Using Rescan

This article explains what the Rescan command does, when to use it, and how it works. Along the way it describes many of the special source code notations that the AppExpert and ClassExpert commands add to files. The information should help programmers understand how markers work, how to create markers, and how to recover from Rescan failures.

Overview

The code-generating Experts are comprised of three major components: AppExpert, ClassExpert and Rescan. This overview describes what each component does. We begin with two important terms:

APX database	Two databases exist for the Experts. The OWLDB.APX file, located in the EXPERT\OWL directory, is a static (read-only) database which contains all information about the OWL base classes (constructors, virtual functions, events, class inheritance, etc.). This database was created by a scanner program which processes each OWL .H file and collects the relevant information. The other database file is the user's .APX file, located in the base directory of the project that AppExpert created. This is a dynamic (read/write) database containing information about each class created by AppExpert or ClassExpert. The content of this database is displayed in ClassExpert's class and event panes. The purpose of the Rescan command is to rebuild the user's database.
snippet	An application is generated from code templates, or snippets, which describe the layout of a generic application. A snippet combines actual code as it would appear if written by a C++ programmer with special controlling lines written in EIL (Expert Interpreter Language). EIL controls how, if, and what code is generated. It is processed by a small interpreter designed specifically for code generation (querying and writing code to various files). EIL lines may appear in many different types of files including source code (.CPP), resource definitions (.RC), rich text format (.RTF), and others. Any file that contains EIL elements is a snippet. All the snippet files are kept in the EXPERT\OWL directory.

Two of three Expert components (AppExpert, ClassExpert, and Rescan) generate code. AppExpert generates the initial code and creates all the files needed to build a default application based on the user's selected options. ClassExpert refines the generated application by creating new classes, overriding virtual functions, handling events, and adding code. Rescan reads the code generated by the Experts in order to rebuild the dynamic .APX database. To generate specific code from snippets, the Experts must gather information about each class either from options stored in the .IDE project file or from the user's .APX database file.

What AppExpert Does

The AppExpert phase of development accepts input from the user and saves it in the .IDE project file. (The user's input determines use of MDI, printing support, help file support, styles, etc.). The classes which will be generated (the base class, the class name, source file and header file) are added to the user's .APX database file. The code generator then queries the user's database for all classes, creates each associated file (.CPP and .H), and adds nodes to the .IDE project file. All possible OWL base classes have an associated snippet file (.OWL) which is then loaded and processed. The processing of the snippet file produces the output files (.CPP, .H, .RC, .BMP, .RH, etc.).

