Getting Started with ShowBasic

We assume here that you are already familiar with the Basic as a language, so we will concentrate on a specific ShowBasic features mostly leaving aside the Basic's programming concepts.

The world's famous method of learning the new language is to write a "Hello, world" program. Let's do it first! There are plenty of ways to write such a simple program using ShowBasic - here they are. Study them one by one.

"Hello, World!" using PAUSE statement.

Sample p01.sb Run

```
 pause "Hello, World!"
```

That's right - this is a complete program. When we run this program, ShowBasic shows the message box with the text and waits for the user to acknowledge by clicking OK button. That's all. The mission is accomplished.

"Hello, World!" using PRINT statement.

Sample p02.sb Run

```
 print "Hello, World!"
```

As you see, this program is not very complicated, but it allows us to introduce several useful ShowBasic concepts. When we run this program, ShowBasic shows the Viewport window with the text "Hello, World!" inside it. What is a Viewport? This is a special window that mostly stays hidden during execution of any ShowBasic program. The only way to output something into this window is to use ShowBasic's *print* statement. Here is the simple rule: when ShowBasic program ends, Viewport will be shown if it is not empty, i.e. if there was some output into the Viewport using *print* statement. To completely finish your program you need to close the Viewport. If Viewport is empty by the time when ShowBasic program ends - it will not be shown, and ShowBasic engine will terminate silently. You can control the visibility of the Viewport window explicitly during the program execution using *Viewport ON* or *Viewport OFF* statements. *Viewport CLEAR* will erase everything from the Viewport whether it is currently visible or hidden.

Why and when do you need the Viewport? Mostly for debugging your applications. It is very convenient to use the *print* statement in the critical points of your program to output the state of the variables and other information. Viewport can also be used as a standard output stream when you write some utilities using ShowBasic. Don't forget that using the *print* statement you can direct your output into some file as well.

"Hello, World!" using PopupTextBox().

Let's get back to our "Hello, World" program. Let's use a little bit more intelligent way to write this program… Here it is:

Sample p03.sb Run

```
PopupTextBox(SBC_CENTER, SBC_CENTER, 300, 100, "Hello, World!")
```

What will this program do? It shows a dialog box with the text field and two buttons (Continue and Stop). The text field has "Hello, World!" text in it. By default, this dialog box is in the modal state, i.e. our program will wait until the user clicks Continue or Stop button and only then the execution of the program continues (in our case the program will end because this is the last statement of our program). This return code of this function allows to detect which button was clicked. Let's modify the program in order to check it:

Sample p04.sb Run

```
ret% = PopupTextBox(SBC_CENTER, SBC_CENTER, 300, 100, "Hello, World!")
If ret% = TRUE Then
     print "You clicked Continue"
Else
     print "You clicked Stop"
EndIf
```

PopupTextBox function is a very simple and convenient way to interrupt temporarily execution of ShowBasic program in order to show some explanatory text and to wait for the response. There are many ways to customize the appearance of the text box and the behavior of this function, you can:

- control the dialog box position on the screen and its size;
- choose the font and center the text; select the range of font sizes so that the font will be selected automatically in order to fit all the text in the box; if text will not fit in the box - the scrollbars will be added automatically;
- select the style of the dialog box (3-D, custom 3-D, plain), colors for the text and the background;
- show from one to four different buttons (for example, Next, Back, Demo, Stop);
- make the dialog box modeless, control its appearance and interact with the user asynchronously;
- show the box for a limited time and continue execution automatically if the user didn't click any button; and more…

Let's change, as an example, the appearance of our text box:

Sample p05.sb Run

```
SBV_STOPBUTTON = SW_HIDE
SBV_FONTFACE = "Courier New"
SBV_FONTWEIGHT = 700
SBV_FONTSIZE = 48
SBV_MINFONTSIZE = 12
SBV_BGRCOLOR = RGB&(0,128,128)
SBV_BCOLOR = RGB&(255,255,0)
SBV_FCOLOR = RGB&(0,0,255)
PopupTextBox(SBC_CENTER, 30, 500, 250, "Hello, World!", SBC_ALIGNCENTER)
```

All variables that have SBV_ prefix are global system variables. Their current values affect the behavior and style of some ShowBasic functions. In our sample we changed the appearance of the text box by assigning different values to some system variables before calling *PopupTextBox()* function. Usually you need to assign values to system variables just at the beginning of your program so that all the dialog boxes and windows in your program will have a consistent look.

