

Editing Rules
DV-Draw User's Guide (new sections)

[Introduction](#)

[Creating Prototypes](#)

[Creating Your Prototype in DVDraw](#)

[Selecting Objects for Rules](#)

[Guidelines](#)

[Managing Rules](#)

[Processing Rules](#)

[Creating Rules](#)

[Guidelines](#)

[Reordering Rules](#)

[Rule Components](#)

[Describing the Event](#)

[Guidelines](#)

[Describing the Condition](#)

[Guidelines](#)

[Describing the Resulting Action](#)

[Guidelines](#)

[Summary of Rule Options](#)

[Rules In Subdrawings](#)

[Additional Guidelines for Creating Prototypes](#)

[Running a Prototype](#)

[Setting the Prototype Environment](#)

[Mandatory DVproto Variables \(Optional for DVDraw\)](#)

[Optional DVproto Variables](#)

[Using Data Sources in a Prototype](#)

[Running a Prototype in DVDraw](#)

[Running a Prototype Outside DVDraw: Using DVproto](#)

[Alphabetical Listing of Configuration File Variables](#)

This document discusses the following topics:

Using Rules

Describing the Event

Describing the Condition

Describing the Resulting Action

Rules in Subdrawings

Guidelines for Using Rules

Setting Up a Prototype

Running Prototypes in DV-Draw and Outside DV-Draw

Related Configuration File Variables

Introduction

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Prototyping lets you build a model of your application interface on which you can base your development. In DV-Draw you can simulate and test your application interface at any time during the design process. This shortens development time by letting you refine your design before programming and implementation, and makes revision easier at all stages of your project life cycle.

When you create a prototype, you describe the interface behavior in terms of **rules** which associate **events** with **actions** or changes in the interface. An event can be a pick or an input object event. An action can be changing to another view, overlaying a view, overlaying an object, popping up an object at a cursor location, or erasing views and overlays.

Running a prototype lets you test the logical connections between your views. You can run a prototype in DV-Draw using the test window, or outside DV-Draw using the *DVproto* script.

Creating Prototypes

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

- The basic steps to creating and using a prototype are: Planning your interface and its behavior.
- Organizing a prototype directory.
- Creating your prototype views by adding rules in DV-Draw.
- Setting the prototype environment.
- Running the prototype.

This document discusses how to create your prototype views by adding rules in DV-Draw, and running the prototype. Planning the interface, organizing the directory, and setting the prototype environment are explained in the *Prototyping* document .

Creating Your Prototype in DV-Draw

The rules associated with your objects describe prototype behavior in terms of the **resulting actions** which occur depending on specified **events** and **conditions**. To define a rule for an object:

- Select the object.
- Pull down the Object Menu and select "Rules."
- In the Edit Object's Rules dialog, click on the Add Rule button.
- Describe the event.
- Describe the condition.
- Describe the resulting action.

You can also attach rules to your view:

- Make sure no object is selected.
- Pull down the Object Menu and select "Rules."
- In the Edit Object's Rules dialog, click on the Add Rule button.
- Describe the event.
- Describe the condition.
- Describe the resulting action.

Selecting Objects for Rules

Rules are attached to objects. Sometimes you want to attach a rule to a single object such as a primitive shape or an input object. However, sometimes you want to attach rules to a complex image. If an image is composed of several objects and you attach rules to only one, the user might not pick the object with the rules. Some suggested solutions are:

- Store the complex image as a subview and create a subdrawing. Then attach rules to the subdrawing object. This lets you pick anywhere on the subdrawing to access the rules without worrying about which component object is selected. Also, to pop up or overlay a subdrawing, you only need one rule instead of a separate rule to display each object in the image.
- Store the complex image as an icon or image object. Then import the icon or image and attach rules to the resulting object.
- Place a single transparent object in front of the complex image and attach rules to the transparent object.
- If one object in the complex image extends to all edges of the image, make that object transparent, place it in front of the other objects, and attach rules to it. This only works if one object covers the whole image area, and you can make it transparent conveniently.
- If the complex image is a button, use a button input object instead.

Guidelines

Note these additional guidelines for attaching rules to objects:

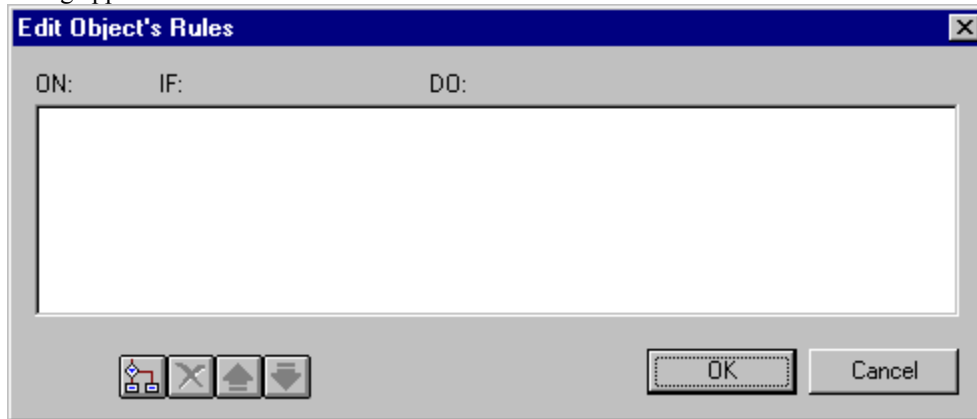
Any object can have rules.

An object can have up to 100 rules.

Managing Rules

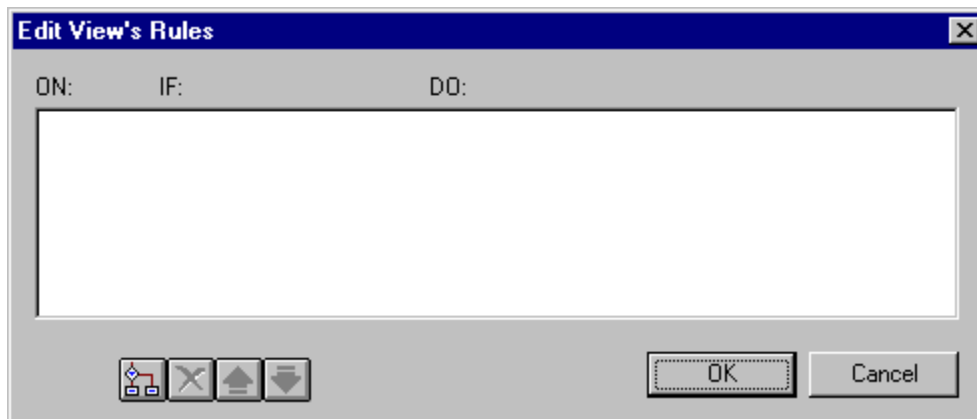
DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Rules can be attached to objects and views, and both can have multiple rules. To add rules to an object, select the object. Pull down the Object Menu or click with the right mouse button. Select "Rules." The Edit Object's Rules dialog appears:



Edit Object's Rules Dialog

To add rules to a view, make sure no object is selected. Pull down the Object Menu or click with the right mouse button. Select "Rules." The Edit View Rules dialog appears:




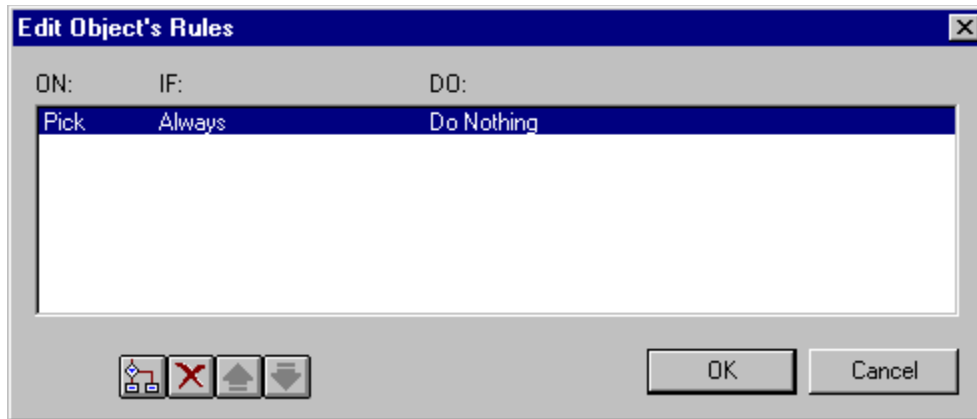
Edit View's Rules Dialog

Processing Rules

Both the Edit Object's Rules dialog and the Edit Object's Rules dialog let you add and organize rules.

Creating Rules

To create a rule, click on the "Add Rule" button, . A default rule is added:



Each time you click on the "Add Rule" button, an additional default rule is added to the list.



Guidelines

Note the following guidelines for creating rules:

- Any object required by an action must be named.
- For a rule to function, the named views and objects must exist. However, you can create rules that refer to the names of views and objects that do not yet exist. This causes an error such as *Cannot open file name* when you run the prototype, but does not prevent the prototype from running.
- Do not use directory names when specifying subdrawings or views in rules. Use the *DVPATH* variable in the configuration file to add the appropriate directories and subdirectories to the search path.

Reordering Rules

All rules with true conditions are processed in the order in which they appear in the list, so organization is important.

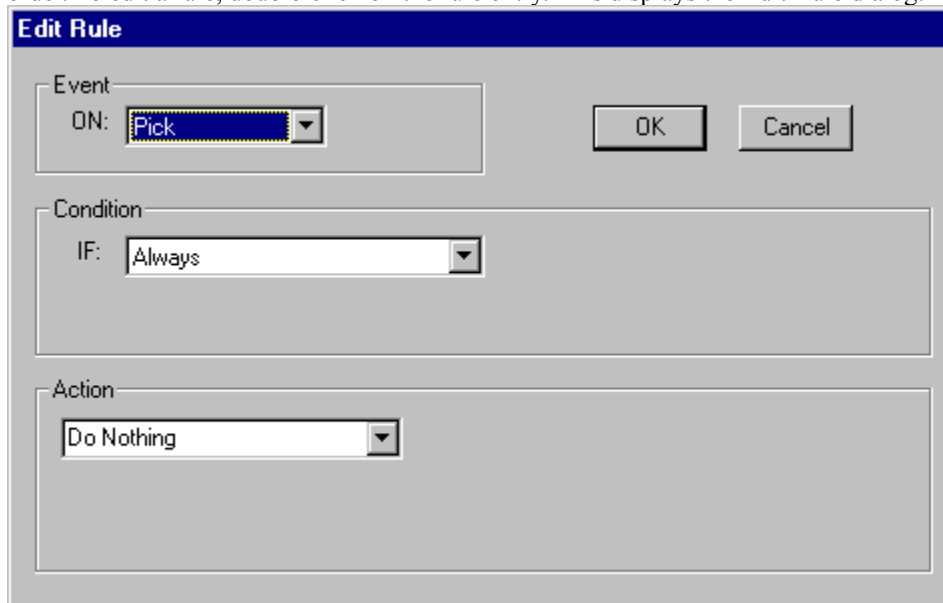
To change the order of existing rules, click on the rule you want to move, and click on the Up or Down arrow buttons,  or , to move the rule up or down in the list.

Rules in subdrawings are processed in ascending hierarchical order. If an object is erased by a rule attached to that object, the remaining rules in its list *are* executed. However, if a rule erases an object that is higher in the hierarchy, no rules attached to the erased object are executed.

Rule Components

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

The components of a rule are its event, condition, and resulting action. You can edit these components in any order. To edit a rule, double-click on the rule entry. This displays the Edit Rule dialog:



The screenshot shows the 'Edit Rule' dialog box. It has a title bar 'Edit Rule' and three sections: 'Event' with 'ON:' and a dropdown menu showing 'Pick'; 'Condition' with 'IF:' and a dropdown menu showing 'Always'; and 'Action' with a dropdown menu showing 'Do Nothing'. There are 'OK' and 'Cancel' buttons in the top right corner.

Describing the Event

An event must occur before the action of a rule can take place. To edit the event selection, pull down the event option menu. Depending on the object type, you can select one of the following events.