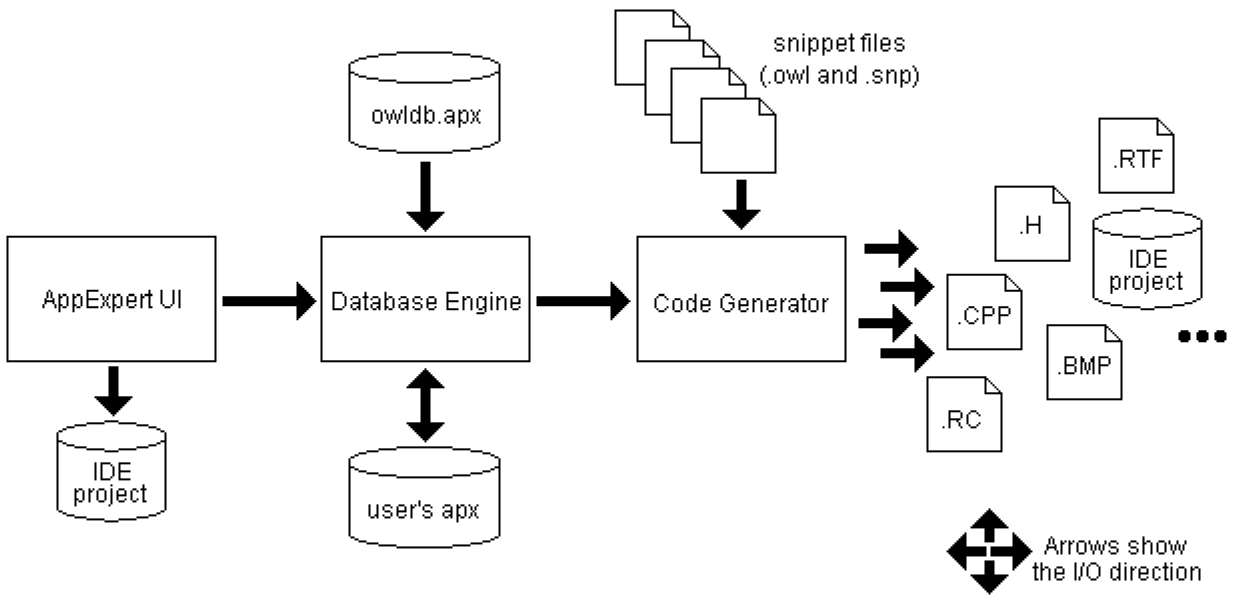

Figure 1. AppExpert

What ClassExpert Does

ClassExpert performs incremental changes. It displays the contents of the user's .APX database in the class and event pane showing such information as classes, events, control notifications, virtual functions, instance data, and document/view pairs.

To create a new class, ClassExpert begins by reading the OWLDB.APX file to display a list of OWL base classes from which to derive the new class. After the user chooses options for the new project, ClassExpert invokes the code generator whose job it is to create the class (.CPP and .H files) and add new nodes to the project file.

Adding event handlers or virtual functions again invokes the code generator which searches for specific code or markers within the generated application to position the new code segment. Similarly when Resource Workshop is used to add, delete, or modify command notifications (menus), to associate a dialog resource with a TDialog-derived class, or to display all controls within a dialog as control notifications (events), the code generator looks for particular locations within the source code to situate the new segments.

Resource Workshop supplies other information, as well. It supplies the resource ID necessary to associate a TDialog object with a DIALOG resource. It provides resource types for handling instance data, and it knows about any .VBX control the dialog contains.

The difference between AppExpert and ClassExpert is apparent in their different uses of EIL commands. The EIL commands determine whether code should be added, how it should be added, and where it should go. The snippets that AppExpert works from contain more lines of straight C++ code than of EIL commands. The snippets that ClassExpert uses contain much more EIL than C++. AppExpert needs a little EIL logic to generate a large framework, but ClassExpert needs a lot of EIL logic to shape and position small blocks of code.

The final files of generated code do not contain any EIL statements. They do, however, contain special comments called markers that flag particular locations within a file.

