

MagicMacros unit

The MagicMacros unit declares the TMagicMacros component, as well as a few exceptions and types that are used by the component.

The following items are declared in the MagicMacros unit:

Components

[TMagicMacros](#)

Exceptions

[ENoSuchMacro](#)

[ENoSuchVHFile](#)

Types

[TMacroState](#)

[TMacroStates](#)

[TPlaySpeed](#)

[TProgressEvent](#)

[TVHNotifyEvent](#)

ENoSuchMacro exception

Unit

[MagicMacros](#)

Description

This exception is raised by the [play method](#) if the macro indicated by the [MacroName property](#) does not exist or is not a valid TMagicMacros macro.

It can also be raised by the [PlayVH](#) method if the specified ID could not be found in the VisualHelp file specified in the [VHFilename](#) property.

ENoSuchVHFile exception

Unit

[MagicMacros](#)

Description

This exception is raised by the [PlayVH](#) method if the VisualHelp file specified by the [VHFilename](#) property does not exist or is not a valid TMagicMacros VisualHelp file.



TMagicMacros component

[Properties](#)

[Methods](#)

[Events](#)

[Tasks](#)

Unit

[MagicMacros](#)

Description

Use the TMagicMacros component to add full macro-support to your application in a snap! It features recording and playback of mouse- and keyboard events and comes with lots of events to ensure smooth integration in your app.

TMagicMacros features:


- [Recording](#) and [playing](#) of macro's.
- [Looping](#) of macro playback.
- 9 different [playback speeds](#), varying from 25% up till 500% of recorded speed.
- [Text-only](#) playback.
- Show or hide [mousemovements](#) during playback.
- [Automatically adapts playback speed](#) to the speed of the PC so playback always run smoothly.
- Show [infoboxes](#) to inform users.
- [Relative journaling](#): the macro automatically adapts to different positions of windows, buttons, toolbars, etc.
- Automatic adaption to different screenresolutions.
- Full [VisualHelp](#) file support, without writing one line of code!

As you may have noticed, the word "automatically" appears very often in the description above. This means TMagicMacros is very powerful, yet easy to use. Just set the desired options in the Object Inspector during designtime, implement some eventhandlers (optional) and add a minimum of code to your application and that's it! Full macro-support in a snap!

Although TMagicMacros is very powerful, there are a few [limitations](#) in what it can do.

Properties

▶ Run-time only

 Key properties

 LockCursor

 LoopPlayback

 MacroName

 PlaySpeed

 RelativeJournaling


 ShowMouseMovements









▶ State

 TextOnly

 VHFilename

Methods




 Key methods

-  [Create](#)
-  [Pause](#)
-  [Play](#)
-  [PlayVH](#)
-  [Resume](#)
-  [ShowInfoBox](#)
-  [StartRecording](#)
-  [StopRecording](#)
-  [WaitForInfoBox](#)



Events

Key events

BeforePlay

-  BeforeRecord
-  BeforeVisualHelp
-  OnCloseInfoBox
-  OnPlay
-  OnPlayCancelled
-  OnPlayEnded
-  OnProgress
-  OnRecord
-  OnRecordCancelled

OnRecordEnded

-  OnShowInfoBox
-  OnVisualHelp
-  OnVisualHelpCancelled
-  OnVisualHelpEnded



About the TMagicMacros component

[TMagicMacros reference](#)

Purpose

Use TMagicMacros to:

- Give the users of your app the possibility to record and playback their own macros to avoid doing the same thing over and over again.
- Make a demo of your application: just record a macro which clearly illustrates how your app works, add a few infoboxes with some extra explanation here and there and distribute a version of your program which automatically plays the macro on startup. As simple as that !
- Make [VisualHelp](#) files: don't tell people how to use your program, show them !

Tasks

- To record a macro, use the [StartRecording](#) method.
- To playback a macro, use the [Play](#) method.
- To pause macroplayback, use the [Pause](#) method.
- To resume macroplayback, use the [Resume](#) method.
- To adjust the playback speed, use the [PlaySpeed](#) property.
- To hide the mousecursor during playback, use the [ShowMouseMovements](#) property.
- To playback only keyboard events, use the [TextOnly](#) property.
- To loop macroplayback, use the [LoopPlayback](#) property.
- To confine the cursor to the bounds of your mainform during recording, use the [LockCursor](#) property.
- To make the macro adapt to different positions of forms, buttons, toolbars,... use the [RelativeJournaling](#) property.
- To generate [VisualHelp](#) files, use the build in component editor.
- To insert infoboxes in your macro, use the [ShowInfoBox](#) and [WaitForInfoBox](#) methods.