- | | |
|-------------------|--|
| Pick | Occurs when you locate the cursor on an object or view and press any key or mouse button. A pick registers first on an object if there is one at the cursor location. If there is no object, the pick registers on an overlay view. If there is no overlay view, the pick registers on the base view. If an object with visibility dynamics is not visible when you pick on it, it does not register the pick event. |
| Done | Occurs when you select a "Done" action of an input object. |
| Cancel | Occurs when you select a "Cancel" action of an input object. |
| Event Used | Occurs when any meaningful event takes place in an input object. Meaningful events include motion that changes the variable value, picks in active areas, done events, and cancel events. |
| Draw | Occurs when the view is drawn, overlaid, or redrawn. |
| Update | Occurs whenever the display is updated. |

Guidelines

Note the following guidelines for using events:

- "Pick" is the only valid event for objects other than views and input objects.
- Input object events are "Pick," "Done," "Cancel," and "Event Used." The default event of an input object is "Done."
- An input object "Pick" event must occur in an active area of the input object. Picks that occur in inactive areas of the input object do not trigger rules of the input object; if the input object is part of a subdrawing, the pick triggers execution of the subdrawing's rules.
- To use "Done" and "Cancel" events, the input objects must define "Done" and "Cancel" areas or keys. An input object updates accordingly before it processes the prototype rule.
- View events are "Pick," "Draw," and "Update." The update event lets actions occur without any user

intervention when data meets specified conditions.

- When an object is selected, events are processed in the following order:
 1. Input object events: "Pick," "Done," "Cancel," and "Event Used."
 2. All true-condition rules, in the order in which they appear in the list.
 3. Update dynamics.

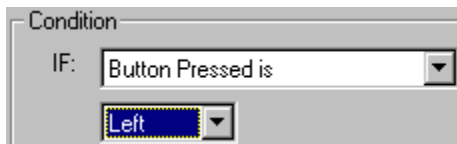
Describing the Condition

The rule's condition determines whether or not the rule's action takes place when the specified event occurs. To edit the rule's condition, pull down the condition option menu. You can select one of the following condition types.

| | |
|--|--|
| Always | Makes the action take place whenever the specified event occurs. |
| Button Pressed | Makes the action take place if the pick was a specified mouse button. The options are "Left," "Middle," and "Right." |
| Key Pressed | Makes the action take place if the pick was one of the specified keyboard keys. For example, the setting <i>Key Pressed is Qq</i> lets you press either the upper or lower case <i>Q</i> . Control and function keys are not allowed. |
| Object's Variable Compares to Value | Makes the action take place if the object's variable has the specified relationship to a value. This condition is only useful with dynamic objects. If a graph or input object has more than one variable, only the first variable is used in the comparison. To compare variables other than the one attached to the object, use one of the "DSVar Compares" condition types. |
| DSVar Compares to Value | Makes the action take place if the specified data source variable has the specified relationship to a value. |
| DSVar Compares to DSVar | Makes the action take place if the specified data source variable has the specified relationship to another data source variable. |

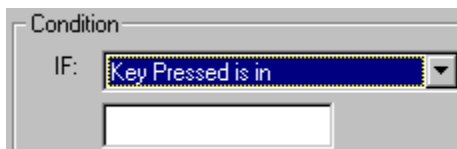
If you select "Always," no further description is required. All other condition types require additional information.

If the condition type is "Button Pressed," the dialog displays additional options:



Pull down the option menu to select the correct mouse button for the action.

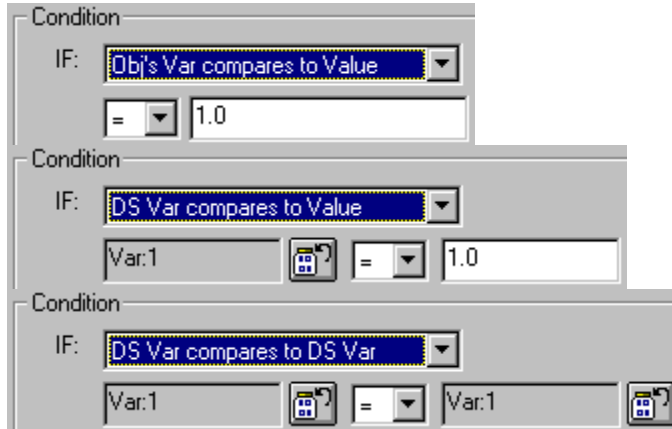
If the condition type is "Key Pressed," the dialog displays additional options:




Enter one or more keyboard keys. Keyboard keys do not need to be separated by other characters. For example, entering *Qq* makes the event happen when you select the object by pressing an upper or lower case *Q*. You can enter multiple keyboard keys, but the condition is true when you press any *one* of them.

If the condition type involves a variable, the dialog displays the variable, an operator, and a comparison value or

variable:



If the variable is the object's variable, you cannot select or change it. To specify a data source variable, click on the data button, , and select a variable from the Data Source List dialog that appears. To specify the relationship, pull down the option menu next to the equal sign, "=", and select the correct operator. To specify a value, enter the numerical value.

Note that a variable can only be tested for one relationship in a single rule. Variables can be numerical or text variables. You cannot test a variable for a value between two other values. To achieve this effect, use the FDSEval function data source.

Guidelines


Note the following guidelines for using conditions:

- Before assigning rules using graph or input object values, check the variables so you know which ones to use in the rules.
- You can use the FDSEval function to set a variable to be used in "DSvar Compares to . . ." condition events. Using FDSEval this way can enhance the power of your prototype.
- The "Button Pressed" condition lets you perform different actions by selecting the same object using different mouse buttons. This is useful when the action choices are limited in number and easy to remember. However, sometimes it's easier to remember which condition controls each action if the condition is a letter associated with the action, such as *Q* for *quit*.
- The "Obj's Var Compares to Value" condition is useful when an object has only one data source variable. However, if an object has more than one data source variable, this condition only recognizes the first one. To compare an object's other data source variables to a value, use the "DS Var Compares to Value" condition. To compare an object's data source variables to each other, use the "DS Var Compares to DS Var" condition.
- If a rule applies to all but one variable value of an input object, you can use a condition such as *Obj's Var != 0* instead of repeating the rule for both variable conditions, such as *Obj's Var = 1* and *Obj's Var = 2*.

Describing the Resulting Action

To describe the resulting action of a rule, pull down the action option menu. You can select one of the following resulting action types. The **base view** is the view currently displayed.

- | | |
|----------------------------|---|
| Go to View | Replaces the base view with the specified view. |
| Go to Previous View | Replaces the base view with the previous view. |
| View | A history of commands is maintained, so you can retrace a progression of base views by using the "Go To Previous" options in successive views. If you will need to go more than |

| | |
|-----------------------------------|---|
| | one level back, use this option instead of "Go to View." |
| Overlay View | Superimposes the specified view over the base view. The objects of the overlaid view appear in front of the objects in the base view. The background color of the base view is not affected. You can overlay any number of views; the only limits are readability and the memory capacity of your system. |
| Overlay Object | Superimposes the specified object over the base view. The object's location matches its placement in the view where the object occurs. You can overlay any number of objects; the only limits are readability and the memory capacity of your system. |
| Popup Object at Cursor | Displays the specified object at the current cursor location. |
| Erase View | Erases the specified overlaid view. The erase method redraws the overlaid view in the background color of the base view. To restore objects in the base view, include a rule with a "Redraw" action after the rule with the "Erase View" action. "Erase View" cannot erase a base view. Base views are erased automatically by the "Go to View" and "Go to Previous" rules. |
| Erase Object | Erases the specified popup or overlaid object. The default erase method is Save Raster. To change this, use the environment variables described in the next section. This erases only objects which have been added to the view as popups or overlays, not objects that are part of the base view or an overlaid view. |
| Erase All Visible Overlays | Erases all overlaid objects and views currently displayed. Objects: The default erase method for objects is Save Raster. To change this, use the environment variables described in the next section. Views: The erase method redraws the overlaid view in the background color of the base view. To restore objects in the base view, use a rule with a "Redraw" action after the rule with the "Erase View" action. |
| Erase All Visible Popups | Erases all popup objects currently displayed. The default erase method is Save Raster. To change this, use the environment variables described in the next section. |
| Set DS Var Value | Sets a data source variable to the specified value. To change the data source variable affected, click on the data button,  , and select a variable from the Data Source List dialog that appears. |
| Increase DS Var Value | Increases the value of a data source variable by a specified amount. |
| Decrease DS Var Value | Decreases the value of a data source variable by a specified amount. |
| Execute Command | Executes the specified command as a system call. Any arguments required by your command must be included in the rule, such as <i>/bin/echo making system command</i> . For example, you can start a process or open a command shell to perform other commands. The prototype continues to run while a command is being executed. |
| Start Dynamics | Starts or resumes reading the data and updating the display. |
| Stop Dynamics | Stops reading the data and updating the display. |
| Increase Update Rate | Increases the run speed by one increment corresponding to the increments of the test window speed scale. |
| Decrease Update Rate | Decreases the run speed by one increment corresponding to the increments of the test window speed scale. |
| Redraw Screen | Redraws the screen. |
| Quit Prototype | Ends the prototype run in either the DV-Draw test window or the <i>DVproto</i> script. In DV-Draw, this command is equivalent to clicking on the "Stop" button. |

Do Nothing

No action takes place. This is the default action selection. Use this as a place holder when you have no action for the rule yet, or to disable a rule for testing purposes.

If you select an action type involving an object, view, or system command, you must complete the description by supplying the appropriate information in the fields that appear in the action area. For example, the "Overlay Object" action provides spaces for the name of the object and the name of the view where the object occurs:



The image shows a graphical user interface form with two input fields. The top field is labeled "View:" and has a small button with three dots to its right. The bottom field is labeled "Obj Name:". Both fields are empty and have a light gray background.

Any object required by an action must be named. There are no naming conventions. When calling an object from another view, the rule uses the first object created with the specified name.

Guidelines

Note the following guidelines for using actions:

- "Quit Prototype" and "Redraw Screen" should be available in every display of every prototype. Make it a habit to add them at the beginning. One way to accomplish this is to assign these actions to visible objects or buttons in each base view. Another way is to assign these rules to a popup menu in each base view. Another way is to attach these rules to each base view itself with mnemonic key presses such as "R" and "Q" as the events.
- Use "Go to View" to go from the current base view to any other view. To provide a choice of destination views, you can give the same rule ("on Pick, Always, Go to View *name*") to different objects. Or you can give different rules to the same object. The rules can differ in their conditions or events. For example: when the object is picked, if the key pressed is *P*, go to the power display; or if the key pressed is *T*, go to the temperature display. Note that you can only use different events if the object is an input object.
- Use "Erase Added View" to erase a specified overlaid view. To erase more than one overlaid view, you can use multiple rules specifying each view in turn, or you can use "Erase Visible Overlays." Note, however, that the "Erase Visible Overlays" option erases both views and objects.
- To pop up the object against the background of the view, give the "Popup Object" rule to the view itself.
- You can overlay any number of views or objects; the only limits are readability and the memory capacity of your system.
- Be careful about overlaying dynamic objects. Dynamics may bleed through other objects. To prevent input object dynamics from bleeding through, you can write a DV-Tools utility to set the `REDRAW_ON_UPDATE` flag of the `VOinPutFlag` routine for the input object. For more information, see the *DV-Tools Reference Manual*.
- Avoid letting objects overlap graphs because graphs bleed through other objects. If an object might overlap a graph, the "Popup Object" action should be preceded by a "Stop Dynamics" action. Use "Start Dynamics" when you erase the overlaid object.
- "Set Value" can be used to set a variable for comparison to other variables, to reset a variable, or to act as a one-item menu. You can change the value using "Increase" and "Decrease."
- "Increase" and "Decrease" can be used to change a memory data source variable for comparison to other variables; to create increment buttons for values that control dynamics such as rotation, movement, or scale; or to implement a counter.

