The Rave Application iNterface Technology (RANT) Specification

"How to Rant with Rave"

Table of Contents

1.0 - Introduction

This document is provided to enable programmers to interface with the RAVE (Report Authoring Visual Environment) Report Designer at a very low level. This document assumes that you are familiar with normal Delphi component development. Many of the concepts and principles for interfacing with Rave are very similar to the code interface of Delphi. If you are already familiar with Delphi component development, you should still pay close attention to this document as there are subtle yet important differences that will allow you to take full advantage of RANT. After reading this document, you should be able to create custom Rave components, property editors, component editors, project editors (also know as wizards) and control the Rave environment in a wide variety of ways.

1.1 - Technical Support

Technical support for items contained within this specification will be limited to Rave specific features. We cannot provide free technical support for programming questions which are common to all Delphi components and classes. We recommend that if you encounter problems with your RANT components or classes, you first refer to the Delphi Component Writer's Guide manual or help file, books on Delphi programming or post a message on a public Delphi forum. An excellent location to post these type of questions is on the Nevrona Forum where you can communicate directly with other Rave users who may have already encountered and solved the same problems you are having. For more instructions on how to join the Nevrona Forum go to http://www.nevrona.com.

2.0 - Creating a Rave Component

One of the most powerful capabilities of Rave is the ability to create and dynamically install custom reporting components. Every component shipped with RAVE was created and registered in the same manner as what is available to the RANT developer. The first step to creating a Rave component is to understand the basic architecture of Rave's class library.

2.1 - Basic Rave Component Classes

The following three classes, TRaveComponent, TRaveControl and TRaveContainerControl are the basic Rave classes that most Rave components descend from. If you are creating a custom Rave component you should either descend from an existing rave component or one of these classes. The relationship of these classes is as follows:

```
TComponent
 |
 +-> TRaveComponent
      |
      +-> TRaveControl
           |
           +-> TRaveContainerControl
```

TRaveComponent (RVClass.pas) - This is the base class for all Rave components and descends directly from Delphi's TComponent. TRaveComponent classes are non-visual in nature and thus should only be used for items which will not be displayed on the page designer or the report. However, the Rave user will still be able to select a TRaveComponent class and modify its properties through the Project Tree.

TRaveControl (RVClass.pas) - This class descends from TRaveComponent and is the base class for all visual components. All TRaveControl classes have Left, Top, Width, Height, Anchor, Mirror and DisplayOn properties (although they may not be visible on all components). TRaveControl classes display themselves through both the Paint method (for the visual designer) and the Print method (for the printed output).

TRaveContainerControl (RVClass.pas) - This class descends from TRaveControl and adds basic parent-child functionality for components that will contain other components. Examples of TRaveContainerControl classes are TRaveBand and TRaveSection. Note that the parent-child methods and properties are defined at the TRaveComponent level and may be overriden from any class level (TRaveContainerComponent for example), however, if you are creating a visual container component you normally will want to descend from this class.

2.2 - Other Important Rave Classes

These classes are used by many of Rave's editor and project management components. You will not normally need to descend your components from these classes, however, you will often interface with components of these class types to perform the tasks that you need to.

```
TComponent
 |
 +-> TRaveComponent
      |
      +-> TRaveDataField
      |
      +-> TRaveContainerComponent
           |
           +-> TRaveProjectItem
                |
                +-> TRavePage
                |
                +-> TRaveDataView
                |
                +-> TRaveReport
                |
                +-> TRaveProjectManager
```

TRaveContainerComponent (RVClass.pas) - This class descends from TRaveComponent and adds basic parent-child functionality similar to TRaveContainerControl. This class is used by Rave's TRaveProjectItem class. If you are creating a visual component that will contain other components, you should descend from the TRaveContainerControl class instead.

TRaveProjectItem (RVClass.pas) - This is the base class for all of Rave's project oriented classes including TRavePage, TRaveDataView, TRaveReport and TRaveProjectManager. This class descends from TRaveContainerComponent and adds a basic streaming and active interface.

TRavePage (RVClass.pas) - This class descends from TRaveProjectItem and is the class used to define each page of a report. All components on a page are owned by the page and can be accessed through the pages Components and ComponentCount properties. There are two types of TRavePage components in a Rave project, Report and Global. Report pages belong to a specific report while Global pages are available to the whole system. The Global boolean property of TRavePage can be used to determine which category a TRavePage class instance falls into.

TRaveDataView (RVData.pas) - This class descends from TRaveProjectItem and is the class used to define a data view. The TRaveDataView class is the owner and parent for all TRaveDataField components belonging to that data view. This class is discussed in further detail in the section "Interfacing with TRaveDataSystem".

TRaveDataField (RVData.pas) - This class descends from TRaveComponent and is the base class for all data field classes (TRaveIntegerField, TRaveStringField, ...). A TRaveDataField component will always be owned and parented by the TRaveDataView class that defines the data view that it is a part of. This class and it's descendents is where your components will interface with the data that's available through the data views. This class is discussed in further detail in the section "Interfacing with TRaveDataSystem".

TRaveReport (RVProj.pas) - This class descends from TRaveProjectItem and is the class used to define all Rave reports. The TRaveReport class is the owner and parent of all report pages that belong to that report but does not own or parent any global pages even though they may be visible in the page designer for that report. The TRaveReport class is also the main starting point to print reports through the Execute method.

TRaveProjectManager (RVProj.pas) - This class descends from TRaveProjectItem and is the class that manages all components in a report project. TRaveProjectManager is the owner and parent of all TRaveReport, global TRavePage and TRaveDataView classes. This is the main interface to load, save, import to and export from a report project. This is also the main interface that you will use to find a specific report or other component. This class is discussed in further detail in the section "Interfacing with TRaveProjectManager".

2.3 - Example Component

For this sample we'll create a simple text component that presets the font color to blue when it is created. First you'll want to create a unit containing your component class definition. This is very similar to the steps you would follow for creating a normal Delphi component. Here's the source for our text component that overrides the default color:

```
  unit NDCsBlue;

  interface

  uses
    Classes, Graphics, RVClass, RVCsStd;

  type
    TNDBlueText = class(TRaveText)
    public
      constructor Create(AOwner: TComponent); override;
    end; { TNDBlueText }

  implementation

    constructor TNDBlueText.Create(AOwner: TComponent);

    begin { Create }
      inherited Create(AOwner);
      FFont.Color := clBlue;
    end;  { Create }

  end.
```

2.3.1 - Designer-Only Library Files

To save size in compiled applications, certain properties and methods are only available when compiling for the visual designer. This is accomplished with the DESIGNER compiler directive and is used throughout the Rave classes. For example, the Paint method is used by the visual designer to display a component to the screen and so it must be defined within {$IFDEF DESIGNER}...{$ENDIF} directives. The Paint method, however, is not needed by a normal Delphi application and so it's code is not included in the executable.

These designer-only files are located in the C:\Rave4\RV directory and can be recompiled with the C:\Rave4\Source\RaveD5.BAT file. When compiling a package for inclusion in the Rave designer, the file in the C:\Rave4\RV directory should be referenced as the Rave library file.

2.4 - Registering a Rave Component

After you've created your Rave component, it's time to register it with the Rave system. The first step is to create a global procedure called RaveRegister. The spelling and capitialization must match exactly or it will not work. The RaveRegister procedure will be called when your component package is added to the Rave system to register your components or other editors. Inside of the RaveRegister procedure you will want to call RegisterRaveGroup, RegisterRaveComponents and RegisterRaveProperties.

2.4.1 - Registering the component's group

The RegisterRaveGroup procedure creates a component group for your component and must be called before you register the component class. The first parameter is the name of the group and the second parameter is the full name. Currently in the Rave designer, each group is stored in a dockable toolbar and the full name is used as the caption for the toolbar. The following source is an example of RegisterRaveGroup being called to create a group called ABC with a caption of "ABC Components".

```
  RegisterRaveGroup('ND','ND Components');
```

2.4.2 - Registering the component class

The RegisterRaveComponents procedure actually registers your component class with the Rave system, loads the component icons and creates a component button on the group toolbar. The first parameter is the name (not the full name) of the group that you want to add this component to and the second is an array of component classes that you want to register. All classes that are registered must descend from TRaveComponent or one of it's descendents. The following source is an example of RegisterRaveComponents being called to create two components, TNDHeaderText and TNDFooterText.

```
  RegisterRaveComponents('ND',[TNDHeaderText,TNDFooterText]);
```

2.4.3 - Registering the component's properties

The RegisterRaveProperties procedure registers the published properties for your components and must be called after the component class is registered. With this procedure you can determine which properties are visible to Beginner or Intermediate level users, which properties are Developer only, and which properties are hidden. The first parameter is the component class you are registering the properties for. The second and third parameters are a list of properties that are available to Beginner and Intermediate level properties.

By default, all published properties are available to Advanced level users. Any properties registered in the Intermediate properties are available to Intermediate and Advanced level users while any properties registered in the Beginner properties are available to all level users. Developer level properties are set with the fourth parameter and will be visible in the programmer version of the Rave designer and not visible in the end-user version of the Rave designer. The last parameter defines properties that are to be hidden to all users and can be used to hide properties that were visible in a ancestor class. The following source is an example of RegisterRaveProperties being called for the TRaveDataText component.

```
    RegisterRaveProperties(TRaveDataText,
     {Beginner}     'DataField;DataView',
     {Intermediate} 'LookupDataView;LookupField;LookupDisplay',
     {Developer}    '',
     {Hidden}       'Text');
```

2.4.4 - Creating Component Icons

Each component has a 24x24 pixel icon that is used to represent it. In the visual desinger, this icon is placed on a button that is used to create the component. There is also a 16x16 pixel icon that is used to represent the component in the Project Tree. Lastly, there is a 24x24 pixel icon that each component group (see section 2.4.1 below) uses to represent itself. In the visual designer, this icon is used on a button on the Toolbar toolbar to show or hide the component group toolbar. The name of the large component icon must match the name of the component class. The small component icon must match the name of the component class plus the characters '16' following it. The component group icon must match the name (not the full name) of the component group. So for our example with TNDBlueText in the ND component group, we would create bitmap resources named TABCBlueText (24x24 pixel component icon), TNDBlueText16 (16x16 pixel component icon) and ND (24x24 pixel component group icon). These bitmap resources will normally be included in a resource file (*.res) with the same name as the unit containing the component class. To register the icons with the unit, include the following source immediately after the implementation keyword in the unit containing the component class.

```
  {$IFDEF DESIGNER}
  {$R *.RES}
  {$ENDIF}
```

2.4.5 - Putting It All Together

Certainly there are plenty of examples of components in the Rave source. There is also a special web page available on the Nevrona Designs web site that contains other useful components (available shortly after release). The following is a complete example of a component and the RaveRegister procedure.

```
unit NDCsBlue;

interface

uses
  RVClass, RVCsStd;

type
  TNDBlueText = class(TRaveText)
  public
    constructor Create(AOwner: TComponent); override;
  end; { TNDBlueText }
  procedure RaveRegister;

implementation

{$IFDEF DESIGNER}
{$R *.RES}
{$ENDIF}

  procedure RaveRegister;

  begin { RaveRegister }
    RegisterRaveGroup('ND','ND Components');

    RegisterRaveComponents('ND',[TNDBlueText]);

    RegisterRaveProperties(TNDBlueText,
     {Beginner}     '',
     {Intermediate} '',
     {Developer}    '',
     {Hidden}       '');
  end;  { RaveRegister }

  constructor TNDBlueText.Create(AOwner: TComponent);

  begin { Create }
    inherited Create(AOwner);
    FFont.Color := clBlue;
  end;  { Create }

end.
```

2.5 - Distribution of Components and Editors

Now that you've created the source for your component, you will need to compile it into a package and other formats depending upon how it is to be used.

