maXbox Starter 16

Start with Event Driven Programming

1.1 Set an Event

Today we spend another small time in programming with a new concept called events. Most modern programming languages (like PHP, Python or Ruby) are dynamically typed, event driven and employ an interpreter for program execution.

Event driven programming are usually message based languages offering the user an interactive dialog allowing commands that are interpreted and immediately executed like a button click see below.

We will concentrate for the first on creating an API call of a DLL using `PlaySound`() from the `winmm.dll` library as the result of a asynchronous timer event. But, first of all I'll explain you what "synchronous" and "asynchronous" events are. In fact there are 2 programming models used in event driven applications:

- Synchronous events
- Asynchronous events

Some event handlers are called immediately when the event occurs, otherwise you have to wait. These are called 'synchronous' events. An example is `CloseButtonClick()`. It gets called as soon as the user hits the close button.

However, some events are called or executed after the event occurs, usually after a short amount of idle time. With these asynchronous events you don't have to wait, so the caller continues without pausing for a response!

All the windows and dialog boxes in your application are based on the class `TForm`.
A VCL form contains the description of the properties of the form and the components it owns. Each form file or a form code section represents a single form, which usually corresponds to a window or dialog box in an application and are able to handle events.
Every Delphi application must have or provide a main form. The main form is the first form created in the body of the application. When the main form closes, the application terminates.

```
procedure CloseButtonClick(Sender: TObject);
begin
  frmMon.Close;
end;
```

So what's event driven programming?
Event-based programs are organized around the processing of events. When a program cannot complete an operation immediately because it has to wait for an event (e.g., the arrival of a packet or

the completion of a disk store transfer), it registers a callback; a function that will be invoked when the event occurs. Event-based programs are typically driven by a loop that polls for events and executes the appropriate callback when the event occurs.
This is the famous message-loop (waiting for events) in the window model, see below chart2:

```
while GetMessage(lpMsg,0,0,0) do begin
    TranslateMessage(lpMsg);
    DispatchMessage(lpMsg);
  end;
```

A callback executes indivisibly until it hits a blocking operation, at which point it registers a new callback and then returns. So in our close event, how does the callback look like?

```
   onClick:= @CloseButtonClick;
   onClose:= @CloseFormClick;
```

As said above, we register a callback `@CloseButtonClick` with the address operator, so the callback function will be executed when the event `onClick` occurs, in our case the button is clicked!

The interesting point is the separation of event and event handler (callback routine).

An event handler is a callback routine that operates asynchronously and handles inputs received into a program (events). In this context, an event is some meaningful element of program information from an underlying development framework, usually from a graphical user interface (GUI) toolkit or some kind of input routine. On the GUI side, events include key strokes, mouse activity, action selections, messages or timer expirations. On the input side, events include opening or closing files or forms and data streams, reading data and so on.

Now we come to the first API event.
Hope you did already work with the Starter 1 to 15 available at:

http://www.softwareschule.ch/maxbox.htm

Remember: What makes the event synchronous or asynchronous is whether it's handled synchronously (blocking) or asynchronously (non-blocking.)
Non Blocking means that the application will not be blocked when the application plays a sound or a socket read/write data. This is efficient, because your application don't have to wait for a sound result or a connection. Unfortunately, using this technique is little complicated to develop a protocol. If your protocol has many commands or calls in a sequence, your code will be very unintelligible.
A synchronous event is an event that – when fired - waits on a response from something called an event handler.

A function which returns no data (has no result value) can always be called asynchronously, cause we don't't' have to wait for a result or there's no need to synchronise the functions.

Let's begin with the application structure process in general of an API call:
The `PlaySound` function plays a sound specified by the given file name, resource, or system event. (A system event may be associated with a sound in the registry or in the WIN.INI file.)

1. Header: Declared in `msystem`.h; include `Windows.h`.
2. Library: Use `Winmm.dll` or the lib.
3. Declaration of the external function
4. Load the DLL and call the API function
    - Static at start time
    - Dynamic at run time
5. Unload the DLL (loaded DLL: `C:\WINDOWS\system32\winmm.dll`)

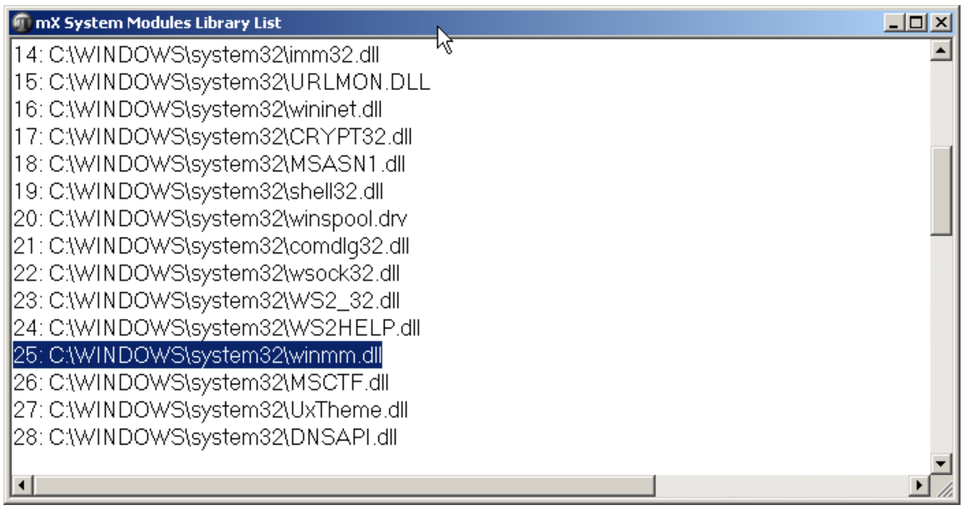
So we call the API function dynamic at runtime and of course asynchronously. The sound is played asynchronously and `PlaySound` returns immediately after beginning the sound. To terminate an asynchronously played waveform sound, call `PlaySound` with `pszSound` set to NULL.