LockCursor property

Applies to

TMagicMacros component

Declaration

property LockCursor: boolean;

Description

If this property is set to True, the cursor is confined to the bounds of the mainform and minimisation of the mainform is disabled (stretching and maximisation is still possible) *during recording*.

You may wish to use this option if you want to record macro's that involve your app only or if you want to prevent that someone records a macro that interacts with other applications.

If macro's should be able to interact with other programs, you should set this property to False.

LoopPlayback property

Applies to

TMagicMacros component

Declaration

property LoopPlayback: boolean;

Description

Set this property to true if you want to continuously playback a macro. This only effects playback. Changing the value of this property *during* playback will *not* affect the current playback.

MacroName property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

```
property MacroName: string;
```

Description

Specifies the filename for the macro. Note that the extension of a MacroMagic-file is always .mcr, regardless of the extension you enter for the MacroName-property.

Important: this property must have a value before recording or playback can start !

See also

[VHFileName](#) property

OnCloseInfoBox event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnCloseInfoBox: TNotifyEvent;

Description

This event is fired *after* the InfoBox has been closed, after the [WaitForInfoBox](#) method returns.

See also

[OnShowInfoBox](#) Event

BeforePlay event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property BeforePlay: TNotifyEvent;

Description

This event is fired *before* playback of a macro is started by the [Play](#) method.

Note that if you're [looping](#) macro playback, this event is fired *every time* the macro starts !

See also

[OnPlay](#) Event

[OnPlayEnded](#) Event

[OnPlayCancelled](#) Event

OnPlay event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnPlay: TNotifyEvent;

Description

This event is fired *after* playback of a macro has started by the [Play](#) method.

Note that if you're [looping](#) macro playback, this event is fired *every time* the macro starts !

See also

[BeforePlay](#) Event

[OnPlayEnded](#) Event

[OnPlayCancelled](#) Event

OnPlayCancelled event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnPlayCancelled: TNotifyEvent;

Description

This event is fired *after* playback of a macro has ended in an abnormal way.

Abnormal termination of macro-playback can be caused by the following reasons:

- the user has pressed one of the system keycombinations (i.e. CTRL-ALT-DEL or CTRL-ESC)
- the user has pressed CTRL-BREAK, indicating that he wishes to stop macro-playback immediatly.
- a serious systemerror has occured and a system modal dialog appears (you know, the horrible blue textscreen saying that your system has become unstable)

Note that if you're [looping](#) macro playback, this event is fired *only once* and the loop *stops* !

See also

[BeforePlay](#) Event

[OnPlay](#) Event

[OnPlayEnded](#) Event

OnPlayEnded event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnPlayEnded: TNotifyEvent;

Description

This event is fired *after* playback of a macro has ended in a normal way (i.e. the macro has been played completed). If playback has been terminated in an abnormal way, the [OnPlayCancelled](#) event is fired instead.

Note that if you're [looping](#) macro playback, this event is fired *every time* the macro has been played completely.

See also

[BeforePlay](#) Event

[OnPlay](#) Event

[OnPlayCancelled](#) Event

OnProgress event

Applies to

TMagicMacros component

Declaration

property OnProgress: TProgressEvent;

Description

This event can be used to visualise the progress of loading and saving of a macro, or loading of a VisualHelp file. During these actions, the OnProgress event is fired several times. The value of PercentDone passes the percentage of the loading or saving that has been completed. You can use this value for instance to visualise the progress with a TProgressBar or a gauge.

BeforeRecord event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property BeforeRecord: TNotifyEvent;

Description

This event is fired *before* recording of a macro is started by calling the [StartRecording](#) method.

See also

[OnRecord](#) Event

[OnRecordEnded](#) Event

[OnRecordCancelled](#) Event

OnRecord event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnRecord: TNotifyEvent;

Description

This event is fired *after* recording of a macro has started by calling the [StartRecording](#) method.

See also

[BeforeRecord](#) Event

[OnRecordEnded](#) Event

[OnRecordCancelled](#) Event

OnRecordCancelled event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnRecordCancelled: TNotifyEvent;

Description

This event is fired *after* the recording of a macro has ended in an abnormal way .

Abnormal termination of recording can be caused by the following things:

- The user has pressed one of the system keycombinations (i.e. CTRL-ALT-DEL or CTRL-ESC)
- A serious systemerror has occured and a system modal dialog appears (you know, the horrible blue textscreen saying that your system has become unstable). Note that the macro is still saved !

See also

[BeforeRecord](#) Event

[OnRecord](#) Event

[OnRecordEnded](#) Event

OnRecordEnded event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnRecordEnded: TNotifyEvent;

Description

This event is fired *after* the recording of a macro has ended in a *normal* way (i.e. a call to StopRecording was made, the maximum recording capacity has been reached (a maximum 100.000 keyboard and mouse-events can be recorded in a single macro) or the user pressed CTRL-BREAK, indicating that he's finished recording.)