2.5.1 - Unique Component, Editor and File Names

As with any library file, you must uniquely name your items to avoid conflicts with items distributed from other authors. Any Rave components or editors distributed with Rave will be located in files beginning with "RP" or "RV". It is recommended that you select a unique prefix of at least 2 characters that will uniquely identify any files that are yours.

You should also apply this uniqueness when creating component and editor classes. All Rave components or editors distributed with Rave will be named with the prefix "TRave". It is recommended that you select a unique prefix that will uniquely identify any classes that are yours.

The following are guidelines that we have found to be a succesful:

- Avoid long file names. Even though Rave is currently only 32-bit, Delphi 1.0 support is planned for the near future and this will cause problems if you are creating a Rave component. Also, certian DOS based compression software and even some network operating systems do not handle long file names very well.

- Select a 2 character unique identifier (e.g. your initials) and use the following naming scheme for the type of item contained in each file (assume ND is the identifier you have chosen and Xxxx is a unique identifier of the item):

* Components: NDCsXxxx.pas
* Wizards (Project Editors): NDWzXxxx.pas
* Property Editor: NDPEXxxx.pas
* Component Editors: NDCEXxxx.pas
* Packages: ND_Xxxxx.dpk

These naming conventions keep file names at or under 8 characters and will help users of your Rave add-ons to understand what is in each file.

2.5.2 - Library Files and Delphi/C++Builder Versions

The 3.0 version of the Rave designer requires that all packages to be installed within it, are compiled with Delphi 4.0. For designer only items such as property editors or report wizards, this is typically the only form that will need to be compiled.

For components, however, this is only one of the forms that the source needs to be compiled for. When an application prints a report that contains a custom component, the component code itself needs to be compiled in the application. So, for example, if a Delphi 2.0 user uses a component that you have written, you will need to provide either .PAS or .DCU files to allow them to compile the component code into their application.

2.5.3 - Creating Packages for Custom Components or Editors

To compile a package containing custom Rave components or editors you will either need to use Delphi 4.0 IDE to create a new package (Component| Install Componet...) or the recommended method of compiling the package with the command line compiler. If you decide to use the Delphi 4.0 IDE, you need to make sure that you only include your own custom units.

The recommended method to compile packages for the Rave designer is to create a .DPK file and use the command line compiler. This is a cleaner method than using the IDE package tools since Delphi does several things that are geared more for it's own packages. The following an example of the .DPK file for our example component. Notice that we include any packages containing units required by our components in the Requires clause. The Contains clause lists the units containing the custom components or editors and the Description compiler directive defines the name of the package that will be displayed in the designer's Edit|Preferences...|Packages dialog.

```
  package ND_Tools;
  {$DESCRIPTION 'ND Tools'}
  requires
    VCL40, RVCL30, RVCLS30;
  contains
    NDCsBlue;
  end.
```

It is also useful to create a batch file to compile your custom package. The following is a sample of the batch file that can be used to compile our example component. This batch file is designed to be used from a directory under C:\Rave4 such as C:\Rave4\Custom. Notice the DESIGNER compiler directive that is added to compile any code that is for the visual designer. Also notice the inclusion of the designer-only units in the RV directory (/U..\RV compiler option).

```
------------------------
Contents of RANTCOMP.BAT
------------------------
  @echo off
  if "%NDD4%"=="" goto endsetup
  REM ********************
  REM Compile Rave Package
  REM ********************
  REM
  REM Usage: RANTCOMP packagename
  REM
  REM Note: The packagename parameter should not include the .dpk extension
  REM
  REM Desc: This batch file will compile a package containing a RAVE add-on
  REM       and place the .BPL in the parent directory (normally C:\Rave4).
  REM       It is meant to be run from a directory under Rave4 such as
  REM       C:\Rave4\RANT. The Rave library files must reside in C:\Rave4\RV.
  REM
  REM ***************************************************
  %NDD4%\bin\dcc32 %1.dpk /b /h /w /LE.. /U..\RV -$d-l-n+p+r-s-t-w- /DDESIGNER
  if errorlevel 1 goto enderror
  del *.dcu >nul
  del %1.dcp >nul
  goto endok
  :enderror
  echo Error!
  goto endok
  :endsetup
  echo You will need to define a variable in your AUTOEXEC.BAT file
  echo called NDD4 that points to your Delphi 4.0 directory and reboot
  echo before using this batch file.  Example:
  echo   SET NDD4=C:\Program Files\Borland\Delphi4
  :endok
```

2.5.4 - Directory Usage for Compiled Binaries

There are several binary directories under the C:\Rave4 main directory. The following is a list of these directories and how to rebuild them.

Directory	Description
C:\Rave4\D4	Delphi 4.0 binary files. Run C:\Rave4\Source\FullD4.BAT to rebuild.
C:\Rave4\D5	Delphi 5.0 binary files. Run C:\Rave4\Source\FullD5.BAT to rebuild.
C:\Rave4\D6	Delphi 6.0 binary files. Run C:\Rave4\Source\FullD6.BAT to rebuild.
C:\Rave4\C4	C++Builder 4.0 binary files. Run C:\Rave4\Source\FullC4.BAT to rebuild.
C:\Rave4\C5	C++Builder 5.0 binary files. Run C:\Rave4\Source\FullC5.BAT to rebuild.
C:\Rave4\C6	C++Builder 6.0 binary files. Run C:\Rave4\Source\FullC6.BAT to rebuild.
C:\Rave4\RV	Designer-only Rave files. Run C:\Rave4\Source\RaveD5.BAT to rebuild. These files should be used instead of the other binary files when compiling a package for inclusion into the Rave designer.

2.6 - Using TRaveComponent.OverrideProperties to handle changes in properties

```
-----------------------------------------------------------------------------
```

The OverrideProperties method is an easy way to intercept the normal reading and writing of your components properties. This is normally done to maintain backwards compatibility with existing project files that contain property data in a older format or under a different name. It may also be used to write data that does not fit into the normal types such as graphics files or record structures.

The first thing you will want to do is override the OverrideProperties method in your component class and call the Filer.OverrideProperty method for each property that you want to override.

```
  procedure TMyRaveComponent.OverrideProperties(Filer: TRaveFiler);

  begin { OverrideProperties }
    Filer.OverrideProperty('MyProperty',ReadMyProperty,WriteMyProperty);
  end;  { OverrideProperties }
```

You will also need to define a read and write method that will be called when a property of name, 'MyProperty', is encountered for your component type.

```
  procedure TMyRaveComponent.ReadMyProperty(Reader: TRaveReader);

  begin { ReadMyProperty }
  // code to read MyProperty from Reader.StreamHandler
  end;  { ReadMyProperty }

  procedure TMyRaveComponent.WriteMyProperty(Writer: TRaveWriter);

  begin { WriteMyProperty }
  // code to write MyProperty to Writer.StreamHandler
  end;  { WriteMyProperty }
```

There are many different methods of the TRaveWrite and TRaveReader classes (defined in RVCLASS.PAS) and you should become familiar with their operation before continuing.

Here's an example of an override that was needed because a property changed from a TStrings type to a string type:

```
  procedure TMyRaveComponent.ReadDescription(Reader: TRaveReader);

  var
    TempStrings: TStrings;
    ValueKind: TValueKind;

  begin { ReadDescription }
    With Reader do begin
      ValueKind := TValueKind(StreamHelper.ReadByte);
      Case ValueKind of
        vkPropList: begin { Read a TStrings object }
          TempStrings := TStrings.Create;
          try
            ReadProperties(TempStrings);
            Description := TempStrings.Text;
          finally
            TempStrings.Free;
          end; { tryf }
        vkString: begin { Read a normal string }
          Description := StreamHelper.ReadString;
        end;
      end; { case }
    end; { with }
  end;  { ReadDescription }
  procedure TMyRaveComponent.OverrideProperties(Filer: TRaveFiler);

  begin { OverrideProperties }
    Filer.OverrideProperty('Description',ReadDescription,nil);
  end;  { OverrideProperties }
```

Notice that we didn't define a write method for this property override. That is because we want the normal output format to be written so there is no need.

Rave also supports Delphi's TPersistent.DefineProperties interface for writing custom data to the stream but this is more limiting than Rave's OverrideProperties interface and should be used if you only want to read and write a custom format for your data.

2.7 - Using the Rave Listener Interface

A special interface has been built into all Rave components to allow them to communicate with each other. You do this by defining methods in a component class to make it a speaker of a specific conversation and then registering another class as the listener of that class. Note that it is the listener object's responsibility to add itself to the conversation of a speaker object at runtime. An example of where this is used in Rave is the TRaveCalcText component (the listener class) and the TRaveDataBand component (the speaker class). The conversation name is 'CalcNewData' and allows DataBand components to notify CalcText components when to perform calculations. The CalcText components adds itself as a listener to a specific DataBand defined by it's Controller property. Now that this conversation has been defined, it would be possible to create other speakers (as is the case for the component, TRaveCalcController). You could also create other listeners (as is the case for the component, TRaveCalcTotal).

2.7.1 - Creating a Rave speaker class

For this example we'll create a conversation type called ABC and create a simple speaker class. At a minimum, the following items must be defined in the speaker class:

```
    TRaveSpeakerClass = class(TRaveControl)
    protected
      ABCListenList: TRaveMethodList;

      procedure Changing(OldItem: TRaveComponent;
                         NewItem: TRaveComponent); override;
    public
      procedure AddListener(Conversation: string;
                            ListenMethod: TRaveListenEvent); override;
      procedure RemoveListener(Conversation: string;
                               ListenMethod: TRaveListenEvent); override;
      function Habla(Conversation: string): boolean; override;
    end; { TRaveSpeakerClass }
```

1: ABCListenList field: A listener list, of type TRaveMethodList, to keep track of all listeners. There needs to be a seperate listener list for each conversation that the speaker class can speak.

