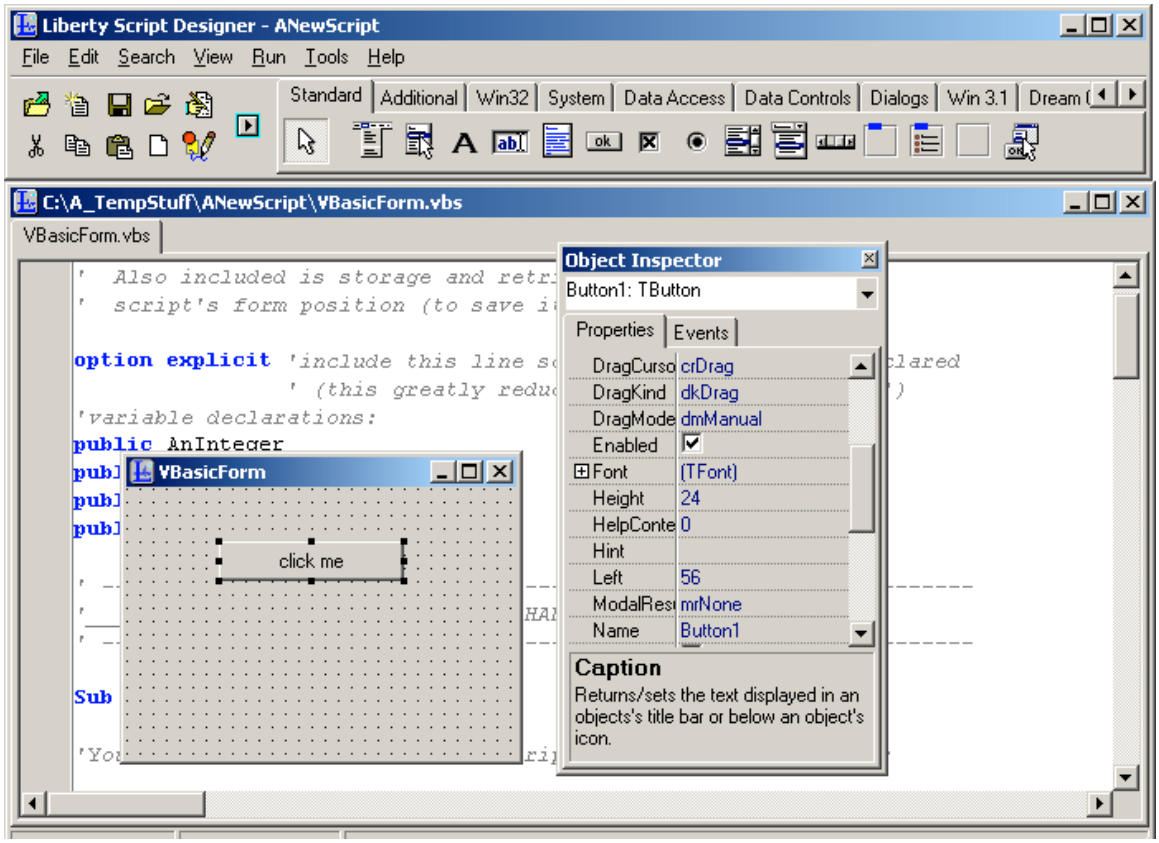
Liberty Instruments, Inc

Praxis Script Programming Guide

V 2.2 January, 2005



About Praxis Scripts

This guide is intended to for use with the PRAXIS v2.0 manual in the development of custom scripts which can be run within the PRAXIS measurement system.

Praxis Scripts are Windows applications in themselves that run from within PRAXIS. They include visual forms on which you can place familiar user interface components like buttons, edit boxes, sliders, text editors, images, etc., and easily give them high level functionality by setting properties and writing code in DelphiScript (based on Borland Delphi) or VBScript (a language based on Microsoft Visual Basic).

The Scripts are developed using the Liberty **Script Designer**, which is an additional, separate application that is installed along with PRAXIS and which can be started from within Praxis.

The Script Designer can be started by using the “Scripts -> Design a Script” menu on the Main Form.

Although Praxis was developed entirely using Borland's Delphi language, and the components used are Delphi "VCL" components, other languages such as VBScript and JavaScript can be used to develop code if you wish. Most procedures and functions that are built into Delphi are also usable in a PRAXIS script, even when the script is programmed in languages other than DelphiScript. The Script Designer provides specific support, such as automatic generation of skeletal scripts, for the VBScript and DelphiScript languages

Developing scripts with the version 2 Script Designer (and with PRAXIS v2) is considerably easier than with the previous versions. For example:

- When creating a new script, the Script Designer will automatically set up your script directory and will provide a template or basic structure for your DelphiScript or VBScript, , even including procedure or subroutine bodies for handling the PRAXIS events.
- The Script Designer makes it easy to edit or create a Picture.bmp and Description.rtf file for your script, so that your script can be easily found and launched from PRAXIS' Script Launcher.
- The Script Designer now provides a "Cut/Copy/Paste/Find" popup menu when you are editing within your script text, to simplify editing.
- PRAXIS scripts can now operate with strict error checking, which causes an error message if any undeclared variables are encountered. This speeds debugging and helps prevent hard-to-find bugs in your code. The error tolerance can be selected via the Main Form’s “Config ->Preferences->Strict Script Running” menu.
- You can now use the very handy "EditRealSpin" (or NVC) controls in your scripts for numerical entry by your users.
- The PRAXIS main form’s “Scripts” menu provides options for zipping a script directory to a compact form, or for installing such a “zipped script”. This makes transferring or sending scripts between installations or users much easier.
- You can also now launch your script in PRAXIS from a button the Script Designer, simplifiying the test and debug cycle for script development

There is also built-in support that can copy an existing script folder under a new folder name so that the existing script can be used as a starting point for the new script. This is often the easiest way to generate a customized PRAXIS script – simply create a new one based on one which is similar to the one you want to create and edit it to provide your new behaviour or features. To use this, just select “Copy an existing Script Folder” in the form that appears when you use the Script Designer’s “File->New PRAXIS Script” menu. Alternately, you can have the Script Designer create a new skeletal script for you based on some selectable parameters.

You can find more information and links about Praxis script writing at `http://www.libinst.com/praxis_script_writers.htm`**.**

Using VBScript

When you start the Liberty Script Designer and choose “New PRAXIS Script” from its menu, you have the option to automatically create a skeletal DelphiScript or a VBScript . The Designer can create a new Script for you, including a form and a code section and blocks for all of PRAXIS’ built-in events.

If you choose Delphi, you can code in PascalScript, which is the only supported script type that operates as compiled code within Praxis (the others are interpreted). This makes DelphiScript run faster and it has more direct control.

But many programmers are more familiar with Visual Basic, of which VBScript is a subset. Those programmers should probably not have difficulty writing PRAXIS scripts in VBScript, but there may be some difficulty in knowing the meaning of various properties of the “VCL” components made available in the Script Designer. These components are the same as those in Borland's Delphi, and documentation for this is available from Borland in the form of a help file “**D5VCL_P.ZIP** “at http://info.borland.com/techpubs/delphi/v5/updates/pro.html.

It is also possible to make and run scripts coded in other script languages (JScript, Python, Perl), but generation of these scripts is not automated by the Script Designer.

Object Pascal

If you choose to create a new "Delphi Application", you can code in PascalScript, which is the only supported script type that operates as compiled code within Praxis (the others are interpreted). This makes DelphiScript run faster and it has more direct control. (*And the programmer of Praxis is heavily biased toward Delphi!*).

If you are not experienced in Delphi programming, don't be intimidated. It is as easy to use as other visual languages (easier, if you tend to think logically). It is an event-based language, very structured and object-oriented.

There are a number of good books about Delphi. Some programmers have even learned it by simply "diving into it" (with use of Delphi's Help files). These help files can be obtained from Borland at http://info.borland.com/techpubs/delphi/v5/updates/pro.html. The files D5VCL_P.ZIP and D5OPLR.ZIP files are particularly useful.

Note that there are some differences between DelphiScript and Delphi's Object Pascal. Most of these are detailed in the section of that name.

Although you could conceivably develop your script using Delphi itself, note that only the controls shown in the Script Designer will work in Praxis (others will prevent the script from running). And

of course, Delphi will not understand the Praxis-Specific procedures, properties, etc, so you could not compile or test in that environment.

Differences between DelphiScript and Object Pascal

The following information is provided by Dream Company, creators of the "Dream Scripter" component used in Praxis's Script operations, and is reproduced here by permission. It is provided as-is -- some portions may not be relevant to use within Praxis.

This document describes the difference between Delphi Script and Object Pascal used in Delphi 5.

- All variables in Delphi Script are always of Variant type. Typecasting is ignored.
- Types in variables declaration are ignored and can be skipped, so these declarations are correct:

```
var a : integer;
var b : integerr;
var c, d;
```

- Types of parameters in procedure/function declaration are ignored and can be skipped. For example, this code is correct:

```
function sum(a, b) : integer;
begin
  result := a + b;
end;
```

- Type of array elements is ignored and can be skipped so these declarations are equal:

```
var x : array [1..2] of double;
var x : array [1..2];
```

- These keywords are ignored:

· *interface, implementation, program, unit*

You can use them but they have no effect.

- "*in*" directive in uses is ignored
- Delphi Script doesn't support type declarations. It tries to skip them but

· complex declarations may cause parser to fail.

- ^ and @ operators are not supported.
- **You can't use the function name to set the return value, use Result to do so**.
- Nested routines are supported but you can't use variables of top level function from the nested one.
- The following keywords are not supported:

· *as, asm, class, dispinterface, exports, finalization, inherited, initialization, inline, interface, is, library, object, out, property, record, resourcestring, set, type*

- The following directives are not supported (note that some of them are obsolete and aren't supported by Delphi too):

· *absolute, abstract, assembler, automated, cdecl, contains, default, dispid, dynamic, export, external, far, implements, index, message, name, near, nodefault, overload, override, package, pascal, private*

protected, public, published, read, readonly, register, reintroduce, requires, resident, safecall, stdcall, stored, virtual, write, writeonly

- These RTL functions aren't supported in Delphi Script:

· *Abort, Addr, Assert, Dec, FillChar, Finalize, Hi, High, Inc, Initialize, Lo, Low, New, Ptr, SetString, SizeOf, Str, UniqueString, VarArrayRedim, VarArrayRef, VarCast, VarClear, VarCopy*

- Open array declaration is not supported.
- *CASE* can be used for any types. So you can write

```
case UserName of
  'Alex', 'John' : IsAdministrator := true;
  'Peter' : IsAdministrator := false;
  else  raise('Unknown user');
end;
```

- *Raise* can be used without parameters to re-raise the last exception. You can also use Raise with string parameter to raise the exception with the specified message string. For example,

```
· Raise(Format('Invalid value : %d', [Height]));
```

- *Threadvar* keyword is treated as *Var*
- [...] set constructors are not supported. You can use MkSet to create a set. For example,

```
· Font.Style := MkSet(fsBold, fsItalic);
```

- Set operator *In* is not supported. Use InSet to check whether a value is a member of set. For example,

```
· if InSet(fsBold, Font.Style) then
ShowMessage('Bold');
```

- Note, that set operators '+', '-', '*', '<=', '>=' don't work correctly. You have to use logical operators. For example,

```
· ASet := BSet + CSet; should be changed to ASet := BSet or CSet;
```

- *CreateObject* can be used to create objects that will be implicitly freed when no longer used. So instead of

```
procedure proc;
var l;
begin
  l := TList.Create;
  try
    // do something with l
  finally
    l.Free;
  end;
end;
```

you can write

```
procedure proc;
var l;
begin
  l := CreateObject(TList);
  // do something with l
end;
```

- Built in function *Evaluate* can be used to interpret string as program code during runtime of a program and execute the code contained in the string. For example, you can write script like

```
Evaluate(ProcNames[ProcIndex]);
```

· and the procedure which name is specified in ProcNames[ProcIndex] will be called.

- Built in directive *UseUnit* can be used to dynamically add to uses section any unit at runtime. For example, the following script

```
if FileExists('Update.pas') then
begin
  UseUnit('Update.pas');
  Evaluate('Update.DoUpdate');
end;
```

checks whether the unit named Update.pas exists and if so calls DoUpdate method from there.

- *UnloadUnit* directive unloads any unit added by "uses" section or with *UseUnit* call.

How Scripts are Stored and Organized

All files related to a single script should be contained in a separate subdirectory (also called a "Folder").

In **Full Mode**, the script directories are most conveniently stored as subdirectories of the directory named "VCLScripts", which itself resides under the same directory as the Praxis executable file "praxis.exe". This will allow the script to appear in the ~~Script Launcher~~{linkID=6100}, for easy access. In most cases, the proper home directory for the script subdirectories will be:

C:\Program Files\Praxis\VCLScripts

In **Demo Mode**, the script directories should be subdirectories of the directory named "DemoScripts", which itself resides under the same directory as the Praxis executable file. This will allow the script to appear in the ~~Script Launcher~~{linkID=6100}, for easy access. In most cases, the proper home directory for the script subdirectories will be:

C:\Program Files\Praxis\DemoScripts

Ideally, each script subdirectory should contain all the following files:

- **Description.rtf** : a "rich text file", which can be created using the "WordPad" application which comes with Microsoft Windows. This should give a brief description of the script, for display in the Script Launcher.
- **Picture.bmp**: a bitmap file, width=150 pixels, height=100 pixels. This can be easily created using a an application such as "Paint" which comes with Microsoft Windows. This picture can give a visual hint about the script in the Script Launcher
- ********.ipr:*** this file is generated using the Liberty Script Editor, and identifies the files that make up the script code and visual forms. There should be ONLY ONE of these ipr files in a script directory.
- ********.dfm***: this contains data about the visual forms. It is created by Liberty Script Editor. There can be a number of these type files in a complex script.
- ********.pas*** or ********.vbs*** or ********.js*** : these files (generated by Liberty Script Editor) contain the Object Pascal (DelphiScript), VBScript, or JScript files which make up the script code. There can be a number of these files.
- any other bitmap or rich text (or other type) files which may be used in the particular script code.

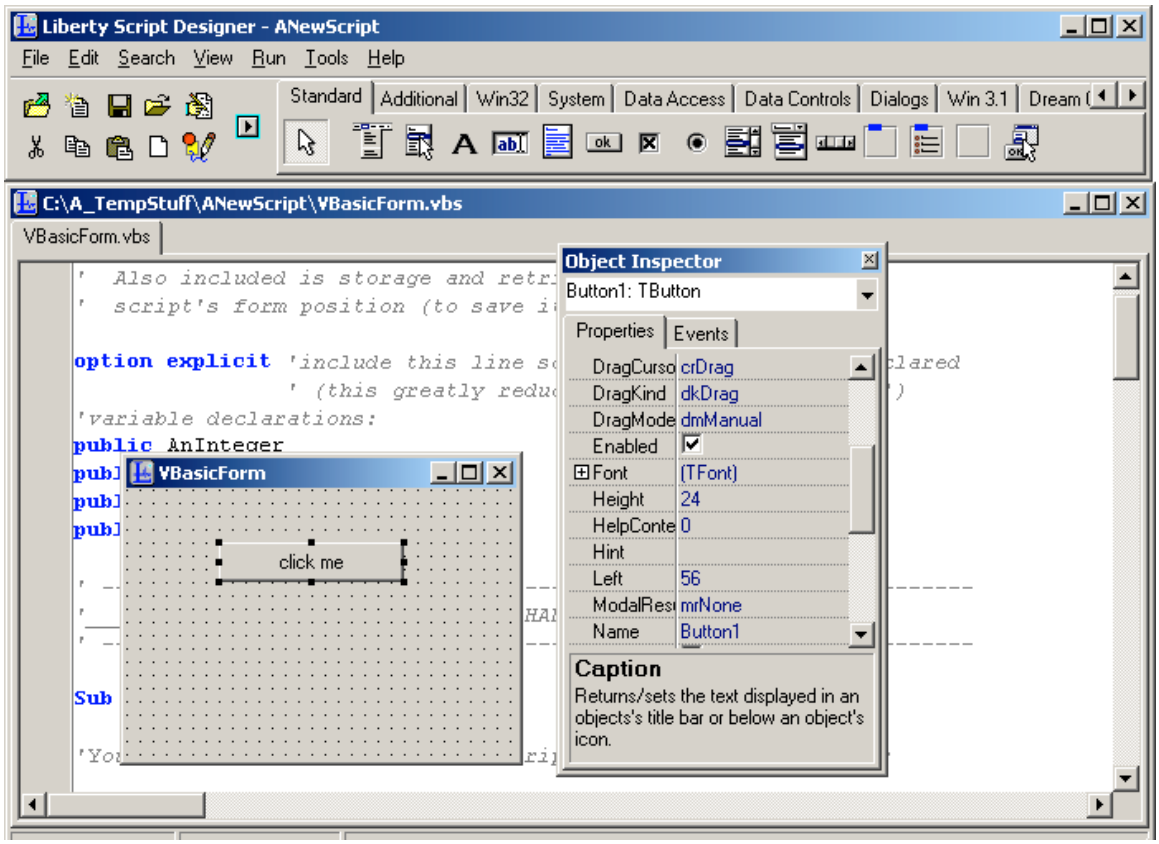
Calling COM servers from Scripts

Very large portions of the Delphi Object Pascal language are usable from your script. This includes the ability to make Windows API calls, including COM methods. That means that scripts can send Praxis data to other applications, or can control other hardware (if it is run by a COM server).

You can use such features to control other hardware (switches, relays, turntables) using third party data control devices (digital I/O), provided those devices provide a COM server for access.

For an example of controlling a COM server, the following script will start up Microsoft Excel (if it is installed on the computer).

```
Procedure TDWForm.OpenExcel; const
{ XlSheetType }
xlChart = -4109;
xlDialogSheet = -4116;
xlExcel4IntlMacroSheet = 4;
xlExcel4MacroSheet = 3;
xlWorksheet = -4167;
{ XlWBATemplate }
xlWBATChart = -4109;
xlWBATExcel4IntlMacroSheet = 4;
xlWBATExcel4MacroSheet = 3;
xlWBATWorksheet = -4167;
Var V :variant;
begin
Try
if VarIsEmpty(V) then
begin
V := CreateOleObject('Excel.Application');
V.Workbooks.Add(xlWBatWorkSheet);
V.Workbooks[1].WorkSheets[1].Name := '-praxis- Some Data';
end;
V.Visible := True;
Except
ShowMessage('Excel could not be opened on this machine!');
End;
end;
```



The Liberty Script Designer

The Script Designer is a separate application that can be started from within Praxis. It can be started from the “Scripts->Design A Script” menu of the Main Form.

Script Designer is essentially a complete development environment, with a component palette, graphical designer forms, text editors for entering Code, and a "project manager". It is much like a streamlined version of the Borland Delphi development environment.

Version 2 of the Script Designer includes a number of enhancements to simply the creation or editing of scripts, and to make the scripts more powerful. A larger number of visual components are available for use in design, including the “EditRealSpin” (also known as a NVC or Numeric Value Control) that is used throughout the PRAXIS system.