"Hello, World!" using Input Simulations.

Our next example of "Hello, World" program is very important. You know already that the major strength of ShowBasic is not in its ability to show dialog boxes and other windows - ShowBasic allows to control other Windows application, in other words we can make other application do what we want them to do. Let's use Notepad for our famous sample:

Sample p06.sb Run

```
Run("notepad.exe", NOWAIT)
Simulate("Hello, World!")
```

The above program starts Notepad and types "Hello, World!" in it. That's all. Consider now that we want not only to show "Hello, world" on the screen but also have it copied to the Clipboard. Let's modify our sample.

Sample p07.sb Run

```
Run("notepad.exe", NOWAIT)
wNotepad% = WinActivate("", "Notepad",5)
Simulate("Hello, World!+{Home}")
WinClickMenuChain(wNotepad%, "&Edit|&Copy", 15)
WinClickMenuChain(wNotepad%, "&File|E&xit", 15)
WinActivate("Notepad", "#32770", 2)
WinClickControl("&No", "Button", 50, 50, 15)
```

The second line of the above sample demonstrates the use of *WinActivate()* function. This function allows us to wait until the Notepad starts and makes its main window active, at the same time it returns the handle of the Notepad window and assigns it to *wNotepad%* variable. Look at the *Simulate()* function - we've changed it a bit, "*+{Home}*" at the end of its parameter string actually allows us to highlight the words that we've typed into the Notepad, this is the equivalent of pressing SHIFT-HOME key combination. The *wNotepad%* variable is used in the following two functions. *WinClickMenuChain()* functions visually simulates selections from the Notepad menu using the mouse. First function selects *Edit->Copy* and the next one does *File->Exit*. Note using the ampersand symbol in words that represent menu items - you have to use this symbol in front of the underlined letter of a menu item. Now look at the last statement. When we select *File->Exit*, Notepad will pop up the dialog box asking whether we want to save the changes. We don't want it - and our last *WinClickControl()* function moves the mouse to the center of "No" button and clicks on it. Now let's concentrate on the previous line - *WinActivate()* function. Why do we need to use *WinActivate()* before simulating the click into "No" button? Because we don't want to rely on timing, and we'd like to make sure that the required message box is up before clicking into it.

Note that in the above sample we were using **visual** simulations with the mouse. This approach is preferable when you really need to show how you control the application. Sometimes, however, you don't care about the visualization - if all you need is to automate some application, it may be easier and faster to use keyboard simulations instead of the mouse simulations. Let's modify our sample:

Sample p08.sb Run

```
Run("notepad.exe", NOWAIT)
wNotepad% = WinActivate("", "Notepad",5)
Simulate("Hello, World!+{Home}%ec%fx")
WinActivate("Notepad", "#32770", 2)
Simulate("n")
```

Look at the last six characters used in the text string passed as a parameter to the first *Simulate()* function - "*%ec%fx*". *%* is a special prefix that allows to simulate the combination of a character with the Alt key, so the above sequence means "press Alt-E and then C". When this sequence is played into active Notepad window, it will select "*Edit->Copy*" from the Notepad's menu. The next three characters simulate Alt-F and then X to exit Notepad (*File->Exit*). The last *Simulate()* function plays N into the confirmation dialog box.

Next sample demonstrates more extensive mouse simulations on Paintbrush (or Paint if the user runs Windows 95).