Summary of Rule Options

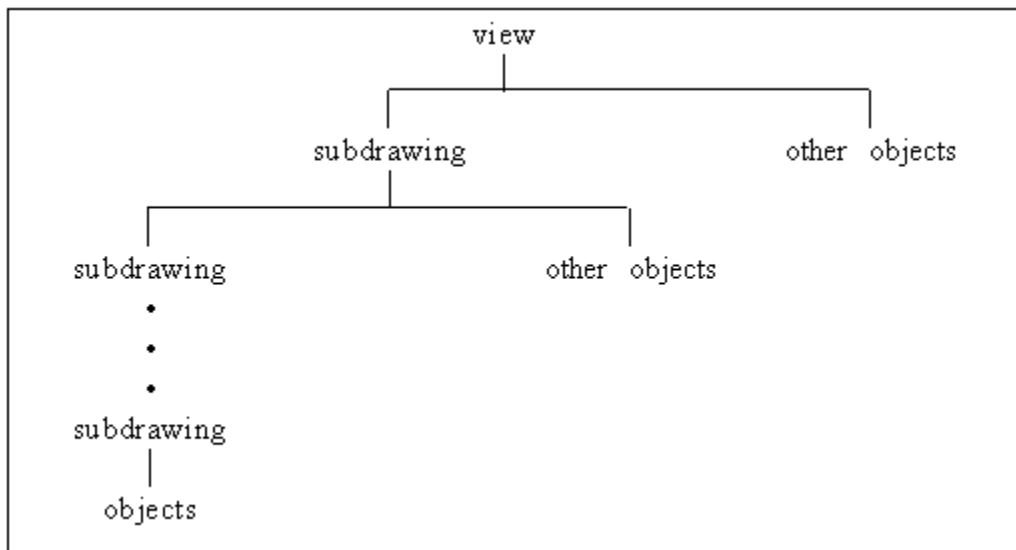
The following table lists the events, conditions, and resulting actions available in DV-Draw.

| Events: | Conditions: | Resulting Actions: |
|----------------|-----------------------------|----------------------------|
| Objects: | Always | Go to View |
| Pick | Button Pressed | Go to Previous View |
| Input Objects: | Key Pressed | Overlay View |
| Pick | Obj's Var Compares to Value | Overlay Object |
| Done | DS Var Compares to Value | Popup Object at Cursor |
| Cancel | DS Var Compares to DS Var | Erase Added View |
| Event Used | | Erase Added Object |
| Views: | | Erase All Visible Overlays |
| Pick | | Erase All Visible Popups |
| Draw | | Set DS Var Value |
| Update | | Increase DS Var Value |
| | | Decrease DS Var Value |
| | | Execute Command |
| | | Start Dynamics |
| | | Stop Dynamics |
| | | Increase Update Rate |
| | | Decrease Update Rate |
| | | Redraw Screen |
| | | Quit Prototype |
| | | Do Nothing |

Rules *In Subdrawings*

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

All rules included in a subdrawing are active when the subdrawing's dynamics are enabled. Therefore, if you assign rules in a view, then use that view as a subdrawing, all of its rules can be active in the final prototype. DV-Draw lets you build deep hierarchies of views and subdrawings. Rules can be attached to objects and subdrawings in views at all levels of the hierarchy. If dynamics are enabled at all levels, all rules are active. Rules are executed starting at the bottom of the hierarchy then proceeding upward. The concept of this hierarchy is illustrated in the following figure:



Representative View Hierarchy

Rules can affect other objects at any level in the hierarchy. However, if a rule erases an object higher in the hierarchy, no rules attached to the erased object are executed. If an object is erased by a rule attached to that object, the remaining rules in its list *are* executed.

Additional Guidelines for Creating Prototypes

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

As you create your prototypes, note the following additional guidelines: A simple form of prototype like a slide show can be created using only "Go to View" to display a series of views in sequence. Using both "Go to View" and "Go to Previous View" lets you go forward and backward in the "slide show."

To display a warning or message when a data value exceeds a critical limit, you can give the view a rule that compares the variable value to a constant or to another variable value on each update. Note that the message appears even if you erase the graph that displays the data. This is because the rule continues to read the data value even if the data is not displayed. This also lets you display a warning or message that depends on data you are not displaying.

To create an overlay view, you can create the overlay objects in the base view, then save them separately as a subview. If you edit the base view at a later time, be sure to check the overlay views and edit them accordingly. You can recombine the views by using "Merge View," edit the overlay objects, and resave the subviews. If a subview has a large number of objects, it helps to give them all the same name before merging the views. This lets you group them "By Name" after editing to resave the subview. Remember that DV-Draw lets you use wild card characters in the name.

Objects cannot be erased from the base view, so to erase an object or set of objects, display them as overlays or popups. Another way to create the effect of erasing an object from the base view is to use visibility dynamics. An object that uses visibility dynamics must be connected to a variable that controls its visibility.

Pull-down menus can be created with the same look and feel by using the same template, even though each has a different number of pickable items. To make a set of similar menus, use a template with the maximum number of items required. To prevent void menu items from showing in the prototype, turn off the Outline option in the Input Object Editor and make the menu background color the same as the drawing background color.

Both horizontal and vertical alignment are important for walking menus. Use the "Grid" option to build the menu template so the height of each item matches the grid. This makes it easy to align the top of each walking menu with the top of its corresponding menu item. You can store all the walking menus in a single view, even if they overlap, to make editing easy.

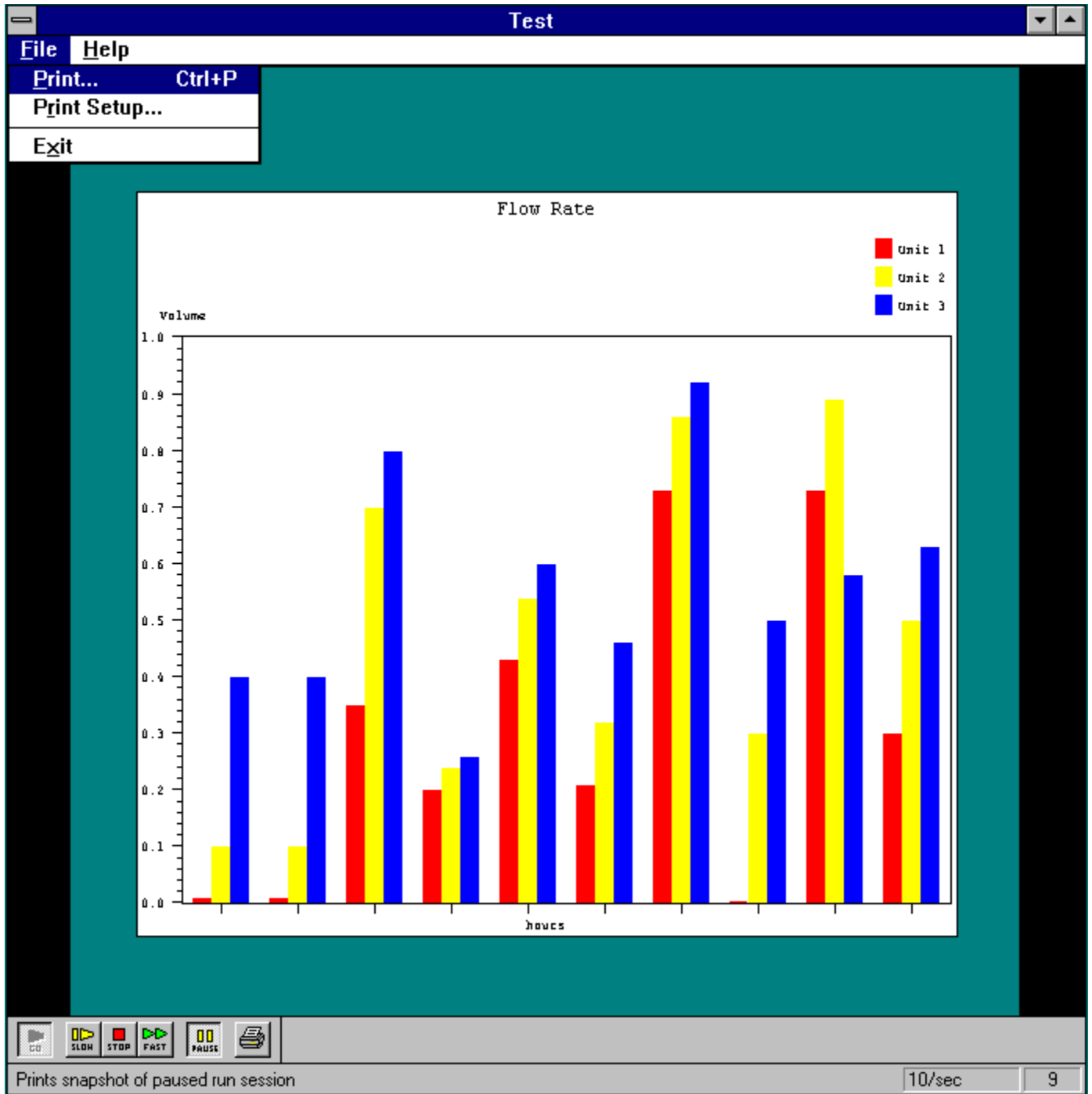
When creating walking menus, make all "Done" events erase all visible overlays to clear the area for the next walking menu or message. If this erases the base menu, too, make all "Done" events also overlay the base menu again. This is easier and faster than erasing each overlaid menu with a separate rule. These rules should be placed first in the list.

Running a Prototype

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Running a **prototype** is exactly like running a view with the addition of all rules being active. Running a prototype lets you sequence from one view to another, and overlay views and objects. You can run prototypes in DV-Draw by using the Run Menu with the "Interpret Rules" option turned on. You can run prototypes outside DV-Draw by using the *DVproto* script.

To run a view or prototype in DV-Draw, pull down the File Menu from the menu bar and select the "Test" option or click on the run button in the palette to display the Test Window:



The buttons at the bottom of the window let you control the test run. In addition to the buttons described in the *DV-Draw User's Guide*, the options include an "Interpret Rules" toggle:

Interpret Rules **Interpret Rules:** Runs the view as a prototype, with rules active. The "Interpret Rules" toggle at the bottom of the test window is tuned off by default and rules are not active.

Setting the Prototype Environment

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

You can run a prototype either inside DV-Draw by using the Run Menu with the "Interpret Rules" option turned on, or outside DV-Draw by using the *DVproto* script.

Before running a prototype using either the DV-Draw Run Menu or the *DVproto* script, you can specify the root view, first view, wait view, and other environment conditions in your configuration file by editing the configuration file as instructed below. The following variables are described in the *Setting the DataViews Environment* appendix.

Set the following variables in your prototype configuration file:

Mandatory *DVproto* Variables (Optional for DV-Draw)

DVPATH

DVDEVICE

DVVIEW

Optional *DVproto* Variables

DVCOLORTABLE

DVPRELOAD

DVROOTVIEW

DVWAITVIEW

DVDSOPENONLOAD

DVDSREADONLOAD

DVSAVERASTERS

DVDYNAMICS_ACTIVE

DVUPDATE_RATE

DVMERGE_ALL_DS

For example, the following configuration file lets you run a prototype starting with the view *top.v*. All views are preloaded, and the view *wait.v* is displayed during preloading. All data sources are opened and merged during preloading. Dynamics are active and the update rate is moderate.

Example: `.DVconfig.proto`

```
# Prototype Configuration File
# Search Path for views and data files
DVPATH = "datafiles drawings images layouts views
          %DVHOME%\etc
          %DVHOME%\lib
          %DVHOME%\lib\views
          %DVHOME%\lib\fonts
          %DVHOME%\lib\icons
          %DVHOME%\lib\images
          %DVHOME%\lib\templates %DVHOME%\lib\templates\drawings"

# Display Device (for use with DV-Draw, DV-Tools, DVplay, and DVproto)
DVDEVICE = w          # Display device

# Variables Specific to DVproto Runtime Environment
DVVIEW = top.v      # Starting view name
DVPRELOAD = YES     # Preload views for DVproto only
DVWAITVIEW = wait.v # View to display during preloading
DVROOTVIEW = top.v  # View for root of tree for preloading
DVDSOPENONLOAD = YES # Open data source when loaded
DVDSREADONLOAD = NO # Read data source when open
DVSAVERASTERS = YES # Save rasters for overlay/popup objs
DVUPDATE_RATE = 6   # Update Rate Slowest = 1, Fastest = 10
DVMERGE_ALL_DS = NO # Merge all ds or just FILE
```

Using Data Sources in a Prototype

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

When setting the *DVPRELOAD*, *DVDSOPENONLOAD*, and *DVDSREADONLOAD* flags in your configuration file, it is helpful to understand their effect on the way your data sources are merged during a prototype run.

A new file data source is only merged into an already loaded data source if the new and loaded data sources share the same name *and* the new data source variables are a subset of the loaded data source variables. Individual data source variables must match exactly.

The order in which data sources are merged is determined by the order in which views are loaded. The *DVPRELOAD* flag determines whether all views are loaded and their data sources merged when you start the prototype or as each view is required. If *DVPRELOAD* is set to *YES*, all views are loaded in hierarchical order when you start the prototype. The order in which data sources within views are merged is determined by the order in which the objects were created. If *DVPRELOAD* is set to *NO*, views are loaded as you access them.