Controls avaialable from the Script Designer are “VCL” controls (controls made by and for Borland Delphi), and specific documentation for most of these can be obtained, if needed, by downloading the help file “**D5VCL_P.ZIP** “ from Borland at http://info.borland.com/techpubs/delphi/v5/updates/pro.html.

Scripts for PRAXIS can be written in a number of scripting languages. However, the Script Designer provides specific support, such as automatic generation of skeletal scripts, only for the VBScript and DelphiScript languages.

You can also find a basic script writing tutorial on the Download page of our web site.

Script Design: Some Tips and Noteworthy Comments

DelphiScript and VBScript, while being in many ways very similar to the Delphi and VisualBasic languages, are not *exactly* the same as their superset languages. There are some special circumstances that may need consideration. Also, the nature of designing one program (your script) to run *within* another (PRAXIS) requires some special treatment. Please note the items below before starting your custom script design, and refer back here if you should run into anything unexpected.

.

- You should **leave the Script Designer open** while testing your script in Praxis. This allows you to go back, make adjustments or corrections, and try again. You can launch your script (into PRAXIS) by using the Script Designer's "Run" menu or button.

- If your script has more than one Form, also include "UseUnit" statements (such as "UseUnit('Unit1.pas'); " or "UseUnit("Unit1.vbs")") in the FormCreate method for each form so that all other forms are made visible to each other and can refer to each other in their code.

- In DelphiScript, all procedures and functions should be written as object methods of a form. For instance, make a "Procedure TForm1.MyProcedure;", rather than a global "Procedure MyProcedure;". When you need to refer to form methods (*OTHER THAN EVENT HANDLERS*) that are in other forms, refer to those methods using the other form's Unit name rather than the Form name: "Unit1.MyProcedure" rather than "Form1.MyProcedure". *EXCEPTION*: For event handlers that are automatically created by the editor, such as "Button1Click(sender)", you should, however, use the Form name to specify it, for example., "Form1.Button1Click(nil)". *This can be confusing, but if PRAXIS doesn't seem to respond to your procedure name, try changing the reference between the unit name and the form name.*

- In PRAXIS's Main Form, set the "Config->Preferences->Strict Script Running" menu to "checked" whenever you are testing your script. This will cause PRAXIS to check for variable declarations and to stop if any errors are found. When writing in VBScript, ***ALWAYS*** use "Option Explicit" at the top of each code file.

- Do not use duplicate variable names in different forms. For instance, don't have a variable "X" in one form and a different variable "X" in another. Use different names, and treat all variables as if they were global. Object variables (such as MyForm.SomeVariable) behave as global variables.

- While developing scripts, toggle OFF the PRAXIS "Backdrop". Otherwise, when you go to test the code in PRAXIS, your Script Designer would be hidden and difficult to access.

- Remember that just because you can see your code and forms in the Script Designer, that *doesn't* mean that PRAXIS can see the latest version of your code! -- be sure to **save your file with the changes** before trying to run your Script from Praxis! PRAXIS and the Script Designer are separate programs and are unaware of what each other are doing.

- **Scripts are very powerful.** And if you've done much programming, then you know that this means: **you are perfectly capable of crashing Praxis, and even the operating system** because of bugs or errors in your script! For this reason, if you have a choice, develop your scripts in one of the more stable versions of Windows (such as Windows2000 – don't even try with WindowsME). Know how to close down a frozen process with Ctl-Alt-Delete.

- None of the Praxis specific items or names (including "Praxis") will be recognized while your Script application is loading, or while its visual forms are being first created. **If you need to initialize any of Praxis parameters at your script's startup time**, do these in the Event Handler method called "**Initscript**" for your main script form. The InitScript event will happen just after the main form is created. At that time "Praxis", and its functions and identifiers, will be known to your script and can be initialized.

- In your script's "**Initscript**" event handler, use the procedure **ScriptConfig** (see under Global functions and Procedures). This will allow you to save the current state of PRAXIS before you script executes, load a full configuration file of settings needed by your script, and cause PRAXIS to automatically restore its settings after your script exits.

- **Be sure to initialize ALL boolean variables before using them**. For example, if you declare a variable "IsOK: boolean", make sure that you set it to an initial value of either true or false in your FormCreate or FormActivate procedure -- otherwise, it may be evaluated as being neither true nor false, or may cause other very confusing behavior.

- You are developing an "**Event Driven**" program, rather than a simple procedural program. In other words, all script code that gets executed occurs in response to Events of some kind. Examples of events are: a mouse click, a button push, a timer going off, PRAXIS doing an Acquisition, etcetera. Much of your code will be grouped into sets of Event Handlers. Of course some of these events will occur multiple times and under different circumstances, so you need to keep track of what conceptual processes are happening as execution of your script progresses. This is needed so that your Event Handlers can adapt their response to the changing situations. You can do this using your own variables, or with the built-in global functions "**SetDoing**", "**IsDoing**", "**IsInDoing**", and "**NowDoing**".

- You shouldn't call "**Praxis.Start**" and "**Praxis.Stop**" within the same script procedure, as this will more or less stop the acquisition process before it has collected anything. Remember that these operations really start or end thread processes (code that happens essentially at the same time as other program operation occur) within Praxis and then almost immediately returns control to your code. You should instead Stop or Start Praxis data acquisitions in response to events.

- If you want to track the sequence of code execution in your script project, insert "**LogString**()" commands in select places within your code and then run your script. LogString pops up a special debugging form that shows the message specified by your call, and does this without interrupting script or PRAXIS program flow. For instance, the call:

  LogString('now in the ButtonClick handler'); //DelphiScript
  LogString("now in the ButtonClick handler") 'VBScript

  will result in the specified message appearing in the debug form (which will appear if not already visible). Multiple messages will appear in order in which they occurred. If you want to track the sequence in which event handlers are called, add a LogString call to each (exception: don't put LogString calls in FormCreate event handlers!).

- Never have your code wait for anything! If your code waits, PRAXIS also waits, and nothing gets done. Instead, if you need to have time pass before continuing, set a Timer component and put the next part of your process in the event handler for the timer. Or better still, put the next part of your process in the event handler of an event that you know happens at at time when your process can continue.

- To enter code for event handlers of Script Designer visual components (like buttons, checkboxes, etc), double click on the component and you will be transported to an event handler cody block for the component. If you need to create or edit a code block for a different

event handler, press the **[F11]** key to access the Object Inspector form, use the “Events” tab, and double click on the event name for which you wish to code.

- In the Script Designer, use the **[F12]** key to change between display of your visual form and its code block.

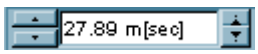
- You cannot create “properties” in your forms or objects in Praxis Scripts. Instead, use functions or variables.

- While DelphiScript constructions such as "With Praxis Do.." are legal in Praxis scripts, you should avoid them. This is because every name within such a construction will be assumed to be a property or method of the PRAXIS object (instead of a method or variable that may be of an item in your script code).

- In DelphiScript code: **If you find that your code does not seem to be responding to the user's use of menus, buttons, etc.**, check for unmatched Begin/End pairs existing just before the event handler code that does not seem to be working! The Script Designer will not indicate unmatched Begin/End pairs, and when such code is executed, entire procedures can become inactive.

Working with NVCs (or “EditRealSpin”) Components

Beginning with PRAXIS version 2 and Liberty Script Designer v2, you can now design with PRAXIS’s exclusive Numerical Value Controls (NVCs) in your scripts. These controls (called “EditRealSpin” or ERS devices in the design environment) provide remarkably versatile and intuitive control over numerical values, a function that is frequently needed during configuration of



measurement or processing parameters.

EditRealSpin controls have a number of properties that should be configured at design time to simplify operation of your script for your users and to minimize the need for range checking in your script code. You can also configure these properties at run time, but that may cause confusion because some property settings are not compatible with others.

These are the EditRealSpin properties that you can configure:

- **Enabled**: detemines whether the user can adjust the ERS control
- **AvoidSetFocus**: this should be left False (unchecked) for script use
- **CardInc**: how much the RealValue will be incremented/decrmented on up/down clicks or mouse wheel rolls when IntOnly is True and StepMode is set to smIntRound.
- **EngUnits**: if true (checked), quantities will be shown using engineering notation. For example, the number 1234.5 would display as “1.2345k”. Engineering units can also be entered by the user into the control.
- **IntOnly**: if true, the RealValue can only be set to whole number values (integers).
- **MaxValue**: the largest number that the control will accept as the RealValue
- **MinValue**: the smallest number that the control will accept as the RealValue

- **RealInc**: how much the RealValue will be incremented/decrmented on up/down clicks or mouse wheel rolls when IntOnly is false and StepMode is set to smReal.
- **RealMult**: how much the RealValue will be multiplied/divided by on up/down clicks or mouse wheel rolls when IntOnly is false and StepMode is set to smLog.
- **RealValue**: this most-used property is used to set or read the numerical quantity that is being represented by the control. It is a floating point value, so if IntOnly is true, you should always round the value of RealValue before using it as an integer quantity. When you set the RealValue from code, the OnValChange event does not occur – that only happens when the user changes the RealValue by manipulating the visual ERS control.
- **ShowDotSpin**: this should normally be left false in scripts
- **ShowUnits**: this determines whether the ERS will display the units string in the control.
- **StepMode**: this determines how the control will change when the up/down buttons are clicked or the mouse wheel is rolled. The user of the script can also change this setting for each ERS by right-clicking the up/down part of the control. Choices are:
    - **sm12345:** will change to the next higher or lower value having one significant digit that is 1,2,3,4 or 5. For example, increasing from 34.56m[Volt] will result in a next value of 40m[Volt].
    - **sm125:** will change to the next higher or lower value having one significant digit that is 1,2 or 5. For example, increasing from 4.567m[Volt] will result in a next value of 5m[Volt].
    - **smFine**: this setting should not generally be used in scripts.
    - **smIntRound**: values will change only to whole numbers, as determined by CardInc. Used with InOnly.
    - **smLog**: values will change multiplied or divided by RealMult.
    - **smReal**: values will change incremented or decremented by RealInc.
- **Units:** this string value is displayed in the control to help the user understand what is being controlled.

There are several events associated with each EditRealSpin controls. In scripts, only these are likely to be useful:

- **OnClick**: occurs when the user clicks on the control.
- **OnValChange**: this occurs when the user changes the RealValue by manipulating the control. It does NOT occur when you change the RealValue in code. If you wish to trigger the OnValChange event in code, call the ERS control's ValChange procedure:

```
MyEditRealSpin.RealValue:=1.234; MyEditRealSpin.ValChange;
```

Praxis-Specific Methods, Functions, Procedures and Properties

Praxis scripts wouldn't be a very useful feature if they weren't able to control the Praxis measurement and formatting operations. To accomplish this, there are large number of procedures (aka "commands", "methods", or "subroutines"), functions, and properties which can be used as if they were part of the programming language.

There are also certain special objects to which you may need to refer when controlling Praxis from a script. One is, of course, the object "**Praxis**", used to control the system. Others are Plot Objects (of type TDataPlot), used to format or access data in various plots.

In the "Properties and Methods" sections to follow, property names are given, followed by their types and their supported uses (Read, Write, or Read/Write). Function names are followed by their types only. Procedure names are defined without a type. For example:

```
Stimulus: integer (Read/Write); ---- is a property
EngFloatToStr(x:double):string; ---- is a function
ShowAcquireCfg(Yes:boolean); ---- is a procedure.
```

IMPORTANT: None of the these names will be recognized as your Script application is first loading, or while its visual forms are being first created. **If you need to initialize any of Praxis's parameters at your script's startup time**, write an Event Handler method called "Initscript" for your main script form (see, for example, the "**procedure TDWform.InitScript(sender: TObject);**" in the WizardModels script) and do the initializations *there*. The InitScript event will happen just after the main form is created. At that time and thereafter, "Praxis" (and its API elements) will be known to your script and you will then be able to use them.

The items documented in these sections contain many methods and properties which are used within Praxis itself. Some of these can be used only in very advanced scripts, and their usage can be complicated. The items documented here are those which we felt could perhaps be helpful in script development -- but there is no guarantee that all the given methods or properties will be useful.

General Tips about Script Development:
Before writing a script to manipulate the data stimulus/acquisition system, you should try to run through those operations manually. This will give you a better idea of the order in which settings and procedure calls should occur. For example, you should always set the Stimulus property before setting the Acquisition property, because the Acquisition property change will not "stick" (Praxis will ignore it) unless it is compatible with the currently specified Stimulus.

In general, while the Acquisition engine is running, you should not try to set values which will be used to configure an Acquisition or Stimulus (the values on the Config Forms which become grayed-out while Praxis is running). Set such values (such as, for example, "Praxis.FftSize") before starting or after stopping the acquisition. In most cases, attempt to change such values, while Praxis.Running is true, will result in the command being ignored. In some cases, however, it is possible that such actions could crash the system.

For DelphiScript: Note that the "arguments" or "parameters" (if used) of procedures or functions are to be enclosed by parentheses, while the indexes of properties (if used) are enclosed by square brackets. For example:

```
PrimaryPlot.TraceColor[1] := clRed; //a property index
SomeFrequency := PrimaryPlot.DomainFromPoint(42); //a function argument
PrimaryPlot.LoadMultiPlotList(mylistname); //a procedure argument
```

Global Constants and Types

Constants

Stimulus types: (example usage: Praxis.Stimulus:=stiChirp;)

```
stiChirp=0;         // Chirp (ReSync)
stiMLSrs=1;         // MLS (ReSync)
stiMLSPnkRs=2;      // MLS/Pink (ReSync)
stiChirpS=17            // Chirp (synchronous)
stiImpulse=3;       // Impulse (synchronous)
stiMLS=4;           // MLS (synchronous)
stiMLSPnk=5;        // MLS/Pink (synchronous)
stiSinStdS=6;       // Fixed Tones (synchronous)
stiSinStd1A=7;      // Fixed Tones (1ch asynch)
stiSinStd2A=8;      // Fixed Tones (2ch asynch)
stiSinSteppedS=9;   // Stepped Tones(synchronous)
stiSinStepped1A=10; // Stepped Tones(1ch asynch)
stiSinStepped2A=11; // Stepped Tones(2ch asynch)
stiWavS=12;         // WAV File (synch)
stiWavA=13;         // WAV File (asynch)
stiWhite=14;        // White Noise
stiPink=15;         // Pink Noise
stiWaveletBurst=16      // Wavelet Burst (synchronous)
stiNone=99;         // None
```

Acquisition types: (example usage: Praxis.Acquisition:=acqTime1;)

```
acqTime1=0;         // Time Domain 1ch
acqTime2=1;         // Time Domain 2ch
acqTimeMLS=2;       // Impulse Response via MLS
acqTimeChirp=3;     // Impulse Response via Chirp
acqFft1=4;          // Spectrum FFT 1ch
acqFft2=5;          // Spectrum FFT 2ch
acqRta1=6;          // Spectrum RTA 1ch
acqRta2=7;          // Spectrum RTA 2ch
acqFreqRespCpx=8;   // Freq. Response MAG and PHASE
acqFreqRespMag1=9;  // Freq. Response MAG 1ch
acqFreqRespMag2=10; // Freq. Resp. MAG 2ch
acqFreqRespMon=11;  // Freq. Response Live Monitor
acqImpulseMon=12;   // Impulse Resp. Live Monitor
acqImpedance=13;    // Impedance
acqCompTrack1=14;   // Component Track 1ch
acqCompTrack2=15;   // Component Track 2ch
acqFRandHD=16;    // Freq. Resp. and distortion via Chirp (Farina's
                  method)
acqFRandIMD=17;   // Freq.Resp.and IM difference distortion via Chirp
acqDispTime=18;   // Displacement vs. Time (NOT IMPLEMENTED)
acqDispFft=19;    // FFT of displacement (NOT IMPLEMENTED)
acqFft2Cpx=20;    // 2 CH fft, complex normalized
acqWaveletMaxLevel=21; // Wavelet Max Level capability
acqWaveletSingleFreq=22; // Wavelet distortion vs level
acqWaveletDist=23;// Wavelet Distortion vs. Frequency (NOT IMPLEMENTED)
acqWaveletDist3D=24;// Wavelet Dist vs. Level vs Freq (NOT IMPLEMENTED)
acqCustom=99;       // Custom Acquisition
```

PostProcess types: (example usage: Praxis.PostProcess:=ppMath;)

```
ppNone=0;           // None
ppReMap=1;          // Re-Map to new Freqs
ppFilter=2;         // Filter
ppMath=3;           // Math
ppPaste=4;          // PasteOver
ppSmoothing=5;      // Smoothing
ppFft=6;            // Fft
ppIfft=7;           // Inverse Fft
ppTSP=8;            // Thiele/Small Extract
ppWF=9;             // CSD Waterfall
```

```
ppStep=10;          //  Step Response
ppHilbert=11;       // Hilbert Transform
ppSchroeder=12;     // Schroeder Curve
ppGroupDelay=13;    // Group Delay
ppPolar=14;         // Polar Data Compilation
ppFRZ=15;           // Combination of Frequency and Impedance data
ppETC=16;           // Analytic Signal (ETC)
ppPCepstrum=17;   // Power Cepstrum transform
ppQAnechFB=18;    // Quasi-Anechoic Bass Restoration
ppProgAvg=19;     // Progressive Averaging
ppTailCorr=20;    // Response Tail Correction
ppUser=99;          // User Defined
```

Stepped Tone Increments: (used in method: ChangeStepFSpecification)
`swmLog=1; swmLin=0;`

TFormStates is a record type to be used with the Praxis.FormStates property, for saving and restoring the visibilities of the Configure forms and the Levels Form. Often, these forms are not needed to be seen in a script and only clutter the screen -- they can be hidden via the script (see the "Show..." procedures in *General properties of the Praxis object*). You can then use the Praxis.FormStates property to restore the Praxis system, when the script ends, to the visiblities the user had when the script started.

Enumerated Types
Use one of the names given in parthesis when a property or method requires a value from these types:

```
dvType = (dvTime,dvFreq);
//used with DomainView button of Praxis

MkrTrackTypes = (mttNone,mttLMax,mttLMin,mttVMax,mttVMin);
//used for MarkerTracking property of Plots.
//"LMax" is local maximum, "VMin" is visible minimum, etc.

TAvgTypes = (atLog, atMag, atRMS, atCoh, atDecay);
// used for AverageType property of Praxis

TFilterShapes = (fsBrickWall,fsButterworth,fsCheb01,fsCheb1,fsBessel);
//used in praxis properties (for PostProcessing) such as
"SmoothingFilter".

TFilterType = (ftLPF,ftHPF,ftBPF,ftBRF,ftAP,ft0dB);
//used in FilterType property of Praxis

TInputSelect = (inProbe1,inProbe2,inMic1,inMic2,inOut1,inOut2,inLineX);
// used for InputSelect property of Praxis

TMathOp = (moSum,moDiff,moMult,moDiv,moMax,moMin,moMagInv,moPhaseInv);
//for Math PostProcess

TPasteMode = (pmAbove,pmBelow,pmClearTarget,pmAll);
//for PasteOver PostProcess

TPrintStyles = (psBW, psColorLines, psColorFill);
//used with Print method of Plots

TRemapFormat = (rfIfftCompat,rfLogSpace,rfFreqList,rfFromSource);
//for Remapping PostProcess

TWndKinds =
```

```
(wnNone,wnRect,wnBH,wnBHh,wnH,wnHh,wnH7,wnH7h,wnHM,wnHMh,wnBG,wnBGh,wnFT
,wnFTh, wnHN, wnHNh);
  //None, Rectangular, Blackman-Harris, Half Blackman-Harris, Hamming,
  // Half Hamming,Hodie7, Half Hodie7, Hodie5, Half Hodie5, Bingham,
  // Half Bingham, FlatTop, HalfFlatTop, Hann, Half Hann.
  //-These are used with Praxis's SetWindowType procedure, and with its
  //WindowType and PosProcWindowType properties.
```

Object Types

- **`TDataPlot`**: this is the type used for Praxis' "Plots". Never use Create, Destroy, or Free with one of these! Use Praxis properties (Praxis.**`PostProcSource, Praxis.PostProcTarget, Praxis.PrimaryPlot, Praxis.AuxPlot`**) to access their properties. Use function **`Praxis.GetNewPlot`** to open a new plot. Use **`Praxis.DeletePlot`** or **`Praxis.CloseAllSecondaryPlots`** to remove secondary plots.