Sample p09.sb

```
Const PlayString$ = "{LeftDown 19 19}.....{LeftUp 323 122}"
Const PaintID$ = "#0"
Const BrushID$ = "#1"
Const PaintClass$ = "Afx:8"
Const BrushClass$ = "pbPaint"
Const VirtSize$ = "{VirtSize 699|477}"

DIM sI As SYSTEMINFO

WinInfoSystem(sI)
IF  MID$(sI.si_szWinVer,1,3) = "W95" THEN
    ID$ = PaintID$
    Class$ = PaintClass$
ELSE
    ID$ = BrushID$
    Class$ = BrushClass$
ENDIF

Run("pbrush.exe", NOWAIT)

wPaint% = WinActivate("", Class$, 10)
IF wPaint% = 0 THEN
    SBV_CONTBUTTON = SW_HIDE
    PopupTextBox(SBC_CENTER, SBC_CENTER, 250,150, \
    "Sorry, can't start or locate Paintbrush or Paint", \
    SBC_THREED or SBC_ALIGNCENTER or SBC_TOPMOST)
    exit
ENDIF

ret% =    PopupTextBox(SBC_CENTER, SBC_CENTER, 350,200, \
"Please select a \"free-hand\" drawing tool \
(pencil, brush or airbrush) and some foreground color.\n\n\
Make sure that the drawing area is big enough and then \
click Continue.", \
SBC_THREED or SBC_ALIGNCENTER or SBC_TOPMOST)

IF ret% = TRUE THEN
    Simulate("{Origin Client Caption|" + ID$ + \
             "}{Origin Client Class|" + Class$ + "}" + \
             VirtSize$ + PlayString$)
ENDIF

SBV_CONTBUTTON = SW_HIDE
PopupTextBox(SBC_CENTER, SBC_CENTER, 250,150, \
"Click Stop to close Paint/Paintbrush and finish.", \
SBC_THREED or SBC_ALIGNCENTER or SBC_TOPMOST)

WinActivate(wPaint%, 1)
Simulate("%fx")
IF WinActivate("", "#32770", 1) Then
    Simulate("n")
ENDIF
```

There is nothing special in this sample - it demonstrates basically the same techniques which you have learned already with the previous simpler samples. Please study it, just pay attention to the following notes:

- The first line in the sample above is incomplete - in reality it is very long, and it was generated using ShowBasic Recorder. This is a series of a mouse movements that draws "Hello, World!" in Paintbrush. You can find the complete sample in the source file.
- We expect that this sample can be run in Windows, Windows NT and Windows 95, we know that on Windows 95 "pbrush.exe" is actually another application - that's why we check the environment (*WinInfoSystem()*) and construct the simulation string depending on the operating system in use.
- The parameter for *Simulate()* function is being constructed from several pieces on the fly - it is using several pseudo-commands: *{Origin Client Caption|XXX}{Origin Client Class|YYY}* ensures that the following mouse movement commands will be played in the window with the specified caption (ID) and Class-name; *{Virtsize W|H}* command ensures that the coordinates in the following mouse movement commands will be converted (scaled) from the given virtual to the actual window's size.

"Hello, World!" using Bitmaps and Metafiles.

Next version of our "Hello, World" program will use ShowBasic's popup windows. There are many varieties of this graphical windows in ShowBasic - there are also dozens of different ways to customize them. We will show here just the basics.

First let's use metafiles and bitmaps to do what we want.

Sample p10.sb Run

```
SBV_BORDER = 10
SBV_BORDERSTYLE=SBC_UP
PopupBitmap(SBC_CENTER, SBC_CENTER, 0, 0, "hellow.bmp")
```

The above sample pops up a modal window (you need to click anywhere inside it to close it and continue) and shows the image from hellow.bmp file inside this window. The window will be centered on the screen (first two parameters of the *PopupBitmap()* control this) and the size of the window will be defined by the native size of the bitmap image (two next parameters specified as zeros control this). The first two lines assign values to system variables which control one of the possible popup window decorations - the border. Note, that our bitmap has black background color. Let's have some fun and show "Hello, World!" drawn transparently over existing screen image - using the same bitmap:

Sample p11.sb Run

```
SBV_TIMER = 5
PopupBitmapTrans(SBC_LEFT, SBC_TOP, 0, 0, "hellow.bmp", RGB&(0,0,0))
```

When you run this sample, you see the "Hello, World!" text from our bitmap on the upper left of the screen. This time there is no black background - we defined the black color (RGB&(0,0,0)) as the "transparent" color - and as a result you see everything whatever happened to be under this color on the screen. Note that we were using another function - *PopupBitmapTrans()*. The first line assigns 5 to the system variable SBV_TIMER - the effect is that if you don't close your popup window by clicking inside it, it will be closed after 5 seconds automatically.