If recording has been terminated in an *abnormal* way, the [OnRecordCancelled](#) event is fired instead.

See also

[BeforeRecord](#) Event

[OnRecord](#) Event

[OnRecordCancelled](#) Event

OnShowInfoBox event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnShowInfoBox: TNotifyEvent;

Description

This event is fired *after* the InfoBox is displayed by a call to the [ShowInfoBox](#) method.

See also

[OnCloseInfoBox](#) Event

BeforeVisualHelp event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property BeforeVisualHelp: [TVHNotifyEvent](#);

Description

This event is fired *before* playback of a [VisualHelp](#) file is started.

See also

[OnVisualHelp](#) Event

[OnVisualHelpEnded](#) Event

[OnVisualHelpCancelled](#) Event

OnVisualHelp event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnVisualHelp: [TVHNotifyEvent](#);

Description

This event is fired *after* playback of a [VisualHelp](#) file has started.

See also

[BeforeVisualHelp](#) Event

[OnVisualHelpEnded](#) Event

[OnVisualHelpCancelled](#) Event

OnVisualHelpCancelled event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnVisualHelpCancelled: [TVHNotifyEvent](#);

Description

This event is fired *after* playback of a [VisualHelp](#) file has ended in an abnormal way.

Abnormal termination of VisualHelp-playback can be caused by the following things:

- The user has pressed one of the system keycombinations (i.e. CTRL-ALT-DEL or CTRL-ESC)
- The user has pressed CTRL-BREAK, indicating that he wishes to stop VisualHelp-playback.
- A serious systemerror has occurred and a system modal dialog appears (you know, the horrible blue textscreen saying that your system has become unstable)

See also

[BeforeVisualHelp](#) Event

[OnVisualHelp](#) Event

[OnVisualHelpEnded](#) Event

OnVisualHelpEnded event

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property OnVisualHelpEnded: [TVHNotifyEvent type](#);

Description

This event is fired *after* playback of a [VisualHelp](#) file has ended in a *normal* way (i.e. the VisualHelp file has been played back completely).

If playback is terminated in a *abnormal* way, the [OnVisualHelpCancelled](#) event is fired instead.

See also

[BeforeVisualHelp](#) Event

[OnVisualHelp](#) Event

[OnVisualHelpCancelled](#) Event

PlaySpeed property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property PlaySpeed: TPlaySpeed;

Description

You can assign a number between 1 and 9 to this property to determine if a macro should be played back faster or slower than recorded. A value of 5 means normal speed.

See [TPlaySpeed](#) type for the interpretation of the different values.

See also

[TPlaySpeed](#) type

RelativeJournaling property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property RelativeJournaling: boolean;

Description

Set this property to true if you want to perform [relative journaling](#).

This only effects playback. During recording, all the nescessary information for relative journaling is always stored.

Changing the value of this property *during* playback will *not* affect the current playback.

See also

[About Relative Journaling](#)

ShowMouseMovements property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property ShowMouseMovements: boolean;

Description

Set this property to false if you don't want to show the mousecursor moving around the screen during playback. Mouse-clicks are still executed.

This only effects playback. Mouse-movements are always recorded, no matter what the value of this property is.

Changing the value of this property *during* playback does *not* affect the current playback.

See also

TextOnly Property

State property

Applies to

TMagicMacros component

Declaration

property State: TMacroStates;

Description

Run-time only. Read-only.

This property indicates the state the macro is currently in. You'll will rarely feel the need to use this property but it is used internally.

TextOnly property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

property TextOnly: boolean;

Description

Set this property to true if you only want to playback the keyboard events recorded in the macro. (mouse-clicks and -movements are ignored)

This only effects playback. Mouse-clicks and -movements are *always* recorded, no matter what the value of this property is.

Changing the value of this property *during* playback will *not* affect the current playback.

See also

[ShowMouseMovements](#) Property

VHFilename property

[See also](#)

Applies to

[TMagicMacros](#) component

Declaration

```
property VHFilename: string;
```

Description

If your application should support [VisualHelp](#) files, you should assign the filename of the VisualHelp file you wish to create or edit to this property.

If you are not using VisualHelp, you can just leave this property blank.

See also

[MacroName](#) property

Create method

Applies to

TMagicMacros component

Declaration

```
constructor Create(AOwner: TComponent); override;
```

Description

Although the declaration of the constructor suggests that any TComponent can be the owner of a TMagicMacros instance, you can only use a TForm or one of its descendants.