.

Global Functions

These functions may be useful for math operations and for text (string) formatting in Praxis script programming.

function **Angle**(Real,Imag:double):double;
// returns angle in degrees of complex number (Real + i*Imag)

function **AmplifiedOutVoltage**(dBOutLevel:double):double;
// returns the peak possible output voltage, after the power amplifer (which has gain of Praxis.PowerAmpdBGain) assuming that Praxis.OutputGain for the desired channel is set to dBOutLevel. Possible power amplifier clipping is not considered.

Procedure **BitmapFileToPrinter(**s:string; UsePrintDialog:boolean=true);
// prints the bitmap file to the current default printer. If UsePrintDialog is true, it
// lets user use the dialog to change settings.

procedure **CopyFileFromTo**(ff,ft:string);
// ff and ft are complete filenames, with paths. Copies the file ff to a new file ft.

//complex math functions, using real,imaginary format*. The type CXN is a record type, defined by*

```
CXN = record
   rep,imp:double;
 end;
```

function **cpxadd**(x,y:CXN):CXN; //x+y
function **cpxsub**(x,y:CXN):CXN; //x-y
function **cpxmult**(x,y:CXN):CXN;//x*y
function **cpxdiv**(x,y:CXN):CXN; //x/y
function **cpxscale**(x:CXN;s:double):CXN; //x*s
function **cpxinv**(x:CXN):CXN; //1/x

function **dBV**(x:double):double;
// returns 20*log(x). Result will be limited to +/-1E5

function **EngFloatToStr(**x:double):string;
// returns x in Engineering notation (for example 0.012 will give "12.00m")

function **FloatLimits**(max,value,min:double):double;
// returns value if it is within min..max. Else, returns min or max.

function **FloatToMinStr**(x:double;digs:integer):string;
// limits output to digs digits after decimal, and if any trailing characters are zeroes,
// they are removed.

function **GetScriptParameters**:string; Returns the string (if any) set by a previous call to SetScriptParameters, and then empties the string. Used to send a string value between scripts (when one script starts another using Praxis.NextScriptToRun).

function **ImagVal**(Mag,Angle:double):double;
// returns the Imaginary part of the complex number Mag/_Angle, angle in degrees

function **IncDecString**(s:string;GoUp:boolean):string;
// Used to generate a series of serial numbers. The string will be changed so
// that any positive numerical characters at the end of the string will be
// incremented or decremented. For example: “MySN1234”-->”MySN1235”

procedure **InOrder**(var v1,v2: double);
// swaps V1,V2 if V2<V1

function **InputGainSettingNeeded**(InChannel:integer;VoltsPeakExpected:double):double;
//returns the dB gain setting that should be used for Praxis.InputGain[Channel] to accommodate an input voltage of VoltsPeakExpected.

function **InputVoltageToClip**(InChannel:integer;InPotGain:double):double;
//returns the input voltage at the given channel and the given input slider setting that would overdrive the input. InPotGain is in dB (and is <=0). Use this to determine whether an expected input level is within the linear input range of the system.

function **IsDoing**(s:string):boolean; //see SetDoing, below

function **IsEven**(i:integer):boolean; //returns true if i is an even integer value.

function **IsInDoing**(s:string):boolean; //see SetDoing, below

function **IsPositive**(Value:double):boolean;
// true if Value >=0

function **IsWin2kPlus**: Boolean; //returns true if the operating system being used is Windows2000 or newer.

procedure **KillLog**; //Call this procedure to clear the data displayed in the "debugging log form" and to hide that form.

function **linterp**(x,x1,y1,x2,y2:double):double;
// linearly interpolates y(x), given 2 defined values y1(x1) and y2(x2)

function **loginterp**(x,x1,y1,x2,y2:double):double;
// interpolates y(log(x)), given 2 defined values y1(log(x1)) and y2(log(x2)). ;x, x1, and x2 must all be positive nonzero values.

procedure **LogBool**(b:boolean;s:string=''); //Call this procedure to add the value of the boolean b (and, if given, the string s) to the messages shown on the debugging log form and to make the form appear if it is not displayed.

procedure **LogDouble**(d:double;s:string=''); //Call this procedure to add the value of the floating point number d (and, if given, the string s) to the messages shown on the debugging log form and to make the form appear if it is not displayed.

procedure **LogInt**(i:integer;s:string=''); //Call this procedure to add the value of the integer i (and, if given, the string s) to the messages shown on the debugging log form and to make the form appear if it is not displayed.

procedure **LogString**(s:string); //Call this procedure to add the string s to the messages shown on the debugging log form and to make the form appear if it is not displayed.

function **LPTin**(LptNumber:integer;var value:byte):boolean;
// Provides direct READ ACCESS of an LPT(parallel printer) port. This works only in
// Windows2000 or WindowsXP (and not in all computers –for more reliable I/O use a
// Labjack device). Returns true if successful, and the byte value that is
// read is returned in the variable "value". See NumberOfLPTPorts

function **LPTout**(LptNumber:integer;value:byte):boolean;
// Provides direct WRITE ACCESS to an LPT (parallel printer) port. A printer should
// NOT be connected to that port! This is intended for controlling external hardware
// using the 8 data lines of the port. . This works only in Windows2000 or
// WindowsXP (and not in all computers –for more reliable I/O, use a
// Labjack device). See NumberOfLPTPorts.

function **Mag**(Real,Imag:double):double;
// returns the magnitude of complex number (Real + i*Imag)

procedure **MicDatFileSensitivityAdj**(fn:string;dB:double);
//Adjusts the Microphone data correction file "fn" so that it causes the mic's sensitivity to be dB higher. The file will be modified, and a backup file of the original will be created.

function **MoveOctaves**(Oct,Value:double):double;
  //takes Value and changes it by Oct octaves (i.e., if Oct is +1, doubles it, if Oct is -1, cuts it in half)

Function **NowDoing**:string; // see SetDoing, below.

function **NumberOfLPTPorts**:integer;
//returns the number of LPT (parallel printer) ports in the computer.
// Works in Windows2000 or WindowsXP only.

function **OctaveChange**(Ffrom,Fto:double):double;
  //if moving -x octaves going from Ffrom to Fto, result is -x

function **OperatingSystemName**:string;
 //returns a string identifying the Windows operating system in use.

function **PadOut**(s:string;slength:integer):string; //returns string of length
// returns a string, based on s, that is exactly slength characters long (extra space at end is filled with blanks)

function **PointInRect**(const x,y:integer; ARect:TRect):boolean;

// For graphic programming. Returns true only if the point defined by x,y is within the ARect area (see Delphi docs);

function **PrimaryPlot**:TDataPlot;
  // This returns the ~~Primary Plot~~ {linkID=9000}object, which is where newly acquired data is placed during a measurement. For instance, when you make an RTA measurement, PRAXIS draws the RTA spectrum result in the Primary Plot. The function PrimaryPlot can be used interchangeably with the PRAXIS property "PrimaryPlot".
  example usage:
     PrimaryPlot.AutoAll; //DelphiScript. Autoscales the primary plot:
     Praxis.PrimaryPlot.AutoAll; //this DelphiScript code does the same thing.
      PrimaryPlot.AutoAll 'VBScript

function **RealVal**(Mag,Angle:double):double;
// returns the Real part of the complex number Mag/_Angle, angle in degrees

function **ReadPortByte**(PortAddr:Integer):Byte;
  // provides DIRECT READ of hardware ports of the computer. Use extreme caution, this lets
  // you do things that Windows does not normally allow, and can cause trouble if used incorrectly.
  // PortAddr is the address of the port, be sure to give the decimal (not hexadecimal!) value for
  // the port. Also see function LPTin

function **RoundToDecimalPoints**(x:double;DecPts:integer):double;
// returns the given value x, rounded to DecPts decimal points.

function **safediv**(num,den:double):double;
// gives num/den, but is protected from divide by zero (instead returns a huge value)

function **safeExp**(x:double):double;
// returns e^x, protected from overflow error.

function **safeExp2**(x:double):double;
// returns 2^x, protected from overflow error.

function **safeln**p(x:double):double;
// returns base elogarithm, but protects from overflow error (by returning huge values)

function **safelog**(x:double):double;
// returns base10 logarithm, but protects from overflow error (by returning huge values)

function **safelog2**(x:double):double;
// returns base2 logarithm, but protects from overflow error (by returning huge values)

function **saferound**(x:double):integer;
// returns value x rounded to the nearest integer (but limits at +/-2^31)

function **safeTenToPwr**(x:double):double;
//returns 10^x, with math error protection.

procedure **SavePraxisConfig**(Fname:string); Saves the current confguration settings to a reloadable configuration file in the fully specified (path and filename) file Fname.

procedure **ScriptConfig**(Fname:string);
// A very handy procedure. When your script starts, you can call ScriptConfig with the name of a PRAXIS software configuration file that is in the same folder as your script. PRAXIS will then save its current configuration (as existed when the script started) and load the new configuration

(making a huge number of program settings for you). When your script (or any other scripts called from our script) exits, PRAXIS will then automatically restore its original configuration without further programming effort.
If Fname is blank (or an empty string), then PRAXIS will still save its current configuration (and restore it after your script is done).

Procedure **SetDoing**(s:string);
// sets an internal string (called "DOING") to the uppercase converted version of string "s". The internal string (accessed with functions IsDoing, IsInDoing, or NowDoing) is useful within scripts to keep track of what the script is trying to do at any time and to determine different actions to be performed within event handler procedures. An event handler could be entered at various times while the script is running, and the string variable will let your script code keep track of changing situations.
Examples:
SetDoing('Measuring First'); // the internal DOING string is set to "MEASURING FIRST"
If IsInDoing('Measuring') then … //true, because the DOING string contains "MEASURING"
If IsDoing('Measuring Second') then…. //false because DOING is not "MEASURING SECOND"
LogString(NowDoing); //shows the DOING string in a debug message

procedure **SetScriptParameters**(s:string);
// used to send parameters (such as the name of a file or directory) between scripts using a string. The next script to run can access the string by using the function GetScriptParameters (which then empties the string).

function **ShellExecute**(hwnd:HWND,lpOperation:LPCTSTR, lpFile: LPCTSTR,lpParameters: LPCTSTR , lpDirectory: LPCTSTR, nShowCmd: INT): HINSTANCE;
//Use this Windows API function to start other Windows programs from within your script. The arguments to the function are not easily explained (see Windows programming documentation if detail is needed) but can be illustrated well enough for many uses using some Delphi examples:

```
ShellExecute(Handle, 'open', 'MSpaint', nil, nil, SW_SHOW);

//open a specific file with a specific applet:
  s:= ""+Praxis.ScriptPath+'Picture.bmp'+"";
  ShellExecute(Handle, 'open', 'MSpaint', s , nil, SW_SHOW);

ShellExecute(Handle, 'open', 'Wordpad', nil, nil, SW_SHOW);
ShellExecute(Handle, 'open', 'sndvol32', nil, nil, SW_SHOW);

//To Open a registered filetype from its registered application:
  s:=""+Praxis.ScriptPath+'Description.rtf'+"";
  ShellExecute(Handle, 'open', s, nil, nil, SW_SHOW); //wordpad, word, or..?

//This can even open a file into PRAXIS:
  s:=""+Praxis.ScriptPath+'planar.px2'+""; //"px2" is registered to PRAXIS
  ShellExecute(Handle, 'open', s, nil, nil, SW_SHOW);
```

function **ShortenFilePath**(p:string):string;
//strips off the path stuff if it is under the directory in which Praxis is stored.
// For example, "C:\Program Files\Praxis\SomeDir\Whatsit.xyz" would become
// "SomeDir\Whatsit.xyz". So filenames are portable within scripts.

function **SignOf(**Value:double):double;
// If value is positive or 0, =1, else =: -1

function **StripLeadingBlanks**(s:string):string;
// returns s with any leading blanks removed

function **StripOutsideBlanks**(s:string):string;
// returns s with both leading and trailing blanks removed

function **ToPwr2**(x:integer):integer;
// returns the equal or next LOWER power of two value near x

function **ToHigherPwr2**(x:integer):integer;
// returns the equal or next HIGHER power of two value near x

function **ToNeatMult**(x:double;r:double):double;
// returns value, nearest x, which is a multiple of r (r is usually from a 1, 2,5 pattern value)

function **VdBToScalar**(x:double):double;
// returns 10^(x/20), result limited to always above 1E-20

function **Within**(max,value,min:double):boolean;
// returns true if value is between max and min (max need not be >=min

procedure **WritePortWord**(PortAddr:Integer;Value:Word);
// provides DIRECT WRITE ACCESS to hardware ports of the computer. Use extreme
// caution, this lets you do things that Windows does not normally allow, and can
// cause trouble if used incorrectly. PortAddr is the address of the port,
// be sure to give the decimal (not hexadecimal!) value for the port.
// Also see function LPTout.

Controlling LabJack hardware

The LabJack U12 (hereafter called the "LabJack" in this documentation) is a USB connected device that allows software, such as a PRAXIS Script, to input and output digital or slow analog signals for controlling or monitoring external devices. It also provides a low-current +5VDC power supply from the USB bus. The LabJack can be used for custom test fixtures, indicators, or DC voltage measurement. LabJack U12 devices can be purchased through Liberty Instruments, Inc or through www.LabJack.com. Further information about the LabJack U12 and other accessories along with a detailed user manual can be found at the LabJack web site.

The LabJack has 20 digital I/O pins, four of which are protected against over voltages and overloads, and available as screw terminals (IO0-IO3). The other 16 digital I/O pins (D0-D15) are available on the "D" connector. These 16 **D0-D-15 pins are NOT protected** against overdrive or overload, so you must make sure to:

- Never drive any of lines D0-D15 high from an external source if that pin has not already been configured as in input. Configure it as input (before it is connected), then drive it. If this condition cannot be assured, you can use series resistors of approximately 1000ohms between the lines and the logic input being driven to provide protection, provided the driving source remains in the range 0 to +5VDC. Use of these resistors is highly recommended.
- Never drive any of the D0-D15 pins beyond the range 0 to +5VDC.
- Never load any of the D0-D15 pins with a load of less than 200 ohms, nor draw more than 25mA current from the pins. A degree of protection from overload can be obtained if you always connect these lines to external circuitry via series resistors of approximately 1000 ohms.
- An inexpensive protection interface (model CB25) is also available to provide overdrive and overload protection for lines D0-D15. The CB25 can be purchased from LabJack Corporation.

LabJack U12 DB25 Pinouts:

1: D0	**6: D5**	**11: +5V**	**16: GND**	**21: D11**
2: D1	**7: D6**	**12: +5V**	**17: GND**	**22: D12**
3: D2	**8: D7**	**13: +5V**	**18: D8**	**23: D13**
4: D3	**9: NC**	**14: GND**	**19: D9**	**24: D14**
5: D4	**10:+5V**	**15: GND**	**20: D10**	**25: D15**

When controlling the LabJack device from PRAXIS, there are two possible sets of controls that can be used: one is the "LabJackS" object, and the other is via the global "LJ_" functions. It is strongly recommended that you use only one of these command sets within a PRAXIS script for controlling digital lines, as mixing calls between the two sets can cause incorrect configurations for the LabJack's digital I/O pins.

The LabJackS (or LJ) Object

The first and easiest method of controlling a single LabJack device is to work through the built-in software object called "LabJackS" (or "LJ"). This LabJackS object can be used only when just one single LabJack device is connected to the computer, and provides for very simple, but more limited input/output control. The names "LJ" and "LabJackS" both indicate the same instance of the same object, and can be used interchangeably. If you refer to this object using "LJ", however, be careful not to confuse that object or its methods with the set of functions whose names begin with "LJ_". The "LJ" object is *not* the same and is not entirely compatible with use of the "LJ_" functions!

When the LabJackS (or LJ) object is used, digital bits D0 through D7 of the LabJack's digital bus are used only as outputs, while bits D8 through D15 of the LabJack's digital bus are used only as inputs. In the documentation below, the LabJackS object name is included in names of the methods and properties to emphasize the form in which they should be used.

LabJackS.OutByte:Integer(Write). This write-only property can be used to write one byte of signals to the lowest order 8 bits (bits D0 through D7) of the LabJack's digital signals. Using this configures all of the lowest 8 digital bits of the LabJack to be logic output signals and sets them to the binary value contained in the lowest 8 bits of the value. DO NOT USE THIS PROPERTY UNLESS YOU ARE SURE THAT NONE OF THE BITS D0 THROUGH D7 ARE BE BEING EXTERNALLY FORCED TO A FIXED STATE. (IN OTHER WORDS, MAKE SURE NONE ARE GROUNDED OR TIED TO A POWER SUPPLY VOLTAGE). USE OF 1000 OHM SERIES RESISTORS IS RECOMMENDED.
Example:

```
LabJackS.OutByte := 63; //DelphiScript
LabJackS.OutByte = 63  'VBScript
```

LabJackS.InByte:Integer(Read). This read-only property can be used to read on byte of signals from data input array (bits D8 through D15) of the LabJack interface. The state of digital line D8 of the LabJack will be indicated in bit 0 of the returned integer, line D1 of the LabJack will be indicated in bit 1 of the returned integer, etc. Bits 8 through 31 of the returned integer will be 0. . Using this property configures all of the second lowest 8 digital bits of the LabJack to be logic input signals. PRIOR TO USING THIS PROPERTY, THE CONFIGURATION DIRECTION OF THESE LINES MAY NOT BE KNOWN. DO NOT ATTEMPT TO DRIVE THESE LINES HIGH FROM AN EXTERNAL SOURCE UNLESS YOU HAVE FIRST USED THIS PROPERTY ONCE, OR UNLESS YOU CONNECT TO THE LINES VIA SERIES PROTECTION RESISTORS (1000 ohms recommended).
Example:

```
OutsideWorldByte := LabJackS.InByte; //DelphiScript  -- result will be no higher than 255
OutsideWorldByte = LabJackS.InByte   'VBScript
```

LabJackS.OutIO[line:integer]:integer (write). This indexed, write-only property is used to configure and define the logic state of the protected screw terminal lines IO0, IO1, IO2 or IO3. When this property is set, that one line will be configured as an output line (the remaining three are unchanged). If the written value is >0, the line will be set, otherwise it will be cleared.
Example:

```
LabJackS.OutIO[0] := 1; //DelphiScript – sets IO0 to the output high state
LabJackS.OutIO[3] := 0; //  clears IO3 to the output low state
LabJackS.OutIO(0) = 1  'VBScript, sets IO0
LabJackS.OutIO(3) = 0  '  clears IO3
```

LabJackS.InIO[line:integer]:integer (read). This indexed, read-only property is used to configure and read the logic state of the protected screw terminal lines IO0, IO1, IO2 or IO3 . When the value of this property is referenced, that one line will be configured as an input line and the returned value will be either 1 if the line was high or 0 if the line was low.
Example:

```
Avalue := LabJackS.InIO[0]; //DelphiScript – Avalue is changed to a 1 if IO0 is high, otherwise it
      is changed to 0.
Avalue = LabJackS.InIO(0)  'VBScript example
```

LabJackS.OutVoltage[line:integer]:single (write). This indexed, write-only property is used to define the analog output DC voltage at one of the screw terminal line AO0 (when line=0) or AO1 (when line=1). These voltages are generated by the LabJack using a 10-bit DAC and the voltages can be configured for between 0 and 5 Volts.
Example:

```
LabJackS.OutVoltage[0] := 2.876; //DelphiScript – the voltage at line AO0 will change to
2.876Volts.
LabJackS.OutVoltage(1) = 3.210  'VBScript example, sets AO1 to 3.210 Volts.
```

LabJackS.InVoltage[line:integer]:single (read). This indexed, read-only property is used to read the analog DC voltage at one of the eight screw terminal lines AI0, AI1, … AI7 or to read the differential DC voltage at one of four pairs of screw terminals. The voltages are converted using 12-bit ADCs. This allows the LabJack to function as a multi-input DC voltmeter.