The API call to the function just works fine, doesn't have any glitch.

☞ On the other side a sound is played synchronously, and `PlaySound` returns after the sound event completes. This is the default behaviour in case you have a list of songs.

If you click on the menu `<Debug/Modules Count>` you can see all the libraries (modules) loaded by your app and of course also the `winmm.dll` with our function `PlaySound()` in it.

```
25 function BOOL PlaySound(
  LPCTSTR pszSound,
  HMODULE hmod,
  DWORD fdwSound
);
```



```
14: C:\WINDOWS\system32\imm32.dll
15: C:\WINDOWS\system32\URLMON.DLL
16: C:\WINDOWS\system32\wininet.dll
17: C:\WINDOWS\system32\CRYPT32.dll
18: C:\WINDOWS\system32\MSASN1.dll
19: C:\WINDOWS\system32\shell32.dll
20: C:\WINDOWS\system32\winspool.drv
21: C:\WINDOWS\system32\comdlg32.dll
22: C:\WINDOWS\system32\wsock32.dll
23: C:\WINDOWS\system32\WS2_32.dll
24: C:\WINDOWS\system32\WS2HELP.dll
25: C:\WINDOWS\system32\winmm.dll
26: C:\WINDOWS\system32\MSCTF.dll
27: C:\WINDOWS\system32\UxTheme.dll
28: C:\WINDOWS\system32\DNSAPI.dll
```

1: Result as of the loaded Modules

1.2 Code Event handling with an API Call

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window shell like a song length protocol result at the bottom.
Included in the package maXbox is the integrated MP3 Songs utility player as an add-on listed on the menu `/Options/Add Ons/`. But we show the tool as a script extract with source, an add-on is a real compiled and integrated script.

There is a special function in Win32 which plays .wav files. You do not need the full Windows Media Control Interface. You can either play the sound by mentioning its filename (in which case the file needs to be distributed with the program) or by making the file into a resource with a line like

```
PlaySound("SoundName", hInst, SND_RESOURCE | SND_ASYNC);
```

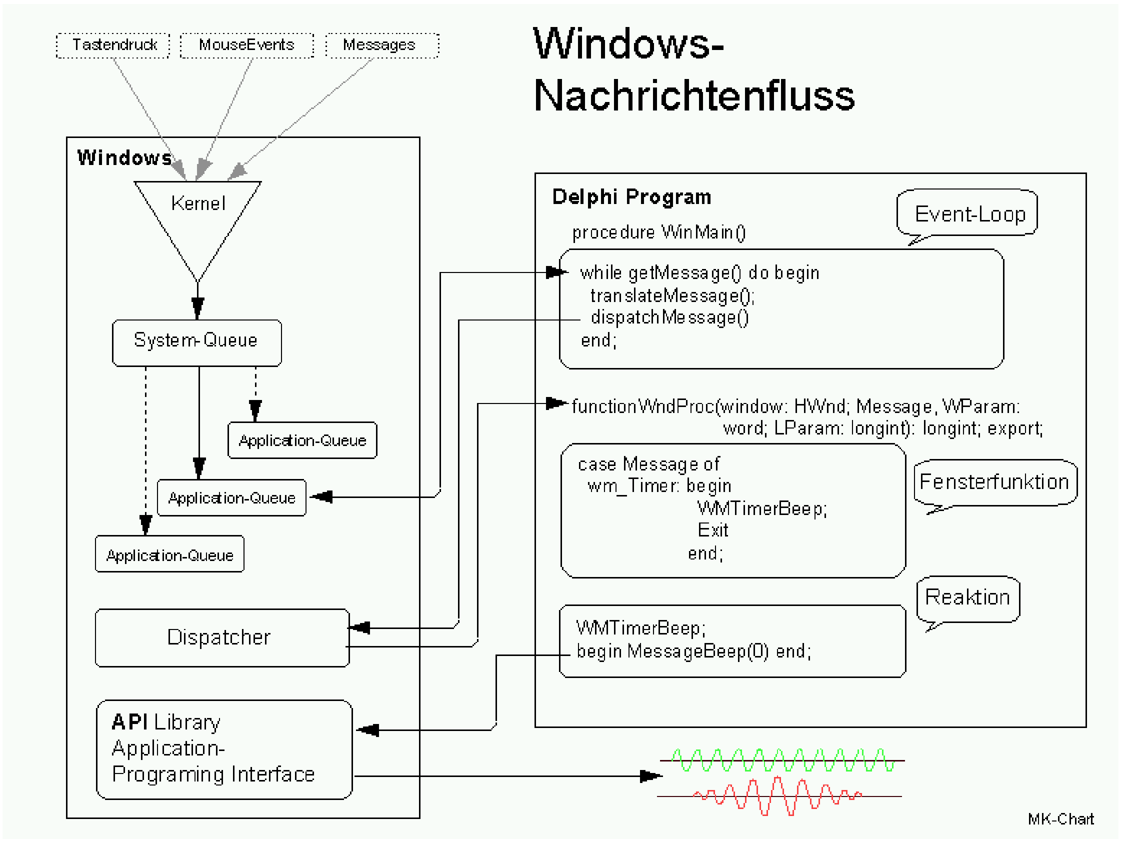
The following example stops playback of a sound that is playing asynchronously:

```
PlaySound(NULL, 0, 0);
```