The *DVDSOPENONLOAD* flag determines when data sources are opened and closed. If *DVDSOPENONLOAD* is set to *YES*, each data source is opened when it is loaded, and closed at the end of your prototyping session. If *DVDSOPENONLOAD* is set to *NO*, *each* data source is opened when it is required by a view or object, and closed when no visible object is using it.

The *DVDSREADONLOAD* flag determines when data sources are read. If *DVDSREADONLOAD* is set to *YES*, all data sources are read continuously, regardless of the view displayed. This can be used to simulate real time behavior. If *DVDSREADONLOAD* is set to *NO*, each data source is read only when the view or object using it is being displayed.

The *DVMERGE_ALL_DS* variable lets you specify whether to merge only file and process data sources, or all data sources (file, process, memory, constant, and function). The default setting of *DVMERGE_ALL_DS* is *NO*. This prevents multiple input objects from using the same memory data source variable. If you *want* input objects or other dynamic objects to share memory data source variables, set the *DVMERGE_ALL_DS* flag to *YES*.

Running a Prototype in DV-Draw

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

You can run a view with rules active or inactive by setting the "Interpret Rules" toggle at the bottom of the test window. This section describes how DV-Draw runs a view with rules active. The way DV-Draw runs a view with rules inactive is described in the *Running Views* chapter.

To run a prototype in DV-Draw, pull down the File Menu from the menu bar and select the "Test" option or click on the run button in the palette. Then turn on the "Interpret Rules" toggle at the bottom of the test window.

When you run a view with rules active, DV-Draw performs all of the steps described in the *Running Views in DV-Draw* section with the addition of the following steps:

1. DV-Draw displays the current editing view. If the *DVPRELOAD* flag is set to *YES*, DV-Draw traverses the initial view or root view and loads all views required by the prototype. If a wait view is specified by the *DVWAITVIEW* configuration variable, it is displayed at this time.
3. DV-Draw opens and reads the data sources according to the setting of the *DVOPENONLOAD* and *DVREADONLOAD* configuration variables.
4. The prototype is ready for user input. All rules, graphs, input objects, and object dynamics are active. The prototype updates dynamics according to the setting of the *DVDYNAMICS_ACTIVE* flag or if the "Start Dynamics" or "Stop Dynamics" actions occur. If the Status Menu is displayed, it displays the current state of the prototype. The prototype reads data according to the setting of the *DVREADONLOAD* flag. If a data source runs out of data, the last value is used repeatedly until you terminate the run.

Satisfying the events and conditions of the prototyping rules causes the specified changes in the display.

If a prototype view cannot be loaded in DV-Draw, an error message appears. You must acknowledge the error message before continuing. If the view is unavailable, rules requiring that view do not work.

If you stop the test run, the display reverts to the view you started with. Exiting the test window returns to the DV-Draw editor. You can exit the test window without stopping the test run, but no matter what view is displayed in the prototype when you exit, DV-Draw returns to the last view being edited.

Running a Prototype Outside DV-Draw: Using DVproto

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

To run a prototype outside DV-Draw, use the *DVproto* script. If your configuration file is set up correctly, no command line options are required. Environment variables which can be set on the command line are *-d* (*DVDEVICE*), *-v* (*DVVIEW*), *-i* (*DVPROTOPATH*), and *-c* (*DVCOLORTABLE*). These variables are explained in the *Setting the DataViews Environment* appendix.

To invoke the *DVproto* script, type the following command on the command line.

```
DVproto
```

If a prototype view cannot be loaded, an error message appears in the window where you invoked *DVproto*. If the view is unavailable when the prototype is running, rules requiring that view do not work.

Resizing the window automatically redraws the screen in X window systems. If you are using any other device, you must redraw the screen with *^L* after resizing the window.

To quit, use the "Quit Prototype" option in your prototype. Quitting the prototype kills the window. In a window system, you can quit the prototype by killing the window.

Alphabetical Listing of Configuration File Variables

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

The following list describes the variables related to prototyping in alphabetical order. Some appear in the *Setting the DataViews Environment* appendix, but their descriptions differ slightly for prototyping.

DVROOTVIEW: Specifies the root view of your prototype to be used for preloading. Use the name of a previously saved view file.

DVSAVERASTERS: Indicates whether or not the raster underlying an object is saved before the object is drawn as a popup or overlay. This option must be set to *yes* in order to restore the raster image when the popup or overlay is erased. If *no*, the erase method redraws the overlaid image in the background color of the base view. To restore objects in the base view, include a rule with a "Redraw" action after the rule with the erase action. The default is *yes*.

DVVIEW: Specifies the view to be pre-loaded when starting *DVplay*, *DVproto*, or DV-Draw. Use the name of a previously saved view file.

The command line equivalent for *DVplay* and DV-Draw is *-v your_view*.

DVWAITVIEW: Specifies a static view to be displayed while the prototype is preloading views. Use the name of a previously saved view file.

Customizing the DV-Draw Data Model
DV-Draw User's Guide (new sections)

[Overview](#)

[Contents of the Examples Directory](#)

[DataViews and Data Sources](#)

[How DVDraw Uses the Data Source DLL](#)

[Designing Your Data Interface](#)

[Coding Your Interface](#)

[Coding Your Dialogs](#)

[Writing the DLL](#)

[The CDvDsDll Class](#)

[The GetIdInfo\(\) Method](#)

[The HandleRequest\(\) Method](#)

[The GetInterface\(\) Routine](#)

[Encoding Information in the Variable Descriptor Name](#)

[Integrating Your Data Model into DVDraw](#)

[Setting the Registry Key](#)

[Editing the Run DLL](#)

[Requirements](#)

[The Context Flags](#)

This document discusses the following topics:

Introduction to Customizing the DV-Draw Data Model

Contents of the Examples Directory

How DataViews Uses Data Sources

How DV-Draw Uses the Data Source DLL

Designing Your Data Interface

Coding Your Interface

Coding Your Dialogs

Writing the DLL

The *CDvDsDll* Class

The *GetIdInfo()* Method

The *HandleRequest()* Method

The *GetInterface()* Routine

Encoding Information in the Variable Descriptor Name

Integrating Your Data Model into DV-Draw

Editing the Run DLL

Requirements

The Context Flags

Overview

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

DV-Draw provides a default data model, with dialogs that let you edit the data source list, specify data source variables, and use data source variables for graphs, input object, object dynamics, nested subdrawings, rules, and grouping. However, if your data structure does not fit the model used by DV-Draw, you can customize the DV-Draw interface to match your own data model. Since DV-Draw is written using separate DLLs for various components of the editor, you can customize the DV-Draw data model by replacing the default data source DLL with one of your own and performing some additional housekeeping tasks. To customize DV-Draw to handle your data model, you must:

Design your data interface.

This step involves analyzing your data model and selecting the data interface options you want DV-Draw to support. You must also design the dialogs that will appear in DV-Draw to let you work with your data structure.

Code your interface.

You can design and code your dialogs using a separate Visual C++ application such as Visual Studio. You must also create a data source DLL, with three methods for the *CDvDsDll* class and a C function to instantiate an instance of your class.

Integrate your data model into DV-Draw.

This step involves coordinating the DLL name and location with the corresponding variable in the registry. If your data model is different from the default model in DV-Draw, and if you want to be able to run views using your data in the test window, you must also edit the Run DLL to support your model.

This release includes a sample of a customized DLL, together with a companion Run DLL, in the *examples* directory under the *dvdDraw* directory. This document uses the example to illustrate various steps in each of the stages listed above.

Contents of the Examples Directory

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

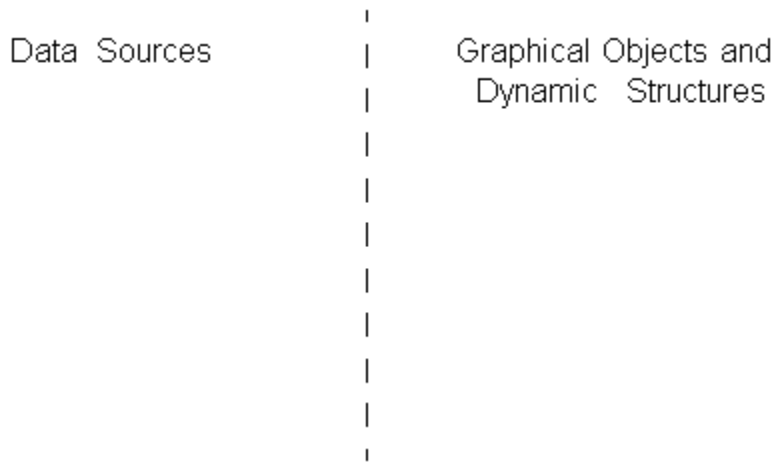
The following list briefly describes some of the files in the *examples* directory under the *dvdraw* directory. The examples directory also contains other files related to the example DLL project, but the list only describes files that are of particular interest in the following discussions. The descriptions below indicate which portions of these files were written by hand, since these are code sections you will have to write in your own files.

| | |
|----------------------------|---|
| <i>DSTEMPLATE.CPP</i> | This is the starting point for a custom data source DLL. It contains the definition of <i>DllMain()</i> . It also has the <i>CMyDsDll</i> class. The <i>Release()</i> method is where you can "tidy up," such as deallocating any resources you have allocated for your data source. The <i>GetIdInfo()</i> method supplies your company name, product name, and revision levels for the About button. The <i>HandleRequest()</i> method produces the appropriate data selection response to each data request situation. |
| <i>VARDEFS.DAT</i> | This file simulates a data source list. It was created entirely by hand. |
| <i>GENERICLISTCTRL.CPP</i> | This code is used by a control inside the dialog. The message handler body was written by hand. |
| <i>GENERICLISTCTRL.H</i> | This is the generated header file for <i>GENERICLISTCTRL.CPP</i> . |
| <i>VARSELECTDLG.CPP</i> | This is the code for the dialog that presents the data source list for data selection. All of the message handler bodies were written by hand. |
| <i>VARSELECTDLG.H</i> | This is the generated header file for <i>VARSELECTDLG.CPP</i> . |
| <i>EXAMPLESDLL.CPP</i> | This contains the <i>CDvDsDll</i> member function <i>HandleRequest()</i> and the <i>GetInterface()</i> C routine for initializing the DLL. |
| <i>EXAMPLESDLL.H</i> | This header file contains the definition of the <i>CExampleDsDll</i> class. |
| <i>EXAMPLESDLL.RC</i> | This is a listing of all the Microsoft Windows resources used by the program. These are the resource that can be edited using the Visual Studio. |
| <i>RESOURCE.H</i> | This is the generated header file used by <i>EXAMPLESDLL.RC</i> . |

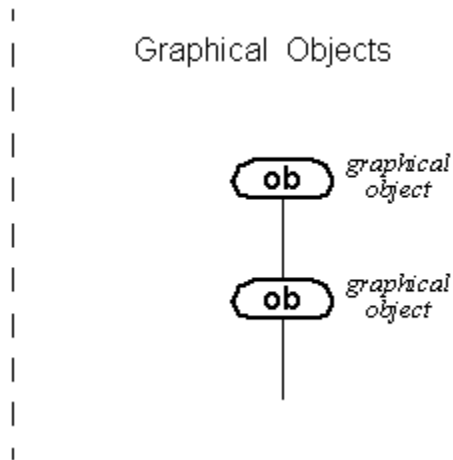
DataViews and Data Sources

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

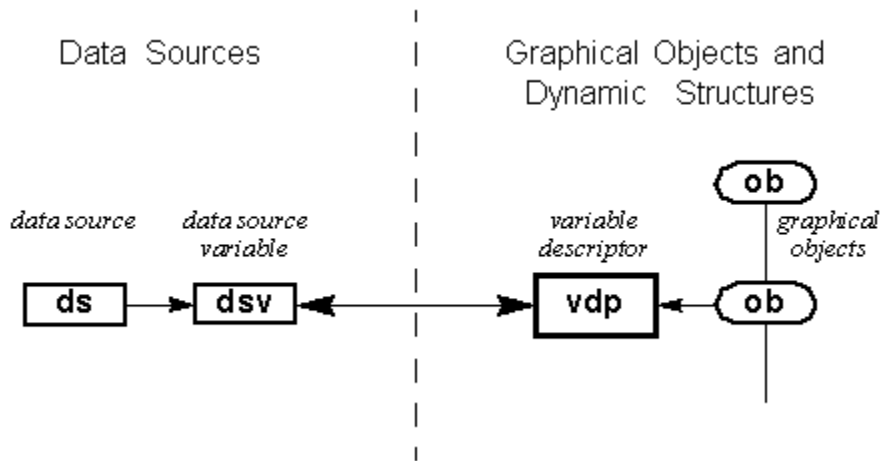
Before we discuss the data source DLL, it helps to understand how DataViews uses data sources. Experienced DataViews programmers are familiar with the data structures diagram in the Structures chapter of the *DV-Tools User's Guide* which shows the structural organization of a DataViews view. For the purpose of understanding the role of data sources in DataViews, we can simplify this to the division between the graphical and dynamic structures and the data source information:



A view that has only static objects and no dynamic objects contains no data source connections:

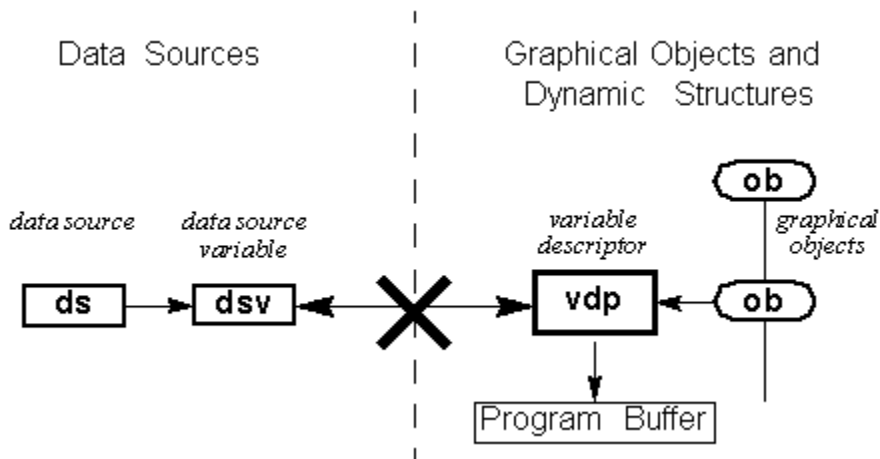


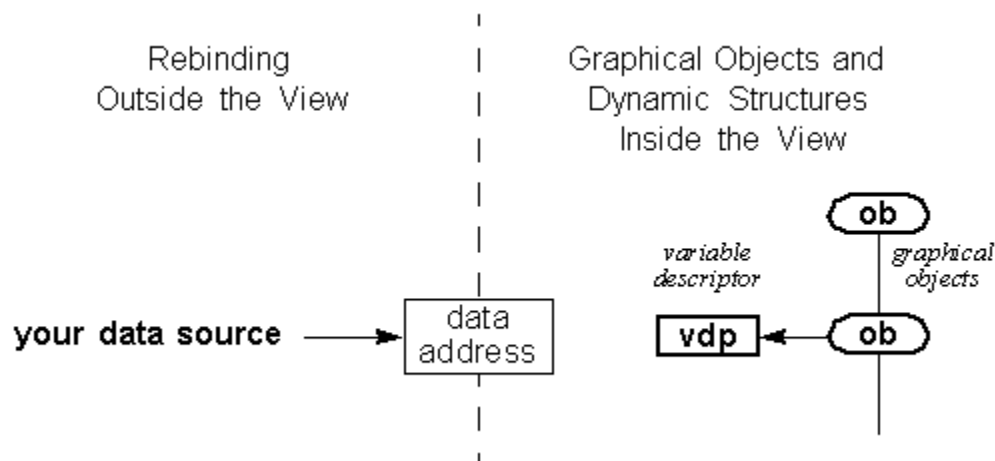
When you add dynamics to a view, you add data source information to the data source side, and a structure called a **variable descriptor**, or *vdp*, to the graphical objects side:



The variable descriptor contains information about where to get the data for this object. The data source list includes information such as the names of the data sources and the names of their variables. At run time, the program uses this information to find the data sources, and the data sources put data into the location pointed to by the variable descriptor.

In previous versions of DataViews, you were limited to the kinds of data you could use to run a view in DV-Draw. You could use data from a file, from a memory data source, or from an evaluator function data source. These are data sources that DV-Draw can use to provide data in the run mode. In order to use other kinds of data source in a DV-Tools application, you use a technique called **rebinding**, which severs the connection between the variable descriptor and the data source variable, and lets you supply data to the variable descriptor in any way you like:





Rebinding lets you use your data in an application, but not in the DV-Draw editor. Customizing the data source DLL is a way to use rebinding within the context of the DV-Draw editor.

Customizing the data source DLL also lets you run views using your own data in the DV-Draw editor, but only if you also write a Run DLL to use the same data model. The Run DLL provided with DV-Draw uses the information provided in the data source list to find the data sources and put data into the location specified by each data source variable. The Run DLL is designed to work with the default DataViews data model. If you write a customized data source DLL to use a different data model, you must also write a companion Run DLL if you want to run views with your data model. This is discussed in [the Editing the Run DLL section](#).

How DV-Draw Uses the Data Source DLL

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Before we discuss the details of customizing the data source DLL, it is useful to understand how DV-Draw uses it. Communication between the DV-Draw executable and the data source DLL begins as soon as you start DV-Draw. When you start DV-Draw, the executable checks the registry for the location of the data source DLL and loads that DLL. If no data source DLL is found, the resulting instance of DV-Draw has no data-related functionality. If a data source DLL is found, DV-Draw queries the DLL to determine which editing contexts it supports. This determines what data options are available in DV-Draw. For example, if the data source DLL does not provide stand-alone editing of the data source list, no data button appears in the palette.

When the DV-Draw user initiates a task that requires access to data, the DV-Draw executable determines the context of the request for data access, such as creating a graph or editing object dynamics, and sends a request to the DLL that includes this context information. The DLL then presents the appropriate dialog. If the user makes any kind of change that affects the content of the view, the DLL sets a dirty flag. When the user finishes using the data interface, DV-Draw checks this flag, and if it is set, DV-Draw sets its own flag to indicate that the view has been changed.

Designing Your Data Interface

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)


- In the design stage, you must: Analyze your data model.
- Select the contexts in which you want to present data source dialogs.
- Design dialogs to present data options in the DV-Draw interface.
- Decide how you want to store data information in the view.

The default DataViews data model is based on data sources and data source variables. In designing your own data interface, you must analyze your own data model and determine how it is different from the DataViews data model. For example, does it use shared memory? Is it a relational database? Is the data structure hierarchical? How your data model differs from the default DataViews data model determines how you present your model in DV-Draw and what additional steps you need to take before DV-Draw can use your model.

DV-Draw supports a specific set of contexts in which data source dialogs are presented. You must select the contexts in which you want to display data source dialogs. The options are:

Stand-alone editing:

The default DV-Draw dialog allows for adding, deleting, and editing data source and data source variable descriptions, without regard to any particular object. This context allows editing the descriptions of your data in a stand-alone dialog. If you choose not to support this context, you can still allow the editing of your data descriptions in the context of assigning data to objects or dynamics. You can also choose not to allow editing the descriptions of your data in any context, but only to allow selecting data for specific purposes. If

this context is not supported, no Data button, , appears in the DV-Draw interface.

Editing graphs, input objects, object dynamics, and rules:

Do you want to allow selection from a list of data sources, or do you want to use only a pre-set data source? Do you want to offer selection from the entire data collection, or do you want to limit selection to data appropriate to this kind of object?

Default data for graphs, input objects, object dynamics, and rules:

When you create a new graph, a new input object, new object dynamics, or a new rule, do you want to present a dialog to let the user make a data selection or do you want to attach a data source by default? If by default, what type of data do you want to assign?

Mapping data source variables:

In the case of nested subdrawings, do you want to be able to replace data sources in a subdrawing view with data sources in the parent view?

Answering these questions should help you design the dialogs you want to present in DV-Draw to allow the assignment of data to objects in a view. The specific options you provide depend on the nature of your data and the amount of control you want to provide in DV-Draw.

For example, in the example DLL, we want to present the data list in the contexts of editing the data selections for object dynamics, object rules, graphs, and input objects. We want the editor to make a default selection when you create a graph or input object, but we want to present the data list when you create object dynamics or rules. The only options we want to support in the editor are selecting a data source and replacing it with a different selection, so we only need one dialog. You cannot edit the list, so we do not support stand-alone editing.

In conjunction with designing your dialogs, you must determine what information you need to store in the view to let the view use your data when you run it. When you use the default data model, DV-Draw saves a data source list as part of the view; when you use your own data source model, it doesn't. However, it does store a variable descriptor for each data assignment, and the variable descriptor name. You must determine where to store the information that tells the program where to find the data for the object. One option is to embed this information in the variable descriptor name. Other options include storing this information elsewhere in the view, such as by using DV-Tools slots, or even by using the view comment field. You can store any kind of information about the data selection, including settings from the fields of your data selection dialog.

This approach lets DataViews maintain a constant view structure no matter what data model you use. Since the variable descriptor structure remains the same no matter what data source model you use, you can run a view in a version of DV-Draw that uses a different data source model, provided you have created a Run DLL to use your data model. Note, however, that if DV-Draw cannot find the data associated with a variable descriptor, it cannot display meaningful data when you run the view.

With this in mind, you can analyze what information must be in your view in order to find the correct data when you run it, and where you want to store that information in the view.

In the example, the variable descriptor name will include the data source name plus tagname information about the data source.

Coding Your Interface

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

In the coding stage, you must provide all of the code components for the data source DLL, including: your own dialogs

the *CMyDsDll* : *public CDvDsDll* class

the *Release()* method

the *GetIdInfo()* method

the *HandleRequest()* method

the *GetInterface()* C routine

Coding Your Dialogs

You can produce most of your dialog code by using a Visual C++ application such as Visual Studio. This lets you efficiently design your dialogs and generate code for their basic functionality. You must add code to the generated files to let your dialogs respond to user input such as text entries and menu selections. How your dialogs connect to your data, and how they insert information into the data specification stored in the DV-Draw view is entirely up to you. The resulting dialog files, header files, resource files, and any other related files will be compiled into the data source DLL.

Also note that all dialogs must be modal.

The example DLL uses a single dialog to present a list of variable descriptor configurations to the user:

| Variable | Type | Min | Max | Shape |
|-------------|--------|-----|-----|-------------------|
| DefFloatVar | FLOAT | 0 | 1 | SCALAR |
| Float:1 | FLOAT | 0 | 1 | SCALAR |
| Float:2 | FLOAT | 0 | 2 | SCALAR |
| Float:3 | FLOAT | 0 | 3 | SCALAR |
| Float:4 | FLOAT | 0 | 2 | SCALAR |
| Float:5 | FLOAT | 0 | 5 | SCALAR |
| RowVector | FLOAT | 0 | 1 | ROW VECTOR[1][5] |
| ColVector | FLOAT | 0 | 1 | COL VECTOR[5][1] |
| Matrix | FLOAT | 0 | 1 | MATRIX[5][5] |
| DefTextVar | TEXT[] | 0 | 1 | ROW VECTOR[1][16] |

We used Visual Studio to design the dialog. Creating this dialog produced the following generated files, which were then edited by hand. These files are in the DV-Draw *examples* directory. Again, the descriptions indicate which portions were written by hand, since these are code sections you will have to write:

- GENERICLISTCTRL.CPP* This subclass of *CListCtrl* lets the data source list dialog get the selected list item. The message handler was written by hand.
- GENERICLISTCTRL.H* This is the generated header file for *GENERICLISTCTRL.CPP*.
- VARSELECTDLG.CPP* This is the code for the dialog that presents the data source list for data selection. All of the message handlers were written by hand.
- VARSELECTDLG.H* This is the generated header file for *VARSELECTDLG.CPP*.

Although we wrote all of the methods in *GENERICLISTCTRL.CPP* and *VARSELECTDLG.CPP*, only three code segments are of particular interest here. One shows context sensitivity in the data source DLL. One shows how the data source DLL meets the requirement for the user to make a valid data selection in the data selection dialog. And the third shows how to call the standard DV-Draw data browser from within your custom DLL.

The following code in *VARSELECTDLG.CPP* demonstrates how the data source DLL can determine the context of a

data request. The example changes the dialog title to reflect the current context:

```
// An example of how a data source DLL can
// accurately discriminate the various contexts
// it is being called from within DV-Draw.

CString context;

if(m_nContext & DV_DSDLL_SELECTDATA)
    context = "Select new variable for ";
else
    context = "Changing variable in ";

if(m_nContext & DV_DSDLL_EDITDYNAMICS)
    context += "DYNAMICS";
else if(m_nContext & DV_DSDLL_EDITGRAPH)
    context += "GRAPH";
else if(m_nContext & DV_DSDLL_EDITINPUT)
    context += "INPUT OBJECT";
else
    context += "!!! UNKNOWN !!!";

SetWindowText((LPCTSTR)context);

return TRUE; // return TRUE unless you set the
             // focus to a control EXCEPTION: OCX
             // Property Pages should return FALSE
```

Another way you could use this information might be to pop up different dialogs for various contexts.