2: AddListener method: Override this method to add a listener to a conversation (listener list). Since the speaker class can be a speaker of several conversations, the Conversation parameter should be checked to add it to the proper listener list. Even if the speaker class only speaks one conversation, the Conversation parameter should still be checked since there is nothing stopping a listener class from attempting it add itself to a conversation that doesn't exist.

```
 procedure TRaveSpeakerClass.AddListener(Conversation: string;
                                         ListenMethod: TRaveListenEvent);

 begin { AddListener }
   inherited AddListener(Conversation,ListenMethod);
   If CompareText(Conversation,'ABC') = 0 then begin
     If not Assigned(ABCListenList) then begin
       ABCListenList := TRaveMethodList.Create;
     end; { if }
     ABCListenList.AddMethod(TMethod(ListenMethod));
   end; { if }
 end;  { AddListener }
```

3: RemoveListener method: Override this method to remove a listener from a conversation (listener list). The same rules specified above for checking the Conversation parameter apply to RemoveListener as well.

```
 procedure TRaveSpeakerClass.RemoveListener(Conversation: string;
                                            ListenMethod: TRaveListenEvent);

 begin { RemoveListener }
   inherited RemoveListener(Conversation,ListenMethod);
   If (CompareText(Conversation,'ABC') = 0) and
    Assigned(ABCListenList) then begin
     ABCListenList.RemoveMethod(TMethod(ListenMethod));
   end; { if }
 end;  { RemoveListener }
```

4: Habla method: Override this method to notify listener classes whether the speaker class can speak a particular conversation. Conversation names should not be case-sensitive so it is best to use the CompareText function when determining a match.

```
 function TRaveSpeakerClass.Habla(Conversation: string): boolean;

 begin { Habla }
   Result := CompareText(Conversation,'ABC') = 0;
 end;  { Habla }
```

5: Changing method: Override this method to track when a listner component has been deleted or replaced. See section 2.8 for more information in how to use the Changing method.

```
 procedure TRaveSpeakerClass.Changing(OldItem: TRaveComponent;
                                      NewItem: TRaveComponent);

 begin { Changing }
   inherited Changing(OldItem,NewItem);
   If Assigned(ABCListenList) and Assigned(OldItem) then begin
     If Assigned(NewItem) then begin
       ABCListenList.ReplaceObject(OldItem,NewItem);
     end else begin
       ABCListenList.RemoveObject(OldItem);
     end; { else }
   end; { if }
 end;  { Changing }
```

6: Speak method: Call this method whenever the speaker class wants to send a message to the listener objects. The first parameter is the listener list and the second parameter is the message. The structure of the Msg parameter is defined by the type of conversation being spoken. For the CalcNewData conversation, no data is needed so the Msg parameter is nil instead of an actual object. For more complicated conversations, actual objects could be sent to transfer more information.

```
 procedure Speak(List: TRaveMethodList; Msg: TObject);
```

2.7.2 - Creating a Rave listener class

Continuing with the previous example of the ABC conversation, we'll now create a simple listener class. At a minimum, a listener method must be defined in the listener class. The format for this method must match as follows (although the name of the method can and should be changed).

```
    TRaveListenerClass = class(TRaveControl)
    protected
      FSpeakerObj: TRaveComponent;

      procedure ABCListen(Speaker: TRaveComponent;
                          Msg: TObject);
      procedure SetSpeakerObj(Value: TRaveComponent);
    published
      property SpeakerObj: TRaveComponent read FSpeakerObj write SetSpeakerObj;
    end; { TRaveListenerClass }
```

1: The listener object must add itself to a specific conversation by calling the AddListener() method of a speaker class object:

```
    SpeakerObj.AddListener('ABC',ABCListen);
```

2: The listener object should make sure to remove itself from a conversation if it no longer wants to receive messages from a specific speaker:

```
    SpeakerObj.RemoveListener('ABC',ABCListen);
```

3: Typically, both of these calls can be combined into a single method for setting the property containing the reference to the speaker object.

```
    procedure TRaveListenerClass.SetSpeakerObj(Value: TRaveComponent);

    begin { SetSpeakerObj }
      If Value = FSpeakerObj then Exit;
      If Assigned(FSpeakerObj) then begin
        FSpeakerObj.RemoveListener('ABC',ABCListen);
      end; { if }
      FSpeakerObj := Value;
      If Assigned(FSpeakerObj) then begin
        FSpeakerObj.AddListener('ABC',ABCListen);
      end; { if }
    end;  { SetSpeakerObj }
```

4: The code that is inside of the ABCListen method will be executed whenever the speaker object calls the Speak() method for the 'ABC' conversation. The speaker object will be passed as the first parameter and the message will be passed as the second parameter. The structure of the Msg parameter is defined by the type of conversation being spoken. For the CalcNewData conversation, no data is needed so the Msg parameter is nil instead of an actual object. For more complicated conversations, actual objects could be sent to transfer more information.

2.7.3 - Property editors for speaker classes

Since several different classes can qualify as speakers for a conversation, the listener component should use the Habla method to determine if a component is a valid speaker rather than checking for specific class types. The following is a sample property editor for our sample speaker and listener classes. By using the Include function of the TRaveComponentPropertyEditor class, only component that speak the ABC conversation will be listed in the drop-down list for the TRaveListenerClass.SpeakerObj property.

```
interface
  TRaveABCSpeakerPropertyEditor = class(TRaveComponentPropertyEditor)
  protected
    function Include(Value: TComponent;
                     Data: longint): boolean; override;
  end; { TRaveABCSpeakerPropertyEditor }

implementation
  procedure RaveRegister;

  begin { RaveRegister }
    RegisterRavePropertyEditor(TypeInfo(TRaveComponent),TRaveListenerClass,
    'SpeakerObj',TRaveABCSpeakerPropertyEditor);
  end;  { RaveRegister }

  function TRaveABCSpeakerPropertyEditor.Include(Value: TComponent;
                                                  Data: longint): boolean;

  begin { Include }
    Result := (Value is TRaveComponent) and TRaveComponent(Value).Habla('ABC');
  end;  { Include }
```

2.8 - The TRaveComponent.Changing method

The Changing method allows components to respond to components that are being added, deleted or replaced during design-time. This is similar to Delphi's Notification method but is done a little differently. For component additions, OldItem will be nil and NewItem will be the component being added. For component deletions, OldItem will be the component being deleted and NewItem will be nil. For component replacements (ususally done during imports), OldItem will be the component being replaced and NewItem will be the component that should replace OldItem. If the component class keeps references to other components, it should check to see if those references match deleted or replaced components through the Changing method and take appropriate actions so that invalid component references are not called.

```
  procedure TSampleClass.Changing(OldItem: TRaveComponent;
                                  NewItem: TRaveComponent);

  begin { Changing }
    inherited Changing(OldItem,NewItem);
    If FComponentReference = OldItem then begin
      FComponentReference := NewItem; { Handles deletes and replaces }
    end; { if }
  end;  { Changing }
```

2.8.1 - Using NotifyChanging procedure at runtime

If you are deleting a Rave component at runtime, you will need to notify the other components in the same report project of this change or they attempt to reference an object that no longer exists. This is done through the NotifyChanging procedure.

```
  procedure NotifyChanging(OldItem: TRaveComponent;
                           NewItem: TRaveComponent);
```

For example, if you are going to delete a dataview you would call:

```
  DataView1 := RaveProject1.ProjMan.FindRaveComponent('ObseleteDataView',nil);
  NotifyChanging(DataView1,nil);
  RaveProject1.ProjMan.DeleteItem(DataView1); // Delete after notifying
```

If you are going to delete a normal component you would call:

```
  Page1 := RaveProject1.ProjMan.FindRaveComponent('CustReport.Page1',nil);
  Text1 := RaveProject1.ProjMan.FindRaveComponent('Text1',Page1);
  NotifyChanging(Text1,nil);
  Text1.Free; // Free after notifying
```

What this is doing above is telling all components that the component referenced in OldItem is going to be changed to NewItem (which is nil in these examples). Not that within Rave at design-time, this notification is normally handled automatically.

If you want to replace a component with another, you can also use NotifyChanging. This will automatically change all references from the old to the new dataview and keep the links intact. If you simply deleted the old dataview and then added a new one, any components such as databands or datatexts would no longer be pointing to a dataview. Here's an example of replacing one dataview with another:

```
  OldDataView := RaveProject1.ProjMan.FindRaveComponent('ObseleteDataView',nil);
  NewDataView := CreateDataView(CreateDataCon(RPDataSetConnection1));
  NotifyChanging(OldDataView,NewDataView);
  RaveProject1.ProjMan.DeleteItem(OldDataView); // Delete after notifying
```

2.9 - Hiding and Moving Rave Components in the Visual Designer

If you want to hide or move certain Rave components to another toolbar there are several ways to do this. The most difficult is to actually modify the RaveRegister procedure to either remove the component reference or assign it to a different component group (toolbar). However, a simpler method exists that will allow you to customize the components contained in the Rave designer for yourself or your end-users. This is done through the Windows registry. To hide a component, set the following string value to '1':

```
HKEY_CURRENT_USER\Software\Nevrona Designs\Rave\Components\TRaveBitmap\Hidden
```

Obviously this string is for the TRaveBitmap component. If you want to hide another component, simply replace TRaveBitmap with the full class name of the component. Setting Hidden to any value other than '1' or not having a Hidden string value will cause the component to appear in the visual designer. Note that reports containing hidden components can still be loaded, edited and printed. The resourceful user could create additional, 'hidden' components simply by copying and pasting any that already exist in the report.

To move a component to another group (toolbar), set the following string value to the name of the destination group (i.e. Report, Standard,...).

```
HKEY_CURRENT_USER\Software\Nevrona Designs\Rave\Components\TRaveBitmap\Group
```

Once again, this is an example of the string value for TRaveBitmap. If you want to move another component, simply replace TRaveBitmap with the full class name of the component.

3.0 - Interfacing with TRaveProjectManager

The Rave project manager class is the owner of all report, global page and data view components in a reporting project. It provides many methods and properties to interface with the reporting project.

If you need to interface with the project manager from within a component or editor, the global variable, ProjectManager of type TRaveProjectManager, provides what you need. To access the ProjectManager variable you must include the unit RVProj in your uses clause.

If you are interfacing with the project manager from within a Delphi or C++Builder application you will want to access the TRaveProject.ProjMan property instead of the global variable, ProjectManager. This is because multiple projects may be open at the same time.