Metafile images can be used in a similar way - let's demonstrate it in the following sample:

Sample p12.sb Run

```
DIM sI As SYSTEMINFO
DIM rc As RECT
WinInfoSystem(sI)
RectSet(rc, sI.si_wXScreen/2, sI.si_wYScreen/2,\
            sI.si_wXScreen/2, sI.si_wYScreen/2)
w% = sI.si_wXScreen/3
h% = sI.si_wYScreen/4
offset% = sI.si_wYScreen/12
SBV_BGRCOLOR = RGB&(0,128,0)
PopupMetafile( offset%, offset%, w%, h%, "hellow.wmf",\
               SBC_FILLBGR or SBC_MODELESS, rc)
SBV_BGRCOLOR = RGB&(0,0,128)
PopupMetafile( -offset%, -offset%, w%, h%,\
               "hellow.wmf", SBC_FILLBGR or SBC_MODELESS, rc)
PopupMetafile( 0, 0, sI.si_wXScreen, sI.si_wYScreen,\
              "hellow.wmf", SBC_MODELESS)
WaitInput(SBC_KEYBOARDINPUT or SBC_MOUSEINPUT)
```

The above sample introduces several more concepts and functions besides the metafile pictures. First two lines declare two variables of predefined types - SYSTEMINFO is a predifined in ShowBasic type which is used just by one function - *WinInfoSysytem()*. This function fills numerous system information data in the structure passed as a parameter. We are using it here to determine the screen dimensions. We also define the variable *rc* of system type *RECT*. This is a structure which defines the rectangle's dimensions. We set this dimensions using *RectSet()* function in the following line (actually, we specify the rectangle with the zero length and height here, basically this is a point). Then we use the current screen dimensions to calculate the size for two metafile windows and the offset to show them on the screen. *SBV_BGRCOLOR* system variable is used to define the background color for the metafile picture. Now take a look at the following *PopupMetafile()* function. The last two parameters are new to us - the first one defines the style of the popup window as *SBC_FILLBGR or SBC_MODELESS*. It is quite obvious what this style is: show the background defined by the current value of *SBV_BGRCOLOR* variable, and make this window modeless, i.e. after showing the window do not wait for   click - continue with the script execution immediately. The last parameter, *rc*, defines the so-called "focus" rectangle. It allows to create a special "focus" decoration for our popup window - you will see what it is when you run the program.

The last (third) *PopupMetafile()* function is a little bit different - it uses the whole screen for the size of its popup window and it doesn't use background color and focus decorations. The result you will see when you run this sample program.

Note that all created modeless windows will be automatically destroyed when the script ends - we would like to delay this destruction so that we can see the actual output, that's why we use *WaitInput()* function at the end. This function stops execution of our script until you click anywhere on the screen or press any button on the keyboard.

"Hello, World!" using PopupText() function.

PopupText() function allows to create a graphical text window look at the following example:

Sample p13.sb Run

```
SBV_FONTFACE = "Times New Roman"
SBV_FONTWEIGHT = 700
SBV_FONTSIZE = 72
SBV_MINFONTSIZE = 10
SBV_BGRCOLOR = RGB&(0,0,255)
SBV_FADECOLOR = RGB&(0,0,128)
SBV_BORDER = 10
SBV_BORDERSTYLE = SBC_FRAMEUP
SBV_FCOLOR = RGB&(255,255,0)
PopupText( SBC_CENTER, SBC_CENTER, 450, 250, "Hello, World!",\
           SBC_ALIGNCENTER or SBC_GRADIENTBGR or SBC_TOPMOST)
```

The function *PopupText()* is quite straightforward - it creates a popup window and draws the given text inside it. There are several interest points in this sample. First, we've created a special "gradient" background for this window. How? Using the style *SBC_GRADIENTBGR* and assigning the range of colors to two system variables: *SBV_BGRCOLOR* and *SBV_FADECOLOR*.

Another interest point is defining the font size for the text inside the window. We assign the desirable font size to *SBV_FONTSIZE* variable, however we are not sure whether our text will fit into the window. The function automatically finds the closest to the desired font size so that the whole text will fit. Word wrap occurs automatically if needed. The range of possible font sizes is limited by the value of *SBV_MINFONTSIZE* variable.