If you set the flag `SND_NOSTOP` we get an interesting constellation.
This specified sound event will yield to another sound event that is already playing. If a sound cannot be played because the resource needed to generate that sound is busy playing another sound, the function immediately returns FALSE without playing the requested sound.
If this flag is not specified, `PlaySound` attempts to stop the currently playing sound so that the device can be used to play the new sound (again).

Discussion:
I have an app that has a background sound and a sound when someone clicks something. But when they click the something sound it stops the background sound, which is supposed to loop. How can I play both sounds and leave the background looping?

Answer:
I'm 99% sure that the `PlaySound` API (which `SoundPlayer.Play` is wrapping) does not support this no matter how you do it. You will have to look at something like DirectX or multithreading if you really want to do this.
I'm trying to get the `SoundPlayer` to play two wav files at the same time. The documentation says is should play asynchronously but when I try, it seems to end the play of the first `SoundPlayer` as soon as the second one starts to play.
I read the documentation, and it doesn't imply that it will merge the two sounds, merely that the method call is asynchronous. Internally `PlaySound` calls the `PlaySound` API from `mmsystem.h`, with the parameters `SND_ASYNC` and `SND_NODEFAULT`. This means so far that "The sound is played asynchronously and `PlaySound` returns immediately after beginning the sound..." and "No default sound event is used. If the sound cannot be found, `PlaySound` returns silently without playing the default sound."



```
procedure WinMain()
while getMessage() do begin
  translateMessage();
  dispatchMessage()
end;

function WndProc(window: HWnd; Message, WParam:
               word; LParam: longint): longint; export;
case Message of
  wm_Timer: begin
              WMTimerBeep;
              Exit
            end;

WMTimerBeep;
begin MessageBeep(0) end;
```

Chart 2: Message Driven Events in Windows

In a Win32/64 application, we start a message loop that fetches messages from a queue, translates them and then dispatches them. Eventually, each message reaches our `WndProc` where the associated event can be handled in a case construct.
So the event is the timer which will be answered be the event handler `WMTimerBeep` and then executed by the `MessageBeep()` or `PlaySound()`.
The OS maintains a message or system-queue, where it puts the events (e.g., from interrupts or other sources). It then sends the messages from that queue to all windows, depending on the message (e.g., it won't send key messages to a window that doesn't have focus).

A message is posted to every window that should receive it. The OS decides depending on the type of message whether a window should receive that message or not.
Most messages are waited for by the system, i.e., the message won't get posted to another window until it was processed by the window. This has a great impact for broadcast messages: if one window doesn't return when handling that message, the queue is blocked and other windows won't receive the message anymore.

So, as far as I can see it implies that the API call queues up the sounds to be played, and the method returns when your sound starts playing, so there's no parallel processing possible!

So after this hard low-level introduction we jump to our app, a small form which counts the characters of a text file and deepen our knowledge with events.

Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from http://sourceforge.net/projects/maxbox site. Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox3.exe` the box opens a default program. Make sure the version is at least 3.8 because `Modules Count` use that. Test it with F2 / F9 or press **Compile** and you should hear a sound and a browser will open. So far so good now we'll open the example:

```
287_eventhandling2.txt (20888 kb)
Or 263_async_sound.txt
```

If you can't find the file use one of those links:
http://www.softwareschule.ch/examples/287_eventhandling2.txt
http://www.softwareschule.ch/examples/287_eventhandling2.htm

Or you use the `Save Page as…` function of your browser[1] and load it from `examples` (or wherever you stored it). One important thing: The mentioned DLL must reside on your harddisk. Now let's take a look at the code of this project first with the declaration of the 2 forms. Our first line is

```
1 Program EventHandler_Tutorial;
```

We have to name the program it's called `EventHandler_Tutorial.` Next we jump to line 19:

```
16 var
17    Exc2: Exception;
18    Image1, Image2: TImage;
19    frmMon, frmMon2: TForm;
```

First in line 19 we ask to set 2 form variables. Forms are important for all IT systems that offer interfaces for user or programming access. Particularly, this includes machines in the private and commercial sector as well as services like electronics or multimedia. In these cases the `var` declaration is usually done by entering a form name and the according class name.

