T Routines

High level routines for managing, loading, displaying, and updating views. The following *#include* files are necessary for using the T level routines.

```
#include "std.h"
#include "dvstd.h"
#include "dvtools.h"
#include "dvGR.h"
#include "Tfundecl.h"
```

TInit, TTerminate	Performs the necessary initialization and clean-up for DV-Tools.
Tdl	Manages data source lists (*dl*).
Tdp	Manages drawports.
Tdr	Drawing access functions.
Tds	Manages data sources (*ds*).
Tdsv	Manages data source variables (*dsv*).
Tlo	Manages location objects.
Tob	Access functions that work on objects that have subobjects.
Tproto	Displays prototypes created in DV-Draw.
Tsc	T level routines for managing screen objects (*sc*).
Tvd	Accesses the display variables associated with drawing objects.
Tvi	View access functions.

TInit and TTerminate

T Routines

Performs the necessary initialization and clean-up for DV-Tools.

Tinit & Tterminate

TInit	Performs the initialization for DV-Tools.
Tterminate	Performs the clean-up for DV-Tools.

TInit

Performs the initialization for DV-Tools.

```
BOOLPARAM
TInit (
       char *search_path,
       char *format_spec_file)
```

TInit performs the initialization for DV-Tools. *TInit* should be the first DV-Tools routine called by your program. *TInit* reads your configuration file and any environment variables or logical names that are set. It also sets the initial heap size for all specified *DVxxINITIALHEAPSIZE* configuration variables.

The first parameter sets the *search_path*, which is the list of directories that are searched for all files, such as view files, data files, and processes. *search_path* is a string of directory paths, separated by spaces, to be checked in order from left to right. The current directory is always searched first. If *search_path* is *NULL*, the value of the environment variable or logical name *DVPATH* is used. If neither *search_path* nor *DVPATH* is set, the search path in the configuration file is used.

The second parameter specifies which format specification file to use. The format specification file, *format_spec_file*, contains information necessary to display graphs. Usually *format_spec_file* is *NULL*, and the default file, *dispforms.stb*, is used. If you do not have *dispforms.stb* in your path and try to run a display formatter, nothing happens. If there are no graphs in your display, you don't need *dispforms.stb*. However, the search path should include *dispforms.stb* so that if you add graphs to the drawing, your display runs correctly.

You can use your own version of *dispforms.stb* to change the number of display formatters, to include your own display formatters, and to rearrange the order of the display formatters as they appear in DV-Draw. To change the number of display formatters, you must add or delete entries in the table found in *ToolNames.c*. For a detailed description of each display formatter, see the *VD Routines* chapter in this manual.

TInit returns *DV_FAILURE* if *format_spec_file* is not provided or if the default file, *dispforms.stb*, can't be found. Otherwise returns *DV_SUCCESS*. *TInit* only executes once within an application. Subsequent calls to *TInit* do nothing, but still return *DV_SUCCESS*.

TTerminate

Performs the clean-up for DV-Tools.

```
BOOLPARAM
TTerminate (
       void)
```

TTerminate performs any clean-up required at the end of a program that uses DV-Tools subroutines. It should be the last DV-Tools subroutine called by your program. Always returns *DV_SUCCESS*.

Tdl (Tdatasourcelist)

Tdl Functions T Routines

Managesdata source lists (*dl*). Data source lists are DataViews private types that maintain lists of data sources (*ds*). Data source lists can belong to one or more views (*vi*), so they maintain reference counts to avoid unexpectedly being destroyed when their views are destroyed.

Tdl Functions

TdlAddDataSource

Tdl Functions T Routines

Adds a data source to the data source list.

```
BOOLPARAM
TdlAddDataSource (
       DATASOURCELIST dsl,
       DATASOURCE ds,
       DATASOURCE ds_reference)
```

TdlAddDataSource adds a data source, *ds*, to the data source list, *dsl*. Adds *ds* before the referenced data source, *ds_reference*. If *ds_reference* is *NULL*, then *ds* is added at the end of the list. Returns *DV_FAILURE* if *dsl*, *ds*, or *ds_reference* are invalid, or if *ds_reference* is not in the *dsl*. Otherwise returns *DV_SUCCESS*.

TdlClone

Tdl Functions T Routines

Copies a data source list.

```
DATASOURCELIST
TdlClone (
       DATASOURCELIST dsl)
```

TdlClone creates and returns a deep copy of a data source list, *dsl*. This routine does not clone bindings between data source variables and the variable descriptors of dynamic objects. Returns *DV_FAILURE* if it is passed an invalid *dsl*.

TdlCloseData

Tdl Functions T Routines

Closes all files and processes.

```
BOOLPARAM
TdlCloseData (
       DATASOURCELIST dsl)
```

TdlCloseData closes all files and processes referenced by every data source in the data source list, *dsl*. Returns *DV_FAILURE* if it is passed an invalid *dsl*. Otherwise returns *DV_SUCCESS*.

TdlCreate

Tdl Functions T Routines

Creates and returns an empty data source list.

```
DATASOURCELIST
TdlCreate (void)
```

TdlDeleteDataSource

Tdl Functions T Routines

Deletes a data source from the data source list.

```
BOOLPARAM
TdlDeleteDataSource (
       DATASOURCELIST dsl,
       DATASOURCE ds)
```

TdlDeleteDataSource removes a data source, *ds*, from the data source list, *dsl*. Returns *DV_FAILURE* if *ds* or *dsl* is invalid. Otherwise returns *DV_SUCCESS*.

TdlDestroy

Tdl Functions T Routines

Conditionally destroys a data source list.

```
int
TdlDestroy (
       DATASOURCELIST dsl)
```

TdlDestroy conditionally destroys a data source list, *dsl*. The reference count is decremented by one and *dsl* is deallocated only if its reference count falls to zero. Otherwise, it is assumed that other views still point to it and no action is taken. The reference count for a data source list is incremented only by a call to *TviMergeAddDataSources* or *TviMergeDataSources*.

Returns the new reference count of *dsl*. If the reference count is zero and no dynamic objects are bound to data sources in the list, destroys *dsl* and returns *0*. If the reference count is zero and *dsl* contains data sources that are still bound to dynamic objects, returns *-1* to indicate the error condition.

If the data source list being destroyed was attached to a view, you must make a subsequent call to *TviPutDataSourceList* to substitute another data source list or a *NULL* data source list in place of the one destroyed.

TdlForEachDataSource

Tdl Functions T Routines

Traverses all data sources.

```
ADDRESS
TdlForEachDataSource (
       DATASOURCELIST dsl,
       TDLFOREACHDSFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           DATASOURCE datasource,
           ADDRESS argblock)
```

TdlForEachDataSource traverses all data sources in the data source list, *dsl*, and calls *fun* for each data source. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the data source being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each data source in *dsl*, or
2. to find a particular data source in *dsl*.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the data source is not found. Otherwise it should return the data source for *ADDRESS*.

Note: You should not alter the list by adding, deleting, or reordering the data sources during traversal.

For an example of a typical function, see the example under *TdrForEachNamedObject*. Note that the example demonstrates the use of a function with three parameters, but *TdlForEachDataSource* requires only two.

TdlForEachVar

Tdl Functions T Routines

Traverses all data source variables.

```
ADDRESS
TdlForEachVar (
       DATASOURCELIST dsl,
       TDLFOREACHDSVFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           DATASOURCE datasource,
           DSVAR dsvar,
           ADDRESS argblock)
```

TdlForEachVar traverses all data source variables in the data source list, *dsl*, and calls *fun* for each data source variable. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*. For a description of *fun*, see *TdlForEachDataSource*. Note that *TdlForEachDataSource* traverses data sources, passing two parameters to *fun*. *TdlForEachVar* traverses data source variables, passing three parameters to *fun*: the data source, the data source variable, and the argument block.

TdlGetNamedDataSource

Tdl Functions T Routines

Gets a named data source from a data source list.

```
DATASOURCE
TdlGetNamedDataSource (
       DATASOURCELIST dsl,
       char *name)
```

TdlGetNamedDataSource gets and returns the first data source with the passed name, *name*. Returns *DV_FAILURE* if it is passed an invalid data source list, *dsl*.

TdlLoad

Tdl Functions T Routines

Loads a data source list.

```
DATASOURCELIST
TdlLoad (
       char *filename)
```

TdlLoad loads a data source list from a file, *filename*. Returns *DV_FAILURE* if the file cannot be opened, or if the loaded file does not contain a valid data source list.

TdlOpenData

Tdl Functions T Routines

Opens all files and processes.

```
BOOLPARAM
TdlOpenData (
       DATASOURCELIST dsl)
```

TdlOpenData opens all files and processes referenced by every data source in the data source list, *dsl*. Returns *DV_FAILURE* if any data sources in *dsl* could not be opened. Otherwise returns *DV_SUCCESS*.

TdlReadData

Tdl Functions T Routines

Reads all data for one iteration.

```
int
TdlReadData (
       DATASOURCELIST dsl)
```

TdlReadData reads one iteration of data for each file and process in the data source list, *dsl*. Returns the number of data sources that have reached the end of the file.

TdlSave

Tdl Functions T Routines

Saves a data source list.

```
BOOLPARAM
TdlSave (
       DATASOURCELIST dsl,
       char *filename,
       int access_mode)
```

TdlSave saves a data source list, *dsl*, to a file, *filename*, using *access_mode*. *access_mode* should be *WRITE_EXPANDED* for ASCII write, or *WRITE_COMPACT* for binary write. Flag values are defined in *VOstd.h*. Returns *DV_FAILURE* if *dsl* is invalid or the file can't be opened. Otherwise returns *DV_SUCCESS*.

TdlValid

Tdl Functions T Routines

Determines if a data source list is valid.

```
BOOLPARAM
TdlValid (
       DATASOURCELIST dsl)
```

TdlValid returns *DV_SUCCESS* if the data source list is valid. Otherwise returns *DV_FAILURE*.