When the user requests a data selection, DataViews requires the user either to cancel the dialog, or to make a valid data selection before selecting "OK." The following code in *VARSELECTDLG.CPP* satisfies this requirement by refusing to close the dialog if the user does not make a valid data selection before selecting "OK."

```
void CVarSelectDlg::OnOK()
{
    // Is variable selection required?
    BOOL needVAR = (BOOL)((m_nContext & DV_DSDLL_SELECTDATA) ||
        (m_nContext & DV_DSDLL_EXCHANGEDATA));
};

if(!needVAR || (needVAR && m_pSelectedVarData))
    CDialog::OnOK();
else
    Beep(300,150);
}
```

The following code segments from *VARSELECTDLG.CPP* illustrate how to call the standard DV-Draw data browser from within your custom DLL. They call the standard DataViews data source DLL, display a warning if none is found, initialize the standard DataViews data sources for the view, and act according to standard DV-Draw data browser behavior for the current context.

First, when the user clicks on the DataViews logo button, this code finds the standard DataViews data source DLL and calls the *DVGetInterface* method to instantiate the interface object to the standard data browser:

```
////////////////////////////////////
//
void CVarSelectDlg::OnStdBrowser()
{
    CDvDsDll* pDsDll;
```

```

// Get the pointer to the standard DataViews data
// browser. Note that the ID used is
// IID_DV_Datasource, not IID_Datasource, which
// would return a pointer to data browser listed
// in the registry, which in this case would be
// ourselves.
//
DVIGetInterface(IID_DV_Datasource, (void**)&pDsDll);

```

Note that *IID_DV_Datasource* is hard coded in a DV-Draw header file so it always points to the standard data source DLL.

If DV-Draw cannot find the standard DataViews data source DLL, we display an error message:

```

if(!pDsDll)
{
    AfxMessageBox("Unable to load standard DataViews Data Browser!");
    return;
}

```

If this is the first time the user has called for the standard data browser, we call the standard DLL's *DV_DSDLL_INITNEWVIEW* method to perform the standard view and data source initialization, which displays the usual default data options in the browser dialog:

```

if(m_FirstTime)
{
    // Our data browser does not work with the view
    // directly, so there are no default datasources
    // created when the view is initialized. For
    // our purposes here as an example, if its the
    // first time, we'll ask the standard data
    // browser to init the view; that way there will
    // be something to see (otherwise its an empty
    // datasource list).
    int real_context = m_pData->m_Context;

    m_pData->m_Context = DV_DSDLL_INITNEWVIEW;
    pDsDll->HandleRequest(m_pData,this);
    m_pData->m_Context = real_context;

    m_FirstTime = FALSE;
}

```

It is not necessary to initialize the browser display this way; if you don't, the user can still create data sources and data source variables. However, displaying a default data source and data source variable can help the user understand the dialog and how to use it.

Finally, we call the standard DLL's *HandleRequest()* method in order to act according to standard DV-Draw data browser behavior for the current context:

```

if((pDsDll->HandleRequest(m_pData,this)) == IDOK) )
{
    // User has selected OK button, so we could do
    // some application-specific things at this
    // point. The assignment below is merely a
    // convenient place for stopping the debugger so
    // you can see what is happening.
    DSVAR newDsvar = m_pData->m_DsVar;
}

```

```
}  
}
```

If you create views that use both your custom data model and the standard DataViews data source model, and you want to run your views in the DV-Draw test window, you must write a custom Run DLL that handles all the data.

Writing the DLL

To use your dialogs in DV-Draw, you must write a DLL. A template DLL file, *DsTemplate.cpp*, is provided in the DV-Draw *examples* directory. The following discussion also provides examples from *EXAMPLESDLL.CPP*.

The includes section of *DsTemplate.cpp* provides the following standard *#include* statements:

```
#include "stdafx.h"  
#include <afxdllx.h>  
#include <afx.h>  
#include "DvDsDll.h"
```

In *EXAMPLESDLL.CPP*, we simply changed the name of the *DvDsDll.h* file to the name of our DLL header file:

```
#include "ExampleDsDll.h"
```

You must also add *#include* statements to include your dialog files. In *EXAMPLESDLL.CPP*, the statement that includes our single dialog is:

```
#include "VarSelectDlg.h"
```

If your data source DLL uses any DV-Tools calls, you must also include the header files for those routines, as demonstrated in the following *#include* statements from *EXAMPLESDLL.CPP*:

```
#include "dvparams.h"  
#include "VUregpaths.h"  
#include "VUfunddecl.h"  
#include "VPfunddecl.h"  
#include "VGfunddecl.h"
```

The following object declaration is provided in *DsTemplate.cpp*. It allows MFC subclassing by creating an extension DLL and requires no changes:

```
static AFX_EXTENSION_MODULE MyDsDLL = { NULL, NULL };
```

In *EXAMPLESDLL.CPP*, we define a convenience macro that indicates which data editing contexts this data source DLL supports. This is used later to set the *m_nContext* flag, and provides a convenient way to specify the supported contexts, as required for the context query:

```
#define STD_SUPPORT (DV_DSDLL_EDITDYNAMICS | DV_DSDLL_EDITGRAPH |  
                   DV_DSDLL_EDITRULES | DV_DSDLL_EDITINPUT) |  
                   DV_DSDLL_USESCUSTOMTAGS
```

The context flags and feature flags are defined in *DVSDLL.H*. The flags used by the *STD_SUPPORT* macro are:

```
DV_DSLL_EDITDS      0x // Editing data sources (read/write).  
DV_DSLL_EDITDYNAMICS 0x // Editing object dynamics.  
DV_DSLL_EDITGRAPH   0x // Creating/editing a graph.  
DV_DSLL_EDITINPUT   0x // Creating/editing an input object.  
DV_DSLL_EDITRULES   0x // Creating/editing rules.
```

The complete list of context flags and feature flags defined in *DVSDLL.H* are presented at the end of this document.

The CDvDsDll Class

To use your data source, you must write a *CDvDsDll* class. You can subclass your *CDvDsDll* class off the base class,

CDvDsDll. The following outline is provided in *DsTemplate.cpp*:

```
class CMyDsDll : public CDvDsDll
{
public :
    void Release() { delete this; }
    void GetIdInfo(CString& mfgName, CString& objName, CString& versionStr);
    int  HandleRequest(DV_DSDLLDATA* pData, CWnd* pParent);
};
```

You can use the *Release()* method as it appears in *DsTemplate.cpp*, or you can add code to perform additional clean-up, such as deallocating any resources you may have allocated for resources. No additional code is provided for this method in *EXAMPLESDLL.CPP*. You must write the body for the *GetIdInfo()* and *HandleRequest()* methods. The hand-written code provided for these methods in *EXAMPLESDLL.CPP* is described below.

The GetIdInfo() Method

The following *GetIdInfo()* method is provided in *DsTemplate.cpp*. All you need to do here is change the strings to reflect your company name, your data source DLL name, and the data source DLL revision levels:

```
void CMyDsDll::GetIdInfo(CString& mfgName,
                        CString& objName, CString& versionStr)
{
    mfgName    = "My Corporation";
    objName    = "My Data Browser";
    versionStr = 1.0.0.1;
}
```

The information provided in the *GetIdInfo()* method is displayed when the user presses the "About" button in the DV-Draw interface. This is useful in debugging, to confirm that you are using the data source DLL you think you are using.

The HandleRequest() Method

The *HandleRequest()* method must determine if the current data request context is supported by the DLL, and respond appropriately to each context that is supported. Each request context is illustrated in the following code samples from *EXAMPLESDLL.CPP* and the *HandleRequest()* method used by the default DV-Draw data source DLL. The *HandleRequest()* method in *DsTemplate.cpp* includes a set of *else if* statements that serve as place-holders for your code.

The *HandleRequest()* method in *EXAMPLESDLL.CPP* starts by initializing the data source variable and the dirty flag:

```
pData->m_DsVar = 0;
pData->m_Dirty = FALSE;
```

The data source variable initialization is required because the example's data model does *not* use data source variables. *DsTemplate.cpp* includes the *m_dirty* flag initialization. The *m_Dirty* flag is defined in *DVDSDLL.H*.

The *HandleRequest()* method in *EXAMPLESDLL.CPP* then determines the data request contexts that are supported by the DLL:

```
if(pData->m_Context & DV_DSDLL_QUERYDLL)
{
    // Initial query request of what we support
    pData->m_Context = (DV_DSDLLCONTEXT) STD_SUPPORT;
    return IDOK;
}
```

This tells DV-Draw how to configure itself, such as whether to display the data button, , and how to behave

under different data request contexts.

Remember that *EXAMPLESDLL.CPP* previously defined the *STD_SUPPORT* convenience macro to specify the supported contexts. An alternative is to specify the supported contexts here.

In the next code segment, the *HandleRequest()* method in *EXAMPLESDLL.CPP* separates the variable descriptor name from any additional information embedded in the name in preparation for displaying the names in the data source list dialog:

```
else if(pData->m_Context & DV_DSDLL_GETNAMEBITS)
{
    GetVarTagName(pData);
    return IDOK;
}
```

The *GetVarTagName()* method extracts the part of the name that you want to display in the data selection dialog. This method is discussed in [the Encoding Information in the Variable Descriptor Name section](#).

The *HandleRequest()* method is where your custom DLL responds to each data support option you support. As stated before, the context flags and feature flags are defined in *DVSDLL.H*. The following code samples illustrate some of these contexts but not all.

The following code sample from the default data source DLL illustrates a response to the creation of a new view, either by starting up DV-Draw or by selecting "New" from the File Menu. This is where you can set up a default data source or perform other initialization tasks associated with creating a new view:

```
else if(pData->m_Context & DV_DSDLL_INITNEWVIEW)
{
    CreateDefaultDotDat(pData);
}
```

The default data source DLL also responds to the case of cutting and pasting an object that has data associated with it. The primary view is the view where the original object is, and the secondary view is the target view when you paste an object to any view other than the primary view.

```
else if(pData->m_Context & DV_DSDLL_RECONNECTDATA)
{
    if(!pData->m_Sview)
        // If pasting to the primary view, we just
        // reconnect the current set of vdp's dsvs from
        // where they point to to some dsvars in the
        // primary view of comparable type.
        TobForEachVdp(pData->m_Ob, ConnectPasteObjectVdps, (ADDRESS)pData);
    else
    {
        // If pasting to a secondary view, we basically
        // do a 'save subview' and need to clone the
        // data sources.
        SUBVIEWARGS svargs;
        svargs.dsHt = VThtcreate("SubViewHT", 0, 0);
        svargs.subviewDsl = TviGetDataSourceList(pData->m_Sview);
        TobForEachVdp(pData->m_Ob, CreateSubviewData, (ADDRESS)&svargs);
        VThtdestroy(svargs.dsHt, 0, 0);
    }
}
```

The *m_Sview* flag is defined in *DVSDLL.H*.

The first case applies when you cut and paste objects within the same view. Note that DV-Draw does not necessarily connect the new object to the same data source variable that the original was connected to. Rather, DV-Draw locates the first data source variable of comparable type and connects to that.

The second case applies when you cut and paste objects from one view to another, or when you save objects to a subview. In this case, DV-Draw simply clones the variable descriptor.

Again, this behavior is provided by the default DV-Draw data source DLL. You can implement a different policy in your custom DLL. Note, however, that you must connect input objects to a data source variable or point them to a memory location, or DV-Draw may crash. Input object connection is demonstrated below.

The next example occurs in both the default data source DLL and in *EXAMPLESDLL.CPP*. It demonstrates a response to a request for default data when you create a graph or input object:

```
else if(pData->m_Context & DV_DSDLL_DEFAULTDATA)
{
    // Default data connection request. Only need to
    // worry about input objects having a memory data
    // source.
    if(pData->m_Context & DV_DSDLL_EDITINPUT)
        AttachVdpToDefMemVar(pData);
    else
        AttachVdpToDefVar(pData);
    return IDOK;
}
```

If the object is an input object, the *AttachVdpToDefMemVar()* method points the variable descriptor to a memory location. Otherwise, the object is assumed to be a graph, and the *AttachVdpToDefVar()* method is called. In the default data source DLL, *AttachVdpToDefMemVar()* attaches a default data source variable; in *EXAMPLESDLL.CPP*, *AttachVdpToDefMemVar()* points the variable descriptor to a memory location.