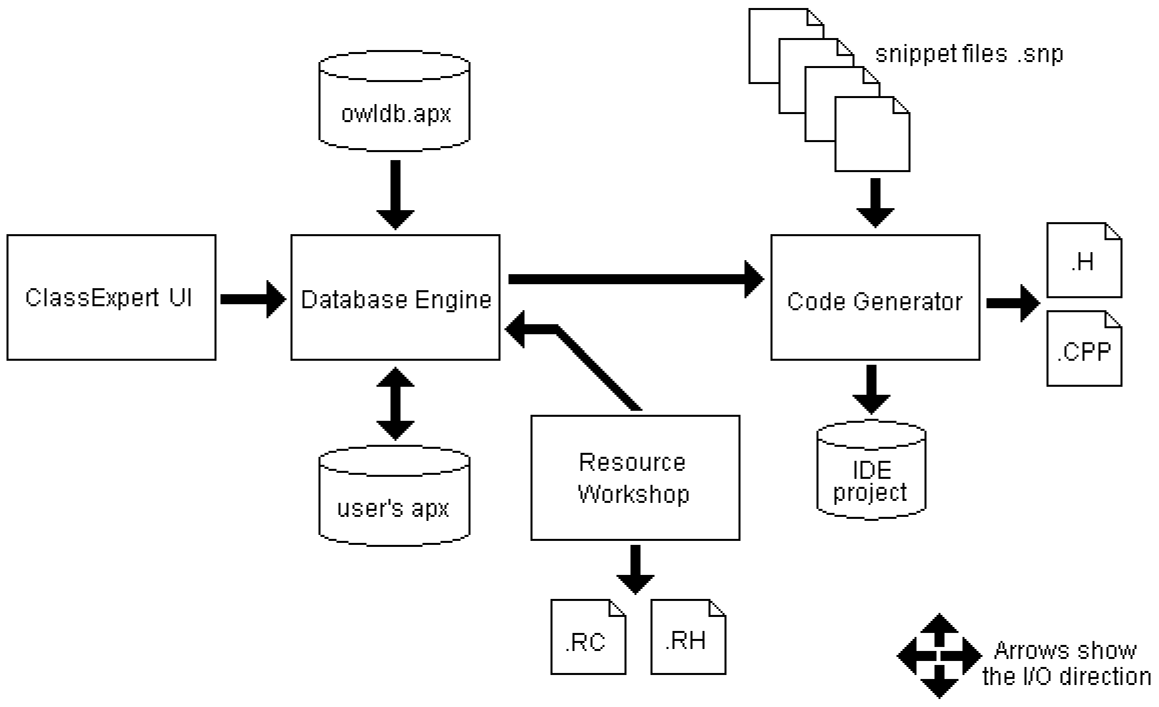

Figure 2. ClassExpert

What Rescan Does

Most generated applications probably make use of both AppExpert and ClassExpert, but Rescan comes into play only when something in the generated application changes unexpectedly. Rescan rebuilds the user's dynamic .APX database file by reading from the source files. Regeneration of the database is necessary for any of these tasks:

- Importing classes from other AppExpert projects
- Changing the name of an existing class
- Deleting an existing class

- Resynching the database to the code after editing outside the ClassExpert

- Fixing an inconsistent or corrupted database

Rescan begins by backing up the original .APX, if it still exists, in a file with the extension .~AP. Then it scans the IDE project nodes. Each file associated with a node in the project is scanned to determine if it contains any class information that belongs in the database. If so, the file and the classes it contains are recorded in a list. This list is then used to create the initial .APX file of class name, base class, and file associations. The first phase of Rescan is complete. (See Figure 3.)

Figure 3. Rescan Phase 1

The second phase of Rescan invokes Resource Workshop to compile the .RC file and create the command notifications for menu items along with the list of dialog boxes and their associated controls. (See Figure 4.)

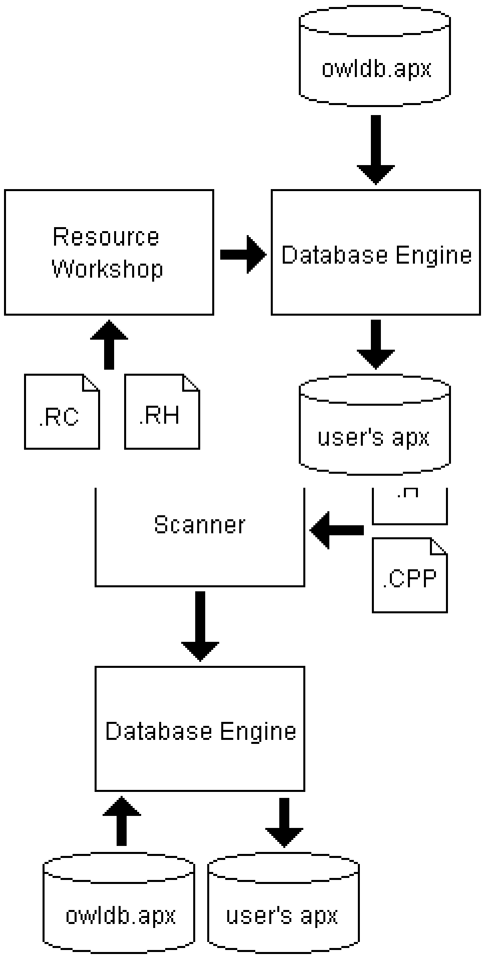

Figure 4. Rescan Phase 2

The last phase of Rescan reads the source code to record event handling, virtual function handling, TDialog to DIALOG resource association, instance data, document/view pairs, and control-notification handling. This is the phase where the bulk of information is transferred to the database and where the majority of problems might arise. (See Figure 5.)