3.1 - Public properties of TRaveProjectManager

property ReportList: TList (read only)
property GlobalPageList: TList (read only)
property DataViewList: TList (read only)

- These properties provide access to the reports, global pages and dataviews of the report project. Since ProjectManager owns all of these components, you can also access these components through the Components property just like any other Delphi component.

property ActiveReport: TRaveReport (read only)

- This property returns the currently active report. To activate a new report use the ActivateReport method.

property DataChanged: boolean (read/write*)

- DataChanged should be set to true whenever a change is made to the report project. Note that once DataChanged is set to true, it will only be set to false again when the project is saved or by calling the ClearChanged method.

property Printing: boolean (read only)

- Printing will be true if a report is currently being printed.

property Version: integer (read only)

- This property is the version of the currently loaded project. Once the project is saved it will be automatically upgraded to the current version.

property FileName: string (read/write)

- This property is the name of the file used by the Save and Load methods.

property StreamParamValues: boolean (read/write)

- This property should not normally be used. It is used internally to transfer parameter values to the end-user Rave designer.

property Saved: boolean (read/write)

- This property can be used to tell if the report project has been saved to the disk or not. When a project is loaded from disk or after is has been saved to disk, Saved will be true.

property Categories: TStrings (read/write/pub)

- This property defines the available report categories for the report project. Each line defines a name for a separate report category.

```
property Parameters: TStrings (read/write/pub)
```

- This property defines the project parameters for the report project. Each line defines a parameter name and values can be set or retrieved using the SetParam and GetParam methods.

```
property Units: TPrintUnits (read/write/pub)
```

- This property defines the units that will be used across the reporting project. Valid values include unInch, unMM, unCM, unPoint and unUser. When changing Units to values other than unUser, UnitsFactor will be set to the appropriate value. unUser allows custom values of the UnitsFactor property to handle any kind of units conversion.

```
property UnitsFactor: TRaveFloat (read/write/pub)
```

- This property defines the relationship between the units used across the report project and inches. For example, when using a Units value of unMM (millimeters), UnitsFactor would be 25.4 (25.4 mm per inch). Custom unit conversions can be set by setting this property to any value desired.

3.2 - Events of TRaveProjectManager

```
TImportConflictEvent = procedure(    CurrentItem: TRaveProjectItem;
                                 var ImportName: string) of object;
```

```
property OnImportConflict: TImportConflictEvent;
```

- This event will be called whenever a naming conflict is encountered during an import. This will allow you to change the name of the imported component to a unique name.

3.3 - Methods of TRaveProjectManager

```
function NewReport: TRaveReport;
function NewGlobalPage: TRavePage;
function NewDataView: TRaveDataView;
```

- These methods can be used to create a new report, global page or data view to the report project. The new item will be returned as the result of the function. For creating dataviews dynamically at runtime you should use the CreateDataView function which is covered in section 4.4.

```
procedure New;
```

- New will create a new report project.

```
procedure Save;
```

- Save will save the report project to the file specified by the FileName property.

```
procedure Load;
```

- Load will load the report project from the file specified by the FileName property.

```
procedure Unload;
```

- Unloads all items from current project.

```
procedure LoadFromStream(Stream: TStream);
```

- Loads a report project from Stream. The Unload method should be called before LoadFromStream is called.

```
procedure SaveToStream(Stream: TStream);
```

- Save the report project out to Stream.

```
procedure ExportProject(ExportFileName: string;
                        Items: TList);
```

- Creates an export report project containing the items listed in Items. Only Reports, Global Page and Data Views should be referenced in Items.

```
function ImportProject(ImportFileName: string;
                       AutoReplace: boolean): boolean;
```

- Imports an exported report project into the current project. AutoReplace will determine if all duplicate items are automatically replaced instead of using the OnImportConflict event.

```
procedure DeactivateReport;
```

- Deactivates the current report.

```
procedure ActivateReport(Report: TRaveReport);
```

- Activates Report as the current report.

```
function FindRaveComponent(Name: string;
                           DefRoot: TRaveComponent): TRaveComponent;
```

- This function can be used to find a component of a given name. The Name parameter can include the owner name as well as the component name (e.g. "Report1.Page2") with the DefRoot parameter set to nil. The other method of calling FindRaveComponent is to pass the component name in the Name parameter and the owner component in the DefRoot parameter.

  NOTE: The ProjectManager component owns all reports, global pages and dataviews. Report components own their report pages and all page components are the owner for all components contained within.

```
function GetUniqueName(BaseName: string;
                       NameOwner: TRaveComponent;
                       UseCurrent: boolean): string;
```

- This function will calculate a unique name for a component. This can be useful when creating components within a report wizard. The BaseName property is the starting point used for the component name. NameOwner is the owner component that will be used to search for naming conflicts. UseCurrent tells the function whether to attempt to use BaseName first before appending unique suffixes. When UseCurrent is true, the BaseName parameter should consist of the class base name appended to the desired name separated by a '|' character. So for a text component that you want to name 'MyText' you would pass the string 'MyText|Text' as the BaseName parameter and UserCurrent set to true.

  Examples:

  ```
  TxtComp.Name := ProjectManager.GetUniqueName('Text',PageComp,false);

  TxtComp.Name := ProjectManager.GetUniqueName('ReportTitle|Text',PageComp,true);
  ```

```
  procedure DeleteItem(Item: TRaveProjectItem;
                       Notify: boolean);
```

- This method is used to delete TRaveProjectItem components such as reports, global pages and dataviews from the report project. In most cases, Notify should be set to true to notify the designer of the deleted item.

```
  procedure SetParam(Param: string;
                     Value: string);
```

- This procedure set the project parameter, Param, to the text specified in Value.

```
  function GetParam(Param: string): string;
```

- This function will return the value of the project parameter, Param.

```
  procedure ClearChanged;
```

- The procedure should be called to set the DataChanged property to false. This procedure should be called with caution since it may cause changes to the report project to not be saved.

4.0 - Interfacing with TRaveDataSystem

The Rave data system is controlled by the TRaveDataSystem class and the RaveDataSystem global variable. Through the RaveDataSystem object you can interface with data connection components to request actions or retrieve data. The RaveDataSystem interface provides more flexibility but is also more complicated. For most cases, the functionality provided by the TRaveDataView component is sufficient and is much simpler.

4.1 - Properties of TRaveDataSystem

```
  property RTConnectList: TStringList (read only)
```

- This is a list of all runtime data connection names. A runtime data connection is one that exists in a running application. The Object array property of RTConnectList contains the TRaveDataConnection object that contains additional information about the runtime data connections.

```
  property DTConnectList: TStringList (read only)
```

- This is a list of all design-time data connection names. A design-time data connection is one that exists on a form currently loaded in Delphi or C++Builder. The Object array property of DTConnectList contains the TRaveDataConnection object that contains additional information about the design-time data connections.

4.2 - Events of TRaveDataSystem

```
  TRaveDataResult = (drContinue,drAbort,drPause);
  TTimeoutEvent = procedure(    DataSystem: TRaveDataSystem;
                                Counter: integer;
                                Timeout: integer;
                                EventType: integer;
                                Connection: string;
                                First: boolean;
                            var DataResult: TRaveDataResult) of object;

  property OnSmallTimeout: TTimeoutEvent
  property OnLargeTimeout: TTimeoutEvent
```

- These events are called when a timeout occurs while communicating with a data connection. The small timeout event will be encountered first and should serve as a warning that communication has stalled. This may occur because of the data connection performing some type of action (i.e. opening a table). The large timeout event will be called after the delay is longer than the configured maximum and the user should then be prompted for action. The DataSystem parameter is the current TRaveDataSystem object that is being used. Counter is the current timing count (in 1/100ths of a second). Timeout is the maximum time that is alotted for this event to execute. EventType is the type of event that is being executed. Connection is the name of the data connection and First is set to true if this is the first time the small or large timeout has occured for this timeout. The timeout events will be continually called every 1/100th of a second so processing should be brief. The DataResult parameter allows the event to control the timeout processing. The default is drContinue which allows continued attempts to perform the data event. drAbort will abort the data event while drPause will suspend attempted retries until a drContinue is passed back.

```
  TRaveDataAction = (daOpen,daClose);
  TDataActionEvent = procedure(DataSystem: TRaveDataSystem;
                               DataAction: TRaveDataAction) of object;
  property OnDataAction: TDataActionEvent
```

- This event can be used to keep track of when data connections are opened and closed by the Rave data system. The Rave designer uses this event to update the Data connection status LED during report execution.

4.3 - Methods of TRaveDataSystem

```
  function GainControl: boolean;
```

- This function should be called at the beginning of a Rave data session. Only one application can have control of the Rave data system at one time. If control cannot be gained, the result of the function will be false. Otherwise if control is successfully gained, the result will be true.

```
  procedure ReleaseControl;
```

- This procedure must be called at the end of a Rave data session to release control of the Rave data system.

```
  function IsUnique(Name: string): boolean;
```

- This function will check for duplicate data connections of the same connection name. This function will not normally need to be called.

```
  procedure UpdateConnections;
```

- This procedure will query the Windows environment for active data connections and update the RTConnectList and DTConnectList properties.

```
  function ReadStr: string;
  function ReadInt: integer;
  function ReadBool: boolean;
  function ReadFloat: extended;
  function ReadCurr: currency;
  function ReadDateTime: TDateTime;
  procedure ReadBuf(var Buffer;
                    Len: integer);
  function ReadPtr(Len: integer): pointer;
```

- These functions will read data from the Rave communication buffers. These functions are not normally needed as the TRaveDataView class handles the reading of the communication buffers.

```
procedure WriteStr(Value: string);
procedure WriteInt(Value: integer);
procedure WriteBool(Value: boolean);
procedure WriteFloat(Value: extended);
procedure WriteCurr(Value: currency);
procedure WriteDateTime(Value: TDateTime);
procedure WriteBuf(var Buffer;
                       Len: integer);
```

- These functions will write data to the Rave communication buffers. These functions are not normally needed as the TRaveDataView class or the CallEvent method handles writing to the communication buffers.

```
procedure ClearBuffer;
```

- This procedure will clear the contents of the communication buffer and reset the buffer pointer to the beginning. As with the ReadXxxx and WriteXxxx methods, ClearBuffer should not normally be called as TRaveDataView or the CallEvent method handles when the buffer needs to be cleared.

```
function OpenDataEvent(AName: string;
                       DataCon: TRaveDataConnection): boolean;
```

- This function will open a data connection for a given connection name. If the connection is opened successfully, the result of the function will be true. This function is normally called by the data view object itself and should not be called directly.

```
procedure CloseDataEvent(DataCon: TRaveDataConnection);
```

- This function will close a data connection that was previously opened by a call to OpenDataEvent. This function is normally called by the data view object itself and should not be called directly.

```
function CallEvent(EventType: integer;
                   DataCon: TRaveDataConnection): boolean;
```

- This function will execute a specific data event for the given data connection. The data connection must already have been opened with a call to OpenDataEvent. EventType must be one of the valid data event constants: DATAFIRST, DATANEXT, DATAEOF, DATAGETCOLS, DATAGETROW, DATASETFILTER, DATAGETSORTS, DATASETSORT, DATAOPEN, DATARESTORE, DATAACKNOWLEDGE, DATAFREEALTBUF. The format of the data in the communication buffer differs depending upon the type of data event. This function is normally called by the data view object itself and should not be called directly.

```
procedure PrepareEvent(EventType: integer);
```

- This method will prepare the communication buffer for the given data event. Any additional data values that are required must still be written using the WriteXxxx methods.

```
procedure CreateAltFileMap(BufIdx: integer);
```

- This method will create an alternate communication buffer. This method is used internally by the data connection components and should not be called directly.

```
  procedure FreeAltFileMap(DataCon: TRaveDataConnection);
```

- This method will free an alternate communication buffer. This method is used internally by the data connection components and should not be called directly.