So if you're creating the component at design-time you *must* drop it on a form. If you're creating the component at run-time, make sure that pass a TForm (or a descendant) as the owner in the Create method.

If you attempt to use anything other than a TForm as the owner, an exception will be raised and the component will *not* be created !

Pause method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

procedure Pause;

Description

Pauses recording or playback.

Call this procedure before doing any serious calculations, loading of tables etc. to make sure that when playing, the macro waits until the operation is completely finished before resuming playback. (See also: [automatic speed adaption](#))

WARNING: A call to this method must always be followed (at some point) by a call to the [Resume](#) method to ensure proper freeing of resources !

This method does absolutely nothing if you're not recording or playing a macro, so you can safely put it in your code, the speed of your application won't be affected since this method returns immediately if journaling is not activated.

See also

[Resume](#) method

[Play](#) method

[PlayVH](#) method

Pause method example

The following code performs an operation that can take a variable amount of time, e.g. the time it takes to load a table from disk depends on speed of harddisk, system occupance, ... To make sure the table is opened and ready before continuing macro-playback, we call Pause. Since a call to Pause **must** be followed by a call to Resume, we are using a Try-Finally block.

```
procedure TimeConsumingProcedure;  
begin  
  MagicMacros1.Pause; //Stop playback of macro  
  try  
    {...Timeconsuming operation here...}  
  finally  
    MagicMacros1.Resume; //Resume playback  
  end;  
end;
```


Play method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure Play;
```

Description

Starts playing the macro.

The [MacroName](#) property must have been assigned before making a call to this procedure, otherwise nothing happens.

If an invalid(=non-existing) macroname has been set, this method raises an [ENoSuchMacro](#) exception.

This method first fires the [BeforePlay](#) event, followed by the [OnPlay](#) event.

See also

[PlayVH](#) method

[Pause](#) method

[Resume](#) method

[StartRecording](#) method

[StopRecording](#) method

Play method example

The following code plays a macro named MyMacro.mcr, located in the c:\Macro directory:

```
begin
  MagicMacros1.MacroName := 'C:\Macro\MyMacro.mcr';
  MagicMacros1.Play;
end;
```

PlayVH method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure PlayVH(ID: integer);
```

Description

Plays the [VisualHelp](#) item with the given ID, in the VisualHelp file specified in the [VHFilename](#) property. If VHFilename is unassigned, this method does nothing.

If an invalid (or non-existing) filename has been specified in VHFilename, this method raises an [ENoSuchVHFile](#) exception.

If there is no macro that matches with the given ID, an [ENoSuchMacro](#) exception is raised.

This method first fires the [BeforeVisualHelp](#) event, followed by the [OnVisualHelp](#) event.

See also

[Play](#) method

[Pause](#) method

[Resume](#) method

PlayVH method example

The following code plays a macro with ID -2000, located in the VisualHelp file named C:\VisualHelp\MyVHF.vhf:

```
begin
  MagicMacros1.VHFilename := 'C:\VisualHelp\MyVHF.vhf';
  MagicMacros1.PlayVH(-2000);
end;
```

Resume method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure Resume;
```

Description

Resumes recording or playback.

Must be preceded by a call to the [Pause](#) method, otherwise this method does nothing.

See also

[Pause](#) method
[Play](#) method
[PlayVH](#) method

Resume method example

The following code performs an operation that can take a variable amount of time, e.g. the time it takes to load a table from disk depends on speed of harddisk, system occupance, ... To make sure the table is opened and ready before continuing macro-playback, we call Pause. Since a call to Pause **must** be followed by a call to Resume, we are using a Try-Finally block.

```
procedure TimeConsumingProcedure;  
begin  
    MagicMacros1.Pause; //Stop playback of macro  
    try  
        {...Timeconsuming operation here...}  
    finally  
        MagicMacros1.Resume; //Resume playback  
    end;  
end;
```

ShowInfoBox method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure ShowInfoBox(Caption: string; Text: String; Duration: integer);
```

Description

Shows an [infobox](#) with the given Caption and a given Text for a certain Duration, during paused playback.

Use this method to give the user some information on what you're doing (very useful when you're using TMagicMacros to make a demo of your application or when making [VisualHelp](#) files).

Duration is the time the infobox needs to be displayed in *milliseconds*.

The Caption is the text being displayed in the titlebar of the infobox. Text is one long string with the text you want to display. The size of the infobox is "shrunk-to-fit" your text and words are wrapped automatically. If you want to force a new line, you can do so by adding a #13-character in the Text-string. (Example: MyText := 'On the first line, blablabla' + #13 + 'On the second line, blablabla';)

The [OnShowInfoBox](#) event is fired after the InfoBox is displayed on the screen.