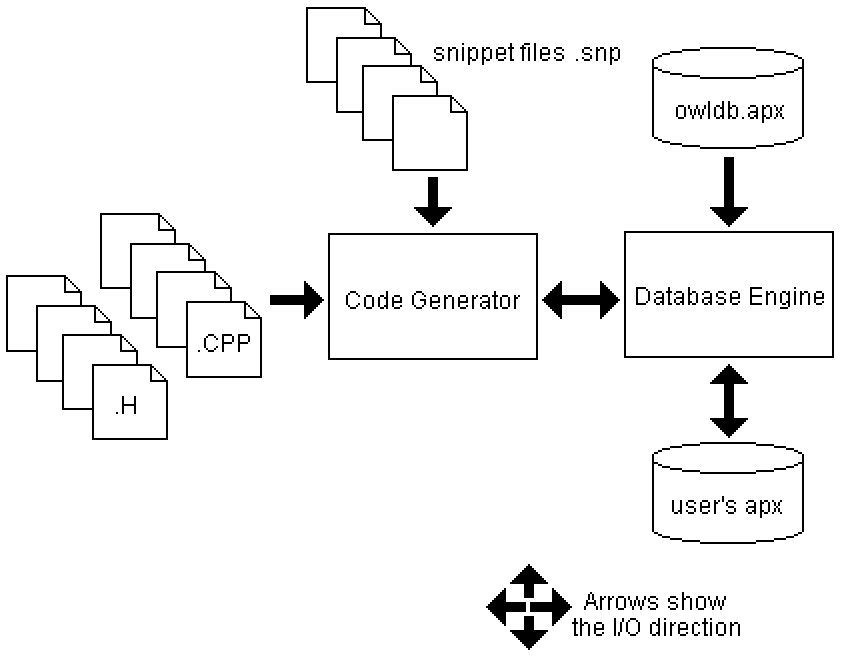

Figure 5. Rescan Phase 3

Rescan is fairly resilient to anomalies and changes in the code. If anomalies occur then Rescan should still complete successfully, but it may generate warnings and some information may not appear in the database or in ClassExpert. If the programmer has modified markers in the generated code, then Rescan may not recognize when a function is overridden or an event is handled.

Understanding Markers

Markers are special comments inserted in the source code to help the Experts find places in the files. Markers usually set off blocks of code, such as virtual functions, event handlers, instance data, and document/view pair associations, that might need to be modified. Markers have the following format:

```
//{{NNN}}
```

w**5**

here NNN is the actual marker. The two code listings that follow in sections a) and b) are keyed to explanations of the markers they contain. The first shows markers that commonly appear in header files. The second shows markers that appear in .CPP files.

To accommodate a limited page width, the listings in this technote are sometimes forced to split single lines of code across two lines of text. Split lines interfere with Rescan. The EIL syntax does not support them, and even where C syntax does, Rescan may not read split lines correctly. Be aware that the code as written does not work; you must rejoin all split lines.

a.) .H File

```
  //{{BaseClassName = ClassName}}        [1]
```

```
struct ClassNameXfer                    [2]

//{{ClassNameXFER_DATA}}                [3]

   char InstanceOne[ 32 ];

   BOOLEAN InstanceTwo;

//{{ClassNameXFER_DATA_END}}            [4]

};


class ClassName : public BaseClassName {

public:

   ClassName (TWindow *parent, \

       TResId resId = IDD_ABOUT, TModule *module = 0);
[5]


//{{ClassNameVIRTUAL_BEGIN}}            [6]

public:                                 [7]

    virtual void InitMainWindow();

    virtual void InitInstance();

//{{ClassNameVIRTUAL_END}}              [8]


//{{ClassNameRSP_TBL_BEGIN}}            [9]

protected:       [10]

   void EvNewView (TView& view);

   void EvCloseView (TView& view);

   void CmHelpAbout ();

   void EvDropFiles (TDropInfo drop);

   void EvWinIniChange (char far* section);

//{{ClassNameRSP_TBL_END}}              [11]
```

```
DECLARE_RESPONSE_TABLE(ClassName);


//{{ClassNameXFER_DEF}}                [12]

protected:       [13]

    TEdit *InstanceOne;

    TCheckBox *InstanceTwo;


//{{ClassNameXFER_DEF_END}}            [14]

};    //{{ClassName}}                  [15]
```

[1]

A class is defined beyond this marker. The marker contains both the base class name and class name. If the instance data transfer structure doesn't exist, it is inserted immediately below this marker. This marker is used during phase one of Rescan to find all classes defined in a .H file.

[2]

For classes derived from TDialog, a transfer buffer is created when instance variables are created. The sequence of fields in the structure initially matches the order in which the instance variables are created in the class's constructor (see [28] - [30] in .CPP file marker definitions.) No data is added to the structure for controls such as TButton that have no data to transfer.

[3], [4]

The fields between these markers contain the actual transferred data. The block contains only data that can be transferred (from objects such as TEdit or TCheckBox.) If an instance variable is deleted. the entry in this block is removed. Any new item added to the list is inserted at the end.