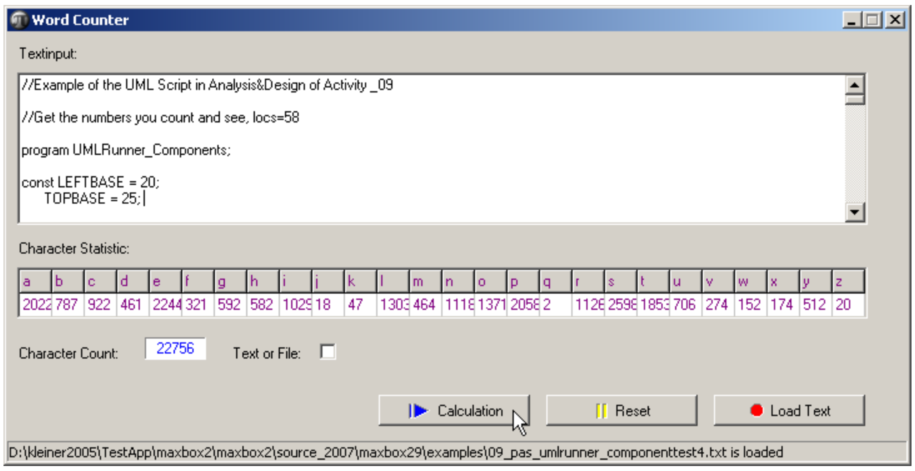
As we now know, `TForm` is able to handle the events as a window function. The Windows API function used to create parent (top-level) and child windows is called `CreateWindow`.
The event handler method must accept four parameters: `hWnd`, the handle for the window that received the message; `nMessage`, the Windows message number; and two Integer parameters, the contents of which vary depending on the Windows message (the documentation for each message describes the values of these parameters).

1.3 Event and Event Handler in a Form

Typically, you create forms for your application from within the IDE with a form designer. In maXbox we can code directly and native in the script your form with controls. When created this way, the forms have a runtime constructor that takes one argument, `Owner`, which is the owner of the form being created. (The owner is the calling application object or form object.) Owner can be NIL.

[1] Or copy & paste

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The owner of a component is determined by the parameter passed to the constructor when the component is created. But there are three or four possibilities passing one: `Application`, `Self`, `NIL` or a `Owner` by Yourself



Textinput:

```
//Example of the UML Script in Analysis&Design of Activity _09

//Get the numbers you count and see, locs=58

program UMLRunner_Components;

const LEFTBASE = 20;
    TOPBASE = 25;
```

Character Statistic:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
2022	787	922	461	2244	321	592	582	1029	18	47	1303	464	1118	1371	2058	2	1126	2598	1853	706	274	152	174	512	20

Character Count: 22756 Text or File: ☐

D:\kleiner2005\TestApp\maxbox2\maxbox2\source_2007\maxbox29\examples\09_pas_umlrunner_componenttest4.txt is loaded

3: Our App with x events on a second form

By default, a form owns all components that are on it. In turn, the form is owned by the application. Thus when the application shuts down and its memory is freed, the memory for all forms (and all their owned components) is also freed.

Hint: Owner is a property of `TComponent` so you don't have to declare it when passing to the Constructor. In our case when dynamically creating one or more components as part of a form, the owner has to be the form itself, so you refer to the form via the self pointer in line 336 and a reference pointer by yourself (`inFrm`).

```
335 begin
336  frmMon:= TForm.Create(self);
348  Image1:= TImage.create(frmMon);
356  Image2:= TImage.create(frmMon);
```

Next we step to the concept of event handlers. Once you determine when the event occurs (later in the main program), you must define how you want the event handled.
Specifying a second event handler object in line 345 or delegate method causes the first binding to be added with the second, simply means the form can handle 2 events:

```
337 with frmMon do begin
338   //FormStyle := fsStayOnTop;
339   Position := poScreenCenter;
340   caption:='Pulsar Universe BitmaX';
341   color:= clblack;
342   height:= 700;
343   width:= 850;
344   Show;
345   onMousedown:= @Image1MouseDown;
346   onClose:= @CloseFormClick; //sync with form2
347 end;
```

`TForm` declares and defines a method `CloseFormClick` which will be invoked at runtime whenever the user presses close button **x** on the window or <Alt><F4>. This procedure is called an event handler (line 346) because it responds to events that occur while the program is running. The `onClose` calls the event handler, previously registered.

```
318 procedure CloseFormClick(Sender: TObject; var Action: TCloseAction);
319 begin
320   Image1.Free; //myImage.Free;
321   Image2.Free;
322   frmMon2.Free;
323   frmMon.Free;
324   frmMon:= NIL
325   Writeln('char counter form has been closed at: '+ TimeToStr(Time));
326 end;
```

Free (Destroy) deactivates the form object by setting Enabled to False before freeing the resources required by the form. The event handler CloseFormClick in the example deletes the 2 forms after it is closed, so the form would need to be recreated if you needed to use a form elsewhere in the application.
If the form were displayed using Show you could not close the form within the event handler because Show returns while the form is still open. So it's better to work with another event handler ButtonCloseClick() which calls with a method inFrm.Close also the above centralized CloseFormClick:

```
328 procedure CloseButtonClick(Sender: TObject);
329 begin
330   frmMon.Close; //calls the onClose eventhandler!
331 end;
```

That means: onClick →CloseButtonClick() → frmMon.Close→onClose → CloseFormClick()

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's Top, Left, Width, and Height properties. You can change these values also with SetBounds in line 458 at runtime to change the position and size of the controls in your forms.
Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

```
455 with panB do begin
456    caption:= '***Outline***';
457    Parent:= frmMon2;
458    SetBounds(LEFTBASE,TOPBASE+40,340,400)
```

Next event we take a look at is the main calculation procedure fired by an OnClick event behind a TBitBtn, declared in the form building, see picture 3 on the form:

```
432    with TBitBtn.Create(frmMon2) do begin
433      parent:= frmMon2;
434      SetBounds(297,290,121,30)
435      Caption:= 'Calculation'
436      glyph.LoadFromResourceName(getHINSTANCE,'CL_MPSTEP');
437      OnClick:= @btnCalcCharsClick
438    end;
```