4.4 - Global functions related to the Rave data system

There are several global functions defined in the RVData unit that are useful for dealing with data related the Rave data system.

```
  function ProcessDataStr(DefaultDataView: TRaveDataView;
                          Value: string): string;
```

- This function will process a DataView and DataText combination and return the text that it evaluates to.

```
  function CreateDataCon(RPConnection: TRPCustomConnection): TRaveDataConnection;
```

- This function will create a TRaveDataConnection object for a given data connection component (TRPCustomConnection or one of its descendents). The result of this function will normally be used in conjunction with the CreateDataView function.

```
  function CreateDataView(DataCon: TRaveDataConnection): TRaveDataView;
```

- This function will create a DataView object for the given TRaveDataConnection object (normally created through a call to the CreateDataCon function). Do not free the DataCon object that is passed into this function since it will belong to the DataView and will be freed by the DataView itself. The following code is how a dataview would normally be created dynamically at runtime:

```
      MyDataView := CreateDataView(CreateDataCon(TableConnection1));
```

```
  function CreateFieldName(DataViewName: string;
                           FieldName: string): string;
```

- This function will create a valid field name for a given data view. This function does not normally need to be called since it is called from within the CreateFields procedure.

```
  procedure CreateFields(DataView: TRaveDataView;
                         DeletedFields: TStringList;
                         ReplacedFields: TStringList;
                         DoCreate: boolean);
```

- This function will create field components for a specific data view. If non-nil values are passed in for the DeletedFields and ReplacedFields parameters, the CreateFields procedure will return existing field components that will be deleted or replaced if the DoCreate parameter is set to true. In the Rave designer, these parameters are used to warn the user that fields are about to be deleted or replaced. This function does not normally need to be called since it is called from within the CreateDataView function.

```
  function PerformLookup(LookupDataView: TRaveDataView;
                         LookupValue: string;
                         LookupValueField: TRaveDataField;
                         LookupField: TRaveFieldName;
                         LookupDisplay: TRaveFieldName;
                         LookupInvalid: string): string;
```

- This function will perform a lookup for the given parameters and return the text that it evaluates to.

```
  procedure DataViewFirst(DataView: TRaveDataView;
                          DetailKey: TRaveFieldName;
                          MasterDataView: TRaveDataView;
                          MasterKey: TRaveFieldName;
                          SortKey: string);
```

- This function will initialize a data view for the given filter values (DetailKey, MasterDataView and MasterKey) and sort order (SortKey).

5.0 - Interfacing with TRaveDesigner

The TRaveDesigner class allows you to view and control the settings of the Rave design environment. Access to the designer object is done through the TRavePage.Designer property. Note that only projects that are loaded in the Rave visual designer will have designers assigned to the TRavePage.Designer property so it is not possible to interface with the TRaveDesigner class from within a Delphi or C++Builder application. If generic access to the currently displayed designer is needed, a global variable, CurrentDesigner, is provided in the RVClass unit.

5.1 - Properties of TRaveDesigner

```
  property GridPen: TPen (read/write)
```

- This property defines the pen used to draw the grid of the current page.

```
  property MinimumBorder: TRaveFloat (read/write)
```

- This property defines the border around the page that will be used by the ZoomPage and ZoomPageWidth methods. The default is 1.0 inches.

```
  property Page: TRavePage (read only)
```

- This property defines the page component that this designer is currently being used to edit.

```
  property Selections: integer (read only)
```

- This property will return the number of components that are in the current selection.

```
  property Selection[Index: integer]: TRaveComponent (read only)
```

- This property will allow access to specific selected component. The Index property is 0 based.

```
  property ZoomFactor: TRaveFloat (read/write)
```

- This property sets or returns the current zoom factor.

5.2 - Methods of TRaveDesigner

```
  procedure AddPip(Index: byte;
                   Control: TRaveControl;
                   Cursor: TCursor;
                   X: TRaveUnits;
                   Y: TRaveUnits);
```

- This method will create a selection/sizing pip at the current position X,Y. When creating most components, the default pip locations will be sufficient, however the pip methods will allow for custom pip placement and functionality. Each pip for a specific component must have a unique Index value. A component can respond to pip movements by overriding the TRaveControl.PipSize method.

```
  procedure UpdatePip(Index: byte;
                      Control: TRaveControl;
                      X: TRaveUnits;
                      Y: TRaveUnits);
```

- This method will update the position of the pip defined by Index.

```
  procedure RemovePips(Control: TRaveControl);
```

- This method removes all pips for the component defined by Control.

```
  procedure SwitchPips(RavePip: TRavePip;
                       SwitchIdx: byte);
```

- This method will switch the pips defined by RavePip and the pip for the same component at index SwitchIdx. This is normally done when a pip of a component is dragged past another pip during resizing.

```
  procedure Modified;
```

- This method should be called whenever modifications have been made to the components in the current designer. This will signal the project manager that the changes need to be saved and will also allow the designer to update it's various displays.

```
{ Selection methods }
  procedure DeselectControl(Control: TRaveComponent);
```

- This method removes the component defined by Control from the current selection.

```
  procedure ClearSelection;
```

- This method removes all components from the current selection.

```
  procedure SelectControl(Control: TRaveComponent);
```

- This method adds the component defined by Control to the current selection.

```
  procedure ToggleControl(Control: TRaveComponent);
```

- This method toggles the selected status of the component defined by Control.

```
  function IsSelected(Control: TRaveComponent): boolean;
```

- This method will return whether the component defined by Control is in the current selection.

```
  procedure DeleteSelection;
```

- This method will delete all components in the current selection.

```
  procedure CopySelection;
  - This method will copy the selected components to the Windows clipboard.

  procedure PasteSelection;

  - This method will paste the components previously copied to the Windows
    clipboard into the current page.

  procedure SelectChildren(Control: TRaveComponent);

  - This method will add all children components of the component defined by
    Control to the current selection.

  procedure SelectType(ProjectItem: TRaveProjectItem;
                       RaveClass: TClass);

  - This method will add all components on the current page that are the same
    type as RaveClass to the current selection.

  procedure MoveSelection(X,Y: TRaveUnits);

  - This method will move the top, left corner of the selected component(s) to
    the position defined by X,Y.

{ Find methods }
  function FindControl(Name: string): TRaveComponent;

  - This method will search the current page for the component of the given
    Name.

  function FindControlAt(X,Y: TRaveUnits): TRaveControl;

  - This method returns the name of the Control at X,Y. If no Control is found
    at that location, then this will return the current Page component.

  function FindContainerAt(X,Y: TRaveUnits;
                           NewChild: TClass): TRaveControl;

  - This method returns the name of a valid parent component at location X,Y
    for a component of type NewChild.  If no valid parent is found, the page
    will be the default container.

{ Position methods }
  function XI2D(Value: TRaveUnits): longint;
  function YI2D(Value: TRaveUnits): longint;

  - These methods convert units measurements (inches) to printer canvas
    measurements (dots).

  function XD2I(Value: longint): TRaveUnits;
  function YD2I(Value: longint): TRaveUnits;

  - These methods convert the printer canvas measurement (dots) to
    unit measurements (inches).

  function SnapX(Value: TRaveUnits): TRaveUnits;
  function SnapY(Value: TRaveUnits): TRaveUnits;

  - These methods will take a Value which could be the result of some
    calculation and converts it to the nearest grid value. For example, if grid
    spacing is set for 0.1 inches and you pass the function a value of 1.12,
    then it will return a value of 1.1.

{ Zooming methods }
  function ZoomToRect(X1,Y1,X2,Y2: TRaveUnits): TRaveFloat;
```

- This method changes both the page position and zoom factor so that the area indicated by X1,Y1,X2,Y2 will occupy the page design. This function examines both the vertical and horizontal sizes when determining what zoom factor should be used so that the full rectangle is shown.

```
  procedure ZoomPage;
```

- This method changes the zoom factor so that the full page plus the Minimum border fills the page designer screen.

```
  procedure ZoomPageWidth;
```

- This method changes the zoom factor so that the form width plus the minimum border fills the page designer screen.

```
  procedure ZoomSelected;
```

- This method changes the zoom factor so that all selected items will fill the page designer screen. If only one item is selected, then the zoom factor will be changed so that single item is full screen.

```
  procedure ZoomIn(X,Y: TRaveUnits);
```

- This method increases the zoom factor by the zoom increment amount set by the preferences.

```
  procedure ZoomOut;
```

- This method decreases the zoom factor by the zoom increment amount set by the preferences.

```
  procedure CenterWindow(X,Y: TRaveUnits);
```

- This method changes the page layout to center at the point indicated by the position, X,Y. This procedure does NOT change the zoom factor.

```
  procedure AlignSelection(AlignType: integer);
```

- This method will perform the alignment action defined by AlignType. Valid values for AlignType include: RaveAlignLeft, RaveAlignHCenter, RaveAlignRight, RaveAlignHCenterInParent, RaveAlignHSpace, RaveAlignEquateWidths, RaveAlignTop, RaveAlignVCenter, RaveAlignBottom, RaveAlignVCenterInParent, RaveAlignVSpace, RaveAlignEquateHeights, RaveAlignMoveForward, RaveAlignMoveBehind, RaveAlignBringToFront, RaveAlignSendToBack, RaveAlignTapLeft, RaveAlignTapRight, RaveAlignTapUp, RaveAlignTapDown, RaveAlignTapHSizeDown, RaveAlignTapHSizeUp, RaveAlignTapVSizeDown and RaveAlignTapVSizeUp.