One more thing to look at - we use *SBC_ALIGNCENTER* as one of the flags for the window's style. As a result, the text inside window is centered.

"Hello, World!" using direct drawing.

One more way to do "Hello, World!" program is to draw directly into the popup window. Here is how it can be done:

Sample p14.sb Run

```
SBV_FONTFACE = "Times New Roman"
SBV_FONTWEIGHT = 700
SBV_FONTSIZE = 48
SBV_BGRCOLOR = RGB&(255,255,0)
SBV_BORDER = 10
SBV_BORDERSTYLE = SBC_FRAMEUP
SBV_SHADOW = 16
hWnd% = PopupWindow(SBC_CENTER,SBC_CENTER,400,300,\
                    SBC_MODELESS or SBC_FILLBGR)
PopupString(hWnd%, "Hello,",20, 20, 400, 150, RGB&(128,0,0),\
            SBC_NULLCOLOR, DT_LEFT or DT_VCENTER or DT_SINGLELINE)
PopupString(hWnd%, "Hello,",15, 15, 395, 145, RGB&(255,0,0),\
            SBC_NULLCOLOR, DT_LEFT or DT_VCENTER or DT_SINGLELINE)

PopupString(hWnd%, "World!",5, 170, 380, 300, RGB&(0,0,128),\
            SBC_NULLCOLOR, DT_RIGHT or DT_VCENTER or DT_SINGLELINE)
PopupString(hWnd%, "World!",0, 165, 375, 295, RGB&(0,0,255),\
            SBC_NULLCOLOR, DT_RIGHT or DT_VCENTER or DT_SINGLELINE)

PopupUpdate(hWnd%)
WaitInput(SBC_KEYBOARDINPUT or SBC_MOUSEINPUT or SBC_IGNOREALLINPUT)
```

Let's see what we do here. First, we assign required values to system variables that control window's appearance, font etc. The new variable here is SBV_SHADOW - if it's value is not a zero, the special shadow decoration will be added automatically to popup windows.

PopupWindow() function creates the empty modeless popup window and fills its background. The core of this sample is *PopupString()* function - this function allows to draw some text into a popup window (first parameter specifies such window's handle, and we use the handle returned by our *PopupWindow()* function call). This function gives you a pretty good "low-level" control in order to place your text at some exact position and to choose specific colors, fonts etc. for every piece of text. In our sample we draw "Hello," in the top part of the window, and "World!" at the bottom. Every string we draw twice in different colors and using a small offset to the upper-left for the second. This creates a "shadowed-text" effect.

Note the very important feature of this function: it does not update the image of the window immediately - you have to call *PopupUpdate()* function to show all changes to the window's image. Why? Because this allows to avoid the "flicker" when you construct the window's image in many steps, this feature is also essential when using special transition effect. *PopupString()* is just one of several "drawing" functions which allow you to draw lines, ellipses, rectangles and polygons into the window's image.

Now let's add some "bells & whistles" to our sample, namely let's use a transitions effects. Here is how we do it.

Sample p15.sb Run

```
DIM rc As RECT
SBV_FONTFACE = "Times New Roman"
SBV_FONTWEIGHT = 700
SBV_FONTSIZE = 48
SBV_BGRCOLOR = RGB&(255,255,0)
SBV_BORDER = 10
SBV_BORDERSTYLE = SBC_FRAMEUP
SBV_SHADOW = 16
hWnd% = PopupWindow(SBC_CENTER,SBC_CENTER,400,300,\
                    SBC_MODELESS or SBC_FILLBGR)
PopupString(hWnd%, "Hello,",20, 20, 400, 150, RGB&(128,0,0),\
            SBC_NULLCOLOR, DT_LEFT or DT_VCENTER or DT_SINGLELINE)
PopupString(hWnd%, "Hello,",15, 15, 395, 145, RGB&(255,0,0),\
            SBC_NULLCOLOR, DT_LEFT or DT_VCENTER or DT_SINGLELINE)

RectSet(rc, 15,15,400,150)
PopupSetEffect(SBC_EFF_SLIDE, SBC_DIR_R2L, 1, 1, 1)
PopupUpdate(hWnd%,  rc, SBC_EFFECT)

PopupString(hWnd%, "World!",5, 170, 380, 300, RGB&(0,0,128),\
            SBC_NULLCOLOR, DT_RIGHT or DT_VCENTER or DT_SINGLELINE)
PopupString(hWnd%, "World!",0, 165, 375, 295, RGB&(0,0,255),\
            SBC_NULLCOLOR, DT_RIGHT or DT_VCENTER or DT_SINGLELINE)

RectSet(rc, 0,165,380,300)
PopupSetEffect(SBC_EFF_SLIDE, SBC_DIR_B2T, 1, 1, 1)
PopupUpdate(hWnd%,  rc, SBC_EFFECT)

WaitInput(SBC_KEYBOARDINPUT or SBC_MOUSEINPUT or SBC_IGNOREALLINPUT)
```