Tdl Functions

TdlAddDataSource	Adds a data source to the data source list.
TdlClone	Copies a data source list.
TdlCloseData	Closes all files and processes.
TdlCreate	Creates an empty data source list.
TdlDeleteDataSource	Deletes a data source from the data source list.
TdlDestroy	Conditionally destroys a data source list.
TdlForEachDataSource	Traverses all data sources.
TdlForEachVar	Traverses all data source variables.
TdlGetNamedDataSource	Gets a named data source from a data source list.
TdlLoad	Loads a data source list.
TdlOpenData	Opens all files and processes.
TdlReadData	Reads all data for one iteration.
TdlSave	Saves a data source list.
TdlValid	Determines if a data source list is valid.

Tdp (Tdrawport)

Tdp Functions T Routines

Manages drawports.

A drawport (*dp*) is a DataViews private structure that contains all the information needed to display a view on a screen. How the view appears is specified by two boundary viewport rectangles contained in the drawport structure: a drawing viewport, specified in world coordinates, that describes the portion of the view to be displayed in the drawport; and a screen viewport, specified in virtual coordinates, that describes the portion of the screen where the view is to be displayed. The drawport also contains transform objects which hold the world-to-screen and screen-to-world coordinate transformation mapping, and information about obscuring drawports to determine clipping. Drawports belong to screen objects. Each screen object maintains an ordered visibility list of its drawports that determines which drawports are on top. A drawport takes up a specific amount of screen real estate and obscures other drawports below it. When a drawport is created, it is placed at the top of the visibility list for its screen. Every drawport contains a pointer to a view and to its own screen object. The screen object represents the device, or window, on which the view is displayed.

Dynamic objects maintain drawport-specific information, so they can only be drawn in one drawport at a time. To draw a dynamic object or a view with dynamics in more than one drawport at a time, clone it first and use the copy in the other drawport. To draw the same dynamic view in different drawports at different times, destroy (or erase) the previous drawport before creating (or drawing) the new drawport. In this case, the view does not need to be cloned.

TdpDraw handles the initial drawing of a drawport. As a result, it initializes data buffers for graphs and input objects. TdpDrawObject is the analogous routine that handles the initial drawing and data buffer initialization for an individual object drawn in a drawport. Other drawing routines, such as TdpDrawNext, TdpRedraw, and TdpErase, are not effective until the drawport is drawn using TdpDraw.

TdpDrawNext or TdpRedrawNext updates the dynamics in the drawport; TdpDrawNextObject updates the dynamics for one object in the drawport. TdpDrawNextchecks all objects in the drawport to determine which are affected by the update and only redraws those objects. TdpRedrawNext redraws all objects in the drawport, whether or not they are affected by the update. Depending on your application, either TdpDrawNext or TdpRedrawNext may be faster. For drawports with many objects (more than several hundred) and many dynamic objects, TdpRedrawNext is usually faster. For drawports with fewer objects and few dynamic objects, TdpDrawNext might be faster. Try each method to determine which is more efficient for your application.

TdpRedraw redraws a drawport after operations such as resizing or zooming. For example, TdpBack, TdpFront, TdpPan, TdpResize, TdpZoom, and TdpZoomTo must be followed by a call to TdpRedraw. TdpRedrawObject is the corresponding routine that redraws an individual object that was drawn using TdpDrawObject.

TdpErase erases the drawport and clears the data buffers for graphs and input objects. It is the opposite of TdpDraw, which draws the drawport and initializes the data buffer. TdpEraseObjectis the analogous routine that erases and clears the data buffers for an individual object drawn into the drawport. To draw the drawport or object again after erasing, you must call TdpDraw or TdpDrawObject, not TdpRedraw or TdpRedrawObject.

TdpDraw, TdpDrawNext, TdpRedrawNext, TdpDrawNextObject, TdpDrawObject, TdpErase, and TdpEraseObject change the value of the current screen, which is an internal global variable, to the drawport's screen.

Tdp Functions

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	Tsc
Tdp	Tlo	Tvd
Tdr	Tob	Tvi

Tdp Functions

TdpBack	Moves a drawport to the back of the visibility list.
TdpCreate	Createsa new drawport.
TdpCreateStretch	Creates a new drawport with stretched coordinates.
TdpDestroy	Destroys a drawport structure.
TdpDraw	Draws the contents of a drawport.
TdpDrawNext	Updates all dynamic objects within a drawport's view.
TdpDrawNextObject	Updates a specific dynamic object within a drawport.
TdpDrawObject	Draws a specific object within a drawport.
TdpErase	Erases the contents of a drawport.
TdpEraseObject	Erases an object within a drawport.
TdpForEachDrawport	Applies a function to all drawports, in all screens.
TdpFront	Moves a drawport to the front of the visibility list.
TdpGetDrawingVp	Gets the drawing viewport rectangle of a drawport.
TdpGetScale	Gets the scale factor of a drawport.
TdpGetScreen	Gets the screen object of a drawport.
TdpGetScreenVp	Gets the screen viewport rectangle of a drawport.
TdpGetView	Gets the view of a drawport.
TdpGetXform	Gets one of the drawport's transformation objects.
TdpIsDrawn	Determines if a drawport has been drawn.
TdpMaskPlanes	Sets the write mask for a drawport.
TdpObsvpGet	Returns a list of obscuring viewports.
TdpPan	Pans a view within its drawport.
TdpRedraw	Redraws a portion of the drawport.
TdpRedrawNext	Updates all dynamic objects and redraws the contents of the drawport.
TdpRedrawObject	Redraws an object in the drawport.
TdpResize	Changes the size and position of a drawport.
TdpScreenToWorld	Converts a point from screen to world coordinates.
TdpWorldToScreen	Converts a point from world to screen coordinates.
TdpZoom	Scales a view within its drawport.
TdpZoomTo	Scales and pans a view within its drawport.

TdpBack

Tdp Functions T Routines

Moves a drawport to the back of the visibility list.

```
BOOLPARAM
TdpBack (
        DRAWPORT dp)
```

TdpBack moves the drawport, *dp*, to the back of the visibility list for the drawport's screen. Does not redraw the drawport. Must be followed by a call to TdpRedraw in order for it effects to be visible. Returns *NO* if it is passed an invalid *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpCreate

Tdp Functions T Routines

Creates a new drawport.

```
DRAWPORT
TdpCreate (
       OBJECT screen,
       VIEW view,
       RECTANGLE *vvp_screen,
       RECTANGLE *wvp_drawing)
```

TdpCreate creates and returns a drawport. The drawport is attached to the screen object specified by *screen*, and is added to its drawport visibility list. *screen* should have been previously created by a call to TscOpenSet. If the *screen* argument is *NULL*, the current screen is used. *view* specifies the view to be displayed on the screen. *wvp_drawing* is the drawing viewport and specifies what part of the view is to be drawn on the screen. It is expressed in world coordinates (-16K to 16K). The *vvp_screen* parameter is the screen viewport and specifies where on the screen that the view is to be displayed. It is expressed in virtual coordinates (0 to 32K).

The *wvp_drawing* and *vvp_screen* viewports define the world-to-screen coordinate transformation of the drawport. It is best to make sure that the aspect ratio of these two viewports are approximately equal. If the aspect ratio of these two viewports is different, TdpCreate uses a best fit algorithm to preserve the aspect ratio of the view. The view is shrunk until it is small enough to fit inside the screen viewport. This leaves extra space on the sides or on the top of the screen viewport.

If *vvp_screen* is *NULL*, the whole screen is used. If *wvp_drawing* is *NULL*, the drawing viewport has the same aspect ratio as the screen viewport and the origin of the view is centered in the screen viewport. A view can have a preferred scale set in DV-Draw or using VOdrSetScale. If there is a preferred scale, the portion of the view that fits is drawn to the scale within the screen viewport. Otherwise, the view is expanded equally in each dimension until it fills the screen viewport. In this case, either the top and bottom or the sides of the view may not be visible.

If both *vvp_screen* and *wvp_drawing* are *NULL*, no preferred scale has been set, and the application is being run in a window or device with the same aspect ratio in which DV-Draw was run when the view was created, the drawport displays the view exactly as it appeared in DV-Draw. TdpCreate was formerly called *TdpSetupDraw*. Returns *DV_FAILURE* if it is passed an invalid *view*.

TdpCreateStretch

Tdp Functions T Routines

Creates a new drawport with stretched coordinates.

```
DRAWPORT
TdpCreateStretch (
       OBJECT screen,
       VIEW view,
       RECTANGLE *vvp_screen,
       RECTANGLE *wvp_drawing)
```

TdpCreateStretch creates and returns a drawport in which the dimensions of the objects in the view are stretched to make the portion of the view specified by *wvp_drawing* exactly fit in the specified screen viewport, *vvp_screen*. Stretching transforms the object's control points differently in the x and y dimensions. Therefore, stretched arcs and circles may change their sizes relative to other object types. If *wvp_drawing* is *NULL TdpCreateStretch* is equivalent to TdpCreate in that it preserves the aspect ratio of *view*. Returns *DV_FAILURE* if it is passed an invalid *view*.

TdpDestroy

Tdp Functions T Routines

Destroys a drawport structure.

```
BOOLPARAM
TdpDestroy (
       DRAWPORT dp)
```

TdpDestroy destroys the drawport structure, *dp*, removing it from its screen's visibility list and freeing the allocated memory. Returns *DV_FAILURE* if it is passed an invalid *dp*. Otherwise returns *DV_SUCCESS*. Formerly called *TdpFree*.