The following code in *EXAMPLESDLL.CPP* handles the *DV_DSDLL_SELECTDATA* and *DV_DSDLL_EXCHANGEDATA* contexts:

```
// Check the context of the request.
else if(pData->m_Context & DV_DSDLL_SELECTDATA || pData->m_Context &
        DV_DSDLL_EXCHANGEDATA)
{
    // Instantiate the selection dialog.
    CVarSelectDlg varDlg(m_pVarDefList, pData, pParent);
    // Make the dialog modal and display it.
    if(varDlg.DoModal() == IDOK)
    {
        CString    varName;
        // Return the data structure for the selection.
        VARDEFDATA* pVarData = 0;

        pVarData = varDlg.GetVarSelection();

        // If the type is anything but UNDEFINED,
        // verify that the requested type is ok
        // for the selected object. If it isn't, by
        // convention we return IDOK, but set the
        // m_DsVar field to NULL. This is specifically
        // used if the user changes a variable in an
        // input object but picks another variable of
        // the wrong type, such as selecting a float
        // although a textvar is required.
        if(pData->m_VarType != V_UNDEFINED && (pData->m_VarType != pVarData-
            >m_nVarType))
        {
            // If the variable is of the wrong type, do
```

```

        // not configure the vdp, but set the
        // variable name to something that indicates
        // there is a problem, then just return.
        VPvdvarname (pData->m_Pvdp, "Unbound_Variable");
        return IDOK;
    }

    // Make the data connection. Here we only
    // configure the vdp.
    if(pData->m_Pvdp)
        AlignVdpToVarDefinition(pData->m_Pvdp, pVarData);

    return IDOK;
}
}

```

Note that you could OR contexts together to perform different actions if the data request were a data selection for an input object or a data selection for a graph. The following context flags can be ORed together to specify the type of request and the type of object:

```

DV_DSLL_EDITDYNAMICS    0x // Editing object dynamics.
DV_DSLL_EDITGRAPH      0x // Creating/editing a graph.
DV_DSLL_EDITINPUT      0x // Creating/editing an input object.
DV_DSLL_EDITRULES      0x // Creating/editing rules.
DV_DSLL_DEFAULTDATA    0x // Request for a default data connection.
DV_DSLL_SELECTDATA     0x // Request for a data connection selection.
DV_DSLL_EXCHANGEDATA   0x // Request for a data selection and exchange.

```

The GetInterface() Routine

The other essential element of the DLL file is the C *GetInterface()* routine. This instantiates an object of base type *CDvDsDll*, which is the interface object of your DLL. You can use the version that appears in the template by making a single edit to change the DLL class name, *CMyDsDll*, to the class name of your DLL.

```

extern "C" {
    DLLEXPORT BOOL
    GetInterface(int idd, void **pInterf)
    {
        *pInterf = 0;
        if (idd == IID_Datasource)
            *pInterf = new CMyDsDll;
        return (pInterf) ? TRUE : FALSE;
    }
}

```

In *examplesdll.cpp*, *CMyDsDll* becomes *CExampleDsDll*:

```

    *pInterf = new CExampleDsDll;

```

Encoding Information in the Variable Descriptor Name

We have mentioned that you can encode data source information into the variable descriptor name. In our example, we use the variable descriptor name to store the name that you want to display in the data selection dialog, and information about the data selection. We refer to the first part as the human-readable name, and to the second part as tagname information. Your DLL must put this information into the following fields of the *pData* structure, defined in *DVDSDLL.H*:

```

char* m_pszVarname; Variable descriptor name.
char* m_pszVartags; Variable descriptor tags (optional).

```

EXAMPLESDLL.CPP uses helper functions to encode tagname information, and to separate the human-readable

name from the tagname information. The methods used for this are *SetVarTagName()* and *GetVarTagName()*.

The *SetVarTagName()* method in *EXAMPLESDLL.CPP* simply encodes some data selection information that is supplied in the data selection dialog. This would not be very useful in a real application, but simply shows how your custom data browser might encode information such as SQL commands that describe how to connect to the real data in a run-time environment:

```
void CExampleDsDll::SetVarTagName(VARDEFDATA* pData, const char* varName)
{
    pData->m_csVarName.Format(
        "%s;%d;%d;%d;%d;Example Data Browser",
        varName,
        pData->m_nRows, pData->m_nCols,
        pData->m_dLowRange, pData->m_dHighRange);
}
```

In *EXAMPLESDLL.CPP*, this method is used in the *ReadVarDefinitions()* method, which gets each variable name, allocates memory for it, and adds it to the list that appears in the data selection dialog.

The *GetVarTagName()* method extracts the human-readable part of the name, which is what you want to display in your data selection dialog. As with the *SetVarTagName()* method, the *GetVarTagName()* method in *EXAMPLESDLL.CPP* is only a rough example of how you can do this:

```
void CExampleDsDll::GetVarTagName(DV_DSDLLDATA* pData)
{
    CString rawVarName= VGvdvarname(pData->m_Pvdp);
    int totalLen      = rawVarName.GetLength();
    g_csVarName      = rawVarName.SpanExcluding(";");
    int realNameLen  = g_csVarName.GetLength();
    g_csTagBits      = rawVarName.Right(totalLen-realNameLen);

    pData->m_pszVarname      = g_csVarName.GetBuffer(0);
    pData->m_pszVartags      = g_csTagBits.GetBuffer(0);
}
```

Notice *g_csVarName* and *g_csTagBits* in the *GetVarTagName()* method. These are used to hold the name for display and the tagname information. They are declared at the beginning of the code:

```
static CString g_csVarName;
static CString g_csTagBits;
```

These string place holders are declared as static globals to ensure that their buffers and the strings in them persist until DV-Draw is finished with them.

Integrating Your Data Model into DV-Draw

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

In the integration stage, you must: Coordinate the DLL name and location with the corresponding variable in the registry.

If you want to run views using your data model, create a companion Run DLL to support your data model.

The first subsection below explains where DV-Draw looks for the *DSDLLNAME* variable, or registry key, and how you can set this registry key to make DV-Draw use your custom data source DLL. The second subsection discusses how to create a companion Run DLL.

Setting the Registry Key

When you install DV-Draw, the installation specifies the default data source DLL in the registry. In order to use your custom data source DLL, you must change the *DSDLLNAME* registry key value before you start DV-Draw. You can use your system's registry editor to do this. The *DSDLLNAME* registry key can be set in one of two places.

When you start DV-Draw, the executable checks the registry for the location of the data source DLL and loads that DLL. If no data source DLL is found, DV-Draw issues a message and the resulting instance of DV-Draw has no data-related functionality. No warning message appears. However, you can verify what DLL is in use by selecting the "About" option under the Help Menu. If no data source DLL is in use, the entry says "NO DATA COMPONENT LOADED."

By default, DV-Draw looks for the data source DLL specified by the following path in the registry:

```
HKEY_CURRENT_USER
  Software
    DataViews
      DV-Draw
        9.8
          General
            DSDLLNAME
```

If no DLL is found by looking under *HKEY_CURRENT_USER*, DV-Draw looks under *HKEY_LOCAL_MACHINE*, using the following path:

```
HKEY_LOCAL_MACHINE
  Software
    DataViews
      DV-Draw
        9.8
          General
            DSDLLNAME
```

The name of the default data source DLL is *DataSource.dll*. If the name is specified using an absolute path, that file is used. If the name is specified without any path information, DV-Draw uses *LoadLibrary* to search the directories specified by the search path.

When you install DV-Draw, the installation specifies the default data source DLL using *HKEY_LOCAL_MACHINE*, and does not specify anything under *HKEY_CURRENT_USER*. Therefore, if no DLL is specified for the current user, DV-Draw uses the DLL specified under *HKEY_LOCAL_MACHINE*. By explicitly setting *HKEY_CURRENT_USER*, you can specify different paths or different data source DLL names for different users.

Given these factors, there are numerous ways to set up the registry to make DV-Draw find the data source DLL you want it to use:

You can store your data source DLL in the same directory as the default DV-Draw data source DLL and change

the name of the data source DLL specified by the default path in the registry. For this method, your data source DLL must have a different name from the default data source DLL.

You can store your data source DLL in a separate directory and change the search path specified in the registry. The directory of the desired DLL should appear before the directory where the default DLL is located. For this method, your data source DLL should have the same name as the default data source DLL.

You can set up the *HKEY_CURRENT_USER* information so that specific users can use different DLLs. For this method, your data source DLL can have a different name from the default data source DLL and be in the same directory as the default data source DLL, or it can have any name in any other directory.

Editing the Run DLL

If your data model is significantly different from the default DV-Draw data model, and if you want to run views using your data model, you must create a companion Run DLL to support your data model. This release includes a sample of a companion Run DLL for the customized data source DLL, located in the *examples* directory under the *dvdraw* directory. The three related files are:

```
TestDlg.cpp
ExampleTestWin.cpp
TestChildWnd.cpp
```

The dialog file, *TestDlg.cpp*, builds the test window and provides the run functionality by handling the messages from the "Go," "Stop," and "Pause" buttons and the slider that controls the update rate.

ExampleTestWin.cpp provides the *CExampleTestWin* class. Like the *CDvDsDll* class, *CExampleTestWin* has a *Release()* method and a *GetIdInfo()* method. Instead of a *HandleRequest()* method, however, it has a *DvTestWin()* method:

```
class CExampleTestWin : public CTestWinDll
{
public:
    CExampleTestWin() {}
    ~CExampleTestWin() {}

    void Release();
    void GetIdInfo(CString& mfgName, CString& objName, CString& versionStr);
    int DvTestWin(VIEW view, RECTANGLE *wvp,
                 char *clut, CFrameWnd *parent);
};
```

In our example, the *DvTestWin()* method simply initializes the test window by verifying that the default resources are being loaded from this DLL and resetting them before displaying the test window:

```
int CExampleTestWin::DvTestWin(VIEW view,
                               RECTANGLE *wvp, char *clut,
                               CFrameWnd *parent)
{
    // Make sure the application's default resources
    // are loaded from this DLL...
    HINSTANCE hInstResourceClient = AfxGetResourceHandle();
    AfxSetResourceHandle(ExampleTestWinDLL.hModule);

    CTestDlg testDlg(view, wvp, clut, parent);

    // Reset the application's default resources.
    AfxSetResourceHandle(hInstResourceClient);

    return testDlg.DoModal();
}
```

```
}
```

ExampleTestWin.cpp also includes the C *GetInterface()* routine to instantiate the *CExampleTestWin* interface object for the Run DLL:

```
extern "C" {
DLLEXPORT BOOL
GetInterface(int idd, void **pInterf)
{
    *pInterf = 0;
    if (idd == IID_Run)
        *pInterf = new CExampleTestWin;
    return (pInterf) ? TRUE : FALSE;
}
```

This is identical to the data source DLL *GetInterface()* routine except that it uses *IID_Run* instead of *IID_Datasource* and instantiates a Run DLL interface object instead of a data source DLL interface object.

The *TestChildWnd.cpp* file is where the example Run DLL handles the data rebinding and display. It demonstrates how to traverse the variable descriptors and extract the human readable portion of the variable descriptor names for display in graph legends. This file is where your Run DLL should rebind and read data, but since the example data source DLL does not use real data, the example Run DLL can only simulate reading data.