Note that we are using *PopupUpdate()* function twice here - first, after drawing "Hello," string, and then after drawing "World!" string. The new feature of the *PopupUpdate()* function is that it is capable of updating not the whole window but a particular part of it defined as a rectangle. The last parameter to *PopupUpdate()* forces it to use the current transition effect (set by the previous *PopupSetEffect()*) function when applying the change. The transition is done between two images: first is the image in the specified rectangle of the window **before** any changes by *PopupString()* and the like functions were applied, and the second one is the image **with** all the changes that has to be applied by *PopupUpdate()*. The result, as you see, is the transition between the empty yellow rectangle and the similar yellow rectangle with some extra words drawn inside it.

"Hello, World!" using custom dialog boxes.

Sample p16.sb Run

```
SBV_STYLE = SBC_3D or SBC_TOPMOST
DIM editText$, indexVar%
DIALOG DlgDialog SBC_CENTER SBC_CENTER 188 98 "Dialog"
      OKBUTTON 39 77 50 14 "Hello,"
      CANCELBUTTON 97 77 50 14 "World!"
      OPTIONGROUP indexVar%
             OPTIONBUTTON 142 41 38 8 "Hello,"
             OPTIONBUTTON 142 55 38 8 "World!"
      END OPTIONGROUP
      EDITTEXT 3 23 72 12 editText$
      CTEXT 4 5 180 10 "Hello, World!"
END DIALOG

indexVar% = 1
editText$="Hello, World!"
PopupDialogBox(DlgDialog)
```

Custom dialog boxes is the most powerful way to provide the interactions with the user. You can use *Popup...* set of functions if all you need in terms of interaction is to output information and to request a simple feedback (like OK or CANCEL, for example). Custom dialog boxes comes handy when you need a complex interaction with the user like forms, multiple-choice questions etc.

The above sample demonstrates a simple (and useless) dialog box just to introduce the concept. The template of the dialog box is defined between pair of *DIALOG* and *END DIALOG* statements. In our sample we define two buttons, group of two radio-buttons, edit field and the static text field. Each control can have a variable associated with it - and this variable always reflects the state of the corresponding control. If the user types a text into some edit field - the corresponding variable gets changed, and there are similar rules for all other types of controls.

Dialog boxes can be modal and modeless. There is also event mechanism in ShowBasic that allow you to react immediately on any change in any control.

There are also many ways to customize the appearance of dialog boxes - the following sample will demonstrate this.

Sample p17.sb Run

```
SBV_STYLE = SBC_TOPMOST
SBV_BGRPICTURE = "hellow.wmf"
SBV_BGRCOLOR = RGB&(128,128,0)
SBV_BCOLOR = SBC_NULLCOLOR
DIM editText$, indexVar%
DIALOG DlgDialog SBC_CENTER SBC_CENTER 188 98 "Hello, World!"
    OKBUTTON 64 77 50 14 "Hello,"
    CANCELBUTTON 122 77 50 14 "World!"
    OPTIONGROUP indexVar%
        OPTIONBUTTON 142 41 38 8 "Hello," 0 13 NULL_EVENT\
                     "MS Sans Serif;8;Bold;0;0,255,0;128,128,0"
        OPTIONBUTTON 142 55 38 8 "World!" 0 14 NULL_EVENT\
                     "MS Sans Serif;8;Bold;0;0,0,255;128,128,0"
    END OPTIONGROUP
    EDITTEXT 3 23 72 12 editText$ 0 15 NULL_EVENT\
             "Times New Roman;10;Bold;0;128,0,0;255,255,255"
    CTEXT 4 5 100 10 "Hello, World!" 0 16 NULL_EVENT\
          "Courier New;10;Bold;0;255,0,255;128,128,0"
END DIALOG

indexVar% = 1
editText$="Hello, World!"
PopupDialogBox(DlgDialog)
```