TdpDraw

Tdp Functions T Routines

Draws the contents of a drawport.

```
BOOLPARAM
TdpDraw (
        DRAWPORT dp)
```

TdpDraw draws the drawport's view on its screen, moving the drawport, *dp*, to the front of the drawport visibility list. Returns *NO* if it is passed an invalid *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpDrawNext

Tdp Functions T Routines

Updates all dynamic objects within a drawport's view.

```
BOOLPARAM
TdpDrawNext (
       DRAWPORT dp)
```

TdpDrawNext updates dynamic objects in the drawport when the values of their variable descriptors change. Objects that use visibility dynamics are only redrawn if they are visible. Updates graphs each time it is called, but only updates other dynamic objects if their data changes. Objects that have visibility dynamics may become invisible when their data changes. In this case, *TdpDrawNext* uses the erase method specified by the dynamic control object and only redraws the affected portions of the drawport. Note that TdpDraw must be called first in order for this routine to work. Returns *NO* if it is passed an undrawn drawport. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpDrawNextObject

Tdp Functions T Routines

Updates a specific dynamic object within a drawport.

```
BOOLPARAM
TdpDrawNextObject (
       DRAWPORT dp,
       OBJECT object)
```

TdpDrawNextObject updates the object, *object*, in the drawport, *dp*, when the values of their variable descriptors change. Objects that use visibility dynamics are only redrawn if they are visible. Updates graphs each time it is called, but only updates other dynamic objects if their data changes. Objects that have visibility dynamics may become invisible when their data changes. Note that TdpDraw or TdpDrawObject must be called first in order for this routine to work. Returns *NO* if it is passed an undrawn *dp*. For more information, see the introduction to this module.

TdpDrawObject

Tdp Functions T Routines

Draws a specific object within a drawport.

```
BOOLPARAM
TdpDrawObject (
       DRAWPORT dp,
       OBJECT object)
```

TdpDrawObject draws the specified object, *object*, in the drawport, *dp* if the object is currently visible. Note that TdpDraw must be called first in order for this routine to work. Returns *NO* if it is passed an undrawn *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpErase

Tdp Functions T Routines

Erases the contents of a drawport.

```
BOOLPARAM
TdpErase (
       DRAWPORT dp)
```

TdpErase erases the drawport, *dp*, by filling it with the background color of its view. Note that TdpDraw must be called first. Returns *NO* if it is passed an undrawn *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpEraseObject

Tdp Functions T Routines

Erases an object within a drawport.

```
BOOLPARAM
TdpEraseObject (
       DRAWPORT dp,
       OBJECT object)
```

TdpEraseObject erases the object, *object*, in the drawport, *dp*, by drawing the object in the background color of the drawing. Note that TdpDraw must be called first in order for this routine to work and that erasing an object does not remove it from the drawing. Objects behind the erased object are not redrawn, except for input objects, where the background is controlled by flag settings. When used to erase a dynamic object, TdpEraseObject clears the dynamic object's data buffer. To draw the object again, you must call TdpDrawObject. Returns *NO* if it is passed an undrawn *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpForEachDrawport

Tdp Functions T Routines

Applies a function to all drawports, in all screens.

```
ADDRESS
TdpForEachDrawport (
       TDPTRAVERSEFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           DRAWPORT dp,
           ADDRESS argblock)
```

TdpForEachDrawport traverses all the drawports on the current screen and calls the function, *fun,* for each drawport, *dp*. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the drawport being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each drawport, or
2. to find a particular drawport.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the drawport is not found. Otherwise it should return the drawport for *ADDRESS*.

Note: You should not alter the drawport list by adding, deleting, or reordering drawports during traversal.

For an example of a typical function, see the example under *TdrForEachNamedObject*. Note that the example demonstrates the use of a function with three parameters, but *TdpForEachDrawport* requires only two.

TdpFront

Tdp Functions T Routines

Moves a drawport to the front of the visibility list.

```
BOOLPARAM
TdpFront (
       DRAWPORT dp)
```

TdpFront moves the drawport, *dp*, to the front of the visibility list for the drawport's screen. Does not erase or redraw any drawports. Returns *NO* if it is passed an invalid *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpGetDrawingVp

Tdp Functions T Routines

Gets the drawing viewport rectangle of a drawport.

```
RECTANGLE *
TdpGetDrawingVp (
       DRAWPORT dp)
```

TdpGetDrawingVp returns a pointer to the drawing viewport rectangle of the drawport, *dp*, specified in world coordinates (-16k,+16k). Before TdpDraw is called, this routine simply returns the drawing viewport parameter used in the drawport creation call. When TdpDraw is called, the drawing viewport may be adjusted to fit the screen viewport, so more of the drawing shows than intended. *TdpGetDrawingVp* returns the intended drawing viewport, not the actual visible portion of the drawing, which can change when the aspect ratio of the screen changes.

For the case where TdpCreate is called with a *NULL* drawing viewport, TdpDraw calculates a “best fit” drawing viewport that is usually less than the entire world coordinates. This best fit drawing viewport becomes the intended drawing viewport.

If the drawport is zoomed out so that the off-drawing area is visible, the returned rectangle represents the entire visible area as if in world coordinates. In this case, one or more coordinates of the rectangle will be outside the world coordinate range. Returns *DV_FAILURE* if it is passed an invalid *dp*.

TdpGetScale

Tdp Functions T Routines

Gets the scale factor of a drawport.

```
double
TdpGetScale (
       DRAWPORT dp)
```

TdpGetScale returns the scale factor of the drawport. The scale factor maps a unit world coordinate to screen coordinates. Returns *DV_FAILURE* if it is passed an invalid drawport.

TdpGetScreen

Tdp Functions T Routines

Gets the screen object of a drawport.

```
OBJECT
TdpGetScreen (
       DRAWPORT dp)
```

TdpGetScreen returns the screen object to which the drawport, *dp*, is attached. Returns *DV_FAILURE* if it is passed an invalid *dp*.

TdpGetScreenVp

Tdp Functions T Routines

Gets the screen viewport rectangle of a drawport.

```
RECTANGLE *
TdpGetScreenVp (
       DRAWPORT dp)
```

TdpGetScreenVp returns a pointer to the screen viewport rectangle of the drawport, *dp*, specified in virtual coordinates (0-32k). Returns *DV_FAILURE* if it is passed an invalid *dp*.

TdpGetView

Tdp Functions T Routines

Gets the view of a drawport.

```
VIEW
TdpGetView (
       DRAWPORT dp)
```

TdpGetView returns the view belonging to the drawport, *dp*. Returns *DV_FAILURE* if it is passed an invalid *dp*.

TdpGetXform

Tdp Functions T Routines

Gets one of the drawport's transformation objects.

```
OBJECT
TdpGetXform (
       DRAWPORT dp,
       int flag)
```

TdpGetXform returns either one of the drawport's transformations depending on *flag*. See also *VOxform*. Valid flags are:

DR_TO_SCREEN	drawing to screen xform
SCREEN_TO_DR	screen to drawing xform

Returns *DV_FAILURE* if it is passed an invalid *dp* or *flag*.

TdpIsDrawn

Tdp Functions T Routines

Determines if a drawport has been drawn.

```
BOOLPARAM
TdpIsDrawn (
       DRAWPORT dp)
```

TdpIsDrawn determines whether the drawport, *dp*, has been drawn. Returns *YES* or *NO*. Returns *NO* if it is passed an invalid *dp*.

TdpMaskPlanes

Tdp Functions T Routines

Sets the write mask for a drawport.

```
LONG
TdpMaskPlanes (
       DRAWPORT drawport,
       LONG mask)
```

TdpMaskPlanes sets the write mask used for all *Tdp* drawing and erasing operations and *TscRedraw*. This routine lets you set up write masks for planemasking on a drawport-by-drawport basis.

By default, the drawport write mask is *0*. This makes the write mask specified by GRmaskplanes, if any, effective for the drawport. If GRmaskplanes also has not been called to set a write mask, the default condition is no masking. To turn off the mask specified by a previous call to TdpMaskPlanes, set *mask* to *0*.

Returns the previous write mask.

TdpObsvpGet

Tdp Functions T Routines

Returns a list of obscuring viewports.

```
RECTANGLE **
TdpObsvpGet (
       DRAWPORT dp)
```

TdpObsvpGet returns a pointer to a *NULL*-terminated array of viewports, in screen coordinates, that obscure the drawport, *dp*.

TdpPan

Tdp Functions T Routines

Pans a view within its drawport.

```
BOOLPARAM
TdpPan (
       DRAWPORT dp,
       DV_POINT *wpt_center)
```

TdpPan pans a view within its drawport, *dp. wpt_center* specifies a world coordinate point in the view's drawing to be brought to the center of the drawport. Does not erase or redraw any drawports. Returns *NO* if it is passed an invalid *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpRedraw

Tdp Functions T Routines

Redraws a portion of the drawport.

```
BOOLPARAM
TdpRedraw (
       DRAWPORT dp,
       RECTANGLE *svp,
       int erase_flag)
```

TdpRedraw redraws the portion of the drawport, *dp*, specified by the screen coordinate rectangle, *svp*. Only that portion of the rectangle within the drawport boundary is redrawn. If *svp* is *NULL*, the entire drawport is redrawn. If *erase_flag* is *YES*, the specified portion of *dp* is erased before being redrawn. Objects that were drawn using TdpDrawObject are not redrawn; for these objects, use TdpRedrawObject.

TdpRedrawNext

Tdp Functions T Routines

Updates all dynamic objects and redraws the contents of the drawport.

```
BOOLPARAM
TdpRedrawNext (
       DRAWPORT drawport)
```

TdpRedrawNext is the same as *TdpDrawNext* except it does not use the erase method specified by the dynamic control object. Instead, *TdpRedrawNext* redraws the whole drawport. Note that TdpDrawmust be called first in order for this routine to work. Returns *NO* if it is passed an undrawn drawport. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpRedrawObject

Tdp Functions T Routines

Redraws an object in the drawport.

```
BOOLPARAM
TdpRedrawObject (
        DRAWPORT dp,
        OBJECT object)
```

TdpRedrawObject redraws an object, *object*, that was drawn using *TdpDrawObject*. The object must be currently visible. Returns *NO* if it is passed an undrawn *dp*. Otherwise returns *YES*.

TdpResize

Tdp Functions T Routines

Changes the size and position of a drawport.

```
BOOLPARAM
TdpResize (
       DRAWPORT dp,
       RECTANGLE *vvp_screen)
```

TdpResize changes the screen viewport rectangle of the drawport, *dp*. The new screen viewport is specified in virtual coordinates by the rectangle parameter, *vvp_screen*. Does not erase or redraw any drawports. Returns *NO* if it is passed an invalid *dp*. Otherwise returns *YES*. For more information, see the introduction to this module.