Remember, the event handler btnCalcCharsClick is invoked by the event.

```
procedure btnCalcCharsClick(Sender: TObject);
```

```
//report char counter distribution
```

But wait a minute: As we have seen Windows is a message-based operating system. System messages are handled by a message handler that translates the message to Delphi event handlers. For instance, when a user clicks a button on a form, Windows sends a message to the application and the application reacts to this new event. If the `OnClick` event for that button `TBitBtn.Create (frmMon2)` is specified it gets executed.
The code to respond to events is contained in Delphi event procedures (event handlers). All components have a set of events that they can react on. For example, all clickable components have an `OnClick` event that gets fired if a user clicks a component with a mouse.
But the procedure is too tight behind the event; it's more flexible to realize a loose coupling between event and event handler by a small footprint like a control layer:

```
procedure btnCalcCharsClick(Sender: TObject);
 begin
   CharsCalculation; //calls the procedure indirect!
 end;
```

The article “How to Detach an Event Handler from a Control Event” describes a simple mechanism to remove an event handling procedure from a control's event.

This form example requires no local variables in the method `Procedure InitFirstForm` / `Init Second Form` by many different classes `TMainMenu`, `TMenuItem`, `TPanel` and `TBitmap`. Because the most controls are building at runtime on behalf of the `with` statement:

```
with TLabel.Create(frmMon2) do begin
  parent:= frmMon2;
  setBounds(9,250,116,13)
  Caption:= 'Character Count:';
end;
```

Next Question will be to show the possibility to share the same event in different controls or in different event handlers. Associating an event with an existing event handler in purpose of code reuse. Suppose we want to have a button and a menu item do the same thing or in our case a form and an Image to do the same thing by the event `onMouseDown`.

```
// of frmMon
   onMousedown:= @Image1MouseDown;

// of Image1
   onMousedown:= @Image1MouseDown;
```

Delphi and other languages allows one coded event to be assigned to many event handlers as long as the methods are of the same type - means, they have the same arguments (eg. sender, btn, shift,x,y in mousedown). The classic example is the `OnClick` event which is shared by many components.

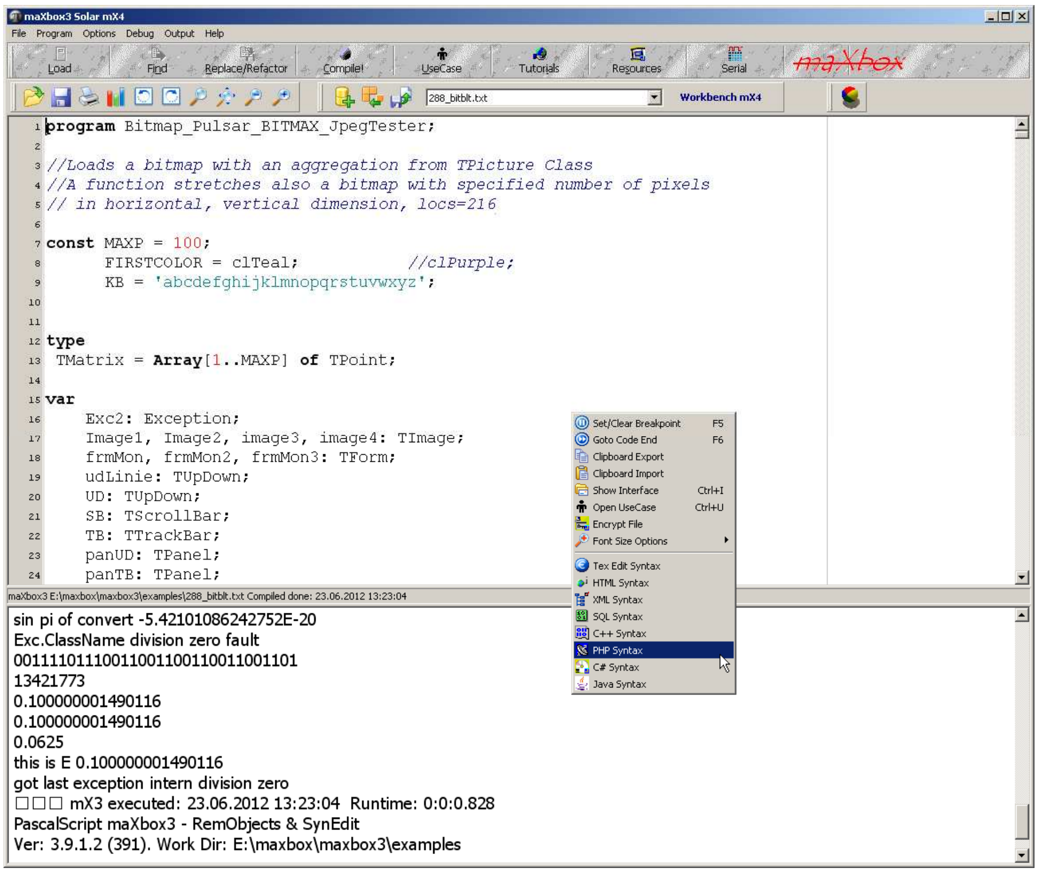
```
procedure Image1MouseDown(Sender: TObject; Btn: TMouseButton;
                            Shift: TShiftState; X,Y: Integer);
var i: byte;
begin
//______________
End.
```