6.0 - Creating a Rave Property Editor

The property editor interface of Rave allows you to create and respond to property editors through the TRavePropertyEditor class (RVTool.pas). The property editor can apply to a specific property of a specific component type, a specific property across all components or all properties of a specific type for a specific component or across all components.

6.1 Types of property editors

- Simple - This property editor only displays an edit box allowing the user to type in a string of data (e.g. TRavePage.Name).

- Listing - Like a Simple property editor, but also displays a drop down list of values to select from (e.g. TRectangle.Color).

- Editor - Like a Simple property editor, but also displays an editor button that brings up a custom dialog to edit the property (e.g. TRaveText.Font).

- Editor/Listing - This is a combination of the Listing and Editor property editors. It allows both a drop down list of values to select from and an editor button that brings up a custom dialog to edit the property (e.g. TRaveDataText.DataField)

6.2 Steps for creating a property editor

1: Create a class descending from TRavePropertyEditor (or one of its descendents). Below is a list of the methods that will normally need to be overriden for the different types of property editors (these methods are described in detail below).

Simple Property Editor - GetOptions, GetValue and SetValue
Listing Property Editor - GetOptions, GetValue, SetValue and GetValues
Editor Property Editor - GetOptions, GetValue and Edit
Editor/Listing Property Editor - GetOptions, GetValue, SetValue, GetValues and Edit

```
      type
        TMyPropertyEditor = class(TRavePropertyEditor)
        public
          function GetOptions: TPropertyOptionsSet; override;
          function GetValue: string; override;
          procedure SetValue(Value: string); override;
        end; { TMyPropertyEditor }
```

2: Create methods for GetOptions, GetValue, SetValue, GetValues and/or Edit as necessary. For examples, there are many property editors defined in this unit and other units in the Rave library.

3: Register the property editor by calling RegisterRavePropertyEditor from within a global scope procedure named RaveRegister (exact case required). The PropType parameter should be the type info structure for the property type you are registering. The function TypeInfo() will return the value you need. The ControlClass parameter defines the class of components that this property editor applies to. If this parameter is nil, then the property editor applies to all component classes. The PropName parameter allows you to limit the property editor to only properties of a specific name. If this field is blank then the property editor applies to all properties with a type info of PropType. The EditorClass property should be the class type of the property editor you are registering. See the RaveRegister procedure in this unit for examples.

```
      procedure RaveRegister;

      begin { RaveRegister }
        RegisterRavePropertyEditor(TypeInfo(integer),TMyComponent,'MyValue',
         TMyPropertyEditor);
      end;  { RaveRegister }
```

4: Include the unit containing the property editor in a Delphi package and register that with RAVE through Edit|Preferences|Components.

6.3 Properties of TRavePropertyEditor

```
  property Instance[Index: integer]: TRaveComponent (read only)
```

- This property returns the Index'th component being edited. See also InstCount.

```
property InstCount: integer (read only)
```

- This property returns the numbers of components that are currently being edited by this property editor (i.e. multiple components are selected). If poMultiSelect is not in the property editor options, this value will always be 1. See also Instance property.

```
property Name: string (read only)
```

- This property returns the name of the property field being edited.

```
property Options: TPropertyOptionsSet (read only)
```

- This property returns the current options for this property editor. See also TPropertyOptions and GetOptions.

```
property PropInfo[Index: integer]: PPropInfo (read only)
```

- This property will return the PPropInfo structure for the instance specified by Index. See Delphi's TypInfo unit for a description of PPropInfo.

```
property Value: string (read/write)
```

- This property sets or returns the value of the property as a string. See also GetValue, SetValue.

6.4 Methods of TRavePropertyEditor

```
procedure Modified;
```

- This method should be called whenever a property value is modified so that the Property Panel can refresh its display.

```
function GetOrdValue(Index: integer): integer;
function GetFloatValue(Index: integer): extended;
function GetStrValue(Index: integer): string;
function GetVariantValue(Index: integer): variant;
```

- These methods should be called to get the property value of the instance specified by Index. If the property is a set, class or pointer type, use GetOrdValue and typecast it appropriately.

```
procedure SetOrdValue(Value: integer);
procedure SetFloatValue(Value: extended);
procedure SetStrValue(Value: string);
procedure SetVariantValue(Value: variant);
```

- These methods should be called to set the property value of all instances connected to this property editor. If the property is a set, class or pointer type, use SetOrdValue and typecast it appropriately. Note however, if the property is a class type and the object is owned by the component (i.e. Font), you should iterate through the Instance property assigning the data instead of overwriting the object pointer which is what SetOrdValue will do.

```
function SameValue(TestComp: TRaveComponent;
                   TestValue: string): boolean;
```

- This method is used internally by the property panel to determine whether the current value is different than the default (Highlight Changes option). You will should not need to call this method directly.

6.5 Virtual methods of TRavePropertyEditor

```
function GetOptions: TPropertyOptionsSet;
```

- This virtual method should be overriden to return the set of options for this property editor. See also Options property and TPropertyOption.

```
    poEditor - This property has a dialog editor
    poListing - This property returns a listing of values
    poNoSort - The listing returned by this property should not be sorted
    poMultiSelect - This property can be edited across a multiple selection
    poLiveUpdate - This property is updated whenever any change is made
    poReadOnly - This property can only be edited through the list or editor
    poPassword - Edit/Display this property with *'s
    poRefreshAll - When this property is changed, all windows will be
                   refreshed.  Used with properties such as Locked and
                   Mirror that drastically change the contents of the entire
                   Property Panel.
```

```
function GetValue: string;
procedure SetValue(Value: string);
```

- These two virtual methods should be overriden to set and return the value of the property as a string. If poLiveUpdate is in the property editors options, SetValue will be called for each keypress the user makes in the properly editor's edit box. See also Value property.

```
procedure GetValues(ValueList: TStrings);
```

- This virtual method should be overriden to return the list of available options in the TStrings object, ValueList. If poNoSort is not in the Options property, the items will be sorted alphabetically. This method will only be called if poListing is in the property editors options.

```
procedure Edit;
```

- This virtual method will be called whenever the user presses the editor button on the property panel. Normally you will want to display a dialog allowing the user to edit a property that is too complicated for a simple text string. This method will only be called if poEditor is in the property editors options.

```
procedure PaintValue(Canvas: TCanvas;
                     Rect: TRect;
                     DefaultValue: string);
```

- This virtual method allows a property editor to customize the contents of the property panel. You will not normally need to override this method but it can be used to provide more visual information on a property value. The color, line style and fill style property editors are examples of where this can be useful.

7.0 - Creating a Rave Component Editor

The component editor interface of Rave allows you to create and respond to menu items that you create on a components popup menu (from a right click).
This is done through the TRaveComponentEditor class (RVTool.pas).
This will allow you to perform actions such as opening a dialog to allow the

user to modify the component's properties. Creating a component editor requires the following steps:

7.1 Steps for creating a component editor

1: Create a class descending from TRaveComponentEditor and override the AddToMenu and RunFromMenu procedures.

```
      type
        TMyComponentEditor = class(TRaveComponentEditor)
        public
          procedure AddToMenu(AddMenuItem: TAddMenuItemProc); override;
          procedure RunFromMenu(ID: integer); override;
        end; { TMyComponentEditor }
```

2: Create a method for AddToMenu and call AddMenuItem for each line you want to create in the component's popup menu (see TAddMenuItemProc definition below). ID should be unique for this component editor and greater than 0 for each menu item. ParentID should be 0 to place the new menu item on the first level. To make a child menu from an existing menu item, pass the parent's ID value as ParentID.

```
      procedure TMyComponentEditor.AddToMenu(AddMenuItem: TAddMenuItemProc);

      begin { AddToMenu }
        AddMenuItem('MenuItem 1',1,0);
        AddMenuItem('MenuItem 2',2,0);
        AddMenuItem('SubMenuItem 2a',21,2);
        AddMenuItem('SubMenuItem 2b',22,2);
        AddMenuItem('-',0,0); // Create a divider
        AddMenuItem('MenuItem 3',3,0);
        AddMenuItem('MenuItem 4',4,0);
      end;  { AddToMenu }
```

3: Create a method for RunFromMenu which will be called whenever a user selects a menu item from a components popup menu. The ID value will be equal to a value given in the previous calls to AddMenuItem from the AddToMenu procedure. If an ID of -1 is given, the default action should be taken (i.e. User double-clicked the component. You can get access to the component currently being edited through the Instance property. NOTE: Menu Items which are parents to other menu items will not cause a RunFromMenu call.

```
      procedure TMyComponentEditor.RunFromMenu(ID: integer);

      begin { RunFromMenu }
        Case ID of
          -1,1: begin
          // MenuItem1 action (double-click default)
          end;
          21: begin
          // MenuItem 2a action
          end;
          22: begin
          // MenuItem 2b action
          end;
          3: begin
          // MenuItem 3 action
          end;
          4: begin
          // MenuItem 4 action
          end;
        end; { case }
      end;  { RunFromMenu }
```

4: Register the component editor by calling RegisterRaveComponentEditor from within a global scope procedure named RaveRegister (exact case required). The ControlClass parameter should be the class type of the component you are defining an editor for and the EditorClass parameter should be the class type of the component editor you are registering.

```
      procedure RaveRegister;

      begin { RaveRegister }
        RegisterRaveComponentEditor(TMyComponent,TMyComponentEditor);
      end;  { RaveRegister }
```

5: Include the unit containing the component editor in a Delphi package and register that with RAVE through Edit|Preferences|Components.

8.0 - Creating a Rave Project Editor

The project editor interface of Rave allows you to create two types of experts.

The first type of project editor can respond to menu items that you create on the Rave Tools menu through the TRaveProjectEditor class (RVTool.pas). This will allow you to perform actions such as starting a report wizard that will create or modify a report and all of the pages and components within. See section 8.1 for more details on menu-based project editors.

The second type of project editor can respond to project events. Examples of project events includes when a new project is opened, after a report is printed or when a data connection is opened. See section 8.3 for more details on event-based project editors.

There is no limitation preventing an event-based project editor from using a menu-based interface as well. In fact, you will find that most of the event-based project editors such as the Rave Data Recorder or Rave Data Randomizer create a menu interface to configure the project editor.