The remaining code in *TestChildWnd.cpp* handles events and messages. Most of this code is straightforward and can probably be used as is in other Run DLLs. One message handler of interest is *OnInitDialog()*, which determines if the current data browser uses custom variable descriptor names, strips out tagname information so the human readable portion can appear in graph legends, and displays the view in the test window:

```
BOOL CTestChildWnd::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Check to see if the current data browser uses
    // custom vdp names. If so, we need to go through
    // all of the vdp's used in graphs and rename
    // them to their human readable form (which the data
    // browser will supply) so that they will make
    // sense in graph legends...
    CDvDsDll* pDsDll = 0;

    VDP_TRAVERSAL_DATA args;

    DVIGetInterface(IID_Datasource, (void**) &pDsDll);

    if(pDsDll)
    {
        DV_DSDLLDATA data;
        data.m_Context = DV_DSDLL_QUERYDLL;
        pDsDll->HandleRequest(&data, this);

        args.renameVdp = (BOOL) (data.m_Context & DV_DSDLL_USESCUSTOMTAGS);
        args.pDsDll = pDsDll;
    }
    else
    {
        args.renameVdp = FALSE;
        args.pDsDll = 0;
    }
}
```

```

OBJECT drawing = TviGetDrawing(m_View);
TobForEachVdp(drawing, CountVdps,
              (ADDRESS) &m_NumVdps);
if(m_NumVdps)
    m_pRebindArray = new float[m_NumVdps];

args.arrayIndex = 0;
args.rebindArray = m_pRebindArray;
TobForEachVdp(drawing, RenameRebindVdp,
              (ADDRESS) &args);

// Create a location object...used to make input
// objects respond...
m_Loc = VOloCreate();

// OK, now we can set up the DataViews screen and
// drawport. One noteworthy point here is that the
// interface to DV-Draw, besides including a cloned
// view of the view being currently edited, it also
// passed in a viewport rectangle if the user
// happens to be zoomed in, allowing you to run the
// view as the user currently sees it. If this is
// the desired behavior, then this is where it would
// be done...

m_Screen = TscOpenSet("W", m_pszClut,
                    V_WIN32_WINDOW_HANDLE, GetSafeHwnd(),
                    V_WIN32_DOUBLE_BUFFER, TRUE,
                    V_ACTIVE_CURSOR, V_END_OF_LIST );

m_Drawport = TdpCreateStretch(m_Screen, m_View, 0, 0);

M_runttype = MRUNNING;

TscErase(m_Screen);
TdpDraw(m_Drawport);

return TRUE; // Return TRUE unless you set
             // the focus to a control.
             // EXCEPTION: OCX Property Pages
             // should return FALSE.
}

```

As with the data source DLL, you must set the appropriate registry key in order to make DV-Draw use your custom Run DLL. As discussed in the previous section, DV-Draw looks for the run DLL specified by the following path:

```

HKEY_CURRENT_USER - or - HKEY_LOCAL_MACHINE
  Software
    DataViews
      DV-Draw
        9.8
          General
            RUNDLLNAME

```

Use the registry editor to set the *RUNDLLNAME* registry key under the *HKEY_CURRENT_USER* or *HKEY_LOCAL_MACHINE* path. For more information about where DV-Draw looks for the *RUNDLLNAME*

registry key and how you can set this registry key to make DV-Draw use your custom Run DLL, see [the Setting the Registry Key section](#).

Requirements

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

This section summarizes the various sets of requirements for coding a custom DLL. One set applies no matter what your data model is. One set applies only if your data model includes data source variables. One set applies only if your data model does *not* include data source variables.

In coding any custom DLL, you must meet the following requirements:

All code related to your data source DLL must be compiled with the Microsoft Visual C++ 5.0 compiler.

If you want to display data-related dialogs, you must provide them.

Dialogs must be modal.

You must determine where to store the information that tells the program where to find the data for each object.

You must provide a *CDvDsDll* class. It must include:

The *Release()* method

The *GetIdInfo()* method.

The *HandleRequest()* method.

You must provide the *GetInterface()* C routine.

For any of the contexts below that are supported, the *HandleRequest()* method must perform the following actions:

When you start DV-Draw, determine what data contexts are supported.

When you start DV-Draw or create a new view, perform any desired initialization, such as setting up a default data source.

When you create an input object, connect the variable descriptor to some persistent memory location of the appropriate type.

When you edit, select, or exchange a data assignment, the dialog must not allow the user to exit until a valid data selection is made or the user cancels the dialog.

If your data model uses data source variables:

You must create data source variables using DV-Tools routines in the code for the data source list dialog.

For any of the contexts below that are supported, the *HandleRequest()* method must perform the following actions:

When you save objects to a subview or cut and paste objects from one view to another, the secondary variable descriptor must be initialized before you perform the data reconnection.

The Context Flags

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

The context flags and feature flags are defined in *DVDSDLL.H*:

Source definition:

```
typedef enum DV-DSDLLCONTEXT {
// Context flags
DV_DSLL_INITNEWVIEW 0x // Initializing a new DV view.
DV_DSLL_EDITDS      0x // Editing data sources (read/write).
DV_DSLL_EDITDYNAMICS 0x // Editing object dynamics.
DV_DSLL_EDITGRAPH   0x // Creating/editing a graph.
DV_DSLL_EDITINPUT   0x // Creating/editing an input object.
DV_DSLL_EDITRULES   0x // Creating/editing rules.
DV_DSLL_DEFAULTDATA 0x // Request for a default data connection.
DV_DSLL_SELECTDATA  0x // Request for a data connection selection.
DV_DSLL_EXCHANGEDATA 0x // Request for a data selection and exchange.
DV_DSLL_RECONNECTDATA 0x // For cut/paste-like operations.
// Query flags
DV_DSLL_QUERYDLL    0x // Query for the DLL's context and feature
                        support.
DV_DSLL_GETNAMEBITS 0x // Request for the variable's name and tag
                        fields.

// Feature flags
DV_DSLL_USESDSVARS  0x // DLL supports/uses data source variables.
DV_DSLL_USESCUSTOMTAGS 0x // DLL uses/embeds custom tag names with
                        variable.
}

```

These additional flags are also defined in *DVDSDLL.H*:

```
typedef struct tagDSDLLDATA {
VIEW m_Pview; // Primary view.
VIEW m_Sviews // Secondary view.
VARDESC m_Pvdp; // Primary variable descriptor.
VARDESC m_Svdp; // Secondary variable descriptor.
DSVAR m_DsVar; // A data source variable.
char* m_pszVarname; // Variable descriptor name.
char* m_pszVartags; // Variable descriptor tags (optional).
OBJECT m_Ob; // Selected object(s).
m_VarType //
OBJECT m_DynControl; // Dynamic control object.
int m_DynType; // Specific dynamic
BOOL m_Dirty; //
DV_DSDLLCONTEXT m_Context; // Context flags.
} DV_DSDLLDATA;

```

Prototyping
DV-Draw User's Guide (new sections)

[Introduction](#)

[Designing a Prototype](#)

[Planning an Interface](#)

[Organizing the Prototype Directory](#)

[Prototyping and DVTools](#)

This document discusses:

Designing a Prototype

Planning an Interface

Organizing the Prototype Directory

Why You Still Need DV-Tools

Introduction

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Prototyping lets you test the logical connections between your views by sequencing through them. You can use this screen sequencing capability to display sequences of views as if they were slides in a slide show. You can run a prototype at any time during the design process. This document describes how to simulate your application. Rule editing and the DV-Draw editing features are described in the *Editing Rules* document. This document assumes that you are already familiar with creating views and adding rules to your objects in DV-Draw.

Designing a Prototype

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

There are five basic steps to creating and using a prototype. They are: Planning your interface and its behavior.

Behavior elements include the display sequence of views, overlays, and popups, and the events and conditions which cause the interface to change.

Organizing a prototype directory and subdirectories. Do this before creating your views.

Creating your prototype views by adding rules in DV-Draw.

Setting the prototype environment.

Running the prototype.

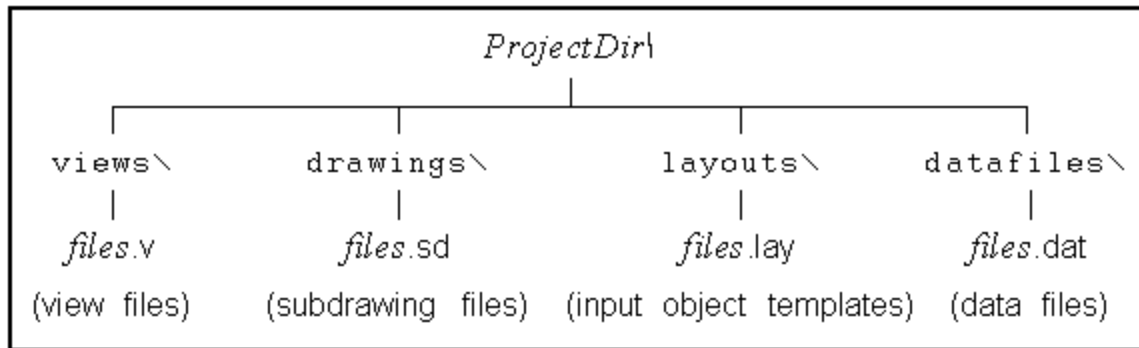
This document discusses planning the interface, organizing the directory, and setting the prototype environment.

Editing the prototype rules in DV-Draw and running the prototype are described in the *Editing Rules* document.

Planning an Interface

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

In the conceptual plan of a graphical interface, you need to organize your information in a logical hierarchical structure. Your views and the transitions between them should parallel this structure, making information access as intuitive as possible. Consider how much information can be displayed and absorbed in a single screen. Each view should contain only the elements needed to make the information clear and easy to understand, and how to indicate what additional information is available at other levels. To associate the available information with visible objects, use the prototype rules. For each rule, determine what *event* triggers each *action* that displays new information, and the *condition* under which the action occurs. For example, when you pick (event) on the state of West Virginia (object), if the temperature is below 30 degrees (condition), a snowfall map replaces the view (action), as illustrated below.



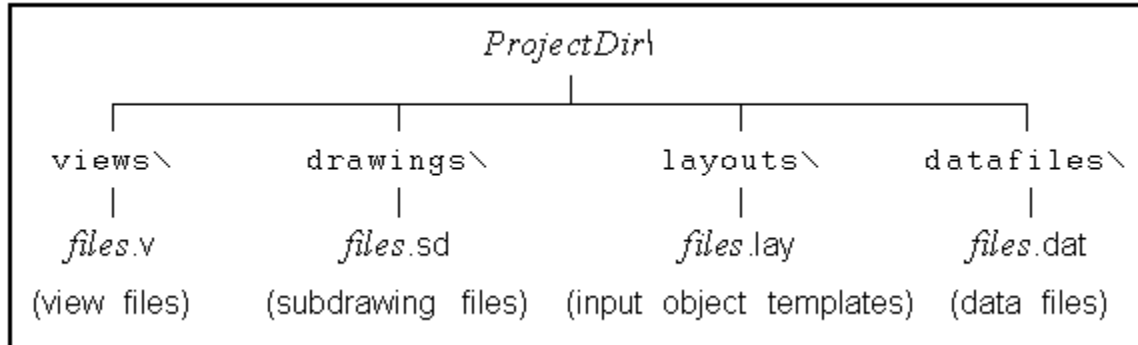
Example Portion of an Interface Design

Organizing the Prototype Directory

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

Before creating the views for your prototype: Set up its working directory.

Each prototype should have its own working directory to avoid conflict with other projects. Use subdirectories and filenames with consistent extensions. The following directory structure is recommended:



A Prototype Directory Structure

References to subdrawings and views should *not* include directory names.

Use the *DVPATH* variable in your configuration file to add the project's subdirectories to the search path.

For example, instead of `<dataviews>\lib\isa\motor`, just enter `motor` and add `<dataviews>\lib\isa\` to *PATH*. This is especially important when you ship the prototype to your customer.

To produce a prototype based on your design:

Create your views and add rules to your objects as described in the *Editing Rules* document.

After adding rules to your views, you can run your prototype either in the test window of DV-Draw or by using the *DVproto* script outside DV-Draw. For instructions on running your prototype, see the *Editing Rules* document.

Prototyping and DV-Tools

DV-Draw User's Guide (new sections) [Chapter Table of Contents](#)

You can activate a prototype from inside a DV-Tools application. This lets you use your prototype intact, with all rules and prototyping behavior active, within a DV-Tools application. This does not, however, let you change or add to the prototype's functionality from within the application. Your prototype is only a simulation of a complete application. To create a complete application, you must use DV-Tools. DV-Tools provides the following features and functionalities that are not provided in DV-Draw:

- Full application development.
- Full runtime.
- Integration with window systems; use multiple windows.
- Connection to your application data.
- Sequencing driven by your application data.
- Control over your real data acquisition; control over the rate of data acquisition.
- Ability to define a wider range of events, conditions, and resulting actions.
- Use of complex input objects (combiner, multiplexor).
- Use of exclusive-and conditions.
- Fully dynamic views within views; more flexible control of multiple views.
- Better message-handling.
- Custom labeling of graph axes.
- Writing new graphs with boilerplate provided.
- Ability to activate and deactivate input objects (in case of overlap).
- Use of button-up and button-down events.
- Rubberbanding and dragging.

For more information on using prototypes with your DV-Tools applications, see the *DV-Tools Reference Manual*.