TdpScreenToWorld

Tdp Functions T Routines

Converts a point from screen to world coordinates.

```
BOOLPARAM
TdpScreenToWorld (
       DRAWPORT dp,
       DV_POINT *spt,
       DV_POINT *wpt)
```

TdpScreenToWorld converts a point in screen coordinates, *spt*, to world coordinates, *wpt*, according to the screen-to-world coordinate transform of the drawport, *dp*. The points are represented as *DV_POINT* structures. Returns *DV_FAILURE* if it is passed an invalid *dp*. Otherwise returns *DV_SUCCESS*.

TdpWorldToScreen

Tdp Functions T Routines

Converts a point from world to screen coordinates.

```
BOOLPARAM
TdpWorldToScreen (
       DRAWPORT dp,
       DV_POINT *wpt,
       DV_POINT *spt)
```

TdpWorldToScreen converts a point in world coordinates, *wpt*, to screen coordinates, *spt*, according to the world-to-screen coordinate transform of the drawport, *dp*. The points are represented as *DV_POINT* structures. Returns *DV_FAILURE* if it is passed an invalid *dp*. Otherwise returns *DV_SUCCESS*.

TdpZoom

Tdp Functions T Routines

Scales a view within its drawport.

```
BOOLPARAM
TdpZoom (
       DRAWPORT dp,
       double scale)
```

TdpZoom changes the scale, *scale*, of the drawing in the drawport, *dp*. If the new scale factor compresses the whole drawing to a single pixel, or expands the world coordinates to be more than five pixels apart, the routine does nothing. Does not erase or redraw any drawports. Returns *NO* if it is passed an invalid *dp* or if no change is made. For more information, see the introduction to this module.

TdpZoomTo

Tdp Functions T Routines

Scales and pans a view within its drawport.

```
BOOLPARAM
TdpZoomTo (
       DRAWPORT dp,
       RECTANGLE *zoom_to_rect)
```

TdpZoomTo pans a view and changes its scale to display the drawing viewport specified by *zoom_to_rect*. If the drawport was created using TdpCreateStretch, the new drawing viewport is stretched to fit the current screen viewport. If the drawport was created using TdpCreate, a new “best fit” is calculated. Does not erase or redraw any drawports. Returns *NO* if it is passed an invalid *dp* or if no change is made. For more information, see the introduction to this module.

Tdr (Tdrawing)

Tdr Functions T Routines

Drawing access functions.

Tdr Functions

TdrForEachNamedObject	Traverses all the named objects in a drawing.
TdrGetNamedObject	Gets a named object from a drawing.
TdrGetObjectName	Gets the name of an object from a drawing.
TdrGetSelectedObject	Gets the selected object from a drawing.
TdrNameObject	Names an object in a drawing.

TdrForEachNamedObject

Tdr Functions T Routines

Traverses all the named objects in a drawing.

```
ADDRESS
TdrForEachNamedObject (
       OBJECT drawing,
       TDRFOREACHNAMEDOBJFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           OBJECT object,
           char *name,
           ADDRESS argblock)
```

TdrForEachNamedObject traverses all the named objects in the drawing and calls *fun* for each named object. Continues traversal while fun returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have three parameters: the object being processed, the name of the object, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each named object in the drawing, or
2. to find a particular object with a given name.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return a *NULL* value for *ADDRESS* if the object is not found. Otherwise it should return the *ADDRESS* of the object.

Note: You should not alter the drawing by adding, deleting, or reordering the named objects during traversal.

The following code fragments illustrate the use of traversal functions. In the first fragment, the function called by *TdrForEachNamedObject* continues the traversal by always returning *NULL*.

```
    VIEW masterview, componentview;
```

```
OBJECT masterdrawing, componentdrawing;

ADDRESS AddToDrawing (OBJECT object, char *name, ADDRESS drawing_1);
int
main (int argc, char *argv[]);
{
   . . .
   masterview = TviLoad ("MasterView");
   masterdrawing = TviGetDrawing (masterview);
   componentview = TviLoad ("ComponentView");
   componentdrawing = TviGetDrawing (componentview);
   TdrForEachNamedObject (componentdrawing, AddToDrawing, (ADDRESS)
             &masterdrawing);
   . . .
}

/* AddToDrawing adds the object and its name to a drawing */
ADDRESS
AddToDrawing (
      OBJECT object,
      char *name,
      ADDRESS args)
{
   OBJECT *drawing_1 = (OBJECT *) args;
   VOdrObAddNamed (*drawing_1, object, name);
   return V_CONTINUE_TRAVERSAL;
}
```

In the following code fragment, the function called by *TobForEachVdp* ends the traversal by returning a non-*NULL* value.

```
VARDESC vdp;
ADDRESS getvdp (OBJECT, VARDESC, ADDRESS);
OBJECT drawing;

/* Get a variable descriptor from the drawing. */
vdp = TobForEachVdp (drawing, getvdp, (ADDRESS)0);

. . .

ADDRESS
getvdp (
      OBJECT obj,        /* not used */
      VARDESC vdp,
      ADDRESS)           /* not used */
{
   return (ADDRESS) vdp;
}
```

TdrGetNamedObject

Tdr Functions T Routines

Gets a named object from a drawing.

```
OBJECT
TdrGetNamedObject (
       OBJECT drawing,
       char *name)
```

TdrGetNamedObject finds the first object in the drawing with the specified name. It returns the named object. Returns *NULL* if the object is not in the drawing or if the object is not named in the drawing.

TdrGetObjectName

Tdr Functions T Routines

Gets the name of an object from a drawing.

```
char *
TdrGetObjectName (
       OBJECT drawing,
       OBJECT object)
```

TdrGetObjectName returns the name of the specified object in the drawing. Returns *NULL* if the object is not named or does not exist in the drawing. This function is typically called after *TdrGetSelectedObject*.

TdrGetSelectedObject

Tdr Functions T Routines

Gets the selected object from a drawing.

```
OBJECT
TdrGetSelectedObject (
       OBJECT drawing,
       OBJECT location_object,
       int check_mode)
```

TdrGetSelectedObject tries to find the object in the drawing that was selected by the location object. Returns the object; *NULL* if no object was selected. If *check_mode* is *NAMED_SEARCH*, only checks named objects in the drawing. If *check_mode* is *FULL_SEARCH*, checks all objects. Returns the selected object. You must use *TloGetSelectedDrawport* to check that the drawport you want is current before calling TdrGetSelectedObject. *TloGetSelectedObject* is an alternate method for selecting an object that does not require a call to *TloGetSelectedDrawport*.

TdrNameObject

Tdr Functions T Routines

Names an object in a drawing.

```
BOOLPARAM
TdrNameObject (
       OBJECT drawing,
       OBJECT object,
       char *name)
```

TdrNameObject names the object in the drawing. If the name is *NULL*, the object's current name is deleted. Returns *YES* if the specified object is in the drawing. Otherwise returns *NO*.

Tds (Tdatasource)

Tds Functions T Routines

Manages data sources (*ds*). A data source represents a single source of data, in the form of a constant, file, function, memory, or process. It contains the name of the source of data, and a list of data source variables (*dsv*) that accept that data. Data sources are contained in data source lists (*dl*) which can belong to views (*vi*).

Function data sources have a special creation routine and other special routines for handling function descriptor sets, function names, function arguments, and auxiliary data. These routines are not useful for other types of data sources.

TInit, TTerminate	**Tds**	Tproto
Tdl	Tdsv	Tsc
Tdp	Tlo	Tvd
Tdr	Tob	Tvi

Tds Functions

TdsAddDsVar	Adds a data source variable to a data source.
TdsClone	Copies a data source.
TdsCloseData	Closes a data source.
TdsClrFcnArg	Clears an argument for a function associated with a data source.
TdsCreate	Creates a new data source.
TdsCreateDsVar	Creates a new data source variable in a data source.
TdsDeleteDsVar	Deletes a data source variable from a data source.
TdsDestroy	Destroys a data source, freeing its memory.
TdsEditAttributes	Changes data source attributes.
TdsFdsCreate	Creates a data source using a function descriptor set.
TdsForEachVar	Traverses all data source variables in a data source.
TdsGetAttributes	Gets data source attributes.
TdsGetAuxData	Gets the auxiliary data buffer of a function data source.
TdsGetFcnArg	Gets an argument for a function associated with a data source.
TdsGetFcnArgCnt	Gets the number of arguments for a function associated with a data source.
TdsGetFcnName	Gets the descriptive name of a function associated with a data source.
TdsGetFdsName	Gets the name of the function descriptor set used by a data source.
TdsGetName	Gets the name of a data source.
TdsGetNamedDsVar	Returns the data source variable with the given name.
TdsLoad	Loads a new data source from a file.
TdsMerge	Merges one data source into another.
TdsMoveDataSource	Moves a data source.
TdsOpenData	Opens all files and processes in a data source.
TdsReadData	Reads data for one iteration of a data source.
TdsSave	Saves a data source to a file.
TdsSetAuxData	Assigns an auxiliary data buffer to a function data source.
TdsSetFcnArg	Sets an argument for a function associated with a data source.
TdsSetFcnByName	Sets the function associated with a data source.
TdsSetFdsByName	Sets the function descriptor set used by a data source.
TdsValid	Determines if a data source is valid.
TdsWriteData	Writes one iteration of data out to a target.

TdsAddDsVar

Tds Functions T Routines

Adds a data source variable to a data source.

```
BOOLPARAM
TdsAddDsVar (
	DATASOURCE ds,
	DSVAR dsvar,
	DSVAR dsvar_reference)
```

TdsAddDsVar adds a data source variable to the data source. The variable, *dsvar*, is added before *dsvar_reference*. However, if *dsvar_reference* is *NULL*, the variable is added to the end of the list of data source variables in the data source. Returns *DV_FAILURE* if *ds*, *dsvar*, or *dsvar_reference*, is invalid, or if *dsvar_reference* is not in the data source. Otherwise returns *DV_SUCCESS*.

TdsClone

Tds Functions T Routines

Copies a data source.

```
DATASOURCE
TdsClone (
       DATASOURCE ds)
```

TdsClone creates and returns a deep copy of the data source, *ds*. Does not clone bindings between data source variables and the variable descriptors of dynamic objects. Returns *DV_FAILURE* if it is passed an invalid data source.