Single ended analog input operation: when the value of "line" is set in the range 0 to 7, the voltage read will be from AI0, A1, A2… or AI7 respectively, and the usable input range is +10V to -10V.

When the value of "line" is set in the range 8 to 11, then the voltage read will be the difference between the voltages at the A0-A1, A2-A3, A4-A5, or A6-A7 respectively. The voltage of each terminal with respect to ground must be within +10V to -10V, but the usable voltage difference between the terminals depends on the AnalogGain property setting (LabJackS.AnalogGain). When using LabJackS.InVoltage in differential mode, the LabJackS.AnalogGain property for the desired pair of pins should configured before reading the voltage. The usable voltage differences are as given in the following table:

LabJackS.AnalogGain[]	**Range**
1	**±20 volts**
2	**±10 volts**
4	**±5 volts**
5	**±4 volts**
8	**±2.5 volts**
10	**±2 volts**
16	**±1.25 volts**
20	**±1 volts**

Example:

```
(DelphiScript:)
  LabJackS.AnalogGain[10]:=4; Avoltage := LabJackS.InVoltage[10];
    //Avoltage is set to the difference between terminals AI4 and AI5,
    // -5V to +5V
  Avoltage := LabJackS.InVoltage[4];
    //Avoltage is set to the voltage at AI4, -10V to +10V
(VBScript:)
  LabJackS.AnalogGain(10) = 4
  Avoltage = LabJackS.InVoltage(10)
    ' Avoltage is set to the difference between terminals AI4 and AI5,
    ' -5V to +5V
  Avoltage = LabJackS.InVoltage(4)
    ' Avoltage is set to the voltage at AI4, -10V to +10V
```

LabJackS.AnalogGain[line:integer]:integer (read/write). This indexed property is used to read or to define the analog gain that will be applied when using the LabJackS.InVoltage to read differential voltages from pairs of AI input terminals. Values of the index "line" must be in the range 8 to 11 (to match the index of the corresponding index used with InVoltage for differential input). This property has no effect when InVoltage is used to read single-ended inputs (non differential inputs using an index less than 8). This gain setting can be used to optimize the resolution and range of the differential voltage inputs. It does not affect the scale of the voltage that is read. For instance, if the value of AnalogGain is set to 1 and a 2.0 volt difference is applied to the corresponding set of AI terminals, the value read by InVoltage will be about the same than as if AnalogGain had been set to 8. But in the first case, the measurement resolution will be 9.8mV, while in the second case the resolution would be much more precise at 1.22mV. Valid values that can be assigned are 20, 16, 10, 8, 5, 4, 2, or 1. See the description of the LabJackS.InVoltage property for usable voltage ranges with each gain setting.
Example:

```
  LabJackS.AnalogGain[9] := 10;  //DelphiScript
  LabJackS.AnalogGain(9) = 10  'VBScript
```

LabJackS.Counter:integer (read). This property returns the current value of the LabJack's internal counter. This hardware counter is incremented on a rising edge at the LabJack's CNT terminal. Triggering signals should pull the CNT line low, then high for each increment.

```
  NumberOfPulses := LabJackS.Counter; //DelphiScript
  NumberOfPulses = LabJackS.Counter  'VBScript
```

Procedure **LabJackS.ResetCounter**. This procedure resets the Labjack counter back to zero.

The LJ_ global functions

The second, more versatile, method of controlling a LabJack device from PRAXIS is through a set of global functions that essentially expose the DLL functions of the LabJack driver. You should use this method only if you need to connect more than one LabJack U12 device to the computer, or if you need to configure the digital input and output pins of the LabJack device differently than the way the simpler LabJackS configures them. The disadvantage of using these functions is that each function call must typically specify a larger number of parameters, making it less intuitive to use for quick programming. Use of the LJ_ global functions will also require greater programming skill and some ability to interpret the LabJack's C-language based documentation.

The LJ_ global functions whose names begin with "LJ_E" are typically simpler to use (though not as simple as the LabJackS object properties), and are generally preferred unless the most detailed control is required.

Some of the lower-level LJ_ functions require use of array parameter types that have been defined internally within PRAXIS, as listed below in the Pascal language :

```
t4Integers = array[0..3] of Integer;
tErrorString = array[0..49] of Char;
tVoltages = array[0..3] of Single;
tVBuffer = array[0..4095] of tVoltages;
tIOBuffer = array[0..4095] of Integer;
tCalMatrix = array[0..19] of Integer;
tCalMatrixList = array[0..126] of tCalMatrix;
t127Integers = array[0..126] of Integer;
```

For detailed description of the LJ_ global functions, please see the LabJack U12 User's Guide. For use within PRAXIS, the function names are changed slightly from those given in the guide, in that each function name is augmented with the prefix "LJ_". For example, the PRAXIS function "LJ_EDigitalOut" will be documented in the LabJack U12 User's Guide as "EDigitalOut". This is done to avoid confusion with other similarly named functions that may be used in PRAXIS scripts. The available LJ_ global functions available from PRAXIS are listed below :

function **LJ_EDigitalOut**(var idnum:integer; demo:integer; channel:integer; writeD:integer; state:integer):integer; DO NOT MIX THIS FUNCTION WITH LABJACKS OBJECT USAGE.

function **LJ_EDigitalIn**(var idnum:integer; demo:integer; channel:integer; writeD:integer; var state:integer):integer; DO NOT MIX THIS FUNCTION WITH LABJACKS OBJECT USAGE.

function **LJ_EAnalogIn**(var idnum:integer; demo:integer; channel:integer; gain:integer; var overvoltage:integer; var voltage:single):integer;

function **LJ_EAnalogOut**(var idnum:integer; demo:integer; analogOut0:single; analogOut1:single):integer;

function **LJ_ECount**(var idnum:integer; demo:integer; resetCounter:integer; var count:double; var ms:double):integer;

function **LJ_AISample**(var idnum: Integer; demo: Integer; stateIO: Integer; updateIO: Integer; ledOn: Integer; numChannels: Integer; channels: t4Integers; gains: t4Integers; disableCal: Integer; var overVoltage: Integer; var voltages: tVoltages): Integer;

function **LJ_AIBurst**(var idnum: Integer; demo: Integer; stateIOin: Integer; updateIO: Integer; ledOn: Integer; numChannels: Integer; channels: t4Integers; gains: t4Integers; var scanRate: single; disableCal: Integer; triggerIO: Integer; triggerState: Integer; numScans: Integer; timeout: Integer; var voltages: tVoltages; var stateIOout: Integer; var overVoltage: Integer; transferMode: Integer): Integer;

function **LJ_AIStreamStart**(var idnum: Integer; demo: Integer; stateIOin: Integer; updateIO: Integer; ledOn: Integer; numChannels: Integer; channels: t4Integers; gains: t4Integers; var scanRate: Single; disableCal: Integer; reserved1: Integer; reserved2: Integer): Integer;

function **LJ_AIStreamRead**(localID: Integer; numScans: Integer; timeout: Integer; var voltages: tVBuffer; var stateIOout: tIOBuffer; var reserved: Integer; var ljScanBacklog: Integer; var overVoltage: Integer): Integer;

function **LJ_AIStreamClear**(localID: Integer): Integer;

function **LJ_AOUpdate**(var idnum:integer; demo: Integer; trisD: Integer; trisIO: Integer; var stateD: Integer; var stateIO: Integer; updateDigital: Integer; resetCounter: Integer; var count: LongWord; analogOut0: Single; analogOut1: Single): Integer;

function **LJ_BitsToVolts**(chnum: Integer; chgain: Integer; bits: Integer; volts: Single): Integer;

function **LJ_VoltsToBits**(chnum: Integer; chgain: Integer; volts: Single; var bits: Integer): Integer;

function **LJ_Counter**(var pl_idnum:integer; demo: Integer; var stateD: Integer; var stateIO: Integer;resetCounter: Integer; EnableStb:integer; var count: LongWord): Integer;

function **LJ_DigitalIO**(var pl_idnum:integer; demo: Integer; var trisD: Integer; trisIO: Integer; var stateD: Integer; var stateIO: Integer; updateDigital: Integer; var outputD: Integer): Integer;
DO NOT MIX THIS FUNCTION WITH LABJACKS OBJECT USAGE.

function **LJ_GetDriverVersion**: Single;

procedure **LJ_GetErrorString**(errorcode: Integer; var errorString: tErrorString);

function **LJ_GetFirmwareVersion**(var idnum: Integer): Single;

function **LJ_ListAll**(var pl_productIDList: Integer; var serialnumList: t127Integers; var localIDList: t127Integers;var powerList: t127Integers; var calMatrixList: tCalMatrixList; var numberFound: Integer; var fcddMaxSize: Integer; var hvcMaxSize: Integer): Integer;

function **LJ_LocalID**(var idnum: Integer; localID: Integer): Integer;

function **LJ_ReEnum**(var idnum: Integer): Integer;

function **LJ_Reset**(var idnum: Integer): Integer;

function **LJ_Watchdog**(var idnum: Integer; demo: Integer; active: Integer; timeout: Integer; reset: Integer; activeD0: Integer; activeD1: Integer; activeD8: Integer; stateD0: Integer;stateD1: Integer; stateD8: Integer): Integer;

function **LJ_ReadMem**(var idnum: Integer; address: Integer; var data3: Integer; var data2: Integer; var data1: Integer; var data0: Integer): Integer;

function LJ_WriteMem(var idnum: Integer; unlocked: Integer; address: Integer;: Integer; data2: Integer; data1: Integer; data0: Integer): Integer;

Events Triggered Within Praxis

You can "Handle" the following events, to respond to certain occurrences happening within Praxis operation. To do this, you must write an event handler procedure (or subroutine). The event handler can be a method (with the specified name) of your main form. Event handlers for Events triggered within Praxis always have one single parameter (called "Sender").

For example, if your main form is called "DWform" (of class "TDWform") then, you might have the following event handler method in your **DelphiScript** code.

```
procedure TDWform.InitScript(sender);
{Be sure to include this method in your Class type declaration!}
begin
 If Praxis.DemoMode then ShowMessage('Welcome Demo Mode user!');
end;
```

In **VBScript**, it would look like this:

```
Sub TDWform.InitScript(Sender)
```

```
  If Praxis.DemoMode then ShowMessage("Welcome Demo Mode User!");
End Sub
```

In Version 2 of the Script Designer, you can automatically create a script from a template that already has blocks for handling all these events. You can follow the sequence of occurance for these events by placing a "LogString" call in each event handler.

_____

The Events are listed here under the method names you should use for them in your code:

- **InitScript(sender)**: This happens immediately after the script is loaded and displayed. This is the event in which to do any settings to the Praxis properties.
- **AcqBurst(sender)**: This happens after Praxis collects an array of data samples and does basic Acquisition processing. For example, in a Stepped Frequency measurement, this happens after data is collected for each applied frequency step.
- **AcquiredOnce(sender)**: This happens every time an Acquisition process has completed. If a number of them are being averaged, it happens on each averaged event.
- **AcqAvgSetDone(sender)**: This happens, if the **AutoStop** property is true, when the entire specified number of averages have been included in the acquisition data.
- **AcqStarted(sender)**: This happens when a Praxis data acquisition is started.
- **AcqStopped(sender)**: This happens when a Praxis data acquisition has stopped.
- **BeforeAcqPlot(sender)**: This happens just before newly acquired data is about to be plotted to the Primary Plot.
- **AfterAcqPlot(Sender)**: This happens just after newly acquired data is plotted to the Primary Plot.
- **UserRedrawPlot(Sender)**: This happens when a plot has its PlotScheme property set to 1, and its bitmap "canvas" needs to be redrawn. It is intended so advanced scripts can draw their own graphics in the resizable plot's graphing area. The single argument of the Event Handler, "Sender", is the plot (a TDataPlot object) which is causing the event and which needs to have its bitmap canvas redrawn. Your method should use the Sender parameter to access the BmpCanvas property of that plot (and can find its dimensions using BmpHeight and BmpWidth), and use Delphi methods to draw your graphics
- **UserAnnotatePlot(Sender)**: This event is called when a plot's bitmap "canvas" has been drawn by Praxis. It allows you to modify the plot bitmap (by adding text, lines, etc. to the bitmap). The single argument of the Event Handler, "Sender", is the plot (a TDataPlot object) which is causing the event and which needs to have its bitmap canvas redrawn. Your method should use the Sender parameter to access the BmpCanvas property of that plot (and can find its dimensions using BmpHeight and BmpWidth), and use Delphi methods to draw your graphics. An example of the use of this event can be found at http://www.libinst.com/praxis_script_writers.htm#Minimal%20Script .
- **AutoLevelDone(Sender)**: This event happens when an automatic input level adjustment process has completed (with the AudPod).
- **Triggered(Sender):** This event happens when the Acquisition is configured to use triggering and the triggering event occurs.
- **AutoStoppedByTimer(Sender):** This occurs when an acquisition autostops by using the Timer setting.
- **AutoStoppedByAvgLimit(Sender):** This occurs when an acquisition autostops from reaching an Average limit (AutoStop Avgs).
- **AutoStarted(Sender);** This occurs when PRAXIS automatically starts an acquisition using the Autostart timer.
- **AppKeyUp(Sender)**: Occurs when the user presses a key while one of the PRAXIS forms has Windows focus.
- **MouseClickInPlot(Sender)**; Triggered when a mouse is clicked in a Plot. This behavior must first be enabled by the plot's "EnableMouseClickEvent" procedure. The value Sender will identify the plot object which as been clicked, and the data values in the plot (corresponding to the horizontal position of the click) can be read using the plot's MouseValue property.

Properties and Methods of the "Praxis" object

Prefix these properties and methods by the indentifier "Praxis". Example:

```
Praxis.Stimulus:=stiChirp; //DelphiScript
or
Praxis.Stimulus = stiChirp 'VBScript
```

The Properties and Methods have been divided, for convenience, into the following groups:

- General
- Data Manipulation
- Input/Output Control
- Primary Plot formatting (via the Praxis object)
- Stimulus control
- Acquisition control
- PostProcess control

.

General

Prefix these properties and methods by the indentifier "Praxis".

AbortPraxis; This procedure causes Praxis to close (after checking with the user, first).

Acquisition:integer(Read/Write). See Global Constants.

AllAcqProcStopped:boolean; This function returns true only if the data engine is not running.

AutoLevel; Attempts to automatically adjust the input levels (AudPod mode only).
AuxPlot: TDataPlot (Read); Provides access to the Auxiliar plot (a special plot used to present data that is compiled in real time from a succession of acquisitions in the Primary Plot).

CalInputIndex:integer (Read/Write). Identifies the AudPod input selection which will be used as the Cal input (for normalized measurements such as Frequency Response or Impedance). This value is NOT saved in the Praxis.Ini value, and defaults to 1 (= Probe#2) at Praxis startup. Same as "RefCalSource".

ClearValueHold; This procedure clears the ValueHold array of variant values (see entry for ValueHold, below).

CloseAllSecondaryPlots; Closes (deletes) all secondary plots. Be sure you are finished with the plots, and the data within them, before using this.

ClosePostProcessReports; Closes the small forms which show Thiele-Small extraction or Room Acoustics reports. This may be wanted within a script to minimize screen clutter.

ConfigAcqPosition(var Left,Top:integer); This procedure can be used to change or read the Left or Top positions of the Acquisition Configure form. To read the values only, but not change them, set Left and/or Top to –1 when calling the procedure.

ConfigAcqSize(var Width,Height:integer); Use this procedure to read the Width and Height of the Acquisition Configure form.

ConfigPostProcPosition(var Left,Top:integer); This procedure can be used to change or read the Left or Top positions of the PostProcess Configure form. To read the values only, but not change them, set Left and/or Top to –1 when calling the procedure.

ConfigPostProcSize(var Width,Height:integer); Use this procedure to read the Width and Height of the PostProcess Configure form.

ConfigStimPosition(var Left,Top:integer); This procedure can be used to change or read the Left or Top positions of the Stimulus Configure form. To read the values only, but not change them, set Left and/or Top to –1 when calling the procedure.

ConfigStimSize(var Width,Height:integer); Use this procedure to read the Width and Height of the Stimulus Configure form.

CreateProcess(command: string; xpos, ypos: integer): THandle; This can be used to start another application from a Praxis script. The "handle" of the application's window is returned as the result of the function (THandle is an integer value, use as a "window handle", for later accessing the opened application so it can be closed again). xpos and ypos are the screen coordinates at which to open the new application's window. For examples, see the code "ProbeCalWizardNoPod.pas" in the "ProbeCal" Demo script. Also see "Calling COM servers from Scripts" for other ways to access other applications.

DataDirectory:string (read); Contains the starting path and directory for saving and loading data files. This is the path that is shown as default in the Save/Load dialog boxes for px2 data.

DataFileLoad(Plot:TDataPlot; filename:string); Loads the data file identified by the path and name in filename, into the identified Plot.

DataFileSave(Plot:TDataPlot; filename:string):boolean; Saves the data contained in the Plot to a file identified by the path and name in filename. Returns true if successful.

DeletePlot(var DP: TDataPlot); Deletes the plot identified by DP. You cannot delete the Primary Plot. Use this method of Praxis, instead of the Free or Destroy methods of the plots (using those methods will likely cause a program crash).

DemoMode: boolean (Read/Write). True if the AudPod was NOT detected when Praxis was started, or has been removed.

DemoRtaMode: boolean (Read/Write). Demo mode, if true, will use mixer settings which the user has set up for RTA operations.

DensityOfAir:double (Read/Write). In grams per cubic meter, nominally 1210. Used in some processing calculations.