IMPORTANT : A call to this method must have been preceded by a call to the [Pause](#) method, otherwise, this method does nothing.

See also

[WaitForInfoBox](#) method

[Pause](#) method

[Resume](#) method

ShowInfoBox method example

The following code demonstrates the use of InfoBoxes, to inform the user of what you are doing, during Macro playback.

```
procedure TForm1.Button1Click(Sender: TObject);
const
  Title = 'MagicMacros rules!!';
  Text = 'Just testing this great component!';
begin
  MagicMacros1.Pause; //Stop macroplayback first!!
  try
    MagicMacros1.ShowInfoBox(Title, Text, 5000); //Display Infobox for at least 5 seconds.

    {... do whatever you want to do here ...}

    MagicMacros1.WaitForInfoBox; //Returns only if Infobox was shown for 5sec or more.
  finally
    MagicMacros1.Resume; //Resume playback
  end;
end;
```

StartRecording method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure StartRecording;
```

Description

Starts the recording of a macro.

The [MacroName](#) property must have been assigned, otherwise this method does nothing. Previously recorded macro's with the same name will be overwritten.

This method first fires the [BeforeRecord](#) event, followed by the [OnRecord](#) event.

See also

[StopRecording](#) method

[Play](#) method

StartRecording method example

The following code records a macro named MyMacro.mcr, located in the c:\Macro directory:

```
begin
  MagicMacros1.MacroName := 'C:\Macro\MyMacro.mcr';
  MagicMacros1.StartRecording;
end;
```

StopRecording method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure StopRecording;
```

Description

Stops the recording of the macro and automatically saves it to disk, with the name specified in the [MacroName](#) property. (Note that the extension is always .mcr, regardless of the extension you specify in the MacroName property)

A call to this method should have been preceded by a call to the [StartRecording](#) method, otherwise, this method does nothing.

The [OnRecordEnded](#) event is fired after recording has stopped.

See also

[StartRecording](#) method

StopRecording method example

The following code stops the recording of a macro:

```
MagicMacros1.StopRecording;
```

WaitForInfoBox method

[See also](#)

[Example](#)

Applies to

[TMagicMacros](#) component

Declaration

```
procedure WaitForInfoBox;
```

Description

Waits for the [InfoBox](#) to close before continuing execution of the application.

If the InfoBox has already been closed because the duration specified in the call to the [ShowInfoBox](#) method has elapsed, this method returns immediately, otherwise this method waits until the duration has elapsed and then returns. This way, you can be sure that the **InfoBox** is shown for exactly the specified duration.

IMPORTANT: A call to this method must have been preceded by a call to the ShowInfoBox method, otherwise this method does nothing.

The [OnCloseInfoBox](#) event is fired after the InfoBox is closed.

See also

[ShowInfoBox](#) method

WaitForInfoBox method example

The following code demonstrates the use of InfoBoxes, to inform the user of what you are doing, during Macro playback.

```
procedure TForm1.Button1Click(Sender: TObject);
const
  Title = 'MagicMacros rules!!';
  Text = 'Just testing this great component!';
begin
  MagicMacros1.Pause; //Stop macroplayback first!!
  try
    MagicMacros1.ShowInfoBox(Title, Text, 5000); //Display Infobox for at least 5 seconds.

    {... do whatever you want to do here ...}

    MagicMacros1.WaitForInfoBox; //Returns only if Infobox was shown for 5sec or more.
  finally
    MagicMacros1.Resume; //Resume playback
  end;
end;
```

TMacroStates type

Unit

MagicMacros

Declaration

```
type TMacroStates = set of TMacroState;
```

Description

The TMacroStates type contains the set of values the State property can assume.

TMacroState type

Unit

[MagicMacros](#)

Declaration

```
type TMacroState = (msRecording, msPlaying, msPaused, msShowingInfo,  
msVisualHelp, msIdle, msUnassigned);
```

Description

These are the possible values that can be included in the [State](#) property.

TPlaySpeed type

[See also](#)

Unit

[MagicMacros](#)

Declaration

type TPlaySpeed = 1 . . 9;

Description

TPlaySpeed is the type of the [PlaySpeed](#) property and is an integer value between 1 and 9, where 5 is the normal speed.

The table below shows the interpretations for the different values.

<u>VALUE</u>	<u>SPEED</u>
1	20 % (very slow)
2	40 %
3	60 %
4	80 %
5	100 % (normal)
6	125 %
7	167 %
8	250 %
9	500 % (very fast)

See also

PlaySpeed property

TProgressEvent type

Unit

[MagicMacros](#)

Declaration

```
type TProgressEvent = procedure (PercentDone: integer) of object;
```

Description

The TProgressEvent type is the method type of the [OnProgressEvent](#) and passes the percentage of a loading or saving action that has been completed in the PercentDone parameter.