You can then assign this procedure to as many event handlers as you wish.

```
Procedure AssignEvents;
Begin
```

```
MyButton.OnClick:= ClickHandlerComponent;
MyLabel.OnClick:= ClickHandlerComponent;
MyCombo.Onclick:= ClickHandlerComponent;  // You Get the message!
```

Each time one of these components is clicked, the above procedure `ClickHandlerComponent` is executed. This can be useful is certain circumstances, but mainly in this tip it serves to illustrate the main point - reduce design time coding by assigning events.
Also an area where you do have many controls sharing code is in respect to Main Menus, Popup Menus and Speed buttons.



```
program Bitmap_Pulsar_BITMAX_JpegTester;

//Loads a bitmap with an aggregation from TPicture Class
//A function stretches also a bitmap with specified number of pixels
// in horizontal, vertical dimension, locs=216

const MAXP = 100;
      FIRSTCOLOR = clTeal;           //clPurple;
      KB = 'abcdefghijklmnopqrstuvwxyz';


type
 TMatrix = Array[1..MAXP] of TPoint;

var
    Exc2: Exception;
    Image1, Image2, image3, image4: TImage;
    frmMon, frmMon2, frmMon3: TForm;
    udLinie: TUpDown;
    UD: TUpDown;
    SB: TScrollBar;
    TB: TTrackBar;
    panUD: TPanel;
    panTB: TPanel;
```

maXbox3 E:\maxbox\maxbox3\examples\288_bitblt.txt Compiled done: 23.06.2012 13:23:04

```
sin pi of convert -5.42101086242752E-20
Exc.ClassName division zero fault
00111101110011001100110011001101
13421773
0.100000001490116
0.100000001490116
0.0625
this is E 0.100000001490116
got last exception intern division zero
□□□ mX3 executed: 23.06.2012 13:23:04  Runtime: 0:0:0.828
PascalScript maXbox3 - RemObjects & SynEdit
Ver: 3.9.1.2 (391). Work Dir: E:\maxbox\maxbox3\examples
```

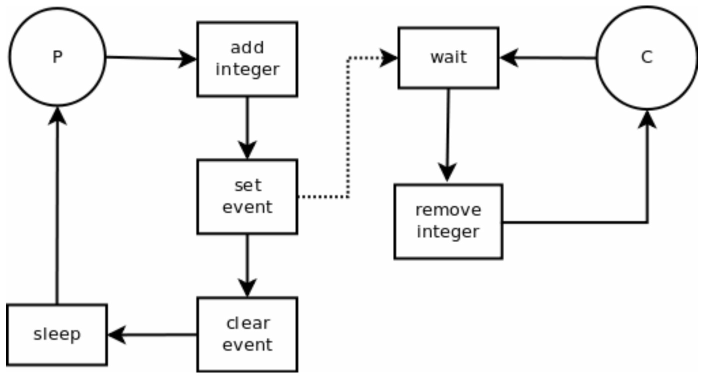
Till now we are discussing the topics of `sync` and `async` calls / events and the way it processes the calls by the receiver or event handler parallel or not. Asynchronous calls or connections allow your app to continue processing without waiting for the process (function) to be completely closed but not always in a simultaneous mode. Therefore we come now to the discussion about real parallel programming with threads (multithreading) or multiprocessing programming.

1.4 Notes about Events or Threads

Events are a better means of managing I/O concurrency in server software than threads: events help avoid bugs caused by the unnecessary CPU concurrency introduced by threads. Event-based programs also tend to have more stable performance under heavy load than threaded programs.
We argue that our libasync non-blocking I/O library makes event-based programming convenient and evaluate extensions to the library that allow event-based programs to take advantage of multi-processors.
We conclude that events provide all the benefits of threads, with substantially less complexity; the result is more robust software. The debate over whether threads or events are best suited to systems software has been raging for decades.
Threaded programs can be structured as a single flow of control, using standard linguistic constructs such as loops across blocking calls. Event driven programs, in contrast, require a series of small callback functions, one for each blocking operation. Any stackallocated variables disappear across

callbacks. Thus, event driven programs rely heavily on dynamic memory allocation and are more prone to memory errors in low-level languages such as C and C++.

Types in Events: Because it is not necessary to declare types of variables and formal arguments of subroutines, these languages are often called type less. This approach can be better characterized as “dynamically typed”, because each value in these languages is associated with a data type.
In many articles, they describe the advantages of interpreted, dynamically typed languages, yet one should be aware of two serious drawbacks. Firstly, these languages usually perform semantic checks (e.g., for the existence of identifiers) when the code is actually executed which leaves many possibilities for typing errors to be undetected.
Secondly, type checks for each operation must be performed at run-time.
This must be done not only once, but every time an instruction is executed. When performing simple operations like the addition of two real numbers, most of the time for the operation will be spent on verifying the types of the arguments, even when it is clear that — because of the design of the algorithm — no other types may occur. While the additional overhead is negligible for interactive execution of commands typed on a command line, it has a serious impact on routines that require intensive calculations.



For ex. thread-1 P appends a number to a list and then set the event to notify the consumer. The consumer’s call to `wait()` stops blocking and the integer is retrieved from the list.