DomainView: dvType (Read/Write). Selects whether the Primary Plot shows the last acquired Frequency domain data (if still available) or the last Time domain data. Similar functions as done by the Time/Frequency Button of the Praxis Main Form.

ExitPraxisDirect. This procedure causes an immediate shutdown both the script and the

PRAXIS program itself, without asking any questions (no chance to save files or configurations). Use with caution.

FormStates: TformStates (Read/Write). This value can be saved to a global variable at script startup time (in the InitScript event handler) and restored when the script closes, to preserve the Users's chosen visibilities for the Stimulus, Acquire, and PostProcess Configure forms and the Levels Form. You may want to hide those forms (use the Show.. procedures) to minimize screen clutter while the scripts are running.

GetConfiguration(FileName:string); This procedure attempts to load the configuration file which is named in FileName. To load the Praxis.ini default Configuration file, set FileName to an empty string.

GetHardwareDependent(FileName:string); This procedure attempts to load the hardware dependent configuration file which is named in FileName. To load the Praxis.hdi default Configuration file, set FileName to an empty string.

GetNewPlot:TDataPlot; Creates and displays a new Plot. The value returned by this function can be used as an identifier to access the new plot's Properties and Methods. Sell also the "DeletePlot" procedure.

GroundResistance: double (Read/Write). Used in Demo Mode (only) for correcting impedance measurements.

KeepFormPositionsAfterScript; Call this procedure method to keep the PRAXIS forms from being restored to their before-script settings after the script exits. By default, PRAXIS will attempt to restore the form sizes and positions.

LevelsFormPosition(var Left,Top:integer); This procedure can be used to change or read the Left or Top positions of the Levels form. To read the values only, but not change them, set Left and/or Top to –1 when calling the procedure.

LevelsFormSize(var Width,Height:integer); Use this procedure to read the Width and Height of the Levels form.

LimitFileDirectory: string (Read/Write); Used to define where Pass/Fail limit evaluation curves are stored or from where they are to be used.

LineFrequency:double (Read/Write); Set this to the power line frequency in your locale. It is used to synchronize coherent stimuli so that averaging will tend to reduce the effects of power line interference in measurements.

LoadPlotFormat(Plot:TDataPlot; filename:string); Formats the specified Plot using the Plot format file named in filename. If no path information is given, the path of the current script is used. Plot format files, with suffix ".pf_", allow one set of data to use the formatting saved previously from a similar data plot.

LoadPlotPreFormat(filename: string); Pre-formats the PrimaryPlot using the Plot format file named in filename. If no path information is given, the path of the current script is used. Plot format files, with suffix ".pf_", allow one set of data to use the formatting saved previously from a similar data plot. A Pre-format operation saves the formatting information and applies it on the *next* data acquisition.

MainForm: TForm (read); Use this property to access the Left, Top, Height and Width properties of the Main Form. Example: TopPosition := Praxis.MainForm.Top;

MakeLimitCurve(DP:TDataPlot;Index,RefField:integer;IsUpper:boolean; minfreq,maxfreq:double; octaveRes:double); This is used to generate a data file named "UPPERLIMIT#.PX2": if IsUpper =True or "LOWERLIMIT#.PX2" if IsUpper=False ("#" is replaced by the characters corresponding to Index). The file will be saved into the Praxis.LimitFileDirectory (see above). RefField determines the data field in the Plot that is to be used (field 0 is normally the frequency, 1 is normally the magnitude, 2 the phase). The values "minfreq" and "maxfreq" determine the frequency range that will be mapped to make the data and octaveRes is the frequency resolution in octaves for the new data. The file is intended for use in quality control Pass/Fail testing.

Mic1FileName: string (Read/Write). The file name of the correction data file for the Mic1 input to the AudPod. The entire path and file name should be used. Writing this causes Praxis to attempt to load the file.

Mic2FileName: string (Read/Write). The file name of the correction data file for the Mic2 input to the AudPod. The entire path and file name should be used. Writing this causes Praxis to attempt to load the file.

Mic1UsesProbe:boolean (Read/Write); If this is true (and when an Audpod is used), Praxis will obtain "Mic1" microphone input signals from the Probe1 jack (rather than the tip connection of the "Mic" jack). This avoids the extra preamplification and bias voltage that is used at the microphone jack, allows for approximately 12dB greater signal headroom, and allows use of balanced microphone inputs. The calibration data for Mic1 still is applied.

NextScriptToRun: string (Read/Write). Set this to a **full** name of the desired Script Directory, to load and start up a different script (from within your script). Will not work if the Acquisition engine is Running (collecting data). And it will be cleared every time the Acquisition engine stops running, so set the value only with then Praxis is not collecting data. You can find the current base directory for Praxis Scripts using the **ScriptsDir** function.

NormalizeWeightNumpoints:integer; (Read/Write); Sets or reads the number of logarithmically spaced frequency points that will be used when generating "Normalize" data (when the user presses the Normalize button on the Weighting tab sheet of the Plot Forms).

Path: string; Returns the Path, less filename and extension, where the Praxis executable file is stored. This can be used to address other files in the same directory, or to address subdirectories where data files may be stored.

PerformPostProcess; Activates the PostProcess operation.

PlotCount:integer (Read). The number of secondary plots currently existing.

PlotExists(DP: TDataPlot):boolean; Returns true if plot DP still exists.

Postprocess:integer (Read/Write). See Global Constants.

PostProcessCfgVisible: boolean (Read/Write). Sets or hides the PostProcess Configure Form.

PrimaryPlot: TDataPlot (Read/Write). Use this to control the Primary Plot's properties and methods, when you wish to use its Plot object methods or properties. There is also a global function "PrimaryPlot" which can be used the same way. In other words, you could use these interchangeably:

```
PrimaryPlot.HorizontalStart := 20;   //using the function
Praxis.PrimaryPlot.HorizontalStart := 20; //using the PRAXIS property
```

PrintToFileGo(PS: TPrintStyles; DP:TDataPlot); Procedure to trigger a Print To File operation (user must respond to menu to supply filename).

ProbeBalance: double (Read/Write). AudPod mode only. Value used for correcting slight gain differences in the probes.

ProbeResistor: double (Read/Write). AudPod mode only. Values of the internal series resistors in the probes. OBSOLETE USE ProbeResistorFor

ProbeResistorFor[index:integer]: double (Read/Write). Value of the total series resistance in each lead of the indicated probe (1 or 2).

ReferenceResistor: double (Read/Write). Value, in ohms, of the reference resistor value to be used in making impedance measurements.

Running: boolean (Read). Indicates whether the Acquisition engine is running. Many script properties and methods can be used only when this value is false.

SavePlotFormat(Plot:TDataPlot; filename:string); Saves the formatting information (scales, etc.) to the Plot format file named in filename. If no path information is given, the path of the current script is used. Plot format files, with suffix ".pf_", allow one set of data to use the formatting saved previously from a similar data plot.

ScriptsDir: string; This function returns the top directory containing all Scripts. For AudPod mode, it is "...\Praxis\VCLScripts\". For Demo mode, it is "...\Praxis\DemoScripts\". Use it to start other scripts, for example:

```
Praxis.NextScriptToRun := ScriptsDir+'WizardModels';
//need to close this script for next one to start...
```

See also the "Path" function.

ScriptPath: string; This function returns the path, less the file name and extension, of the currently running script. It can be used to access other files in the same script directory, for example:

```
RichEdit1.Lines.LoadFromFile(Praxis.ScriptPath+'Welcome.rtf');
```

ScriptVariable[Index:integer]: variant (Read/Write). (Index can be in range: 0..32). All variables created or declared within a script become invalid after the script ends or if it calls a new script. These ScriptVariables, however, will be available during the entire Praxis session and can be used to save values between calls of scripts, so scripts can record values from past calls or pass values with other scripts (that run before or after). ScriptVariables are Pascal *variant* types, and can contain data of various other types including dynamic arrays. The ScriptVariables are initialized to *Null* when Praxis starts.

SelectedPlayDevice: string; returns the text name of the currently selected soundcard playback device.

SelectedRecDevice: string; returns the text name of the currently selected soundcard recording device.

ShowAcquireCfg(Yes:boolean); Shows or hides the Acquisition Configure form.

ShowLevelsCfg(Yes:boolean); Shows or hides the Levels Configure form.

ShowPostProcCfg(Yes:boolean); Shows or hides the PostProcess Configure form.

ShowPrimaryPlot(Yes:boolean); Shows or hides the Primary Plot.

ShowStimulusCfg(Yes:boolean); Shows or hides the Stimulus Configure form.

SpeedOfSound:double (Read/Write); In meters/second, nominally 344. Used in some processing calculations.

Start; Starts the Praxis Stimulus/Acquisition engine for measurements.

Stimulus:integer (Read/Write). See Global Constants.

Stop; Stops the Praxis Stimulus/Acquisition engine for measurements. The engine may also stop if an error occurs or if "AutoStop" is enabled.

SupportsPlay(SampRate, SampRes: integer): boolean; checks for claimed playback support from the current soundcard for the given sample rate and resolution.

SupportsRecord(SampRate, SampRes: integer): boolean; checks for claimed recording support from the current soundcard for the given sample rate and resolution.

TimeWindowBegin:double (Read/Write). Accesses the setting (in units of seconds) of the Time windowing starting edge. This is used for windowing out regions of time domain data before transformation to frequency domain. The time domain data must be from a Synchronous or ReSync measurement.

TimeWindowEnd:double (Read/Write). Accesses the setting (in units of seconds) of the Time windowing ending edge. This is used for windowing out regions of time domain data before transformation to frequency domain. The time domain data must be from a Synchronous or ReSync measurement.

UnitsMetric: boolean (Read/Write). True if users's "Preference,Units" selection is for metric units (rather than American English units).

UsesBackdrop: boolean (Read/Write). Use this to determine whether a backdrop is used behind the PRAXIS forms.

UseSingleSoundDevice; sets the system to use the same record device as is currently being used for the play device, and the record sample rate and resolution to equal the play sample rate and resolution. These are the normal settings for Praxis operation, and should be used in all scripts

UseFlexWindow:boolean (Read/Write). shows or controls whether new FFT operations will use the FlexWindow to allow for gradual transitions from windowed to unwindowed operations..

UseDashed:Boolean (Read/Write). Same function as the Dashed Gridlines menu item in the MainForm. Controls whether plots use solid or dashed gridlines.

ValueHold[index:integer]:variant (Read/Write). A storage array which can be used by Scripts to save values or to communicate with other scripts that will run later.

Data Manipulation

Prefix these properties and methods by the indentifier "Praxis".

ChangeDataSampleRate(DP:TDataPlot;Value:integer); This procedure changes the effective sample rate of the data in DP. The data in other fields is not adjusted or changed, only the values of the time parameters (if the data is in the Time Domain). For advanced use, only. Always check that DP still exists before using (use the PlotExists function of Praxis)!

ChangeNumberOfDataPoints(DP:TDataPlot; NewNumPoints:integer):boolean; This function, if the boolean result returns true, changes the number of data points in the data set which is contained in plot object DP. If the number is increased from its previous value, added values are assigned 0 (except for the frequency and time domain fields which are assigned arbitrary ascending values). For advanced use, only. Always check that DP still exists before using (use the PlotExists function of Praxis)!

GetDataPoint(DP:TDataPlot;SField,Index:integer;var Value:double):boolean; This function, if the boolean result returns true, sets the parameter Value to the Index data point (first index is 0) of the field SField, from the data set contained in plot DP. Field 1 is the frequency or time domain array. Field 2 is the first field of the data, etc. Used to read raw values from the data sets. For advanced use, only. Always check that DP still exists before using (use the PlotExists function of Praxis)!

SetDataPoint(DP:TDataPlot;SField,Index:integer; Value:double):boolean; This function, if the boolean result returns true, sets the Index data point (first index is 0) of the field SField, from the data set contained in plot DP to the value Value. Field 1 is the frequency or time domain array. Field 2 is the first field of the data, etc. Used to read raw values from the data sets. For advanced use, only. Always check that DP still exists before using (use the PlotExists function of Praxis)! Instability may result if invalid values are applied to some plot types.

Input/Output Control

Prefix these properties and methods by the indentifier "Praxis".

InputGain[channel:integer]:integer (Read/Write). Set the AudPod's input gain setting, in dB, for each input channel. Max=0dB, Min= -78dB.

InputPeak[Index:integer]: double (Read). Peak value of the last buffer from the Output A/D. This gets cleared on some internal operations. Index indicates the channel.

InputPeakRaw[Index:integer]: double (Read). Peak raw value of the last buffer from the Output A/D. This is normalized to 1.0=full scale input.

InputResolution: integer(Read/Write). A/D sample resolution, for specialized use only. Works ONLY if the user has first selected separate sound cards or separate sample resolutions for record and play using the menus in the Levels window. *For normal uses, scripts should first call Praxis.UseSingleSoundDevice, and then set sample resolution using Praxis.OutputSampleRate.* Value must be supported by current hardware (16 must always be supported). Only values of 16 or 24 are supported.

InputRms[Index:integer]: double(Read). Rms value of the last buffer from the Output A/D. This gets cleared on some internal operations. Index indicates the channel.

InputSampleRate: integer (Read/Write). A/D sample rate, for specialized use only. Works ONLY if the user has first selected separate sound cards or separate sample rates for record and play

using the menus in the Levels window. *For normal uses, scripts should first call Praxis.UseSingleSoundDevice, and then set sample rates using Praxis.OutputSampleRate.* Value must be supported by current hardware (48000 is always supported).

InputSelect[Index:integer]: TInputSelect (Read/Write). Select the input source with the AudPod for each channel (channel 2 can not be set in Synchronous mode).

InterSilence:double (Read/Write). The minimum time, in seconds, between successive stimulus bursts for non-consecutive stimulus types. Can be used for cooling-off, or to wait for reverberation decay.

LatestInputHeadroom(chan:integer):double; This function returns the decibels of headroom from the last acquistion in input channel chan (1 or 2). It can be used to determine whether the InputGain setting can be increased for an identical input.

Mic1UsesProbe:boolean (Read/Write); If this is true (and when an Audpod is used), Praxis will obtain "Mic1" microphone input signals from the Probe1 jack (rather than the tip connection of the "Mic" jack). This avoids the extra preamplification and bias voltage that is used at the microphone jack, allows for approximately 12dB greater signal headroom, and allows use of balanced microphone inputs. The calibration data for Mic1 still is applied.

Mic2UsesProbe: boolean (Read/Write); If this is true (and when an Audpod is used), Praxis will obtain "Mic2" microphone input signals from the Probe2 jack (rather than the tip connection of the "Mic" jack). This avoids the extra preamplification and bias voltage that is used at the microphone jack, allows for approximately 12dB greater signal headroom, and allows use of balanced microphone inputs. The calibration data for Mic2 still is applied. This is intended for when two balanced microphone inputs are to be used simultaneously.

OutputEnabled[index:integer]:boolean (Read/Write). Determines whether the stimulus output is allowed to come from the soundcard. *THIS VALUE IS NOT SAVED IN PRAXIS CONFIGURATIONS.*

OutputGain[channel:integer]:double (Read/Write). Output Gain, in dB, for each channel

OutputPeak[Index:integer]:double (Read/Write). Peak value of the last buffer sent to the Output D/A. This gets cleared on some internal operations. Index indicates the channel. Writing to it causes praxis to attempt to set the output for the specified output level, based on the latest buffer.

OutputResolution: integer (Read/Write). D/A sample resolution (16 or 24). Other than in specializes uses, use this to also control the A/D sample resolution (unless the user has first selected separate sound cards or separate sample resolutions for record and play, using the menus in the Levels window). *To assure the state of the system, scripts should normally call Praxis.UseSingleSoundDevice to assure one soundcard is being used.* Value must be supported by current hardware (16 must always be supported). Only values of 16 or 24 are supported.

OutputSampleRate:integer (Read/Write). D/A sample rate. Other than in specializes uses, use this to also controls the A/D sample rate (unless the user has first selected separate sound cards or separate sample rates for record and play, using the menus in the Levels window). *To assure the state of the system, scripts should normally call Praxis.UseSingleSoundDevice to assure one soundcard is being used.* Value must be supported by current hardware (48000 is always supported).

PowerAmpdBGain:double (Read/Write). The voltage gain of the power amplifier (at the present setting of its volume control). This is used in the Wavelet Burst output capability measurements and to determine linear drive levels.

PowerAmplifierPeakVCap:double (Read/Write). The maximum voltage output (peak) that the power amplifier (following the stimulus output of the Audpod) is capable of. This is used in the Wavelet Burst output capability measurements and to determine linear drive levels.

StimulusVCap:double; This function returns the maximum linear voltage that the stimulus output of the AudPod is capable of (according to the most recent calibration for the selected soundcard). For most stimulus types, this voltage will be reached when the Output Gain of stimulus channel 1 is set to 0dB.

UseAverage:boolean (Read/Write); Can be used to enable or disable the averaging function of data acquisitions. This duplicates the function of the "Averaging On" checkbox of the Acquire configuration form.

Primary Plot Formatting via the Praxis Object

Prefix these properties and methods by the indentifier "Praxis".

Note: These all act on the Primary Plot. Many of these can also be done using Properties and Methods of Plot Objects.

HorizontalCenter: double (Read/Write). Sets the center of the horizontal display range (time or frequency) of the Primary Plot. You can also do this using the Plot.HorizontalCenter property. You should normally set the HorizontalSpan first.

HorizontalSpan: double (Read/Write). Sets the span of the horizontal display range (time or frequency) of the Primary Plot. You can also do this using the Plot.HorizontalSpan property.

HorizontalStart: double (Read/Write). Sets the start of the horizontal display range (time or frequency) for the Primary Plot. You can also do this using the Plot.HorizontalStart property. Must be less than the Horizontal Stop value.

HorizontalStop: double (Read/Write). Sets the end of the horizontal display range (time or frequency) for the Primary Plot. You can also do this using the Plot.HorizontalStop property. Must be more than the Horizontal Start value.

Marker[Trace,Index:integer]: double (Read). Used to read the marker value on the Primary Plot, at the marker's currently placed location. Trace can be 1 (A) or 2 (B). Index can indicate marker 1 through 5. The indicated marker must be enabled. See description of the MarkerFrequency property. You can use the general Plot.Marker property also.

MarkerFrequency[Index:integer]: double (Read/Write). Used to place or read the position of the indicated marker (1 to 5), specified by the frequency or time domain value. The indicated marker must be enabled. If the value is not on an exact data point for the frequency, the marker will be moved to the next higher or lower valid frequency. Use MarkerNext and MarkerPrev to adjust it to an exact data point.

MarkerNext(Index:integer); Moves the specified marker to the next valid data point to the right.

MarkerPrevious(Index:integer); Moves the specified marker to the next valid data point to the left.

MarkerSearch(Max,Local:boolean;CurveIndex,MkrIndex:Integer); Will cause marker number MkrIndex (1 to 5) to search along the visible areas of trace # CurveIndex (1 or 2). If Max is true, the search will be toward a maximum value, otherwise the search will be toward a minimum value. If Local is true, the search will be only along adjoining points until no further upward slope is encounter. If Local is false, the marker will go to the highest (or lowest) displayed data point on the trace.

Stimulus Control

Prefix these properties and methods by the indentifier "Praxis".