You can use this value to visualise the progress by using a TGauge or a TProgressBar.

TVHNotifyEvent type

Unit

[MagicMacros](#)

Declaration

```
type TVHNotifyEvent = procedure (Sender: TObject; ID: integer) of object;
```

Description

The TVHNotifyEvent type is the method type of the [BeforeVisualHelp](#), [OnVisualHelp](#), [OnVisualHelpEnded](#) and [OnVisualHelpCancelled](#) eventhandlers and passes the ID of the [VisualHelp](#) item that is being played.

Registering TMagicMacros

The trialversion of TMagicMacros is fully functional but it will only work while the IDE is running. The registered version does not have this limitation.

If you like this product and wish to register, there are a few ways to do this:

VERY VERY VERY VERY IMPORTANT!!!

Whatever method you choose, **ALWAYS** provide me with a **VALID E-MAIL ADDRESS!!** I will send you the registered component through e-mail, and e-mail only (I can not send the component on floppy, for several reasons).

If you do not supply an e-mail address, I will not be able to send the component to you!!

I use a shareware registration service called Kagi to process the orders. Kagi is a large, professional company, so you can be sure that your personal information is safe with them. You pay using any method you prefer: cash, US cheque (no local currency please!), MasterCard, Visa, DinnerClub, etc, etc...

You can register online from my website: <http://magicmacros.8m.com>. All transimisions are secure so your creditcard information is protected.

If you don't like to register online, you can also use snailmail or fax. I've included a special registration program called "Register.Exe" for this purpose. The usage of Register is very simple, just run it and fill in the required information. If need any help, check out the helpfile: register.hlp.

For the latest information and prices, please visit my website: <http://magicmacros.8m.com>

If you are unable to reach my website, you can send an e-mail to the following address:

Niels.Vanspauwen@Kagi.Com

Contacting the author

If you have any questions or problems with my component, please check this helpfile first: I did my best to make it as clear and complete as possible.

Another good source for information are the set of demo projects that are included in the Zipfile.

Third, I've included a FAQ on my website: point your browser to <http://magicmacros.8m.com>

Fourth, if you have purchased the version with full sources, you might want to check them out: the code is fully documented.

And finally, if you can't seem to find the answer to your questions, you can contact me by e-mail at: Niels.Vanspauwen@Kagi.Com (or just [click here](#) to send mail now.)

About Automatic Speed Adaption

One of the most powerful features of TMagicMacros, is the ability to adjust the playback speed of the macro to the current speed of the PC it's playing on.

An example will make this a bit clearer: say you have a button which, when clicked upon, loads a table. The time it takes to load a table depends on lots of things such as the speed of the harddisk, system occupation, the size of the table, etc. Of course, when you have recorded a macro, you'll want the macro to wait until the table is fully loaded before continueing playback. By adding only two method calls, TMagicMacros can accomplish this !

Here's how to do it: just before you load the table (in the OnClick eventhandler of the button) make a call to the [Pause](#) method. Notice that during recording or playback, this completely disables mouse and keyboard. However, when you're not recording or playing a macro, the Pause method immediatly returns, so the normal operation of your application is *not* affected. After the table is loaded, call the [Resume](#) method to continue playback. (if you were recording, the mouse and the keyboard are enabled again.)

For an other example of how to use this feature, take a look at the Button1Click method in the Macro_Rec demo-project.

About InfoBoxes

When you're building a demo of your app using TMagicMacros or when you're building [VisualHelp](#) files, you'll want to inform the "watcher" of what you're doing.

This is where *InfoBoxes* come into the picture. You can easily make an InfoBox pop up for a given amount of time using the [ShowInfoBox](#) method and make the macro wait until the infobox is closed again using the [WaitForInfoBox](#) method.

Please note that you have to make a call to the [Pause](#) method before making a call to ShowInfoBox and a call to the [Resume](#) method after calling WaitForInfoBox, otherwise nothing will happen.

When you're not playing a macro, calls to Pause, ShowInfoBox, WaitForInfoBox and Resume return *immediatly*, so you can savelly leave them in your code: the speed of your app won't be affected !

The demo projects that are included with this component also use Infoboxes frequently, so you can check them for examples.

About Relative Journaling

Relative journaling is perhaps the most important new feature in version 2.0 of TMagicMacros. In earlier versions, the macro was always recorded in absolute screencoordinates. This caused problems if the window for which the macro was recorded had been moved before playback.