The changes we've done in this sample allowed us to use metafile picture as a dialog box's background and to use different fonts and colors for dialog controls.

"Hello, World!" with multimedia.

Do you think we've ran out of "Hello, World" samples? No! Here comes yet another one-liner:

Sample p18.sb Run

```
PlaySound("hellow.wav")
```

What else can we say about this sample? Let's move on and make it show something.

Sample p19.sb Run

```
DIM rc As RECT
GLOBAL wAvi%
DECLARE Sub OnPlay()
DECLARE SUB OnExit()

SBV_BORDER = 10
SBV_BORDERSTYLE = SBC_BORDERCOLOR
SBV_DARKCOLOR = RGB&(192,192,192)
wAvi% = PopupAvi( SBC_CENTER, SBC_CENTER, 0, 0, "hellow.avi",\
                  SBC_FIXEDPOS or SBC_MODELESS or SBC_NOPLAY or SBC_TOPMOST)

ON EVENT SBEP_DLGCONT OnPlay
ON EVENT SBEP_DLGSTOP OnExit

SBV_CONTTEXT = "&Play"
SBV_STOPTEXT = "&Exit"

RectWin(wAvi%, rc)
wText% = PopupTextBox(rc.left, rc.bottom, rc.right-rc.left, 90,\
            "Click Play to watch the movie", SBC_MODELESS or SBC_FIXEDPOS\
            or SBC_3D or SBC_ALIGNCENTER or SBC_TOPMOST)
sleep 0

static SUB OnPlay()
    PopupChangeAvi(wAvi%, SBC_FIRST)
    PopupChangeAvi(wAvi%, SBC_PLAY)
End Sub

static SUB OnExit()
    wakeup
End Sub
```

The above sample is the most complicated among those we've written so far. Not because it is using the new functions that allow to work with AVI files, but rather because it is introducing the concepts of events. ShowBasic events allow your program to react asynchronously to user interactions with the dialog boxes. Events are also used in some other cases (notably with the group of *Wait...()* functions), but we will concentrate on the dialogs here.

Let's recall that dialog boxes (and popup windows) can be shown in two modes: MODAL and MODELESS. When the dialog box is shown in modal state, ShowBasic delays the execution of your program until user clicks OK or CANCEL buttons (they can be named differently, of course, like Continue and Stop, for example). Only after that ShowBasic program continues. With MODELESS dialogs and windows the things work differently. Immediately after creation of the modeless dialog, ShowBasic continues to execute your program until it reaches the end of the program or execution is temporarily delayed somehow. So how to work with the modeless dialogs? Using the events, of course. Events allow you to execute a ShowBasic procedure (defined by you) when user clicks any control on a dialog. Now let's look at our sample and see how it works.

First of all, look how two subroutines declared and defined in our program:

```
DECLARE Sub OnPlay()
DECLARE SUB OnExit()
. . . . . . . . . .
static SUB OnPlay()
    PopupChangeAvi(wAvi%, SBC_FIRST)
    PopupChangeAvi(wAvi%, SBC_PLAY)
End Sub

static SUB OnExit()
    wakeup
End Sub
```

Subroutines and functions can be used both with events and simply as means of structuring your program - and you probably know already how they work. The regular subroutines and functions can be called from anywhere inside your program. In our case, we use subroutines as the event handlers, i.e. those procedures that are called automatically when some event is triggered. We need to tell ShowBasic on which event to call those subroutines - here is how we do it:

```
ON EVENT SBEP_DLGCONT OnPlay
ON EVENT SBEP_DLGSTOP OnExit
```