AddStepFSpecification(Channel:integer; FreqStart,FreqStop,LevStart,LevStop:double; FreqMode,LevMode:Integer); This procedure adds a new tone to the Stepped Tone specification for the Channel (1 or 2). For FreqMode and LevMode, the constants `swmLog` or `swmLin` can be used.

ChirpLinearSweep: boolean (Read/Write). True if chirp will use a Linear Sweep. False, if it uses Log Sweep.

ChirpStartFreq: double (Read/Write). The start frequency when Chirp Stimulus is used for impedance measurements (otherwise, chirp measurements are always full-band).

ChangeStepFSpecificatio(Channel:integer; FreqStart,FreqStop,LevStart,LevStop:double; FreqMode,LevMode:Integer); This procedure changes the last added (or only remaining) tone in the Stepped Tone specification for the Channel (1 or 2). For FreqMode and LevMode, the constants `swmLog` or `swmLin` can be used.

ChirpStopFreq: double (Read/Write). The stop frequency when Chirp Stimulus is used for impedance measurements (otherwise, chirp measurements are always full-band).

ChirpTimeLength: double (Read/Write). length, in seconds, of a Chirp Stimulus.

CompensateForResp:boolean (Read/Write). If true, the acqFRandHD or acqFRandIMD acquisition types used with Chirp Stimulus will adjust measured harmonic or intermodulation distortion levels for the effect of the (simultaneously measured) frequency response of the measurement.

DenormalizeDist:boolean (Read/Write). If true, the acqFRandHD or acqFRandIMD acquisition types used with Chirp Stimulus will NOT show the distortion levels in decibels relative to the fundamental (frequency response) values, and will instead show values of the product levels.

EnvelopeLength:integer (Read/Write). The length, in samples, of the envelope to be applied to certain output stimuli. Outside this envelope, the stimuli will be 0 (no output).

EnvelopeType:integer (Read/Write). The type of envelope to be applied to certain output stimuli. 1=rasied cosine, 2=cosine squared, 3=custom, 0=None.

FreqOfOutTone(OutChan,ToneIndex:integer):double; Returns the Stepped Tone (number

ToneIndex) next to be output in channel OutChan (1 or 2). Returns -1 if an invalid parameter is used.

FreqRatio:double (Read/Write). The ratio (less than 1, typically about 0.94) between the two frequencies in a dual-frequency chirp used to measure intermodulation distortion and frequency response using an acqFRandIMD acquisition. This acquisition type is experimental at this time.

InterSilence:double (Read/Write). The minimum time, in seconds, between successive stimulus bursts for non-consecutive stimulus types. Can be used for cooling-off, or to wait for reverberation decay.

MaxDistOrder:integer (Read/Write). This determines the maximum distortion order (2 through 10) that will be analyzed using the Chirp Stimulus and the acqFRandHD or acqFRandIMD acquisition types.

MaxWindow:boolean (Read/Write). Relevant when using the Chirp Stimulus and the acqFRandHD or acqFRandIMD acquisition types. When true, the time windows used to measure each product will be automatically adjusted to the longest time period that could isolate the separate products. If false, the windowing for the time domain impulse reponse will apply (which can be used to remove echoes in some measurement circumstances, and may be needed in cases where impulse responses decay slowly). In neither case will the windowing period used be longer than the calculated longest time period for product isolation.

MlsSeqLength:integer (Read/Write). Length of MLS sequence to be used with MLS or MLSPink type stimuli. Legal range = 4096 to 524288.

SetToOneStepFSpecification(Channel:integer); This procedure deletes all but the first tone of a stepped tone specification for the indicated channel (1 or 2). Use this as the start of a new definition of a stepped tone specification, followed by use of ChangeStepFSpecification and AddStepFSpecification.

StepFParameterFile:string (Write). List File name to be used with Stepped Frequency stimulus . Setting this value causes the Stepped Frequencies to be determined "by list file". This merely names the file, it does not load it, nor check it for existence or correctness.

StepsInStepFSpecification:integer (Write). Number of steps to be used with Stepped Frequency stimulus . Setting this value causes the Stepped Frequencies to be determined "by specification". Min=2, max=1000.

Stimulus: integer (Read/Write). See Global Constants.

UseDcCancel: boolean (Read/Write). When true (and when used in acquisitions which can be coherently averaged), this will cause alternate acquisitions to occur in inverse polarity, synchronized to whole cycles of the power line frequency. The data obtained in the acquistion will be also inverted. When coherently averaged, this results in DC offsets in the measuring system (soundcard) and power line interference signals cancelling over time. When false, the inversion process is not used on alternate stimuli but the bursts are synchronized to an odd number of half cycles of the line frequency – resulting in cancellation of line interference but not of DC offsets. UseDcCancel should be set FALSE whenever distortion measurements are being made, as even harmonics will also tend to cancel.

WaveFormSetAdd(channel:integer; WavShape:Integer;Frequency:double; Amplitude:double;Phase:double; DutyCycle:double); Adds a new waveform element to the

channel (1 or 2). Frequency is specified in Hertz, Amplitude as normalized scalar (0.00 to 1.00), phase in radians. DutyCycle is 0 to 1 (normally 0.5).
The new element has a waveshape defined by:
0=sine, 1=rectangular, 2=step (use phase field for delay), 3=impulse, 4=triangle.

WaveFormSetClear(channel:integer); Clears all waveforms for the channel (1 or 2), as defined in the Waveform Component Editor.

WaveFormSetLoad(channel:integer;Filename:string); Load the set of waveforms to the channel (1 or 2), as defined in the Waveform Component Editor, from the path and file named by Filename.

WaveFormSetPeak(channel:integer;ScaleThem:boolean); Adjust the amplitudes of all the waveforms in the set for the channel (1 or 2), as defined in the Waveform Component Editor so that the sum is no larger than 1.00. If ScaleThem is true, all amplitudes are scaled in their present proportions. Otherwise, they are all set to the same amplitude.

WaveFormSetSave(channel:integer;Filename:string); Save all waveforms for the channel (1 or 2), as defined in the Waveform Component Editor to the path and file named by Filename.

WaveFormSetToSingleFullTone(channel:integer;Frequency:double); Sets the channel (1 or 2) to use a single tone at the specified frequency, with amplitude 0.707.

WaveletFrequencyStart:double (Read/Write). The starting (or fixed) frequency of a Wavelet burst stimulus.

WaveletEnvelopeShape:integer (Read/Write). Determines the shape applied to the Wavelet Burst stimulus. 0=None (continuous wave), 1=RaisedCosine, 2=CosSquared, 3=BlackmanHarris, 4=Hodie5, 5=Hodie7, 6=FlatTop.

WaveletFrequencyStep:double (Read/Write). The step (in octave fraction) used between successive stimulus frequencies when Wavelet Acquisition measurements are made as a function of stimulus frequency.

WaveletFrequencyStop: double (Read/Write). The maximum frequency of a Wavelet burst series of stimuli. Used when Wavelet Acquisition measurements are made as a function of stimulus frequency.

WaveletLength:double (Read/Write). The length (in cycles of the first tone frequency in a Wavelet Burst stimulus) of the shaped tone burst. Will be rounded to the nearest number of half cycles.

WaveletLevelStart:double (Read/Write). The starting (or fixed) level of a Wavelet burst stimulus.

WaveletLevelStep:double (Read/Write). The step (in decibels) used between successive stimulus levels when Wavelet Acquisition measurements are made as a function of stimulus level.

WaveletMaxLevelFile:string (Read/Write). The file named here, in Weight File format (ASCII: frequency, dB), will be used to limit the output stimulus level as a function of frequency. Values of 0dB or higher will not be limited. Used only if WaveletUseMaxLevelFile is true.

WaveletUseMaxLevelFile: boolean (Read/Write). Determines whether the MaxLevelFile will be used to limit output stimulus levels.

WavFile:string (Read/Write). The file name to try to use for a WAV file stimulus.

Acquisition Control

Prefix these properties and methods by the indentifier "Praxis".

AcqSize: integer (Read/Write). Sets the size of the Acquisition array which will be acquired for each screen update. *Depending on other settings, this value may be over-ridden or adjusted by the system when Acquisition starts.* This value cannot be set while the acquisition engine is running.

Acquisition:integer(Read/Write). See Global Constants.

AutoLevelStatus: integer (Read). Indicates whether an attempted AutoLevel adjustment process has completed, and the result. -1=failed, 0=in process, 1=succeeded

AvgFrameRate:double (Read). Read this property to find the update rate (frames per second) for FFT or RTA acquisitions. The value will depend on computer speed, FFT size, sample rate and resolution.

AutoStartTime: TdateTime (Read/Write). This uses the Delphi TDateTime format to specify a time and day at which PRAXIS will automatically begin acquiring (as if the Start button were pushed at that time). For this to occur, the computer must be left on (with PRAXIS active) after the setting is made, and the computer's sleep or hibernate modes must be disabled. Also, UseAutoStart must be set to True.

AutoStop: boolean (Write). Set true to make the Acquisition engine automatically stop after a total of Praxis.AverageLimit acquisitions have been included in the ongoing average. If the system automatically stops in this way, an AcqAvgSetDone event will occur.

AutoStopTime: double (Read/Write). The time in seconds after which acquisitions will stop if UseAutoStopTime is true.

AverageLimit: integer (Read/Write). The number of acquired data sets which will be included in the ongoing average result. If Praxis.AutoStop is true, acquisition will stop when this number of averages has been included.

AverageType:TAvgTypes (Read/Write). Type of averaging to use. Value must be compatible with Stimulus and Acquisition types.

AvgReset; This Procedure resets the average counter to 0, so the averaging process starts fresh with the next acquisition.

CalcDisplayedFreqsOnly:boolean; Used with SpectrumTotal or ShouldCalcTotal properties. When true, only the energy in the displayed frequency range of the Primary Plot is summed. If false, all energy in the curve is included.

FFTsize: integer (Read/Write). Sets the size of the FFT which will be used for Acquisitions. Should be specified as a power of 2 value. *Depending on other settings, this value may be over-*

ridden or adjusted by the system when Acquisition starts. This value cannot be set while the acquisition engine is running.

FindWinEdges; This procedure causes Praxis to try to find the leading and ending edges, for removing echoes from acquisitions which work via Impulse Response measurements (normally, using Chirp and MLS stimuli). Has no effect with Async stimulus types, impedance measurements, or measurements not utilizing an impulse response.

FindWinEdgesEachAcq:boolean (Write). Use to determine whether Praxis should try to find the leading and ending edges, for removing echoes from acquisitions which work via Impulse Response measurements (normally, using Chirp and MLS stimuli). Has no effect with Async stimulus types, impedance measurements, or measurements not utilizing an impulse response.

FreeRunAfterTrig: boolean (Read/Write). When true, acquisition will continue without needing further triggering (when triggering is active). When false, triggering will not occur again until conditions for another trigger event are satisfied.

HideLineBW:**integer** (Read/Write). The bandwidth value, in FFT bins (one bin is SampleRate/FFTsize) that will be hidden in an FFT acquisition for each stimulus tone when the property SpectrumHideStimulus is used.

IsolateDistortion:boolean (Read/Write). If true, and if the ReferenceIRFile is valid, then FFT or Time Domain acquisitions will be split into linear parts and distortion (and noise) parts. This is done by calculating an "Expected" signal from the stimulus and measured IR of the system being measured, and subtracting the expected (linear) signal from the measured signal (which consists of the linear result plus distortion and noise components). See "Isolating Distortion".

MaxHarmonic: integer (Read/Write). The maximum harmonic number (2 to 100) to use in analyzing Total Harmonic Distortion. Analyzed harmonics are also constrained to be below (samplerate/2).

IsTriggered: boolean (read); Indicates whether triggering has occurred (valid only if triggering is enabled and acquisition has been enabled via Start).

ReferenceIRFile:string (Read/Write). The full path and filename of the Reference Impulse Response file . This data MUST be made using a “Chirp Synchronous” stimulus and and “Impulse Resp norm via Chirp” acquisition, and with precisely the same arrangement of equipment as will be used in the corresponding Isolate Distortion FFT or Time Domain measurement. See “Isolating Distortion”.

ReferenceResponseFile:string (Read/Write). A frequency response file that is used to correct distortion analaysis done with Wavelet Burst stimuli and Wavelet type acquisitions.

RtaResolution:integer (Read/Write). For RTA acquisitions. For example, a value of 3 would give results in 1/3 octave resolution. Min value=1, max value =48. This will also, with the FFTSize property, affect how low, in frequency, the measurement can resolve.

SetExpNeedsUpdate. Call this procedure to force PRAXIS to generate a new "expected" spectrum (or time domain) data curve before doing an "Isolate Distortion" operation. This might be needed, for example, should anything in the acquisition settings changes. The expected curve must be precisely generated using the same settings and conditions as will be used for the measurement, in order for the isolation to be able to remove the linear portion of the response.

SetWindowType(SyncOrResynch: boolean; const Value: TWndKinds); Sets the type of data window to use with Async acquisitions (if SyncOrResync=false) or with Synchronous or ReSynched measurements.

ShouldCalcTotal[Index:integer]: boolean (Read/Write). Used to determine whether the channel indicated by index (1 or 2) should display its calculated average at the bottom of the Primary Plot for FFT or RTA data. Uses CalcDisplayedFreqsOnly property.

SpectrumHideStimulus[Index:integer]:integer (Read/Write). Legal values are –1, 1 or 2. Index is the input (or FFT) channel. Causes the display to hide the spectral content near the fixed tones of the indicated stimulus output channel. For example,

```
Praxis.SpectrumHideStimulus[1] := 2;
```

will cause the FFT display for channel #1 (normally "Trace A") to hide tones that are in the output channel #2. The lines are hidden within the range "HideLineBW" by showing a flattened level to the spectral data just outside that range. The purpose for this is to improve visibility of distortion content during Spectral Contamination measurements, by supressing display of intentional output tones.

SpectrumNormalize:boolean (Read/Write). If 2-channel RTA or 2-channel FFT is being used for acquisition, will cause Channel 1's data to be normalized with Channel 2's data during acquisition. This corresponds to the "Normalize" checkbox of the "Acquire Configure" form.

SpectrumTotal[index:integer]:double (Read). When used with FFT type acquisitions, this property returns the total energy in the Primary plot for the curve indicated by index. Property CalcDisplayedFreqsOnly determines whether the entire data curve (or just the region currently displayed in the PrimaryPlot) is used to calculate the sum. This calculation includes the effects of SpectrumHideStimulus, that is, this can be used to calculate only energy at non-stimulus frequencies.

StepFDomainOutCh:integer (Read/Write). Indicates which channel will be used to determine the frequency points (using that channel's Tone 1) against which the data will be plotted. Valid values are 1 or 2. For Component Tracking Acquisitions with Stepped Frequency Stimuli.

StepFField1MultA, StepFField1MultB, StepFField2MultA, StepFField2MultB :integer (Read/Write). These are the multipliers for the indicated trace (field) and tone (A or B), used to specify which harmonic or intermodulation mixing product will be tracked. See the Component Tracing Configuration Editor for details.

StepFField1Ref, StepFField2Ref :integer (Read/Write). Specifies which tone's level will be used as the reference level for determining relative (dB or %) results in component tracking for each trace (field). Tone A and B are specified by the StepFField?Tone? properties. 0=none, 1=A, 2=B.

StepFField1THD, StepFField2THD :boolean (Read/Write). If true, Total Harmonic Distortion products will be tracked for the given field (trace) and reported per tone #1 of the specified output channel (StepFDomainOutCh). If false, the frequency component to be tracked will be as specified by the other StepFField... properties.

StepFField1ToneA, StepFField1ToneB, StepFField2ToneA, StepFField2ToneB :integer (Read/Write). These identify which tone will be used (from the related StepFField?OChan? output channel) to specify which harmonic or intermodulation mixing product will be tracked. See the Component Tracing Configuration Editor for details.

TimeConstant :double (Read/Write). The time constant value used when AverageType is atDecay.

TimeOfFlightAsynch:double (Read/Write). Same as the Propagation Time parameter used for some acquisition types with Async or Chirp stimuli.

TimeOfFlightSyncFft:double (Read/Write). Propagation time, used to hold off acquisition of synchronous data when a considerable time delay exists in the measurement path.

TrigChannel: integer (Read/Write). The input channel which is monitored for the triggering level and slope.

TrigPolarity: integer (Read/Write). The polarity of the level at which Triggering will be activated. If set to zero, triggering is OFF. 1 means postive, 2 means negative, three means either. For instance, if the Triggering threshold is set to 1V and TrigPolarity is 2, then acquisition will be triggered when the voltage at the designate input crosses through -1V at the specified slope direction.

TrigPosPercent: double (Read/Write). The position in the acquisition time period at which the triggering event will be placed before display or transformation. For instance, suppose that TrigPosPercent is 25%. Then when the triggering event occurs, in the first following display of the Primary Plot, 25 percent of the information (or transformed by FFT or RTA) will be from the time before the event and 75% will be for the time after the event. If the acquisition were acqTime1, and the acquisition were to be 100msec long, then the event would be found at the 25msec point.

TrigSlope: integer (Read/Write). Specifies the direction the signal must be going in order to cause a trigger to occur. 0 means rising, 1 means falling, 2 means either.

TrigThreshold: double (Read/Write). This is used with TrigPolarity to determine the time domain value at which triggering can occur (provided the slope as the point is approached is as specified by the TrigSlope value).

UseAutoStart:boolean (Read/Write). When true, can activate an Automatic acquisition start at the time specified in AutoStartTime.

UseAutoStopTime:boolean (Read/Write). When true, acquisitions will automatically stop after AutoStopTime seconds have passed since the acquisition has started.

UseAverage:boolean (Read/Write). Reads or specifies the state of the "On" checkbox in the Averaging region of the Acquire Configure form (this checkbox controls whether acquisition averaging is to be used).

UseMaxHold:boolean (Read/Write). When true, single channel FFT and RTA curves will maintain a MaxHold response curve of the highest value encountered at each frequency point during the current acquisition.

UseMinHold:boolean (Read/Write). When true, single channel FFT and RTA curves will maintain a MinHold response curve of the lowest value at each frequency point encountered during the current acquisition.

WaveletAnalysisBandwidth:double (read/write). The octave fractional bandwidth overwhich a spectrum will be analyzed (near fundamental and harmonic center frequencies) in Wavelet type acquisitions.

WaveletAvgOptions:integer (read/write). A binary set of flags (sum the options desired) used to control how averaging is adjusted when searching for maximum levels with Wavelet Burst stimuli. 1=constant number of averages, 2=force an even number of averages, 4=scale the number according to output stimulus level, 8=scale the number of averages according to the relative response in the reference response (if used). All are adjusted relative to the number of averages

and the level of the first level and first frequency in the search.

WaveletDistResolution:double (Read/Write). Also can be called **WaveletLevelResolution**. The decibel range tolerance within which a searched-for distortion (limit value) or maximum level can be assumed to be “found”.

WaveletLimitCondition:integer (Read/Write). An integer value that determines what kind of measurement result is used to find maximum output level capability with Wavelet Burst Stimuli. 0=compression, 1=Total HD (Harmonic Distortion), 2=2nd HD, 3=3rd HD.