With relative journaling, this problem has been solved: TMagicMacros *knows* all the controls on the screen and records coordinates relative to the *positions* of the controls and windows under the cursor. Different positions of windows, buttons even (floating or docked) toolbars are no longer a problem: the effect of the macro is always the same and always correct!

Relative Journaling uses several techniques to do "window-recognition". Only one of them is 100% fullproof: using the tag property of every component. By assigning a unique tag to every visual component on a form, you provide TMagicMacros a way to identify each control flawlessly. Doing this manually, however, is a terrible job: that's why I've build the **TMagicMacros Tagger Utility**. This utility comes **free** with TMagicMacros and is super-easy to use: just open a project, select the forms you want to tag and click start. If you need any more information, you can consult the [Tagger Helpfile](#) (tagger.hlp).

If, for some reason, you can't use the tag property, TMagicMacros will use other techniques to identify windows. While these methods have been fully optimised, they are not fullproof. They will, however, generate the correct result in 90% of the cases. Controls that will often fail if you *don't* use tags are SpeedButton or other Non-Windowed controls or controls whose text changes frequently (e.g. TMemo's).

You can enable relative journaling by setting the [RelativeJournaling](#) property to true.

Installation notes

Adding the component to your component library

The TMM_Comp.Pas (or TMM_Trial.Pas for the trialversion) file contains the component definition. This file should be added to your component-library (as always, make a backup of your current library-file before adding a new component)

All the .PAS and .DCU files should be placed in the same directory.

Two demo projects are included in the ZIP:

- The Macro_Rec demoproject shows you how to build a simple macrorecorder and illustrates most events and methods.
- The SimpleEditor project is a VERY simple texteditor, with examples of VisualHelp (called from the WinHelp file or from the menu) and also an example on how to implement a demo of your application in seconds.

Integrating this helpfile with Delphi's helpsystem

To install this helpfile into your Delphi IDE, follow these steps:

1. Copy this helpfile (MagMcr2.hlp and MagMcr2.cnt) into the same directory as the component.
2. Copy the keyword file (MagMcr2.kwf) to your Delphi\Bin directory.
3. Run HelpInst. (Comes with Delphi, located in Delphi\Help\Tools).
4. Open you Delphi Header File (delphi.hdx, normally located in Delphi\Bin)
5. Click on Add Keyword File and select the MagMcr2.kwf file.
6. If there is any keywordfile that is not found (indicated in the right column of the grid), click Options - Search Paths and locate the wanted .kwf files.
7. Now click "Save" and wait while HelpInst merges the keyword files into the header.

Context-sensitive help for TMagicMacros is now installed!

Note: you may have to restart Delphi before the context-sensitive help is activated.

Known Limitations

Although TMagicMacros is a very powerful component, there are a few limitations in what it can do:

- A maximum of 100.000 keyboard and mouse events can be recorded.
- Due to limitations in the Windows environment, the keycombinations CTRL-ALT-DEL and CTRL-ESC cannot be recorded. All journaling is stopped by Windows if one of these system keycombinations are pressed. The ALT-TAB can't be recorded either, but this doesn't stop the recording.
- Avoid using the special "Windows keys" that can be found on some keyboards. Windows tends to screw up journaling activities when these keys are pressed. This guideline came directly from Micro\$oft, so don't blame me!

Having trouble with the Relative Journaling feature? Be sure to read the [About Relative Journaling](#) section in this helpfile!

Credits

The TMagicMacros component, the demo projects, this helpfile and the Tagger Utility were written by Niels Vanspauwen.

The Lzh.Pas file is based on the Japanese C version of November 29th, 1988. LZSS coded by Haruhiko OKUMURA. Adaptive Huffman Coding coded by Haruyasu YOSHIZAKI. Edited and translated to English by Kenji RIKITAKE. Translated from C to Turbo Pascal by Douglas Webb (2/18/91). Update and bug correction of TP version (4/29/91). Added Delphi exception handling May 9th, 1996 by Danny Heijl (Danny.Heijl@cevi.be).

My eternal gratitude goes out to my girlfriend Martine for her love and support and for putting up with me, spending so much time behind my computer.

Special thanks go out to my beta-testers for bugreports, tips and suggestions. The beta-testers are (in alphabetical order): Brian Armstrong, Erik B.Berry, Marc Di Meo (a.k.a. Hellfire), Bruce Glover (a.k.a. Private), Andreas Hase, Ben Hicks, Tim Hofstee, Bernd Schachinger and George Shin.

Thanks for fruitful discussions and suggestions: Freddy Biets and Gerrit Wolsink.

About VisualHelp

This is where the fun starts: support for advanced options and writing zero (0.000!) lines of code. TMagicMacros offers you the capability to **show** your users how to use your application, instead of telling them about it.