[5]

To associate a TDialog-derived class with a DIALOG resource, Rescan reads the constructor to find the default value for the resId parameter.

[6], [7]

Any overriden virtual functions known by OWL are placed between these markers. Only OWL virtual functions should appear here. Function definitions within the block are limited to one line and may not contain carriage returns.

[8]

The "public:" line should not be removed from the block. If the [6] and [8] markers exist, then Rescan assumes the "public:" line exists. If there are no virtual functions between [6] and [8] then the entire marker block, including [6], [7], and [8], can be deleted as a group.

[9], [11]

The response table markers enclose the list of functions associated with events. This list is maintained in conjunction with the actual response table (with its message crackers) in the .CPP file (see [25] - [26] in the .CPP file marker definitions). If an application contains no event handlers in its response table, then the entire marker block--[9], [10], and [11]--can be removed as a group. The DECLARE_RESPONSE_TABLE(ClassName) macro must also be removed from the header file along with the response table entry in the .CPP file. (See [25] and [26] in the .CPP file marker descriptions.)

[10]

The protected line should not be removed from the block. It is assumed to exist if the [9] and [11] markers exist. If there are no handlers between [9] and [11] then the entire marker block, including [9], [10], and [11], can be deleted as a group.

[12], [14]

The instance variable markers enclose the variable names associated with each instance of a control (used only for TDialog derived classes). This list of data members matches exactly the instantiation of each control class within the constructor of the TDialog derived class (see notes [28] - [30] in the .CPP file annotations.)

[13]

The "protected:" line should not be removed form the block. It is assumed to exist if the [12] and [14] markers exist. If no instance variables appear between [12] and [14] then the entire marker block, including [12], [13], and [14], can be removed as a group.

[15]

This marks the end of a class. The ClassExpert looks for this marker when it has to insert markers for a new block of virtual functions, response table event handlers, or instance variable markers.

b.) .CPP file

```
//{{ClassName Implementation}}         [20]


//{{DOC_VIEW}} [21]

DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, TEditView, \
   DocType1);

//{{DOC_VIEW_END}}                     [22]


//{{DOC_MANAGER}} [23]

DocType1 __dvt1("All Files (*.*)", "*.*", 0, "TXT",

   dtAutoDelete | dtUpdateDir);

//{{DOC_MANAGER_END}}                  [24]
```

```
DEFINE_RESPONSE_TABLE1(ClassName, BaseClassName)
//{{ClassNameRSP_TBL_BEGIN}}           [25]
   EV_OWLVIEW(dnCreate, EvNewView),
   EV_OWLVIEW(dnClose,  EvCloseView),
   EV_COMMAND(CM_HELPABOUT, CmHelpAbout),
   EV_WM_DROPFILES,
   EV_WM_WININICHANGE,
//{{ClassNameRSP_TBL_END}}             [26]
END_RESPONSE_TABLE;


static ClassNameXfer ClassNameData;    [27]


ClassName::ClassName (TWindow *parent, TResId resId,
   TModule *module)
   : TDialog(parent, resId, module)
{
//{{ClassNameXFER_USE}}                [28]
   instanceOne = new TEdit(this, IDC_EDIT1, 32);
   instanceTwo = new TCheckBox(this, IDC_CHECK1);


   SetTransferBuffer(&ClassNameData);  [29]
//{{ClassNameXFER_USE_END}}            [30]
```

[20]

All class-specific member functions, along with an OWL response table, are implemented below this marker. During phase one of Rescan this marker is used to associate a source file with a header file class definition. Each header can be associated with only one source file.

[21], [22], [23], [24]

Rescan parses the values between these markers to determine how documents are paired with views.

The markers appear only in applications that use the document/view paradigm. Markers [21] and [22] define the association of a TDocument-derived class with a TView-derived class. Markers [23] and [24] define the attributues of this pairing: its styles, its file type, and the file-type string in the file-open dialog box. All the information between [21] and [24] is handled as a block each time a document/view pair is created or deleted.

[25], [26]

This is the actual OWL response table which dispatches event notifications to the particular functions that handle them. This table is maintained in parallel with the response table marker block in the .H file (see [9] - [11]). If there are no response table entries then the entire block may be removed, beginning with the line preceding marker [25] (DEFINE_RESPONSE_TABLE...) and ending with the line following marker [26] (END_RESPONSE_TABLE).