WaveletLimitVal:double (Read/Write). The dB limiting value that is used with WaveletLimitCondition

WaveletUseWindow:boolean (Read/Write). Determines whether PRAXIS will use time domain windowing with microphone input during wavelet acquisition analysis (Wavelet Maximum Level Capability or Wavelet Single Frequency Acquisition).

WaveletWindowMinCycles:integer (Read/Write). When WaveletUseWindow is true (and the input is from a microphone), the later edge of the time domain window will be moved out to accomodate at least this many cycles of the stimulus wavelet test frequency. Valid only with wavelet acquisition types (Wavelet Maximum Level Capability or Wavelet Single Frequency Acquisition).

WindowType[SyncOrResync:boolean]: TWndKinds (Read ONLY). The type of data window to be used with Async acquisitions (if SyncOrResync=False) or with Synchronous or ReSynched measurements (if True). To set the Window type, use the SetWindowType procedure (see above).

PostProcess Control

Prefix these properties and methods by the indentifier "Praxis".

CalculateSTI:boolean (Read/Write). Determines whether Speech Transmission Index will be calculated with other parameters when a Scroeder Curve postprocess operation is performed. You may wish to set this false if it is not needed, as the calculation can be quite time consuming.

CurveFitFreeAir:boolean; Starts a curve-fitting operation to the modeled Impedance data which is in the PostProcess Source plot, assumed to be for a woofer measured in free air. Returns true if the operation appears to be successful.

CurveFitMassLoaded:boolean; Starts a curve-fitting operation to the modeled Impedance data which is in the PostProcess Source plot, assumed to be for a woofer measured with mass loading (see TSMass, below). Returns true if the operation appears to be successful.

FilterMagOnly: boolean (Read/Write). If true, PostProcess filtering options will not affect phase response.

FilterOrder:Integer (Read/Write). For PostProcess Filter operations.

FilterShape:TFilterShapes (Read/Write). (Butterworth, etc.) For PostProcess Filter operations.

FilterType:TFilterType (Read/Write). (High pass, etc.) For PostProcess Filter operations.

LF_TruncFixMethod: integer (Read/Write). Used in the Quasi-Anechoic Bass Restoration postprocess. Valid values are 0 and 1. A value of 1 corresponds to selecting the "flat method", 0 means that option is unselected.

MathOp: TMathOp (Read/Write). For the Math PostProcess.

PasteOverMode: TPasteMode (Read/Write). For the PasteOver PostProcess.

PerformPostProcess; Activates the PostProcess operation.

PolarAngle: double (Read/Write). In Polar Compilation, defines the angle at which the curve data is to indexed when included into the Polar data , or the angle for which a curve is to be removed.

PolarDoInclude: boolean (Read/Write). In Polar Compilation, defines whether the curve in the PostProcess Source is to be included or removed when PerformPostProcess is called.

Postprocess:integer (Read/Write). See Global Constants.

PostProcessCfgVisible:boolean (Read/Write). Sets or hides the PostProcess Configure Form.

PostProcDelay: double (Read/Write). Delay removed, in seconds, in certain PostProcess operations.

PostProcDivAfterFft:boolean (Read/Write). If true, and if time domain data being subjected to a postprocess FFT operation is dual channel, this will cause the two spectra to be complex divided, yielding relative spectrum magnitude and relative phase information.

PostProcFreq1: double (Read/Write). If one frequency parameter is adjustable for PostProcess operation, use this value. If two frequency parameters are adjustable, this is: (start frequency, corner frequency, or bandwidth)

PostProcFreq2: double (Read/Write). If two PostProcess frequency parameters are adjustable, this is: (stop frequency or center frequency)

PostProcFftSize:integer (Read/Write). The size to use in FFTs for certain PostProcess operations.

PostProcGain: double (Read/Write). Gain, in dB, applied to data in certain PostProcess frequency domain operations.

PostProcSource:TDataPlot (Read/Write). This accesses the PostProcess Source plot. Make sure the plot you assign to this still exists!

PostProcTarget:TDataPlot (Read/Write). This accesses the PostProcess Target plot. If assigned "nil", a new plot will be created and displayed when the PostProcess operation is applied.

PostProcTimeGainDb1:double and **PostProcTimeGainDb2:double** (Read/Write). These properties control the gain that is applied if the TimeGainMode property is set to 2 ("Specify dB", on the Postprocess configure form for time domain operations)..

PostProcUseFlexWindow:boolean (Read/Write). shows or controls whether FFT postprocess operations will use the FlexWindow to allow for gradual transitions from windowed to unwindowed operations.

PostProcWindowType: TWndKinds (Read/Write). The type of data window to use in PostProcess operations such as PostProcess FFT.

RemapFreqFormat: TRemapFormat (Read/Write). For the ReMap PostProcess.

RemapListFile: string (Read/Write). List File to use for ReMap PostProcess. Must have complete path and extension.

RemapNumberOfPoints:Integer (Read/Write). Used with ReMap PostProcess.

RemapSampleRate: Integer (Read/Write). Used when RemapFreqFormat is `"rfIfftCompat".`

SchroederFilter:TFilterShapes (Read/Write). The filter shape (Butterworth, brick wall) used to bandlimit Schroeder curve PostProcesses.

SchroederFilterCenter: double (Read/Write). Center frequency for filtered PostProcess Schroeder curve.

SchroederFilterOrder:integer (Read/Write). Filter order for PostProcess Schroeder curve.

SchroederOctBandwidth: double (Read/Write). Bandwidth used, around SchroederFilterCenter, in PostProcess Schroeder Curve. **Set to 0 for full band.**

SchroederRoomName:string (Read/Write). For PostProcess Schroeder curve.

SmoothingFilter:TFilterShapes (Read/Write). Filter type for PostProcess Smoothing.

SmoothingFilterOrder:integer (Read/Write). Filter order for PostProcess Smoothing.

SmoothIncPhase:boolean (Read/Write). Determines if Phase is also included during PostProcess Smoothing.

SmoothingOctBandwidth: double (Read/Write). Octave bandwidth for PostProcess Smoothing.

SmoothingOctLimit: double (Read/Write). Octave band limit for PostProcess Smoothing.

SpecifiedRe: double (Read/Write). If set to 0, Praxis will attempt to approximate the DC resistance of the measured woofer. Otherwise, it will assume, during Thiele-Small parameter extraction, that the DC resistance is this value.

StepTrim:double (Read/Write). The value to use for adjusting offset in Step Response (PostProcess) operations.

TailCorrAutoLowerSlope; This procedure automatically sets the TailCorrLowerSlope from the shape of the data near TailCorrLowerFreq.

TailCorrAutoUpperSlope; This procedure automatically sets the TailCorrLowerSlope from the shape of the data near TailCorrLowerFreq.

TailCorrLowerFreq:double (Read/Write). The lower frequency below which tail correction will be applied using TailCorrLowerSlope.

TailCorrLowerSlope :double (Read/Write). The slope, in dB/Octave, at which the response curve will be tapered off below TailCorrLowerFreq.

TailCorrUpperFreq:double (Read/Write). The upper frequency above which tail correction will be applied using TailCorrUpperSlope

TailCorrUpperSlope :double (Read/Write). The slope, in dB/Octave, at which the response curve will be tapered off above TailCorrUpperFreq.

TimeChan1From:integer and **TimeChan2From:integer** (Read/Write). Used with the Time Domain Channels postprocess. These properties specify where the data to be placed into the Target Plot's new channel 1 (or 2) is to come from (i.e, from the Source plot or the original Target plot). 0 indicates that the curve is to come from the Source plot and 1 indicates it is to come from the Target plot. A value of 2 ("none") can also be specified for the TimeChan2From property to indicate that the Target plot is to contain only one channel after the postprocess. Also see the TimeChan1UseChan property

TimeChan1UseChan:integer and **TimeChan2UseChan:integer** (Read/Write). These properties specify which channel (1 or 2) of the "From" plot is to be used for the new Channel 1 (or 2) of the Target plot, when a Time Domain Channels postprocess operation is performed. The From plot (for each channel to be put into the Target Plot) is specified in the TimeChan1From or TimeChan2From properties.

TimeGainMode:integer (Read/Write). Determines how gain will be applied to time domain data for Time Domain Postprocessing operations (Time Domain Length/Channels or Math). A value of 0 specifies that each channel of the Postprocess Target plot will be individually normalized so that its largest magnitude is 1. A value of 1 specifies that both channels are normalized together by the same scale factor so that the largest of either is scaled to a magnitude of 1 (used for keeping proportions of binaural files unchanged). A value of 2 specifies that the gains are to be adjusted according to the PostProcTimeGainDb1 and the PostProcTimeGainDb1 properties. To prevent any change to the levels of the processed Target plot, specify TimeGainMode to be 2 with PostProcTimeGainDb1 and PostProcTimeGainDb2 set to 0dB.

TimeLengthAdjustVal:double (Read/Write). This value, in seconds, will be used in Time Domain Length postprocesses when the TimeLengthMode property is set to 1,2, or 3 ("Extend beginning", "Extend end" or "delay").

TimeLengthMode:integer (Read/Write). Determines how the length will be changed in a Time Domain Length postprocess. 0 indicates that the Target is to consist of that portion of the Source that is between the window edges of the source. 1 means "Extend at the beginning", 2 means "Extend at the end" and 3 means "delay". Settings of 1, 2 or 3 will use the value specified in the TimeLengthAdjust property for extension or delay.

TimeMathMode:integer (Read/Write). Determines the math operation that will occur when a Time Domain Math operation is performed. 0 means "Copy from Source", 1 means "Invert from Source", 2 means "Target + Source", 3 means "Target-Source", 4 means "Convolve Source, Target" and 5 means "Deconvolve Target via the Source as reference". Gain will be adjusted per the TimeGainMode property.

TSanimate:boolean (Read/Write). Determines whether Thiele-Small curve fitting operations will be "animated".

TSClearCurveFits; Clears any existing Thiele-Small curve fitting data from the system. Used to start fresh for measuring a new driver.

TSCsvFileSave:boolean; Attempts to save the set of T/S parameters in a "comma separated variable" format (readable by spreadsheets) to a file name by TSname. Returns true if successful.

TSdia:double (Read/Write). The effective diameter of the woofer in Thiele-Small extractions. Value is specified in units of meters.

TSExtract:boolean; Extracts the Thiele-Small parameters from the results of the free-air curve fit and the mass-loaded curve fit operations. Returns true if successful.

TSmass:double (Read/Write). The added mass value used in Thiele-Small extractions. Value is specified in units of grams.

TSname:string (Read/Write). The name identifying the woofer in a Thiele-Small measurement. Used for printing reports or saving data files.

TSPrint:boolean; Prints a report of the T/S parameters to the printer. Returns true if successful.

TSTextFileSave(filename:string):boolean; Attempts to save the text report of the T/S parameters to the named path and filename. Returns true if successful.

TsUseAddedMass:boolean (Read/Write). When true, the added mass method will be used for T/S parameter estimation of Vas. When false, the "delta compliance" or "Box Method" is used.

TSvolume:double (Read/Write). In Litres. Used in "delta compliance" or "Box Method" measurement of Theile/Small parameters.

UseFastTSExtract:boolean (Read/Write). If true, only the traditional method of analyzing the impedance curve (finding the peak and the -3dB points) will be used. This is extremely fast, but does not model voice coil inductance. If false, a full curvefit will be performed.

WfSteps:integer (Read/Write). Maximum number of steps (sheets) to use for PostProcess Waterfall plot generation.

WfTimeInc:double (Read/Write). Seconds between sheets, for PostProcess Waterfall plot generation.

Handling Secondary Plots in a script

To get access to a secondary plot in PRAXIS script, the plot should be one that is created within the script (otherwise, there's no way to get hold of it). The type of any plot is TDataPlot.

(Declaration)

```
MyNewPlot: TDataPlot; //DelphiScript
Dim MyNewPlot as TdataPlot ‘VBScript
```

(Creation from a script):

```
MyNewPlot := Praxis.GetNewPlot; //DelphiScript: makes a new plot
MyNewPlot = Praxis.GetNewPlot  ‘VBScript
```

Now, you can use any of the methods or properties of Plot objects with it, or do things like:

```
Praxis.PostProcSource := MyNewPlot;  //DelphiScript.
  //You can use MyNewPlot wherever a parameter expects a TDataPlot object

MyNewPlot.CopyPlot(Praxis.PrimaryPlot);
  //copies the data and formatting from PrimaryPlot

Praxis.DataFileLoad(MyNewPlot, 'C:\XLERB\SomeData.px2');
Saved:=Praxis.DataFileSave(MyNewPlot, 'C:\SomeFolder\SomeData.px2');
MyNewPlot.FormLeft := 100;
```

To delete the plot, use:

```
Praxis.DeletePlot(MyNewPlot); //DON'T FREE OR DESTROY IT!
```

To see if it still exists, use something like:

```
DidntDeleteItYet := Praxis.PlotExists(MyNewPlot);
```

Properties and Methods of Plot Objects

Prefix the following properties and methods using the complete indentifier for the plot.
Examples:

```
Praxis.PrimaryPlot.Print(psBW);
PrimaryPlot.Print(psBW); //using the global function  PrimaryPlot
Praxis.PostProcSource.Print(psBW);
MyPlot:=Praxis.GetNewPlot;
MyPlot.Print(psBW);
```

Setting Position and size of the Plots:

Do **not** use the properties Top, Height, Left, or Width for the Plot Objects.

Instead, use the special integer properties: **FormTop, FormHeight, FormLeft, and FormWidth.**

Handling the "LiveTime", "LiveIR" and "AuxPlot":

The position and dimensions (Left, Top, Width and Height) of these three special plots can be read or set using these methods:

```
Praxis.LiveTimeFormDimensions(var Left,Top,Width,Height:integer);
Praxis.LiveIRFormDimensions(var Left,Top,Width,Height:integer);
Praxis.AuxPlotFormDimensions(var Left,Top,Width,Height:integer);
```

All arguments to these methods must be variables of the script. To set the values, assign the variables as desired before calling the method. To leave any of them unchanged, set the corresponding variable(s) to a negative value. After the method executes, the variable will be loaded with the current (positive) value of the plot setting.

You can obtain parameters of the data that is currently obtained in a normal PRAXIS plot by using the
procedure GetDataStats(var DS:DataStats). This is given a data record of type "DataStats" that will be filled by the procedure. The DataStats record is defined as follow:

```
DataStats = record
  datatype:string;  //example: "TimeDomain", etc.
  NumberOfPoints,NumberOfFields:integer;
  HorizontalMin,HorizontalMax:double; //range over which the horizontal axis is defined
  Description:string;
  DateAcquired,MonthAcquired,YearAcquired:word;
  HourAcquired,MinuteAcquired,SecondAcquired:word;
  SampleRate:integer;
  WindowLeft,WindowRight:double; //window edges for time domain data
end;
```

AddAPlotToPlotSet(DP:TDataPlot); See *Plot Sets*, below.

AddMultiPlotFile(FileName:string); Adds a single data file to be included in the set of "Multi" traces for the plot. If UseMultiPlot is true, and the first field of the data file is compatible with the first field of the data held by the plot, a trace will be shown from this file. You can also load a list of such file names using LoadMultiPlotList.

AllMkrOff(Sender: TObject); Set the value 'Sender' to nil or to some object (it does not matter). This procedure turns off all markers from the plot. See the Property UseMarker[index], for turning each marker on or off individually.

AutoAll; Attempts to both AutoScale and AutoReference all displayed traces in the plot. This is equivalent to clicking on the small "lighting bolt" button at the bottom of the plot.

AutoRef(Index:integer); Attempts to AutoReference the data curve (1 or 2) indicated by Index.

AutoScale(Index:integer); Attempts to AutoScale the data curve (1 or 2) indicated by Index.

AutoScaleRef(Index:integer); Attempts to both AutoScale and AutoReference the data curve (1 or 2) indicated by Index.

BackColor: TColor; (Read/Write) Sets or reads the color used to draw the backgrounds of the plots when UseCustomBitmap is false.

BmpHeight: integer; (Read) Returns the height, in pixels, of the bitmap area of the graph, for custom plotting.on the BmpCanvas.

BmpWidth: integer; (Read) Returns the width, in pixels, of the bitmap area of the graph, for custom plotting.on the BmpCanvas.

BmpCanvas: TCanvas; (Read) Returns a TCanvas object for use in custom plotting of graph areas using Delphi canvas methods, or for annotating graph areas drawn by Praxis. Custom plotting is best handled by setting PlotScheme to 1 (to prevent Praxis from drawing on the canvas, other than the background) and by putting the code to redraw the plot in an Event Handler method of the main form (in your script) named UserRedrawPlot. The single argument of the Event Handler, the "Sender", is the plot (a TDataPlot object) which is causing the event and which needs to be redrawn.

CalcTHD(F0:double;var F0Level:double;trace:integer):double; This function returns the total harmonic distortion value, in percent, calculated using the current spectrum data (assuming that the plot contains spectral lines for the fundamental and its harmonics, obtained from measurement of a single-tone stimulus). The fundamental frequency is F0, and the level at F0 is returned in variable F0Level.

ClearMultiPlotFiles; This procedure removes any "Multi" traces from the plot.

CustomBitmapName: String; (Read/Write) The complete path, file name, and extension of the bitmap file used for plot backgrounds when UseCustomBitmap is true. If an invalid filename is given, or if an empty string is assigned, then the default Liberty Instruments "embossed logo" bitmap is used.

CustomBitmapStretched: Boolean; (Read/Write) Controls the way plot background bitmaps are drawn when UseCustomBitmap is true. If true, the bitmap is stretched to cover the background. If false, multiple copies of the bitmap are used to tile the background.

D3BaseColor:TColor; (Read/Write) Sets or reads the color used for the "base" or "floor" of 3D waterfall plots.

D3FillColor:TColor; (Read/Write) Sets or reads the color used for the highest altitudes in the fill areas of 3D waterfall plots, when gradient color is used. If this is set to the same value as D3MinColor, then gradient color is not used (and plotting will then be much faster!).

D3MinColor:TColor; (Read/Write) Sets or reads the color used for the lowest altitudes in the fill areas of 3D waterfall plots, when gradient color is used. If this is set to the same value as D3FillColor, then gradient color is not used (and plotting will then be much faster!).

D3HorizAngle:double; (Read/Write) Accesses the apparent user horizontal viewing angle (relative to the "floor"), in degrees, for display of waterfall plots. Allowable range is -80 to +80 degrees. The graphed structure is always resized for most efficient use of the available display area.

D3VertAngle:double; (Read/Write) Accesses the apparent user vertical viewing angle (relative to the "floor"), in degrees, for display of waterfall plots. Allowable range is 0 to 80 degrees. The graphed structure is always resized for most efficient use of the available display area, so adjusting this parameter may also appear as if it were adjusting the height of the graph.

DefineLimitFile(IsUpperLimit:boolean;LimFileName:string;Limitindex:integer); Declares the data in the .px2 format data file indicated by LimFileName (full path and filename) to be used as the upper (if IsUpper is true) or lower limit (if false) for Pass/Fail evaluations. LimitIndex is normally 1, but can be assigned other values if a set of limit curves is needed (see function **IsWithinLimits**).

