

```
{$i MSXDOS2.inc }
{$i MAPPER.inc }
{$i MBPLAYER.inc }

begin
    { check if there is a mapper }
    if (MapperSupport(MapperCount)=false) then begin
       writeln('No mapper found.');
       halt;
    end else writeln('This system has ', MapperCount, ' mapper(s) installed');

    { Check music chips }
    if GetMusicChips > 2 then begin
       {
            GetMusicChips returns:
                0 for MSX Audio
                1 for MSX Music
                2 for MSX Audio and MSX Music
                255 for PSG only
     }
     writeln('Could not find MSX Audio or MSX Music!');
     halt;
    end;

    { load moonblaster player }
    if not InitMbmPlayer then begin
      writeln('Error loading Moonblaster player!');
      halt;
    end;

end.
```

Now all we have to do is to load some song and play it.

```
{$i MSXDOS2.inc }
{$i MAPPER.inc }
{$i MBPLAYER.inc }

begin
    { check if there is a mapper }
```

```
    if (MapperSupport(MapperCount)=false) then begin
       writeln('No mapper found.');
       halt;
    end else writeln('This system has ', MapperCount, ' mapper(s) installed');

    { Check music chips }
    if GetMusicChips > 2 then begin
       {
            GetMusicChips returns:
                  0 for MSX Audio
                  1 for MSX Music
                  2 for MSX Audio and MSX Music
                  255 for PSG only
        }
        writeln('Could not find MSX Audio or MSX Music!');
        halt;
    end;

    { load moonblaster player }
    if not InitMbmPlayer then begin
       writeln('Error loading Moonblaster player!');
       halt;
    end;

    { Load MBM song }
    LoadMBMFile('PROFARMX.MBM');

    { Start the music }
    StartMbmPlay;
    Writeln('Press any key to stop the music!');

    Repeat
    Until Keypressed;

    { Stops the music }
    StopMbmPlay;
end.
```

You can also use the **ContinueMbmPlay** function to Resume a song stopped by **StopMbmPlay**.

The player will run on background, so you can do whatever you want in your program and the music won't stop. If you start getting long delays while doing certain tasks, you can always force an IRQ to happen using a **INLINE($FF)** in your program.