From now on whenever *SBEP_DLGCONT* event happens, the *OnPlay()* subroutine will be called, and whenever *SBEP_DLGSTOP* event happens, the *OnExit()* subroutine will be called. The two events we just mentioned are system events predefined in ShowBasic. The first one is automatically triggered when OK (Continue) button is clicked on any dialog box, and the second one is triggered when CANCEL (Stop) button is clicked. In our sample here we use just these two system events, but ShowBasic allows you to define your own events and to connect them to any control on your custom dialog box - therefore you'll be able to react an any state change of any dialog box control, and even to any Windows message processed by a dialog box. Although we never use it in our sample, it is worth mentioning that you can also disconnect your event handlers from processing the events at some other point of your program. Here is how you'd do it:

```
ON EVENT SBEP_DLGCONT
ON EVENT SBEP_DLGSTOP
```

Now let's talk about one more useful ShowBasic statement:

```
sleep 0
```

This statement stops the processing of our program forever. (The parameter passed to *sleep* is the number of seconds to delay the execution of the program, but 0 is the special case - it means "forever"). So what is the use to stop our program forever, we definitely need a way to finish it sooner or later! That is where one more statement comes to the rescue:

```
wakeup
```

This statement simply tells ShowBasic to continue execute our program if it is currently in a "sleeping" state. You make ask, but how can we execute *wakeup* when our program is sleeping? Of course, inside the event handler. The event handlers are called asynchronously, and even though the main program execution is halted by *sleep 0*, the event handler will be called and executed when the correspondent event triggers.

Now, with all this knowledge about the events, you can figure out how our sample works, let us just comment on some new functions that are used there.

PopupAvi() function allows to create a popup window capable of playing an AVI animation inside it. This window will play the associated AVI file automatically upon creation, unless the special *SBC_NOPLAY* flag is used.

PopupChangeAvi() function allows to control the AVI movie inside the previously created AVI-window. Using this function, you can start/stop playing the movie, or to control what exactly frame is shown in the window.

RectWin() function allows to find the coordinates and the dimension of the rectangle occupied by some window on the screen. We use this function to place our dialog textbox exactly below the window with the AVI movie.

"Hello, World!" using Windows API.

Our next sample is somewhat artificial, but nevertheless it demonstrates that the full power of Windows API is in your hands when you program in ShowBasic.

Sample p20.sb Run

```
declare function SendMessage lib "User" (hwnd as integer, w as integer,\
                              wParam as integer, lp as any) as integer
declare function IsWindow    lib "User" (hwnd as integer) as integer

DIM sI As SYSTEMINFO
DIM rc As Rect

WinInfoSystem(sI)

Run("notepad.exe", NOWAIT)
wNotepad% = WinActivate("", "Notepad",5)
RectSet(rc, sI.si_wXScreen/2-200, sI.si_wYScreen/2-60,\
            sI.si_wXScreen/2+200, sI.si_wYScreen/2+60)
WinSetPosSizeState(wNotepad%, rc, SBC_WINPOSSIZE or SBC_WINTOPMOST)
SBV_PLAYTICK = 0
Simulate("Close Notepad to stop this madness.")
While IsWindow(wNotepad%)
    SendMessage(wNotepad%, WM_SETTEXT, 0, "Hello,")
    sleep 1
    If IsWindow(wNotepad%) then
        SendMessage(wNotepad%, WM_SETTEXT, 0, "World!")
        sleep 1
    EndIf
    If IsWindow(wNotepad%) then
        SendMessage(wNotepad%, WM_SETTEXT, 0, "Hello, World!")
        sleep 1
    EndIf
Wend
```

The above sample starts Notepad and continuously changes its window's caption until is closed. We use two Windows API functions here: *IsWindow()* and *SendMessage()*. You can use any API functions in ShowBasic using its ability to call functions in any 16-bit or 32-bit external DLL. The whole Windows API is just a set of functions in several system DLLs - that's why you can use it. The declarations of all API functions, structures and constants can be found in *winapi.inc* file supplied with ShowBasic, however it is not always reasonable to include the whole file into the program using '$INCLUDE statement - this will slow down the compilation speed. That's why we simply extracted the required declarations from *winapi.inc* and included them at the beginning of our sample.