Consider the following example: you're trying to explain some advanced feature of your application in a WinHelp file. The user is by definition stupid and doesn't understand what the hell you're talking about. Now, wouldn't it be nice if you could include a button in your WinHelp file with the caption "Please show me!" which the user could press, sit back and watch YOU showing him what you mean ?

TMagicMacros makes this possible! Just record some macros illustrating the difficult parts of your app, stuff 'em in a VisualHelp file and add a button in your WinHelp file and that's it !

To learn all about making VisualHelp files, click [here](#).

Making VisualHelp files

Simply follow these steps to make [VisualHelp](#) files:

- 1) Drop a TMagicMacros component on your mainform.
- 2) Record some macros.
- 3) Rightclick the TMagicMacros component and select "Include macro in VHFFile" in the PopUp window. The component editor is shown. (I have included a bitmap of the component editor at the bottom of this topic, which you can click on to get more information).
- 4) Enter a name for your VisualHelp file.
- 5) In the grid, add the full path of the macro you want to include in the VisualHelp file (or alternatively, double click the grid to get an Open-dialog) and assign a unique ID (= a negative integer below -999) to the macro. Repeat this step for all the macros you wish to include.
- 6) Click OK to generate the VisualHelp file.
- 7) Open your favorite WinHelp editor and include a button to which you assign a WinHelp macro called ShortCut, with the following parameters: the classname of your mainform (e.g. TMyMainForm), "", the ID (remember, the negative integer of step 5) of the macro you wish to play, 0. In case you're not using a WinHelp editor, just type the following to add a button that starts the macro with ID = -1000:

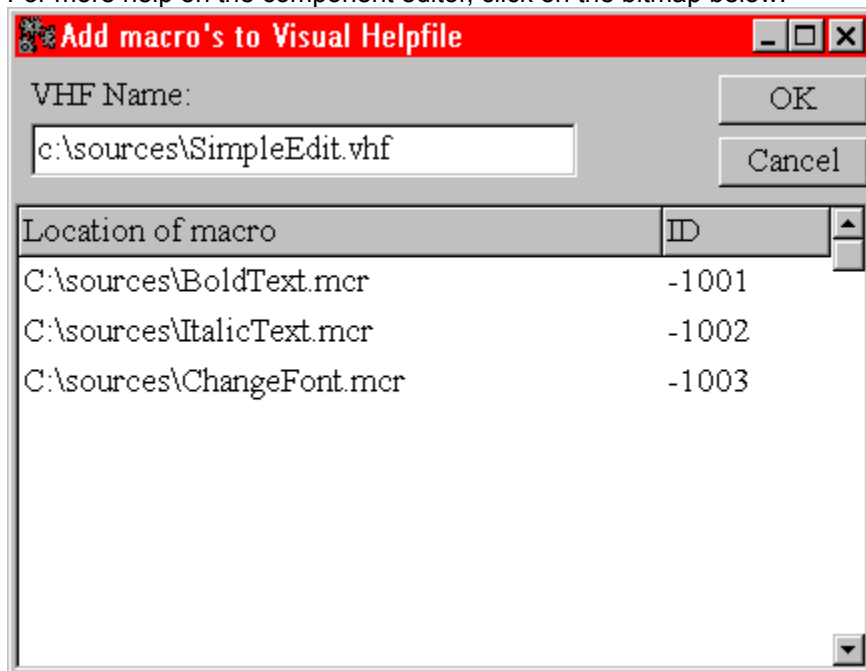
```
{button Please show me, shortcut(TMyMainForm, "", -1000, 0)}
```

- 8) Compile your helpfile and you're done !

IMPORTANT: The VisualHelp file is NOT compiled into your WinHelp file, so you need to distribute your .HLP file as well as your .VHF file with your application !

Distributing the .MCR files is *not* needed: they are packed in the .VHF files.

For more help on the component editor, click on the bitmap below:



Here you must enter the filename of the VisualHelp file you want to create or edit. By default, this editbox contains the value of the VHFilename property.
If you change this field, the VHFilename property will be changed also when you press OK.

Here you must specify the full path of the macro you wish to include in the VisualHelp file.
If you doubleclick a cell in the grid, an OpenFileDialog pops up automatically.

Here you must enter a negative integer (smaller than -999) that uniquely identifies the macro within the VisualHelp file.

It is this ID that you must pass to the PlayVH method and also that is passed to you by all the VisualHelp events.

By pressing OK, all the changes you have made will be saved and the VisualHelp file will be generated.

Pressing cancel discards all the changes you have made. The VisualHelp file is not generated (or altered if one already exists).