- Avoiding bottlenecks. With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- Organizing program behaviour. Often, a program's run can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases.
- Multiprocessing. If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

One word concerning a processing thread: In the internal architecture there are 2 threads categories.

- Threads with synchronisation (blocking at the end)
- Threads without synchronisation (non blocking at all)

In case you’re new to the concept, a thread is basically a very beneficial alternative (in most cases) to spawning a new process. Programs you use every day make great use of threads whether you know it or not but you have to program it.
The most basic example is whenever you’re using a program that has a lengthy task to achieve (say, downloading a file or backing up a database), the program (on the server) will most likely spawn a thread to do that job.
This is due to the fact that if a thread was not created, the main thread (where your main function is running) would be stuck waiting for the event to complete, and the screen would freeze.

For example first is a listener thread that “listens” and waits for a connection. So we don't have to worry about threads, the built in thread will be served by for example Indy though parameter:

```
   IdTCPServer1Execute(AThread: TIdPeerThread)
```

When our DWS-client is connected, these threads transfer all the communication operations to another thread. This technique is very efficient because your client application will be able to connect any time, even if there are many different connections to the server.
The second command "`CTR_FILE`" transfers the app to the client:

Or another example is AES cryptography which is used to exchange encrypted data with other users in a parallel way but not a parallel function. It does not contain functions for key management. The keys have to be exchanged between the users on a secure parallel channel. In our case we just use a secure password!

```
procedure EncryptMediaAES(sender: TObject);


 with TStopwatch.Create do begin
    Start;
    AESSymetricExecute(selectFile, selectFile+'_encrypt',AESpassw);
    mpan.font.color:= clblue;
    mpan.font.size:= 30;
    mpan.caption:= 'File Encrypted!';
    Screen.Cursor:= crDefault;
    Stop;
    clstBox.Items.Add('Time consuming: ' +GetValueStr +' of: '+
          inttoStr(getFileSize(selectFile))+' File Size');
    Free;
 end;
```

And that’s how to get a feeling of the speed of AES with a protocol extract:

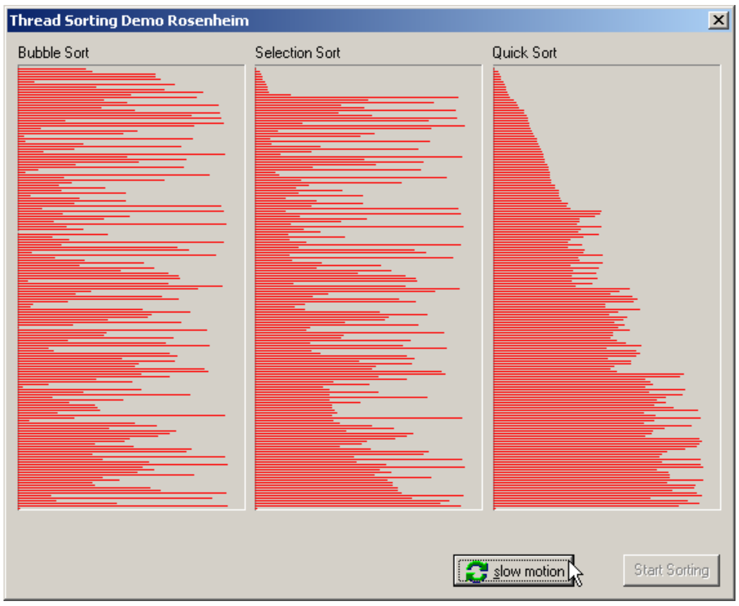
```
Use native.AES-256 cipher and native.CBC chaining mode.
Ciphertext size = 21726 bytes.
Decryption succeeded. 21726 bytes processed.
Speed of decryption was 1326 KiB per second.  //4 GByte about 40 Minutes
```

✌To understand threads one must think of several programs running at once. Imagine further that all these programs have access to the same set of global variables and function calls.

Each of these programs would represent a thread of execution and is thus called a thread. The important differentiation is that each thread does not have to wait for any other thread to proceed. If they have to wait we must use a synchronize mechanism.

All the threads proceed simultaneously.
To use a metaphor, they are like runners in a race, no runner waits for another runner. They all proceed at their own rate.

☞ Because Synchronize uses a form message loop, it does not work in console applications. For console applications use other mechanisms, such as a mutex (see graph below) or critical sections, to protect access to RTL or VCL objects.



6: 3 Threads at work with Log

The point is you can combine asynchronous calls with threads! For example asynchronous data fetches or command execution does not block a current thread of execution.
But then your function or object has to be thread safe. So what's thread-safe. Because multiple clients can access for example your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables.