8.1 Steps for creating a menu-based project editor

1: Create a class descending from TRaveProjectEditor and override the AddToMenu and RunFromMenu procedures.

```
      type
        TMyProjectEditor = class(TRaveProjectEditor)
        public
          procedure AddToMenu(AddMenuItem: TAddMenuItemProc); override;
          procedure RunFromMenu(ID: integer); override;
        end; { TMyProjectEditor }
```

2: Create a method for AddToMenu and call AddMenuItem for each line you want to create in the Tools menu (see TAddMenuItemProc definition above). ID should be unique for this project editor and greater than 0 for each menu item. ParentID should be 0 to place the new menu item on the first level. To make a child menu from an existing menu item, pass the parent's ID value as ParentID.

```
      procedure TMyProjectEditor.AddToMenu(AddMenuItem: TAddMenuItemProc);

      begin { AddToMenu }
        AddMenuItem('MenuItem 1',1,0);
        AddMenuItem('MenuItem 2',2,0);
        AddMenuItem('SubMenuItem 2a',21,2);
```

```
    AddMenuItem('SubMenuItem 2b',22,2);
    AddMenuItem('-',0,0); // Create a divider
    AddMenuItem('MenuItem 3',3,0);
    AddMenuItem('MenuItem 4',4,0);
  end;  { AddToMenu }
```

3: Create a method for RunFromMenu which will be called whenever a user selects a menu item from the Tools menu. The ID value will be equal to a value given in the previous calls to AddMenuItem from the AddToMenu procedure. NOTE: Menu Items which are parents to other menu items will not cause a RunFromMenu call.

```
  procedure TMyProjectEditor.RunFromMenu(ID: integer);

  begin { RunFromMenu }
    Case ID of
      -1,1: begin
      // MenuItem1 action (double-click default)
      end;
      21: begin
      // MenuItem 2a action
      end;
      22: begin
      // MenuItem 2b action
      end;
      3: begin
      // MenuItem 3 action
      end;
      4: begin
      // MenuItem 4 action
      end;
    end; { case }
  end;  { RunFromMenu }
```

4: Register the project editor by calling RegisterRaveProjectEditor from within a global scope procedure named RaveRegister (exact case required). The EditorClass parameter should be the class type of the project editor you are registering.

```
  procedure RaveRegister;

  begin { RaveRegister }
    RegisterRaveProjectEditor(TMyProjectEditor);
  end;  { RaveRegister }
```

5: Include the unit containing the project editor in a Delphi package and register that with RAVE through Edit|Preferences|Components.

8.2 Special notes for creating components at design-time
----

(1) Most of the interface to the components of a project are through the ProjectManager component. ProjectManager is of type TRaveProjectManager and is located in the unit RVProj.pas. To create a new report call ProjectManager.NewReport. To create a new GlobalPage call ProjectManager.NewGlobalPage and to create a new data view call ProjectManager.NewDataView. To create a new report page, call ProjectManager.ActiveReport.NewPage. To create other types of components follow the directions below.

(2) You will need to assign the owner for the component using the AOwner parameter of the Create constructor. Any TRaveField components should have the appropriate TRaveDataView component as their owner. Any other components that are placed on a Global Page or Report Page should use the

TRavePage object as their owner.

(3) You will need to assign the Parent property for all components. For components being placed on a Global Page or Report Page, the parent should be the graphical owner of the component (e.g. If you want to place a component in a TRaveSection, you need to set the Parent property to the TRaveSection component). For all other components, the Parent property should be set to the same as the Owner property.

(4) You will need to set the Name property for all components. Call ProjectManager.GetUniqueName to make sure you are using a unique name. The BaseName parameter is the base part of the name that you want to use (i.e. "Section"). The NameOwner parameter is the Owner component whose namespace you will be checking against (normally the TRavePage object). The UseCurrent parameter tells whether you want to try the current BaseName before appending digits on the end (i.e. Try "CustomerName" before trying "CustomerName1", "CustomerName2",..).

(5) For visual components (anything deriving from TRaveControl) you will need to initialize the Top, Left, Width and Height properties. These should be the location relative to the upper left corner of the component's Parent.

(6) For all components, you will need to call AddComponent(Component) to signal the visual designer that the component has been added to the visual designer. If your wizard is deleting component that have already been added to the report project, your will need to call DeleteComponent(Component) before calling the components Free method.

8.3 Steps for creating an event-based project editor

There are several project events that can be handled and a single project editor can handle as many project events that it needs to. The following is a list of project events currently supported (Item and Param values list the type of data that is passed in for each project event type). Before events such as peBeforeAppClose will use the return value of HandleEvent to determine whether to continue with (true) or cancel (false) the specific action.

peAfterAppOpen - Called after the Rave visual designer is opened
(Item = ProjectManager, Param = nil)

peBeforeAppClose - Called before the Rave visual designer is closed
(Item = ProjectManager, Param = nil)

peAfterProjectOpen - Called after a project is opened
(Item = ProjectManager, Param = nil)

peBeforeProjectClose - Called before a project is closed
(Item = ProjectManager, Param = nil)

peBeforeNewProject - Called before a new project is created
(Item = ProjectManager, Param = nil)

peAfterNewProject - Called after a new project is created
(Item = ProjectManager, Param = nil)

peBeforeNewReport - Called before a new report is created
(Item = nil, Param = nil)

peAfterNewReport - Called after a new report is created
(Item = the new TRaveReport object, Param = nil)

peBeforeNewReportPage - Called before a new report page is created
(Item = nil, Param = nil)

peAfterNewReportPage - Called after a new report page is created
(Item = the new TRavePage object, Param = nil)

peBeforeNewGlobalPage - Called before a new global page is created
(Item = nil, Param = nil)

peAfterNewGlobalPage - Called after a new global page is created
(Item = the new TRavePage object, Param = nil)

peBeforeNewDataView - Called before a new data view is created
(Item = nil, Param = nil)

peAfterNewDataView - Called after a new data view is created
(Item = the new TRaveDataView object, Param = nil)

peBeforeProjectSave - Called before a project is saved
(Item = ProjectManager, Param = nil)

peAfterProjectSave - Called after a project is saved
(Item = ProjectManager, Param = nil)

peBeforeReportPrint - Called before a report is printed
(Item = ProjectManager, Param = nil)

peAfterReportPrint - Called after a report is printed
(Item = ProjectManager, Param = nil)

peDataConOpen - Called when a data connection is opened
(Item = nil, Param = the TRaveDataConnection object)

peDataConClose - Called when a data connection is closed
(Item = nil, Param = the TRaveDataConnection object)

peDataConConnect - Called when a data connection is made
(Item = nil, Param = the TRaveDataConnection object)

peDataSystemOpen - Called when the Rave data system is opened
(Item = nil, Param = the TRaveDataSystem object)

peDataSystemClose - Called when the Rave data system is closed
(Item = nil, Param = the TRaveDataSystem object)

peDataSystemCallEvent - Called before a Rave data event is executed
(Item = nil, Param = pointer to a TDataSystemEventData record)

peDataSystemEventCalled - Called after a Rave data event is executed
(Item = nil, Param = pointer to a TDataSystemEventData record)

peShowAboutDialog - Called before the About dialog is shown. This will allow an alternate About Box dialog to be displayed. If the result of HandleEvent is false, the normal Rave about box will not be shown. NOTE: Any replacement About Box must clearly display the Nevrona Designs copyright as follows "Copyright © 1995-2000, Nevrona Designs". Failure to do so will be a violation of the software license.
(Item = nil, Param = nil)

peGetAppTitle - Called when the designer is initializing to retrieve an alternate title bar text. The RaveTitle string variable (defined in RVDefine.pas) should be set to the desired text.
(Item = nil, Param = nil)

peBeforePageChange - Called before the page designer is changed from one page to another. Refer to the CurrentDesigner variable (defined in

```
 RVClass.pas) for information about the designer that is being moved
 deselected.
 (Item = nil, Param = nil)

peAfterPageChange - Called after the page designer is changed from one
 page to another or when the Page Designer tab is selected.  Refer to the
 CurrentDesigner variable (defined in RVClass.pas) for information about the
 newly selected designer.
 (Item = nil, Param = nil)

peAddShortCuts - This event is called when the designer is building the list
 of shortcuts that it will support.  By default, all project editors
 that create tool menu items will have short cut entries created, but this
 peoject event allows you to create additional shortcuts.  Call
 RaveCreateShortCut (defined in RVClass.pas) to add a short cut to the list
 as follows:

   procedure RaveCreateShortCut(Desc: string; // Description used in IDE
                                Name: string = ''; // Registry Key
                                Item: TComponent = nil; // Action component
                                Key1: TShortCut = 0; // Short-cut key
                                Key2: TShortCut) = 0; // Second key (if any)

 If Name is blank, the value of ('Rave@' + Desc) will be used.  Name is the
 value used to identify the shortcut in the Windows registry.  Once Name is
 defined, you should not change it or the link will be broken for existing
 settings.  The action component defined in Item is the component that will
 be "executed" when the short cut is performed.  TAction.OnExecute,
 TMenuItem.OnClick and TControl.OnClick are the only recognized actions.  If
 Item is nil, a title bar will be created within the ShortCut Editor.  Rave
 supports 2 key short cuts (i.e. ^K-H). To define one, define the first key
 in Key1 and the second key in Key2.
 (Item = nil, Param = nil)

peAfterZoomChange - This event is called whenever the zoom factor is
 changed.  You can access the current zoom factor through the
 CurrentDesigner.ZoomFact property.
 (Item = nil, Param = nil)

peAfterZoomToolChange - This event is called when the Zoom Tool state of the
 current designer is changed.  You can access the current state of the zoom
 tool with the Zooming boolean variable (defined in RVDefine.pas).
 (Item = nil, Param = nil)

peAfterSelectionChange - This event is called whenever the selection of
 components is changed.  You can access information about the currently
 selected components through the CurrentDesigner.Selections and Selection
 properties.
 (Item = nil, Param = nil)

peLoadState - This event is called when the current state of the designer is
 being loaded from the registry.
 (Item = nil, Param = nil)

peSaveState - This event is called when the current state of the designer is
 being saved to the registry.
 (Item = nil, Param = nil)

peAfterPropertiesModified - This event is called whenever a property is
 changed in the Property Panel.
 (Item = nil, Param = nil)

peInitialize - This event is called when the IDE is being initialized.  Use
 this event instead of FormCreate or a similar event.
```

(Item = nil, Param = nil)

peSelectComponent

peShowPage - This event is called whenever a new page needs to be shown in the Page Designer. The Item can contain either the page to be shown or a component (in which case, that's component's page will be shown).
(Item = Page or Component, Param = nil)

peBeforeReportActivate -

peAfterReportActivate -

pePrepareViewChange

peAfterViewChange

peShowControlPopup

The following steps will show how to create an event-based project editor.