TdsCloseData

Tds Functions T Routines

Closes a data source.

```
BOOLPARAM
TdsCloseData (
       DATASOURCE ds)
```

TdsCloseData closes the file or process associated with the data source, *ds*. Returns *DV_FAILURE* if it is passed an invalid data source. Otherwise returns *DV_SUCCESS*.

TdsClrFcnArg

Tds Functions T Routines

Clears an argument for a function associated with a data source.

```
BOOLPARAM
TdsClrFcnArg (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype,
       int argindex)
```

TdsClrFcnArg clears an argument for a specific type of function within the function descriptor set. Only optional arguments can be cleared. *ds* is the data source which is using the function descriptor set, *fcntype* is the type of function, and *argindex* is the index within the argument list. Valid types of functions are listed in TdsSetFcnByName. Returns *DV_SUCCESS* if successful. Returns *DV_FAILURE* if *argindex* is too large, *argindex* refers to a required argument, or no such function type is defined in the function descriptor set.

TdsCreate

Tds Functions T Routines

Creates a new data source.

```
DATASOURCE
TdsCreate (void)
```

TdsCreate creates and returns a new data source, *ds*. Use TdsFdsCreate to create a data source that gets its data from a function descriptor set.

TdsCreateDsVar

Tds Functions T Routines

Creates a new data source variable in a data source.

```
DSVAR
TdsCreateDsVar (
       DATASOURCE ds)
```

TdsCreateDsVar creates a new data source variable and adds it to the end of the list maintained by the data source, *ds*. By default, the data source variable is created as a scalar float. Its default name is “Var:n,” where n is determined by the number of data source variables created so far. To set the attributes of the new data source variable, call TdsvEditAttributes after calling this routine. If the data source is a function data source and its function descriptor set includes a data source variable creation function, *TdsCreateDsVar* calls this function. If this function fails, the creation is aborted. Returns the new data source variable if successful. Otherwise returns *NULL*. To create and add a data source variable elsewhere in the list, use *TdsvCreate* and TdsAddDsVar.

TdsDeleteDsVar

Tds Functions T Routines

Deletes a data source variable from a data source.

```
BOOLPARAM
TdsDeleteDsVar (
       DATASOURCE ds,
       DSVAR dsvar)
```

TdsDeleteDsVar removes but does not destroy a data source variable, *dsv*, from the data source, *ds*. Returns *DV_FAILURE* if *ds* or *dsvar* is invalid, or if *dsvar* is not in the data source. Otherwise returns *DV_SUCCESS*.

TdsDestroy

Tds Functions T Routines

Destroys a data source, freeing its memory.

```
BOOLPARAM
TdsDestroy (
       DATASOURCE ds)
```

TdsDestroy destroys a data source, *ds*, freeing its memory. Does nothing and returns *DV_FAILURE* if it is passed an invalid data source, or an attempt is made to destroy a data source which is bound to variable descriptors of dynamic objects. Otherwise returns *DV_SUCCESS*.

TdsEditAttributes

Tds Functions T Routines

Changes data source attributes.

```
BOOLPARAM
TdsEditAttributes (
       DATASOURCE ds,
       int type,
       int format,
       char *source)
```

TdsEditAttributes changes the attributes of the data source. Any field can be *NOCHANGE*, indicating no changes for that attribute.

Valid *type* flags:

DSPROCESS	DSFILE	DSCONSTANT
DSFUNCTION	DSMEMORY	

Valid *format* flags:

DSASCII	*DSBINARY*

format flags are valid only for *type* file or process. If *ds* is a file or a process, then *source* must match the name of the file or process used as a data source; if *ds* is a constant, memory, or function data source then *source* is only a label used to identify the data source. If *source* is *NULL*, *default.dat* is used. Returns *DV_FAILURE* if it is passed an invalid *ds*. Otherwise returns *DV_SUCCESS*.

TdsFdsCreate

Tds Functions T Routines

Creates a data source using a function descriptor set.

```
DATASOURCE
TdsFdsCreate (
       char *fds_name)
```

TdsFdsCreate creates a data source and associates a function descriptor set, *fds_name*, with it. If the function descriptor set contains a data source creation function, *TdsFdsCreate* calls this function immediately after it creates the data source. Returns the new data source of type *DSFUNCTION*.

TdsForEachVar

Tds Functions T Routines

Traverses all data source variables in a data source.

```
ADDRESS
TdsForEachVar (
       DATASOURCE ds,
       TDSFOREACHVARFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
          DSVAR dsvar,
          ADDRESS argblock)
```

TdsForEachVar traverses all of the data source variables in the data source and calls *fun* for each data source variable. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the data source variable being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each data source variable in the data source, or
2. to find a particular data source variable in the data source.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the data source variable is not found. Otherwise it should return the data source variable for *ADDRESS*.

Note: You should not alter the data source by adding, deleting, or reordering the data source variables during traversal.

For an example of a typical function, see the example under TdrForEachNamedObject. Note that the example demonstrates the use of a function with three parameters, but *TdsForEachVar* requires only two.

TdsGetAttributes

Tds Functions   T Routines

Gets data source attributes.

```
BOOLPARAM
TdsGetAttributes (
       DATASOURCE ds,
       int *type,
       int *format,
       char **source)
```

TdsGetAttributes gets data source attributes.

Valid *type* flags:

DSPROCESS	DSFILE	DSCONSTANT
DSFUNCTION	DSMEMORY	

Valid *format* flags:

DSASCII	*DSBINARY*

format flags are valid only for *type* file or process. If *ds* is a file or a process, then *source* must match the name of the file or process used as the data source; if *ds* is a constant, memory, or function data source then *source* is only a label used to identify the data source. Returns *DV_FAILURE* if it is passed an invalid *ds*. Otherwise returns *DV_SUCCESS*.

TdsGetAuxData

Tds Functions T Routines

Gets the auxiliary data buffer of a function data source.

```
ADDRESS
TdsGetAuxData (
       DATASOURCE ds)
```

TdsGetAuxData gets the address of the auxiliary data buffer from the data source, *ds*. The data buffer is used to store data for a function descriptor set. For more information, see *TdsSetAuxData*. Returns the address if the query is successful. Returns *NULL* if there is no address, if the data buffer was freed, or if an error occurs.

TdsGetFcnArg

Tds Functions   T Routines

Gets an argument for a function associated with a data source.

```
BOOLPARAM
TdsGetFcnArg (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype,
       int argindex,
       int *typep,
       ANYTYPE *valuep)
```

TdsGetFcnArg gets an argument for a specific type of function within the function descriptor set. *ds* is the data source which is using the function descriptor set, *fcntype* is the type of function to query, and *argindex* is the index within the argument list. Valid types of functions are listed in TdsSetFcnByName.

Returns the argument value in *valuep* and the type of argument in *typep*. Valid argument types are *V_T_TYPE* (text), *V_L_TYPE* (long), *V_D_TYPE* (double), or *V_DSV_TYPE* (data source variable).

Returns *DV_SUCCESS* if the query is successful. Returns *DV_FAILURE* if no argument corresponds to the index, no such function type is defined in the function descriptor set, or if an error occurs.

TdsGetFcnArgCnt

Tds Functions T Routines

Gets the number of arguments for a function associated with a data source.

```
BOOLPARAM
TdsGetFcnArgCnt (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype,
       int *req_arg_cntp,
       int *opt_arg_cntp)
```

TdsGetFcnArgCnt gets the count of the required and optional arguments for a specific type of function within the function descriptor set. *ds* is the data source which is using the function descriptor set and *fcntype* is the type of function to query. Valid types of functions are listed in TdsSetFcnByName.

Returns the number of required arguments in *req_arg_cntp* and the number of optional user-defined arguments in *opt_arg_cntp*. Returns *DV_SUCCESS* if the query for the argument count is successful; *DV_FAILURE* if no such function type is defined or if an error occurs.

TdsGetFcnName

Tds Functions T Routines

Gets the descriptive name of a function associated with a data source.

```
char *
TdsGetFcnName (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype)
```

TdsGetFcnName gets the name of the function of a specific type used by the data source. *ds* is the data source which is using the function descriptor set. *fcntype* is the type of function to query. Valid types of functions are listed in TdsSetFcnByName. Returns the descriptive name of the function if it exists. Returns *NULL* if there is no name or if an error occurs.

TdsGetFdsName

Tds Functions T Routines

Gets the name of the function descriptor set used by a data source.

```
char *
TdsGetFdsName (
       DATASOURCE ds)
```

TdsGetFdsName gets the name of the function descriptor set used by the data source. *ds* is the data source to query. Returns the name of the function descriptor set if it exists. Returns *NULL* if an error occurs.

TdsGetName

Tds Functions T Routines

Gets the name of a data source.

```
char *
TdsGetName (
       DATASOURCE ds)
```

TdsGetName returns the name of the data source, *ds*. Returns *DV_FAILURE* if it is passed an invalid *ds*.

TdsGetNamedDsVar

Tds Functions T Routines

Returns the data source variable with the given name.

```
DSVAR
TdsGetNamedDsVar (
       DATASOURCE ds,
       char *name)
```

TdsGetNamedDsVar returns the first data source variable with the name, *name*, if one exists. Returns *NULL*. Returns *DV_FAILURE* if it is passed an invalid *ds*.

TdsLoad

Tds Functions T Routines

Loads a new data source from a file.

```
DATASOURCE
TdsLoad (
       char *filename)
```

TdsLoad loads a data source, *ds*, from the file, *filename*. Returns *DV_FAILURE* if the file could not be opened or if the loaded file does not contain a data source.

TdsMerge

Tds Functions T Routines

Merges one data source into another.

```
BOOLPARAM
TdsMerge (
       DATASOURCE ds1,
       DATASOURCE ds2,
       int matchflag)
```

TdsMerge attempts to merge the data source, *ds2*, into the data source *ds1* according to the matchflag.

DS_EXACTMATCH	Merges if *ds2* exactly matched *ds1*.
DS_SUBSETMATCH	Merges if *ds2* is a subset of *ds1*.
DS_NAMEMATCH	Merges if the name of *ds2* matches the name of *ds1*.