Tasks for advanced studies:

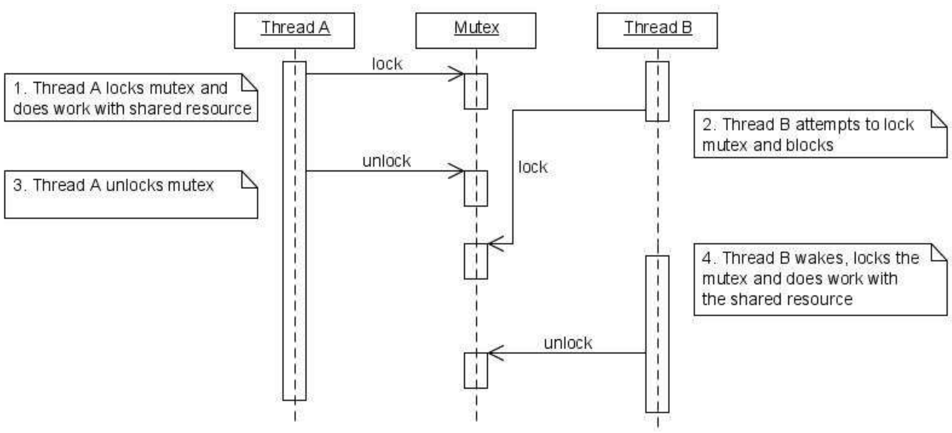
How can you crack a password with a massive parallel concept? Study all about a salt. A 512-bit salt is used by password derived keys, which means there are `2^512` keys for each password. So a same password doesn't generate always a same key. This significantly decreases vulnerability to 'off-line' dictionary' attacks (pre-computing all keys for a dictionary is very difficult when a salt is used). The derived key is returned as a hex or base64 encoded string. The salt must be a string that uses the same encoding.

We also use a lot of more multi scripts to teach and the wish to enhance it with a thread, simply take a look in the meantime at `141_thread.txt, 210_public_private_cryptosystem.txt` and `138_sorting_swap_search2.txt.` At least let's say a few words about massive threads and parallel programming and what functions they perform.

Do not create too many threads in your apps. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.
For events, in Win32 Delphi OOP, only one method can be assigned as a handler to a component's event! But you can build a Win32 Delphi object that maintains a list of the methods its event(s) is handled by - thus creating a simulation of multicast event handling.

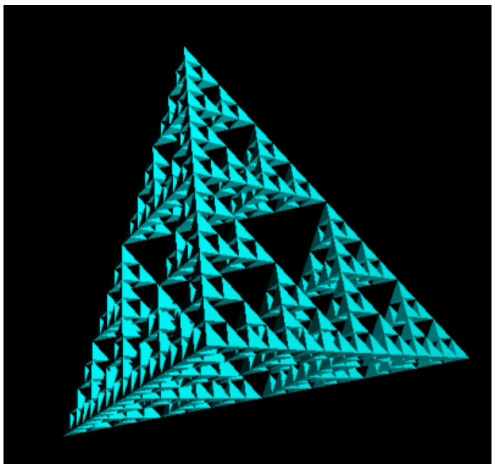
Short a design study: First, we'll define a simple class (TMultiEventClass) with only one event of TMultiEvent type. In order to store a list of methods that will be called when event is fired (using the FireMultiEvent) we'll use a fMultiEventHandlers `TList` object. The `TList` Delphi class is designed to be used as a storage mechanism for an array of pointers. By design, each method (function or procedure) in Delphi code can be represented with a `TMethod` Delphi type.


Thread A
Mutex
Thread B
lock
1. Thread A locks mutex and does work with shared resource
2. Thread B attempts to lock mutex and blocks
unlock
lock
3. Thread A unlocks mutex
4. Thread B wakes, locks the mutex and does work with the shared resource
unlock

7: A real Thread with a synchronisation object

One note about async execution with fork on Linux with libc-commands; there will be better solutions (execute and wait and so on) and we still work on it, so I'm curious about comments, therefore my aim is to publish improvements in a basic framework on sourceforge.net depends on your feedback ;)

Try to find out more about symmetric/asymmetric encryption systems and the probability to crack it. Due to their large computational requirements, a asymmetric key system is used in practice only for encrypting so called "session keys". The pay load is encrypted with a symmetric algorithm using the session key. This combination of two methods is called hybrid encryption (asymmetric/symmetric). This implies no security deficit if you use a good random number generator for the session key, like a HW generator below!



Try to change a second event handler `Image1MouseDown2` one for each event:

```
234 procedure Image1MouseDown(Sender: TObject; Btn: TMouseButton;
                                    Shift: TShiftState; X,Y:
```

Try to find out more about the callback function schema and the question how it works in a synchronous or asynchronous mode.

Check the source of LockBox to find out how a thread is used:

```
E:\maxbox\maxbox3\source\TPLockBoxrun\ciphers\uTPLb_AES.pas;
```

Try to add/change the event handler procedure `btnLoadClick`(Sender: TObject); with a second event with a message when the text is loaded:

```
procedure btnLoadClick(Sender: TObject);
//Set a new text file to count
var i: integer;
  selectedFile: string;
begin
 edText.Text:=''; panB.Caption:='';
 SBtnLoad:= true;
 chk.checked:= false;
```

max@kleiner.com
Links of maXbox and Asynchronous Threads of DelphiWebStart:

[2] 059_timerobject_starter2_ibz2_async.txt

[7] Jensen, K., and Wirth, N. (1974), PASCAL - User Manual and Report, Springer.

http://sourceforge.net/projects/delphiwebstart

Event-driven Programming for Robust Software:

http://pdos.csail.mit.edu/~rtm/papers/dabek:event.pdf

http://www.softwareschule.ch/maxbox.htm
http://sourceforge.net/projects/maxbox
http://sourceforge.net/apps/mediawiki/maxbox/

My Own Experience:
http://www.softwareschule.ch/download/armasuisse_components.pdf

SHA1 Hash of maXbox 3.9.1.2 Win: B8686418D5F31B618E595F26F8E74BBCD39AA839