1: Create a class descending from TRaveProjectEditor and override the HandleEvent function.

```
type
  TMyProjectEditor = class(TRaveProjectEditor)
  public
    function HandleEvent(ProjectEvent: TProjectEvent;
                         Item: TRaveComponent;
                         Param: pointer): boolean; override;
  end; { TMyProjectEditor }
```

2: Create a method for HandleEvent and check the ProjectEvent parameter for the specific project event(s) that you want to handle.

```
function TMyProjectEditor.HandleEvent(ProjectEvent: TProjectEvent;
                                      Item: TRaveComponent;
                                      Param: pointer): boolean;

var
  Report: TRaveReport;

begin { HandleEvent }
  Result := true;
  Case ProjectEvent of
    peAfterNewReport: begin
      Report := Item as TRaveReport;
    { Insert code to get a description and assign to Report.Description }
    end;
  end; { case }
end;  { HandleEvent }
```

3: Register the project editor by calling RegisterRaveProjectEditor from within a global scope procedure named RaveRegister (exact case required). The EditorClass parameter should be the class type of the project editor you are registering.

```
procedure RaveRegister;

begin { RaveRegister }
  RegisterRaveProjectEditor(TMyProjectEditor);
end;  { RaveRegister }
```

4: Include the unit containing the project editor in a Delphi package and register that with RAVE through Edit|Preferences|Components.

8.3.1 - New Rave 4 Event System

The following steps will show how to create an event-based project editor.

1: Create a class descending from TRaveProjectEditor and override the Handles function. This function defines which events this project editor defines methods for. If there are more than one event type that are defined, separate with semicolons (e.g. 'DataEvents;ToolMenuEvents').

```
TMyProjectEditor = class(TRaveProjectEditor)
public
  function Handles: string; override;
end; { TMyProjectEditor }

function TMyProjectEditor.Handles: string;
begin
  Result := 'DataEvents';
end;
```

2: Define the event methods that you want to handle. Use the same name and structure as the methods defined in the event handler interface structure. It is not necessary to define all methods of the interface.

Data event handler interface structure in RVData.pas (this is already created, no need to define it yourself):

```
IRaveDataEventHandler = interface
  ['{E89848DC-28E3-4BBB-A8C8-ADC36904EDA4}']
  procedure DataConOpen(DataCon: TRaveDataConnection);
  procedure DataConClose(DataCon: TRaveDataConnection);
  procedure DataConConnect(DataCon: TRaveDataConnection);
  procedure DataSystemOpen(DataSystem: TRaveDataSystem);
  procedure DataSystemClose(DataSystem: TRaveDataSystem);
  procedure DataSystemCallEvent(DataSystem: TRaveDataSystem;
                                EventData: TDataSystemEventData);
  procedure DataSystemEventCalled(DataSystem: TRaveDataSystem;
                                  EventData: TDataSystemEventData);
end; { IRaveDataEventHandler }
```

New project editor source:

```
TMyProjectEditor = class(TRaveProjectEditor)
public
  function Handles: string; override;
  procedure DataConOpen(DataCon: TRaveDataConnection);
end; { TMyProjectEditor }

procedure TMyProjectEditor.DataConOpen(DataCon: TRaveDataConnection);
begin
// Do whatever you want to do when a data connection is opened
end;
```

3: Register the project editor by calling RegisterRaveProjectEditor from within a global scope procedure named RaveRegister (exact case required). The EditorClass parameter should be the class type of the project editor you are registering.

```
procedure RaveRegister;

begin { RaveRegister }
  RegisterRaveProjectEditor(TMyProjectEditor);
```

```
     end;  { RaveRegister }
```

8.3.2 - Creating and calling an event handler

In 8.3.1 we showed how to create a project editor to handle existing event handler system. This section will show how to create and call your own event handler systems.

1: Define an event handler interface structure. This is the methods or events that a project editor can define. The GUID that is on the second line can be generated by pressing Shift-Ctrl-G within the Delphi IDE.

```
     IMyEventHandler = interface
       ['{36E1A0F0-F7EE-487E-AE66-F291400A1052}']
       procedure MyProcA(ID: integer);
       procedure MyProcB(Value: string);
     end; { IMyEventHandler }
```

2: Define an event handler class.

```
   // Event methods type
     TMyEventMethod = (meMyProcA,meMyProcB);

   // Event handler class
     TMyEventHandler = class(TRaveEventHandler, IMyEventHandler)
     protected
     // Parameter fields
       FID: integer;
       FValue: string;
       FMyEventMethod: TMyEventMethod;
     // Overrides
       procedure Process; override;
       function Handles: string; override;
     // Interface property
       property ProjectEditor: TRaveProjectEditor read FProjectEditor implements
        IMyEventHandler;
     public
     // Shell Interface methods
       procedure MyProcA(ID: integer);
       procedure MyProcB(Value: string);
     // Broadcast methods
       procedure DoMyProcA(ID: integer);
       procedure DoMyProcB(Value: string);
     end; { TMyEventHandler }

   // Overrides
     procedure TMyEventHandler.Process;

     var
       IItem: IMyEventHandler;

     begin { Process }
       IItem := self;
       Case FMyEventMethod of
         meMyProcA: IItem.MyProcA(FID);
         meMyProcB: IItem.MyProcB(FValue);
       end; { case }
     end;  { Process }

     function TMyEventHandler.Handles: string;

     begin { Handles }
       Result := 'MyEvents';
     end;  { Handles }
   // Shell Interface methods
     procedure TMyEventHandler.MyProcA(ID: integer); begin end;
     procedure TMyEventHandler.MyProcB(Value: string); begin end;

   // Broadcast methods
     procedure TMyEventHandler.DoMyProcA(ID: integer);

     begin { DoMyProcA }
       FID := ID; // Assign parameter(s) to field variables
       FMyEventMethod := meMyProcA; // Determine which method to execute
       Broadcast; // Broadcast event to all project editors
     end;  { DoMyProcA }

     procedure TMyEventHandler.DoMyProcB(Value: string);

     begin { DoMyProcB }
       FValue := Value; // Assign parameter(s) to field variables
       FMyEventMethod := meMyProcB; // Determine which method to execute
       Broadcast; // Broadcast event to all project editors
     end;  { DoMyProcB }
```

3: Define a variable of the same type as your event handler.

```
     var
       MyEventHander: TMyEventHandler;
```

4: Register the event handler by calling RegisterRaveEventHandler from within a global scope procedure named RaveRegister (exact case required). The EventHandlerClass parameter should be the class type of the project editor you are registering. The function will return an instance of your event handler class that you should store.

```
     procedure RaveRegister;

     begin { RaveRegister }
       MyEventHandler := RegisterRaveProjectEditor(TMyEventHandler);
     end;  { RaveRegister }
```

5: To execute an event for all project editors simply call the broadcast method for the event handler such as:

```
     MyEventHandler.DoMyProcB('This is a test');
```

This would call the MyProcB method for all project editors that handle the MyEvents event type and define a method called MyProcB.

6: To execute an event for a single project editor instance (e.g. AProjectEditor) assign the project editor using the SetProjectEditor method, define an interface variable and assign your event handler to it then call the interface method (not the DoXxxxx broadcast method) of the interface variable (not the event handler).

```
   var
     IEventHandler: IMyEventHandler;

     MyEventHandler.SetProjectEditor(AProjectEditor);
     IEventHandler := MyEventHandler;
     IEventHandler.MyProcB('This is a test');
```

```
------------------------------------------------------------------------------
```

1: Create a directory under C:\Rave4 that is the same name as the component package that you will be creating (for this example: C:\Rave4\ND_AbtBx).

2: In C:\Rave4\ND_AbtBx create a form (.PAS and .DFM) called NDCsAbt and place the following code in there (place whatever controls on the form that you want for the about box):

```
    unit NDCsAbt;

    interface

    uses
      Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
      StdCtrls, RVClass, RVTool;

    type
      TNDAboutBoxForm = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        Button1: TButton;
      private
        { Private declarations }
      public
        { Public declarations }
      end;

      TNDAboutBoxProjectEditor = class(TRaveProjectEditor)
      public
        function HandleEvent(ProjectEvent: TProjectEvent;
                             Item: TRaveComponent;
                             Param: pointer): boolean; override;
      end; { TNDAboutBoxProjectEditor }

      procedure RaveRegister;

    var
      NDAboutBoxForm: TNDAboutBoxForm;

    implementation

    {$R *.DFM}

      procedure RaveRegister;

      begin { RaveRegister }
        RegisterRaveProjectEditor(TNDAboutBoxProjectEditor);
      end; { RaveRegister }

    (******************************************************************************}
    ( class TNDAboutBoxProjectEditor
    (******************************************************************************)

      function TNDAboutBoxProjectEditor.HandleEvent(ProjectEvent: TProjectEvent;
                                                    Item: TRaveComponent;
                                                    Param: pointer): boolean;

      begin { HandleEvent }
        Result := true;
        Case ProjectEvent of
          peShowAboutDialog: begin
            With TNDAboutBoxForm.Create(Application) do try
              ShowModal;
              Result := false; // Signal not to show original about box
            finally
              Free;
            end; { with }
          end;
        end; { case }
      end;  { HandleEvent }

    end.
```

3: Also in C:\Rave4\ND_AbtBx create a package unit called ND_AbtBx.dpk and place the following code in there:

```
    package ND_AbtBx;

    {$DESCRIPTION 'About Box Replacement Property Editor (ver 1.0)'}
    {$IMPLICITBUILD OFF}

    requires
      VCL40,
      RVCL30;

    contains
      NDCsAbt;

    end.
```

4: Create a batch file in C:\Rave4 called RANTCOMP.BAT and place the following commands in there:

```
    @echo off
    if exist setenv.bat call setenv.bat
    ..\source\computil SetupD4
    if exist setenv.bat call setenv.bat
    if "%1%"=="" goto showhelp
    goto docomp
    :showhelp
    echo ***************************************************
    echo Compile RANT Package (for the Rave Visual Designer)
    echo ***************************************************
    echo :
    echo Usage: RANTCOMP packagename
    echo :
    echo Note: The packagename parameter should not include the .dpk extension
    echo :
    echo Desc: This batch file will compile a package containing a RAVE add-on
    echo       and place the .BPL in the parent directory (normally C:\Rave4).
    echo       It is meant to be run from a directory under Rave4 containing
    echo       the source for the add-on.  The Rave library files must reside
    echo       in C:\Rave4\RV.
    echo ***************************************************
    goto endok
    :docomp
    %NDD4%\bin\dcc32 %1.dpk /b /h /w /z /LE.. /U..\RV -$d-l-n+p+r-s-t-w- /DDESIGNER
%2 %3 %4
    if errorlevel 1 goto enderror
    goto endok
    :enderror
    echo Error!
    :endok
```

5: Change to the C:\Rave4\ND_AbtBx directory and run "..\RANTCOMP ND_AbtBx"

6: Start Rave and install the new package file. The BPL will be located in C:\Rave4.

7: If the RANT package defines any components, place the .PAS and any other relavent files in the library C:\Rave4\D4 (or C4 for C++Builder 4.0, D5 for

Delphi 5.0, ...)