[27]

For TDialog-derived classes with instance variables, the transfer buffer is created statically so that values may be put in the buffer before the dialog box is created and taken from it after the dialog box is destroyed.

[28], [30]

The instance variables are placed between markers [28] and [30]. They must correspond to the entries between [12] and [14] in the .H file. Also, the fields in the transfer data structure defined between markers [2] and [4] in the .H file must match the order of these instance variables exactly.

[29]

This line associates the transfer buffer with a TDialog-derived class. The blank line preceding the SetTransferBuffer command must not be removed.

EXPERT.INI File

Some of the Expert configuration options can be set from an initialization file called EXPERT.INI. We recommend that you create the .INI fil**10**

e in the \WINDOWS directory because it loads 6-7 times faster from there, but the file may also be in the EXPERT\OWL directory. This example shows the file switches. All are optional. Unless otherwise noted, 1 means ON and 0 means OFF.

```
[Expert]

Snippet=EXPERT\OWL\



[Annotation]

Author=John Q. Smith

Company=Borland International, Inc.

Copyright=Copyright © 1993 by [[CompanyName]]. All Rights Reserved.
```

```
Version=1.0

Description=Expert OWL Application


[Directory]

; If directories are empty then Expert uses normal mechanism

SourceExt=.CPP

HeaderExt=.H


[Application]

; Startup Values:  1 = Maximized, 2 = Minimized, 3 = Normal

Startup=3

; Control Values:  0 = 3D,  1 = 2D (normal Windows control),

; 2 = BWCC

Control=1

; Model Values:  1 = MDI, 2 = SDI

Model=1

; Document/View ON or OFF

DocView=1

; DragDrop ON or OFF

DragDrop=1

; Printing ON or OFF

Printing=1

; Speedbar ON or OFF

SpeedBar=1

; Status bar ON or OFF

StatusBar=1

; Generate help file ON or OFF

Help=0
```

```
AboutDescription=About Box Text

AboutConstant=IDD_ABOUT

; Comment Values:  1 = Terse, 2 = Verbose

Comment=2


[Classes]

; List of customized classes followed by an entry for each

; class listed.

TApplication=

TWindow=


; Override the TApplication defaults.

[TApplication]

Title=Main Application

Icon=

; Background colors are RGB values of the form R G B

; (e.g., 255 255 255)

Background=

Style=

Client=TEditFile

ClassPrefix=TAppl


; Override the TWindow defaults.

[TWindow]

Title=Window

Icon=

Background=
Style=

Client=TWindow

ClassPrefix=TWin


[DocView]

Description=All Files (*.*)

Filter=*.*

Ext=TXT

Style=1140850688

Dir=


[CGenOptions]

; Generate .MAK file

MAK=1

; Dump info on rescan or code generation error to the file

; AXCGEN.ERR.

Dump=1

; Exception handling?

Exception=FALSE
```

AXCGEN.ERR File

Setting `Dump=1` in the [CGenOptions] section of the EXPERT.INI file makes Rescan produce a log file called AXCGEN.ERR. Even if no errors occur, the log file records what occurs as the user's database is rebuilt. The two annotated listings that follow show what the log file looks like if Rescan succeeds and if it fails.

Rescan log file without errors:

```
========================================================
  [1]

---------------------------RESCAN-----------------------
```

```
========================================================

ScanVEntry function name=SetupWindow     [2]

Adding Virtuals to database PreviewWindow::SetupWindow



Adding Response Entry to database PreviewWindow::WM_NCLBUTTONDOWN
  [3]

Adding Response Entry to database PreviewWindow::WM_CLOSE



ScanViewEntry     [4]

   Template = DocType1

   Document = TFileDocument

   View = TListView

ScanDocEntry       [5]

   Style =

   Description = All Files (*.*)

   Filter = *.*

   Ext = TXT

Adding DocView Entry to database TFileDocument - TListView
  [6]



ScanRespEntry:  [7]

   MenuType= COMMAND

   Constant= CM_HELPABOUT

   Function= CmHelpAbout

   Event=

Adding Response Entry to database TAppltll3App::CM_HELPABOUT
  [8]
```

```
ScanDialog resource ID=IDD_ABOUT         [9]

DBDialog ClassName=TDLGtll3AboutDlg, DialogResource=IDD_ABOUT
  [10]



=========================================================
  [11]

------------------------END RESCAN-----------------------

=========================================================
```

[1]

Rescan phase 3 is starting.