TdsMerge returns *DV_SUCCESS* or *DV_FAILURE*.

TdsMoveDataSource

Tds Functions T Routines

Moves a data source.

```
BOOLPARAM
TdsMoveDataSource (
       DATASOURCE dstomove,
       DATASOURCE dstoinsertbefore)
```

TdsMoveDataSource is used to change the position of a data source. It moves *dstomove* from its current location to before *dstoinsertbefore*. Both data sources can be in the same data source list or in different ones. If *dstoinsertbefore* is *NULL*, the routine puts *dstomove* at the end of its own data source list. Returns *DV_FAILURE* if *dstomove* or *dstoinsertbefore* are invalid. Otherwise returns *DV_SUCCESS*.

TdsOpenData

Tds Functions T Routines

Opens all files and processes in a data source.

```
BOOLPARAM
TdsOpenData (
       DATASOURCE ds)
```

TdsOpenData opens the file or process associated with the data source, *ds*. Returns *DV_FAILURE* if *ds* is invalid or cannot be opened.

TdsReadData

Tds Functions T Routines

Reads data for one iteration of a data source.

```
BOOLPARAM
TdsReadData (
       DATASOURCE ds)
```

TdsReadData reads all the data for one iteration of the data source, *ds*, into its data source variables. Returns *DV_FAILURE* if *ds* is invalid, not open, or has reached the end of the file. Otherwise returns *DV_SUCCESS*.

TdsSave

Tds Functions T Routines

Saves a data source to a file.

```
BOOLPARAM
TdsSave (
       DATASOURCE ds,
       char *filename,
       int access_mode)
```

TdsSave saves a data source, *ds*, to a file, *filename*, using *access_mode*. *access_mode* should be *WRITE_EXPANDED* for ASCII write, or *WRITE_COMPACT* for binary write. Flag values are defined in *VOstd.h*. Returns *DV_FAILURE* if *ds* is invalid or if the file cannot be opened for writing. Otherwise returns *DV_SUCCESS*.

TdsSetAuxData

Tds Functions   T Routines

Assigns an auxiliary data buffer to a function data source.

```
BOOLPARAM
TdsSetAuxData (
       DATASOURCE ds,
       ADDRESS data,
       TDSFREEFUNPTR freefcn)

   void
   freefcn (
          ADDRESS data)
```

TdsSetAuxData associates a user-defined auxiliary data buffer, *data*, and its free function, *freefcn*, with the data source, *ds*. The auxiliary data buffer is created and maintained by the function descriptor set for use by its functions. Setting *data* to *NULL* clears the data buffer.

The free function is optional. If it is specified, it is called automatically when TviCloseData, TdlCloseData, or TdsCloseData is called. The free function frees the buffer and clears the address. If a free function is not specified, the buffer remains unless freed by the data source destroy function of the function descriptor set.

Returns *DV_SUCCESS* if the data buffer and free function are set successfully. Otherwise returns *DV_FAILURE* and aborts the changes.

TdsSetFcnArg

Tds Functions T Routines

Sets an argument for a function associated with a data source.

```
BOOLPARAM
TdsSetFcnArg (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype,
       int argindex,
       int type,
       ANYTYPE *valuep)
```

TdsSetFcnArg sets an argument for a specific type of function within the function descriptor set. *ds* is the data source which is using the function descriptor set and *fcntype* is the type of function. Valid types of functions are listed in TdsSetFcnByName.

argindex is the index within the argument list. If the index does not refer to a current argument, it must refer to a new optional argument at the end of the list. *valuep* specifies the new value of the argument. *type* specifies the type of the argument, which you can change only if the argument is an optional argument rather than a required argument declared in the function descriptor set. Valid argument types are *V_T_TYPE* (text), *V_L_TYPE* (long), *V_D_TYPE* (double), or *V_DSV_TYPE* (data source variable). To delete an optional argument, use TdsClrFcnArg.

Returns *DV_SUCCESS* if the arguments were set successfully. Returns *DV_FAILURE* if no such function type exists in the function descriptor set, *argindex* does not refer to an existing argument, *type* conflicts with the defined type of a required argument, or an error occurs.

TdsSetFcnByName

Tds Functions T Routines

Sets the function associated with a data source.

```
BOOLPARAM
TdsSetFcnByName (
       DATASOURCE ds,
       V_FDS_FCN_ENUM fcntype,
       char *fcnname)
```

TdsSetFcnByName changes the function used by the data source for a specific type of function within the function descriptor set. *fcnname* is the descriptive name of the function. *ds* is the data source which is using the function descriptor set and *fcntype* is the type of function to change. Valid types of functions are:

V_FDS_FCN_OPEN	The Open function, called by TviOpenData, TdlOpenData, or TdsOpenData.
V_FDS_FCN_READ	The Read function, called by TviReadData, TdlReadData, or TdsReadData.
V_FDS_FCN_CLOSE	The Close function, called by TviCloseData, TdlCloseData, or TdsCloseData.
V_FDS_FCN_WRITE	The DS-Write function, called by TdsWriteData.
V_FDS_FCN_DS_CREATE	The DS-Create function, called by TdsFdsCreate.
V_FDS_FCN_DS_DESTROY	The DS-Destroy function, called by TviDestroy, TdlDestroy, or TdsDestroy.
V_FDS_FCN_DS_SAVE	The DS-Save function, called by any of the *Tvi* saving routines, TdlSave, or TdsSave.
V_FDS_FCN_DS_RESTORE	The DS-Restore function, called by any of the *Tvi* loading routines, TdlLoad, or TdsLoad.

Returns *DV_SUCCESS* if the function is successfully changed. Returns *DV_FAILURE* if an error occurs.

TdsSetFdsByName

Tds Functions T Routines

Sets the function descriptor set used by a data source.

```
BOOLPARAM
TdsSetFdsByName (
       DATASOURCE ds,
       char *fds_name)
```

TdsSetFdsByName changes the function descriptor set used by the data source, *ds*, to the function descriptor set specified by *fds_name*. When the function descriptor set is changed, all function arguments are cleared. The new functions and their arguments are set using defaults in the function descriptor set. Returns *DV_SUCCESS* if successful. Returns *DV_FAILURE* and aborts the change if no function descriptor set is found with the specified name or an error occurs.

TdsValid

Tds Functions T Routines

Determines if a data source is valid.

```
BOOLPARAM
TdsValid (
       DATASOURCE ds)
```

TdsValid returns *DV_SUCCESS* if the data source is valid. Otherwise returns *DV_FAILURE*.

TdsWriteData

Tds Functions T Routines

Writes one iteration of data out to a target.

```
BOOLPARAM
TdsWriteData(
       DATASOURCE ds)
```

TdsWriteData calls user-supplied write functions to write the data from the data source out to another part of the application. Currently this routine works only for function data sources that have a user-supplied DS-Write function assigned to them. In addition to the DS-Write function, the data source variables can each have their own DSV-Write function. *TdsWriteData* calls the DS-Write function first, then calls each data source variable's DSV-Write function. Returns *DV_FAILURE* if *ds* is invalid or not open. Otherwise returns *DV_SUCCESS*.

Tdsv (Tdatasourcevariable)

Tdsv Functions T Routines

Manages data source variables (*dsv*). Data source variables are DataViews private types that maintain buffers for storing data from data sources. A data source variable contains information about the type, size and dimensionality of its data, and a name. Data source variables are usually bound to one or more variable descriptors (*vdp*). Data sources variables are managed by data sources.

Data source variables in function data sources have special routines for handling function names, function arguments, and auxiliary data. These routines are not useful for data source variables in other types of data sources.

TInit, TTerminate	Tds	Tproto
Tdl	**Tdsv**	Tsc
Tdp	Tlo	Tvd
Tdr	Tob	Tvi

Tdsv Functions

TdsvAttachVdp	Attaches a variable descriptor to a data source variable.
TdsvClone	Copies a data source variable.
TdsvClrFcnArg	Clears an argument for a function associated with a data source variable.
TdsvCreate	Creates a new data source variable.
TdsvDestroy	Destroys a data source variable.
TdsvDetachVdp	Detaches variable descriptor from a data source variable.
TdsvEditAttributes	Edits data source variable attributes.
TdsvForEachVdp	Traverses the variable descriptors bound to a data source variable.
TdsvGetAttributes	Gets data source variable attributes.
TdsvGetAuxData	Gets the auxiliary data buffer of a data source variable in a function data source.
TdsvGetBuffer	Gets data source variable buffer address.
TdsvGetDataSource	Gets the data source of a data source variable.
TdsvGetFcnArg	Gets an argument for a function associated with a data source variable.
TdsvGetFcnArgCnt	Gets the number of arguments for a function associated with a data source variable.
TdsvGetFcnName	Gets the descriptive name of a function associated with a data source.
TdsvGetGlobalFlag	Gets the global flag of a data source variable.
TdsvGetName	Gets the name of a data source variable.
TdsvGetSize	Gets the size of a data source variable.
TdsvGetType	Gets the type of a data source variable.
TdsvReadData	Reads data separately for one data source variable.
TdsvSetAuxData	Assigns an auxiliary data buffer to a data source variable in a function data source.
TdsvSetFcnArg	Sets an argument for a function associated with a data source variable.
TdsvSetFcnByName	Sets the function associated with a data source variable.
TdsvSetGlobalFlag	Sets the global flag for a data source variable.
TdsvSetInitialValue	Sets the initial value for a constant data source variable.
TdsvSetTypedValue	Sets a value in a data buffer.
TdsvSetValue	Sets a *double* in a data buffer.
TdsvValid	Determines if a data source variable is valid.
TdsvWriteData	Writes data from one variable out to a target.

TdsvAttachVdp

Tdsv Functions   T Routines

Attaches a variable descriptor to a data source variable.

```
BOOLPARAM
TdsvAttachVdp (
        DSVAR dsvar,
        VARDESC vdp)
```

TdsvAttachVdp binds the variable descriptor, *vdp*, to the data source variable, *dsvar*. More than one variable descriptor can be bound to a data source variable, but each variable descriptor can only have one data source variable attached to it. Changes the name of *vdp* to match the name of *dsvar*. Returns *DV_FAILURE* if it is passed an invalid *dsvar* or *vdp*. Otherwise returns *DV_SUCCESS*.