DefineLimitPlot(IsUpperLimit:boolean;Source:TDataPlot;Limitindex:integer); Declares the data contained in a different plot object Source to be used as the upper (if IsUpper is true) or lower limit (if false) for Pass/Fail evaluations. LimitIndex is normally 1, but can be assigned other

values if a set of limit curves is needed (see function **IsWithinLimits**).

Delay: double; (Read/Write) Accesses the delay compensation value for removing delay effects from phase data of frequency domain plots. The value entered is the amount of delay to be removed.

DisplayedCurve[index: integer]: integer; (Read/Write) Select the curves (traces) to display. Index is 1 (trace A) or 2 (trace B). Assigning a value of -1 disables display of the curve. For frequency domain data, the first data field is 1 -- do not assign a value 0 to either trace (0 is the frequency array). For time domain data, 0 is the first data field.

DistortionAsPercent: boolean; (Read/Write) Sets or reads a value which determines whether normalized distortion product measurements use dB format (if false) or percent (%) format (if true).

DomainFromPoint(point: integer): double; This function returns the frequency or time value at the point indexed by point (the first point is 0).

EnableMouseClickEvent; This procedure allows mouse clicks in the plot form to trigger a MouseClickInPlot event in a running script. To disable this again, use the MouseNormal procedure of this Plot object.

FindNextDomainPoint(domain: double; IsUp: boolean): double; Domain is the time or frequency value of a data point in the plot. If IsUp is true, this function returns the next higher data point at which data is stored. If IsUp is false, this function returns the next lowr data point at which data is stored. For example, to find the next stored data point above 1kHz that is used in the plot, use: x= FindNextDomainPoint(1000,true);

FormTop, FormHeight, FormLeft, and FormWidth. See "Setting Position.." at the top of this section.

Freq3D: double; (Read/Write). The frequency at which Polar Plots are displayed (as shown in the Polar Format Tab).

FreshRedraw; This procedure immediately redraws the current data plot.

GetDataStats(var DS:DataStats). This procedure returns parameters of the data that is currently contained within the Plot. (The DataStats record is documented at the beginning of this section). Example:

```
Var DS:DataStats;
…
GetDataStats(DS);
PointsInTheData:=DS.NumberOfPoints;
```

GridColor: TColor; (Read/Write) Sets or reads the color used to draw the grid lines on the plot.

HasSameFreqPoints(DP:TDataPlot):boolean; This function returns true if the frequency data points of the plot object uses the exact same frequency points as does the plot DP. If true, the data at each point in the two plots can be related, otherwise one plot must first be mapped (by a Postprocess operation).

HoldThisData; This procedure sets the current data to be the persistent "Hold" data (a special multiplot used to quicky define and show a reference plot, when variations are being monitored).

HorizontalCenter: double (Read/Write). Sets the center of the horizontal display range (time or frequency) of the Plot. You can also do this using the Plot.HorizontalCenter property. You should normally set the HorizontalSpan first.

HorizontalLog: boolean; (Read/Write) Sets or reads a value reflecting whether frequency domain data uses logarithmic format (if true) or linear format (if false). Should always be set False for time domain data.

HorizontalSpan: double (Read/Write). Sets the span of the horizontal display range (time or frequency) of the Plot.

HorizontalStart: double (Read/Write). Sets the start of the horizontal display range (time or frequency) for the Plot. Must be less than the Horizontal Stop value.

HorizontalStop: double (Read/Write). Sets the end of the horizontal display range (time or frequency) for the Plot. Must be more than the Horizontal Start value.

InvertPhase: boolean; (Read/Write) causes any displayed phase values to be offset by 180 degrees (inverted).

ImportMLSSA(FileName:string;IsASCII:boolean=true); This procedure is intended to import data files from another measurement system. (However, its implementation is not yet complete nor tested, so results will vary). IsASCII should be true if the file type is text, and false if the file to be read is binary.

IrPolarityPositive:boolean; This function returns true If the plot has time domain data and its polarity is positive, otherwise false (valid only for impulse response data).

IsInAPlotSet:boolean; See *Plot Sets*, below.

IsInSamePlotSet(DP:TDataPlot):boolean; See *Plot Sets*, below.

IsolateFromPlotSet; See *Plot Sets*, below.

IsWithinLimits(var BadFreq:double;Mask:integer; EvalAbsolute:boolean; CurveID:integer; LimitIndex:integer):boolean; This function is used to perform Pass/Fail evaluations. These evaluations can be performed only from a script. The function returns true (passing) if the trace identified by CurveID (the Field of the data) lies in its displayed region entirely **above the lowerLimit curve** for the specified LimitIndex **and below the upperlimit curve** for the specified LimitIndex. Otherwise the function returns false (failing). That is, the function is true when the curve being tested is within the limit curves. (Any region in which the data curve or the limit curves is not defined will be considered to be "passing"). Mask should be set to 3 for evaluation both upper and lower limits; to 1 for evaluating only relative to a lower limit; and to 2 for evaluation relative to only an upper limit. If EvalAbsolute is false, then the curve being evaluated can be shifted vertically to try to make it fit within the limit curves; otherwise, it will be evaluated at its absolute settings.

LoadMultiPlotList(ListName:string); This procedure attempts to use the file indicated by ListName (complete path and filename, with extension ".lil") as a list of data files for "Multi" display. If successful, the file names listed are appended to any other file names already being used for Multi display. To first clear the list, use ClearMultiPlotFiles.

LoadWeightFile(filname:string); This procedure attempts to use the file indicate by filname (including path, filename, and extension of ".wgt"). As a weight file for the plot.

Marker[Trace,Index:integer]: double (Read). Used to read the marker value on the Plot, at the

marker's currently placed location. Trace can be 1 (A) or 2 (B). Index can indicate marker 1 through 5. The indicated marker must be enabled. See description of the MarkerFrequency property.

MarkersColor: TColor; (Read/Write) Use this to set the colors to be used by Praxis' triangular data markers for this plot. If the value is assigned the value "clBlack", then the markers will use the same color as the data traces.

MarkerDelta: integer; (Read/Write) Sets or reads the marker which is to be used as the "delta reference" (all other markers will list their range values in relation to this marker). The designated marker must be enabled (see UseMarker, below), and its range values will list as 0. If the assigned value is outside the range (1 to 5), all markers will list normally.

MarkerFrequency[Index:integer]: double (Read/Write). Used to place or read the position of the indicated marker (1 to 5), specified by the frequency or time domain value. The indicated marker must be enabled. If the value is not on an exact data point for the frequency, the marker will be moved to the next higher or lower valid frequency. Use MarkerNext and MarkerPrev to adjust it to an exact data point.

MarkerNext(Index:integer); Moves the specified marker to the next valid data point to the right.

MarkerPrevious(Index:integer); Moves the specified marker to the next valid data point to the left.

MarkerSearch(Max,Local:boolean; CurveIndex,MkrIndex:Integer); This procedure will cause marker number MkrIndex (1 to 5) to search along the visible areas of trace # CurveIndex (1 or 2). If Max is true, the search will be toward a maximum value, otherwise the search will be toward a minimum value. If Local is true, the search will be only along adjoining points until no further upward slope is encounter. If Local is false, the marker will go to the highest (or lowest) displayed data point on the trace.

MarkerTracking[Curve,Index:integer]: MkrTrackTypes; (Read/Write) Sets or reads the type of tracking for marker # Index (1 to 5), tracking on the indicated Curve (0 or 1). Only the last assigned tracking type for this marker will be effective. The marker must be enabled (see UseMarker, below).

MouseButtonLeft:boolean; (Read) Reads true if the last mouse button clicked in the plot's graph was the left mouse button.

MouseButtonShift:integer; (Read) Reads the states of the "Shift, Alt and Ctrl" keys at the time that the mouse button was last clicked in the plot's graph.

- If Shift was pressed, then (ThisPlot.MouseButtonShift AND 1)=1.
- If Alt was pressed, then (ThisPlot.MouseButtonShift AND 2)=2.
- If Ctrl was pressed, then (ThisPlot.MouseButtonShift AND 4)=4.

MouseDomain:double; (Read) Reads the horizontal data value (usually Hertz or seconds) of a plot's graph, where the mouse was last clicked. Does not work correctly for waterfall or polar plots.

MouseNormal; This procedure disables mouse clicks in the plot form fromto triggering a MouseClickInPlot event in a running script. To enable this event, use the EnableMouseClickEvent procedure of this Plot object.

MouseValue[index:integer]:double; (Read) Reads the value of the data traces (for the data field corresponding to index) at the horizontal position in the plot where the mouse was last clicked. Read this in response to a MouseClickInPlot event. (See MouseNormal and

EnableMouseClickEvent, above).

MultiPlotTraceColor[index:integer]:TColor; (Read/Write) Sets or reads the color of the multiplot trace having the specified index (the index is the order in which the trace was added to the MultiPlot Trace list).

NewTimeDomain(Fields,Points,Rate:integer); This procedure fills the plot's data with a blank time domain record of the specified number of points and of up to two fields (traces). Rate is the sample rate of the new data to be created, so the time extent of the new data will be (points-1)/samplerate [seconds].

NumberOfPoints: Integer; (Read) Returns the number of points in the domain (time or frequency) of the data set contained in the plot.

NumberVerticalDivisions:integer; (Read/Write); (Read/Write) Sets or reads the Vertical Number of Divisions for the plot.

Offset: double; (Read/Write) Accesses the offset value, in dB, to be used for displaying frequency response or spectrum data.

Open(FileName:string); This procedure attempts to load and display the data from the compatible file with path, name, and extension given in FileName.

PlotDescription:String (Read/Write); This accesses the string which is entered in the "title" tab of the plot's formatting control.

PlotNowShownFromSet:TDataPlot; See *Plot Sets*, below.

PlotScheme: integer; (Read/Write); This value can be set to 1, if your script will take responsibility for drawing the graph area (on the BmpCanvas). You shoud not set this to any other value. Put the code to redraw the plot in an Event Handler method of the main form (in your script) named "UserRedrawPlot(Sender)". The single argument of the Event Handler, "Sender", is the plot (a TDataPlot object) which is causing the event and which needs to have its bitmap canvas redrawn.

Plot Sets:

A plot set is a grouping of a number of plots so that they occupy the same on-screen form. They can be manipulated from scripts using the following methods of the Plot object:

- **IsInAPlotSet:boolean**; True if this Plot is part of a PlotSet.
- **IsInSamePlotSet(DP:TDataPlot):boolean**; True if this Plot is in the same plot set with plot DP.
- **StartAPlotSet**; This procedure makes this Plot into a PlotSet, if it is not already part of one.
- **AddAPlotToPlotSet(DP:TDataPlot)**; This procedure adds the plot DP to the same PlotSet as this Plot object is in.
- **IsolateFromPlotSet**; This procedure removes this Plot from any plot set it may be in. After this IsInAPlotSet will return false.
- **PlotNowShownFromSet:TDataPlot**; This function retuns the Plot that is now showing in the PlotSet that contains the Plot object from which the method is called.
- **ShowPlot**; This procedure shows the Plot object if it is in a PlotSet, or brings it to the front if it is not.

  PointFromDomain(domain: double): integer; This function returns the index in the data field arrays corresponding to the given domain value (in Hz or sec).

  PointFromX(X: integer; DfieldNum: integer): integer; returns the index into the field arrays

for which the domain value exists at horizontal coordinate X in the BmpCanvas. The DfieldNum parameter is not used, and can be set to 0.

Polar Frequency -- USE property "**Freq3D**" for this.

PolarIsFilled:boolean; (Read/Write) Indicates whether the Polar plot is to be shown filled or only as an outline.

PolarSymmetry:integer; (Read/Write) Indicates whether the Polar plotting process should assume Right/Left symmetry (if the value is 1), Front/Back symmetry (if the value is 2), both symmetries (3) or no symmetry (0).

PolarUseNormalize:boolean; (Read/Write) Indicates whether the Polar Plot should be shown formatted with the largest value in the curve just touching the outside of the polar circle (VScale can still be used to adjust the dB/division or the number of divisions).

Print(PS: TPrintStyles); Prints the current data plot to the printer.

PrintToFile(PS: TPrintStyles;FileName:String); Prints the current data plot to a graphic file. The extension given for the file name determines the type ("jpg" or "bmp").

RestoreLatestData; Loads the data from the plot which was used for Praxis's startup data (the data which was displayed last time Praxis shut down with the user approving the configuration save), or the last data which was in use when SaveLatestData was called from a plot..

SaveLatestData; Saves the data in this plot to be used as Praxis's startup data, or for recall using RestoreLatestData.

SetAutoRef(Index:integer;SetOn: boolean); Sets AutoReference for the data curve (1 or 2) indicated by Index, so that the automatic process attempts to activate continuously (i.e., with every redraw of the plot).

SetAutoScale(Index:integer;SetOn: boolean); Sets AutoScale for the data curve (1 or 2) indicated by Index, so that the automatic process attempts to activate continuously (i.e., with every redraw of the plot).

SetLFLimit(limit:double); This procedure sets the low frequency resolution limit of frequency response data to the specified value. Data below this frequency limit will not be displayed. Set this to 0 to allow all data to be displayed to the lowest frequency.

ShowMultiPlotLegends: boolean; If "Multi" curves are displayed, this also controls whether their legends (the Titles they were saved with) will also be displayed.

ShowPlot; This procedure shows the Plot object if it is in a PlotSet, or brings it to the front if it is not.

SpectrumAvgLogFreq(fstart,fstop:double;field:integer=1):double; This function is intended for spectral contamination or Rub/Buzz testing, when the plot contains an FFT spectrum. It returns a result (in dB) that is equally weighted in each octave (over the frequency range from fstart to fstop), of the integration of all the power in the range. The result is "relative", that is, it should be used only to compare similar situations (not to measure absolute power). If the "Hide tones" option for FFT was used, the result will ignore the same stimulus tones. In that case, this function then behaves like SpectrumNoStimSum, except that it is octave weighted. Use this to evaluate the "residual spectrum" left by a device, as compared to that left by a known-good device.

SpectrumNoStimSum(fstart,fstop:double;StimChan:integer;bins:double;field:integer=1):double; This is similar to the SpectrumSum function (see below) except that it ignores tones that are near the frequencies of applied stimulus tones (of the output channel StimChan). In other words, this can be used for spectral contamination measurements. The width of the region ignored around each tone is determined by the value 'bins' (which should usually be approximately 5). Also see SpectrumAvgLogFreq.

SpectrumSum(fstart,fstop:double;field:integer):double; This function returns the summation of power (Result is in volts,Pa, etc, rms) over the frequency range between fstart and fstop for the indicated field. Data should be spectrum data and field should be 1 or 2. The summation is over equal bandwidths (in Hz), so that, for instance, noise in a higher frequency octave would sum to a higher total than would noise in a lower frequency octave.

SpectrumSumWeighted(fstart,fstop:double;field:integer):double; Like the SpectrumSum function, but the effect of the weighting curve (if in effect) is included.

StartAPlotSet; See *Plot Sets*, above.

StimTonesSum(fstart,fstop:double;StimChan:integer;bins:double;field:integer=1):double; This function is like SpectrumNoStimSum except that it gives the power ONLY at the stimulus tones.

Trace[index:integer]: integer; (Read) Reads the Field being currently displayed in the indicated trace (1=A, 2=B).

TraceColor[Index:Integer]: TColor (Read/Write) Used to read or set the colors used by the data trace (Index=1 is for trace A, index=2 is for trace B).

TraceWidth:integer; (Read/Write) This property reads or sets the widths of trace lines as drawn in a plot (valid range = 1 to 5).

UnitsOfMeasure(Field:integer):string; Returns a string identifying the units of measure (such as "V", "dB", "Pa", etc) for the field of this plot.

UpdatePlot; This procedure redraws only the graph area of the plot -- marker readouts, etc, are not redrawn.

UseCustomBitmap: Boolean; (Read/Write) Specifies whether to use a bitmap file for the background of plots. Otherwise, the background is drawn as a solid color using BackColor.

UseMarker[Index:integer]: boolean; (Read/Write) Determines if the indexed marker (1 to 5) is enabled.

UseMultiPlot: boolean; (Read/Write) Determines whether "Multi" curves (from compatible data files) should be shown with the data contained in the plot.

ValOfY(y:integer; DfieldNum: integer): double; This function returns the vertical data value (dB, volts, etc) for the given field number of the data in the plot, that would correspond to the vertical graphic position y. It is useful to, for instance, find the implied data value at a mouse postion on a plot. The vertical graphic position 0 is the top of the bitmap (see BmpCanvas, above), position 1 is the next line down, etc. DFieldNum is indexed, here, starting at 0 and for time domain data is 0 or 1. For frequency domain data, do not use a value of 0, as that is used for the frequencies.

ValueAtPoint(point:integer;Field:integer):double; This function returns the data value of Field "Field", for point index "point". Use the PointFromX or PointFromDomain functions to find the

point related to a graphic pixel position or a floating point domain value. For example, for the dB magnitude of the point nearest 1kHz in a frequency response plot, use:

```
v:= ThisPlot.ValueAtPoint(ThisPlot.PointFromDomain(1000),1).
```

Field 0 of a frequency domain plot is the frequency value, Field 1 is usually the dB magnitude. Field 0 of a time domain plot is the data of the first trace (the time value is inferred from the point index and the sample rate). You can use the UnitsOfMeasure function to find what type of data is stored in a field, and the NumberOfPoints property to find how many points are in the data -- valid points range from 0 to NumberOfPoints-1, and each trace (Field) contains separate data for each domain point.

ValueAtX(X: integer; DfieldNum: integer): double; This function returns the raw value in the data field indicated by DfieldNum, for the domain value corresponding to position X. X is a horizontal coordinate in the BmpCanvas.

VerticalRefDiv[Index: integer]: integer; (Read/Write) Sets or reads the VRef division value for the trace (1=A, 2=B).

VerticalReference[Index:integer]:double; (Read/Write) Sets or reads the Vertical Reference value for the trace (1=A, 2=B).

VerticalScale[Index:integer]:double; (Read/Write) Sets or reads the Vertical Scale for the trace (1=A, 2=B).

WaterfallAsPeriod: boolean (Read/Write) If true, waterfall plots will be plotted with the depth dimension shown in number of cycles ("Q"), rather than in units of time.

WindowLeft[Index:Integer]:double; (Read/Write) Reads or sets the windowing start edge to the assigned value, for use in subsequent transformations to frequency domain (such as by FFT). Index should always be 1.

WindowRight[Index:Integer]:double; (Read/Write) Reads or sets the windowing stop edge to the assigned value, for use in subsequent transformations to frequency domain (such as by FFT). Index should always be 1.

XFromDomain(domain: double): integer; This function returns the horizontal coordinate, in the BmpCanvas, of the data using the given domain value (in Hz or sec). This function may be useful for making custom annotation on a graph drawn by Praxis.

XFromPoint(point: integer): integer; This function returns the horizontal coordinate, in the BmpCanvas, of the point whose domain value would be indexed by "point". This function may be useful for making custom annotation on a graph drawn by Praxis.

YAtPoint(point: integer; DfieldNum: integer; TraceNum: integer): integer; This function returns the vertical coordinate within the BmpCanvas of the data point (indexed in the array, starting at 0) corresponding to the DFieldNum (assuming that it is now plotted on the graph). TraceNum value is not used and can be set to 0. This function may be useful for making custom annotation on a graph drawn by Praxis.