[2]

Virtual entry SetupWindow was found and added to the user's database.

[3]

Window message handlers were found in the response table and added to the user's database.

[4], [5], [6]

Document/view markers were found and document/view pairs were added to the database.

[7], [8]

A command handler was found in the response table. MenuType can be COMMAND or ENABLER. The command handler was added to the database.

[9], [10]

Rescan found the DIALOG resource ID associated with a TDialog-derived class and added it to the database.

[11]

The rescan operation completed without errors.

The AXCGEN.ERR file will contain many more lines of Rescan information than we have shown here. The error file text may have many more sections depending on the number of virtuals, events, command handlers, instance data variables, and document/view pairs that occur in classes that the ClassExpert recognizes. (It recognizes OWL classes with marker comments.) Each section will have the same general format, but the class, function, and variable names will vary.

Rescan log file with errors:

When Rescan fails the AXCGEN.ERR file will contain the following additional information.

```
========================================================
  [20]

--------------------------RESCAN-----------------------

========================================================


                 .

                 . [21]

                 .


ScanVEntry function name=SetupWindow     [22]

Adding Virtuals to database TDLGtll3AboutDlg::SetupWindow

ScanVEntry function name=MySetupWindow

--------------------------------------------------------------

Mon Jan 17 13:53:33 1994                 [23]


Project Name: tll3.exe                   [24]

Base directory: c:\tll\

Directory for .H: c:\tll\          Directory for .CPP: c:\tll\

Module path: C:\BORLANDC\EXPERT\OWL\


Current template/snipit file: rescan.snp

Current line number: 21

Current offset: 828

Current line information: rescan.snp  (21):  ##:

DBVirtual(ClassName, FunctionName)

Current class name: TDLGtll3AboutDlg

     : public TDialog  [class ID = 6]    [25]
```

```
File type: 2

Project node: 0x60803

AppGen ONLY: 0x0

Error Message: Unable to add to dynamic DB, invalid ##| directive at line
  21 (rescan.snp)


HandlerREC: NULL


File I/O

========

Output file name: c:\tll\tdlgt3ad.h

     NOTE: Check BufferID_t's below for validity.
  [26]

Virtual file buffer open BufferID_t: 0xffffffff

Edit buffer open BufferID_t: 0x4

FEditor buffer open BufferID_t: 0xffffffff


Control Stack:

==============

[1] Class Start         TRUE

[2] --BEGIN--           TRUE

[3] Lines to run 0      TRUE

[4] Lines to run 0      TRUE

[5] --BEGIN--           TRUE

[6] --BEGIN--           TRUE

[7] --BEGIN--           TRUE

[8] --FOR--             TRUE

[9] Lines to run 0      TRUE

[10] Lines to run 1      TRUE
```

```
Symbol Stack:

=============

ClassName = TDLGtll3AboutDlg        [CStack Mark = 1]
  [27]

StartLine = 33        [CStack Mark = 2]  [28]

EndLine = 35        [CStack Mark = 3]

VEntries = 2        [CStack Mark = 7]

FunctionName = MySetupWindow        [CStack Mark = 8]
  [29]



Nested includes: NONE




<EOF>
```

[20], [21]

The first part of the file does not differ from the log for a successful Rescan. Notice that because a failure occurred the End Rescan message is not at the end of the .ERR file.

[22]

The last successful Rescan operation completed was finding SetupWindow and adding it to the database. MySetupWindow was being processed as a virtual function. (The likely cause of this particular error is that user has manually pasted a non-OWL virtual function inside the virtual markers).

[23]

Here the code generator begins to dump its internal state. The dump begins with a date-and-time stamp showing when the error occurred. This is important because if the .ERR file already exists, the new output is appended to it.

[24]

The project is being rescanned.

[25]

This line shows the class currently being scanned (class name and base class name)

[26]

This line shows the file currently being scanned (`c:\tll\tdlgt3ad.h`).

[27]

As a double-check, the class being scanned. This makes sure we have the correct class.

[28]

This is the line number where the error occurred. It refers to a position within the file named at [26].

[29]

Name of function begin scanned when the error occurred.