TdsvClone

Tdsv Functions T Routines

Copies a data source variable.

```
DSVAR
TdsvClone (
       DSVAR dsvar)
```

TdsvClone creates and returns a copy of a data source variable, *dsvar*. Returns *DV_FAILURE* if it is passed an invalid *dsvar*. Does not clone the bindings between data source variables and dynamic objects.

TdsvClrFcnArg

Tdsv Functions T Routines

Clears an argument for a function associated with a data source variable.

```
BOOLPARAM
TdsvClrFcnArg (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype,
       int argindex)
```

TdsvClrFcnArg clears an argument for a specific type of function within the function descriptor set. Only optional arguments can be cleared. *dsvar* is the data source variable in the data source using the function descriptor set, *fcntype* is the type of function, and *argindex* is the index within the argument list. Valid types of functions are listed in TdsvSetFcnByName. Returns *DV_SUCCESS* if successful. Returns *DV_FAILURE* if *argindex* is too large, *argindex* refers to a required argument, or no such function type is defined in the function descriptor set.

TdsvCreate

Tdsv Functions T Routines

Creates a new data source variable.

```
DSVAR
TdsvCreate (void)
```

TdsvCreate creates and returns a new data source variable. See also *TdsCreateDsVar* to create a data source variable and add it to a data source in one step. Always use TdsCreateDsVar to create data source variables for a function data source.

TdsvDestroy

Tdsv Functions T Routines

Destroys a data source variable.

```
BOOLPARAM
TdsvDestroy (
        DSVAR dsvar)
```

TdsvDestroy destroys a data source variable, *dsvar*. Does nothing and returns *DV_FAILURE* if *dsvar* still has dynamic objects attached to it or is invalid. Otherwise returns *DV_SUCCESS*.

TdsvDetachVdp

Tdsv Functions T Routines

Detaches variable descriptor from a data source variable.

```
BOOLPARAM
TdsvDetachVdp (
       DSVAR dsvar,
       VARDESC vdp)
```

TdsvDetachVdp detaches the variable descriptor, *vdp*, from the data source variable, *dsvar*. No data is displayed for a dynamic object which uses *vdp* until the variable descriptor is attached to another data source variable, *dsvar*. Returns *DV_FAILURE* if it is passed an invalid *dsvar* or *vdp*. Otherwise returns *DV_SUCCESS*. See also *TdsvAttachVdp*.

TdsvEditAttributes

Tdsv Functions T Routines

Edits data source variable attributes.

```
BOOLPARAM
TdsvEditAttributes (
       DSVAR dsvar,
       char *name,
       int type,
       int rows,
       int columns,
       int delimiter)
```

TdsvEditAttributes sets the various attributes of a data source variable, *dsvar*. *name* should contain a new name string. If *name* is *NULL*, a unique name is assigned in the form *VAR:n*, where *n* is an integer. *type* should contain a flag indicating the variable type. Valid flags are listed below in TdsvGetType. *rows* and *columns* indicate the number of dimensions for matrix variables. For scalar variables, set rows and columns to 1; for vectors, set columns to 1 and rows to the dimension of the vector.

delimiter contains the delimiter character for text variables. For fixed-length text, set *delimiter* to *NULL*. The following delimiters are allowed, in addition to any single character:

In the data file:	*delimiter* Value:
<Return>, *<NewLine>*, or *<LineFeed>*	’\n’
<Tab>	’\t’
a double-quote before and after each string	V_DOUBLE_QUOTED
a single-quote before and after each string	V_SINGLE_QUOTED
one or more double-quote between each pair of strings	’"’
one or more single-quote between each pair of strings	’’’

Any attribute set to *NOCHANGE* remains unchanged. Returns *DV_FAILURE* if it is passed an invalid *dsvar*. Otherwise returns *DV_SUCCESS*. If the application accesses the buffer of the data source variable, call *TdsvGetBuffer* after calling this routine because the buffer address may have changed.

Note that this routine not only changes the name of the data source variable specified, but also applies the same new name to every variable descriptor that refers to this data source variable by internally calling *VPvdvarname* on every variable descriptor in the data source variable’s reference list.

TdsvForEachVdp

Tdsv Functions T Routines

Traverses the variable descriptors bound to a data source variable.

```
ADDRESS
TdsvForEachVdp (
       DSVAR dsvar,
       TDSVFOREACHVDPFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
          VARDESC vdp,
          ADDRESS argblock)
```

TdsvForEachVdp traverses the list of variable descriptors bound to the data source variable, *dsvar*, and calls *fun* for each variable descriptor. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the variable descriptor being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each variable descriptor attached to *dsvar*, or
2. to find a particular variable descriptor attached to *dsvar*.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the variable descriptor is not found. Otherwise it should return the variable descriptor for *ADDRESS*.

Note: You should not alter the list by adding, deleting, or reordering the variable descriptors during traversal.

For an example of a typical function, see the example under *TdrForEachNamedObject*. Note that the example demonstrates the use of a function with three parameters, but *TdsvForEachVdp* requires only two.

TdsvGetAttributes

Tdsv Functions T Routines

Gets data source variable attributes.

```
BOOLPARAM
TdsvGetAttributes (
       DSVAR dsvar,
       char **name,
       int *type,
       int *rows,
       int *columns,
       char *delimiter)
```

Error! Reference source not found.*TdsvGetAttributes* gets the various attributes of a data source variable, *dsvar*. *name* gets a pointer to the name string, *type* contains a flag indicating the variable type. Valid flags are listed below in TdsvGetType. *rows* and *columns* contains the number of dimensions for matrix variables, and *delimiter* contains the delimiter character for text variables. *NULL* attributes are interpreted as setting the value to 0. Returns *DV_FAILURE* if it is passed an invalid *dsvar*. Otherwise returns *DV_SUCCESS*.

TdsvGetAuxData

Tdsv Functions T Routines

Gets the auxiliary data buffer of a data source variable in a function data source.

```
ADDRESS
TdsvGetAuxData (
       DSVAR dsvar)
```

TdsvGetAuxData gets the address of the auxiliary data buffer from the data source variable, *dsvar*. The data buffer is used to store data for a function descriptor set. For more information, see *TdsvSetAuxData*. Returns the address if the query is successful. Returns *NULL* if there is no address, if the data buffer was freed, or if an error occurs.

TdsvGetBuffer

Tdsv Functions T Routines

Gets data source variable buffer address.

```
ADDRESS
TdsvGetBuffer (
       DSVAR dsvar)
```

TdsvGetBuffer queries the data source variable, *dsvar*, for the address of its data buffer. Returns the *ADDRESS* of the buffer. Returns *DV_FAILURE* if it is passed an invalid data source variable. To make sure the correct data source variable buffer address is being used, call this routine after a call to TdsvEditAttributes. For a text variable with delimiter, the buffer address may also change after reading new data, so call this routine after calling TviReadData, TdlReadData, or TdsReadData.

TdsvGetDataSource

Tdsv Functions T Routines

Gets the data source of a data source variable.

```
DATASOURCE
TdsvGetDataSource (
       DSVAR dsvar)
```

TdsvGetDataSource returns the data source to which the data source variable, *dsvar*, belongs. Returns *NULL* if it is passed an invalid data source variable or if the data source variable does not currently belong to any data source.

TdsvGetFcnArg

Tdsv Functions T Routines

Gets an argument for a function associated with a data source variable.

```
BOOLPARAM
TdsvGetFcnArg (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype,
       int argindex,
       int *typep,
       ANYTYPE *valuep)
```

TdsvGetFcnArg gets an argument for a specific type of function within the function descriptor set. *dsvar* is the data source variable in a function data source, *fcntype* is the type of function to query, and *argindex* is the index within the argument list. Valid types of functions are listed in TdsvSetFcnByName.

Returns the argument value in *valuep* and the type of argument in *typep*. Valid argument types are:

V_T_TYPE	text string
V_L_TYPE	LONG
V_D_TYPE	double
V_DSV_TYPE	DSVAR

Returns *DV_SUCCESS* if the query is successful. Returns *DV_FAILURE* if no argument corresponds to the index, no such function type is defined in the function descriptor set, or if an error occurs.

TdsvGetFcnArgCnt

Tdsv Functions T Routines

Gets the number of arguments for a function associated with a data source variable.

```
BOOLPARAM
TdsvGetFcnArgCnt (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype,
       int *req_arg_cntp,
       int *opt_arg_cntp)
```

Error! Reference source not found.*TdsvGetFcnArgCnt* gets the count of the required and optional arguments for a specific type of function within the function descriptor set. *dsvar* is the data source variable which is using the function and *fcntype* is the type of function to query. Valid types of functions are listed in TdsvSetFcnByName.

Returns the number of required arguments in *req_arg_cntp* and the number of optional user-defined arguments in *opt_arg_cntp*. Returns *DV_SUCCESS* if the query for the argument count is successful. Returns *DV_FAILURE* if no such function type is defined in the function descriptor set or if an error occurs.

TdsvGetFcnName

Tdsv Functions T Routines

Gets the descriptive name of a function associated with a data source.

```
char *
TdsvGetFcnName (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype)
```

TdsvGetFcnName gets the name of the function associated with the data source variable. *dsvar* is the data source variable in a function data source. *fcntype* is the type of function to query. Valid types of functions are listed in TdsvSetFcnByName. Returns the descriptive name of the function if it exists. Returns *NULL* if there is no name or if an error occurs.

TdsvGetGlobalFlag

Tdsv Functions T Routines

Gets the global flag of a data source variable.

```
int
TdsvGetGlobalFlag (
       DSVAR dsvar)
```

TdsvGetGlobalFlag returns the global flag of the data source variable, *dsvar*. The global flag controls whether or not the data source variable, if referenced by a subdrawing, can be mapped to another data source variable in the higher-level view. See VOsubdrawing for more information on mapping. Returns *DV_FAILURE* if *dsvar* is invalid. Otherwise returns the global flag. Valid values for the returned flag are:

V_GLOBAL	Can be mapped.
V_LOCAL	Cannot be mapped.

TdsvGetName

Tdsv Functions T Routines

Gets the name of a data source variable.

```
char *
TdsvGetName (
       DSVAR dsvar)
```

TdsvGetName returns the name of the data source variable, *dsvar*. This is a pointer to an internal variable which should not be modified. Returns *DV_FAILURE* if it is passed an invalid *dsvar*.

TdsvGetSize

Tdsv Functions T Routines

Gets the size of a data source variable.

```
int
TdsvGetSize (
       DSVAR dsvar,
       int *rows,
       int *columns)
```

TdsvGetSize queries the data source variable, *dsvar*, for the size of its data buffer. Returns the total number of bytes in the current buffer. Returns *DV_FAILURE* if it is passed an invalid *dsvar*. The function also gets the number of *rows* and *columns* in *dsvar*. A scalar variable contains 1 row and 1 column. A vector variable has *columns* set to 1 and *rows* set to the size of the vector.

TdsvGetType

Tdsv Functions T Routines

Gets the type of a data source variable.

```
int
TdsvGetType (
        DSVAR dsvar)
```

TdsvGetType returns a flag indicating the type of the data source variable, *dsvar*. Possible flag values are:

Flag	**Data Type**	**Size in bits**
V_C_TYPE	char	8
V_UC_TYPE	unsigned char, UBYTE	8
V_S_TYPE	short	16
V_US_TYPE	unsigned short	16
V_L_TYPE	int, LONG	32
V_UL_TYPE	unsigned int, ULONG	32
V_F_TYPE	float	32 (or 64 for some systems)
V_D_TYPE	double	64 (or 128 for some systems)
V_T_TYPE	NULL-terminated string	no set size

If the format of the data source is ASCII, the only valid types are *V_T_TYPE* and *V_F_TYPE*. If the format of the data source is binary, then all types are valid. Returns *DV_FAILURE* if it is passed an invalid *dsvar*.

TdsvReadData

Tdsv Functions T Routines

Reads data separately for one data source variable.

```
BOOLPARAM
TdsvReadData(
       DSVAR dsvar)
```

TdsvReadData reads data for only one data source variable, in contrast with TdsReadData, which reads data for all the variables in a data source. Call this routine when you need to update a data source variable outside the normal read cycle. This routine is most useful for variables in function or memory data sources. For file and process data sources that contain several variables in a particular order, you must read the individual variables in the formatted order. Returns *DV_FAILURE* if the data source is invalid, not open, or has reached the end of the file. Otherwise returns *DV_SUCCESS*.

TdsvSetAuxData

Tdsv Functions T Routines

Assigns an auxiliary data buffer to a data source variable in a function data source.

```
BOOLPARAM
TdsvSetAuxData (
       DSVAR dsvar,
       ADDRESS data,
       TDSVFREEFUNPTR freefcn)

   void
   freefcn (
          ADDRESS data)
```

TdsvSetAuxData associates a user-defined auxiliary data buffer, *data*, and its free function, *freefcn*, with the data source variable, *dsvar*. The auxiliary data buffer is created and maintained by the program for use by the functions in a function descriptor set. Setting *data* to *NULL* clears the data buffer.

The free function is optional. If it is specified, it is called automatically by TviCloseData, TdlCloseData, and TdsCloseData. The free function frees the buffer and clears the address. If a free function is not specified, the buffer remains unless freed by the data source variable or data source destroy function of the function descriptor set.

Returns *DV_SUCCESS* if the data buffer and free function are set successfully. Otherwise returns *DV_FAILURE* and aborts the changes.

TdsvSetFcnArg

Tdsv Functions T Routines

Sets an argument for a function associated with a data source variable.

```
BOOLPARAM
TdsvSetFcnArg (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype,
       int argindex,
       int type,
       ANYTYPE *valuep)
```

Error! Reference source not found.*TdsvSetFcnArg* sets an argument for a specific type of function within the function descriptor set. *dsvar* is the data source variable in a function data source and *fcntype* is the type of function. Valid types of functions are listed in TdsvSetFcnByName.

argindex is the index within the argument list. If the index does not refer to a current argument, it must refer to a new optional argument at the end of the list. *valuep* specifies the new value of the argument. *type* specifies the type of the argument, which you can change only if the argument is an optional argument rather than a required argument declared in the function descriptor set. Valid argument types are *V_T_TYPE* (text), *V_L_TYPE* (long), *V_D_TYPE* (double), or *V_DSV_TYPE* (data source variable). To delete an optional argument, use TdsvClrFcnArg.

Returns *DV_SUCCESS* if the argument was successfully set. Returns *DV_FAILURE* if no such function exists in the function descriptor set, *argindex* does not refer to an existing argument, *type* conflicts with the defined type of a required argument, or an error occurs.

TdsvSetFcnByName

Tdsv Functions T Routines

Sets the function associated with a data source variable.

```
BOOLPARAM
TdsvSetFcnByName (
       DSVAR dsvar,
       V_FDS_FCN_ENUM fcntype,
       char *fcnname)
```

TdsvSetFcnByName changes the function used by the data source variable for a specific type of function within the function descriptor set. *fcnname* is the descriptive name of the function. *dsvar* is the data source variable in a function data source and *fcntype* is the type of function to change. Valid types of functions are:

V_FDS_FCN_SELECT	The Select function, called by TviReadData, TdlReadData, or TdsReadData for each data source variable in a function data source, or by TdsvReadData for a particular data source variable.
V_FDS_FCN_SELECT_WRITE	The DSV-Write function, called by TdsWriteData for each data source variable in a function data source, or by TdsvWriteData for a particular data source variable.
V_FDS_FCN_DSV_CREATE	The DSV-Create function, called by TdsCreateDsVar.
V_FDS_FCN_DSV_DESTROY	The DSV-Destroy function, called by TdsvDestroy.

Returns *DV_SUCCESS* if the function is successfully changed. Returns *DV_FAILURE* if an error occurs.

TdsvSetGlobalFlag

Tdsv Functions T Routines

Sets the global flag for a data source variable.

```
BOOLPARAM
TdsvSetGlobalFlag (
       DSVAR dsvar,
       int flag)
```

TdsvSetGlobalFlag sets the value of the global flag for a data source variable, *dsvar*, to *flag*. The global flag controls whether or not the data source variable, if referenced by a subdrawing, can be mapped to another data source variable in the higher-level view. See VOsubdrawing for more information on mapping. Returns *DV_FAILURE* if *dsvar* is invalid. Otherwise returns *DV_SUCCESS*. Valid values for *flag* are:

V_GLOBAL	Can be mapped.
V_LOCAL	Cannot be mapped.

TdsvSetInitialValue

Tdsv Functions T Routines

Sets the initial value for a constant data source variable.

```
int
TdsvSetInitialValue(
       DSVAR dsvar,
       double initial_value)
```

TdsvSetInitialValue sets the initial value for a constant data source variable. It always returns *DV_SUCCESS.*

TdsvSetTypedValue

Tdsv Functions T Routines

Sets a value in a data buffer.

```
BOOLPARAM
TdsvSetTypedValue (
        DSVAR dsvar,
        int valtype,
        ADDRESS valptr,
        LONG row,
        LONG column)
```

Error! Reference source not found.*TdsvSetTypedValue* sets an element, identified by *row* and *column*, in the data buffer of the data source variable, *dsvar*, to the value pointed to by *valptr*. *row* and *column* are 0-based indices. *valtype* is a flag that indicates the type of datum pointed to. See *TdsvGetType* above for valid flag values for *valtype*. TdsvSetTypedValue treats *valptr* as a pointer to a value and puts the value in the dsvar's buffer, recasting the value to match the datum type of *dsvar*.

For example, if *valptr* points to 10.0, *valtype* is *V_F_TYPE*, and the *dsvar* is *V_C_TYPE*, the value 10 is put into the first byte of the dsvar's buffer. If *valtype* is *V_T_TYPE* and *dsvar* is *V_T_TYPE*, the routine copies as much of the string as fits into the *dsvar* buffer, starting at the position defined by *row* and *column*. Note that text dsvars are usually one dimension, so *row* is usually one. For scalar data, both *row* and *column* are zero.

Returns *DV_FAILURE* if *dsvar*, *valtype*, *row*, or *column* is invalid. Otherwise returns *DV_SUCCESS*. When the data type is *double*, TdsvSetValue can be used instead of this routine.

TdsvSetValue

Tdsv Functions T Routines

Sets a *double* in a data buffer.

```
BOOLPARAM
TdsvSetValue (
       DSVAR dsvar,
       double val,
       LONG row,
       LONG column)
```

Error! Reference source not found.*TdsvSetValue* sets an element in the data buffer of the data source variable to the specified value, *val*. When necessary, the value, which is passed as a *double*, is converted to match the data type of *dsvar*. The position of the element is identified by *row* and *column,* which are 0-based indices. Returns *DV_FAILURE* if *dsvar*, *row*, or *column* is invalid. Otherwise returns *DV_SUCCESS*.

TdsvValid

Tdsv Functions T Routines

Determines if a data source variable is valid.

```
BOOLPARAM
TdsvValid (
       DSVAR dsvar)
```

TdsvValid returns *DV_SUCCESS* if the data source variable is valid. Otherwise returns *DV_FAILURE*.

TdsvWriteData

Tdsv Functions T Routines

Writes data from one variable out to a target.

```
BOOLPARAM
TdsvWriteData(
        DSVAR dsvar)
```

TdsvWriteData calls a user-supplied write function to write the data from a data source variable out to another part of the application. Currently this routine works only for function data source variables that have a DSV-Write function assigned to them.

If you want to write data from all the variables in the function data source, you do not need to call this routine. Instead, you can call *TdsWriteData* by itself.

Returns *DV_FAILURE* if the data source is invalid or not open. Otherwise returns *DV_SUCCESS*.

Tlo (Tlocationobject)

Tlo Functions T Routines

Manages location objects. Location objects contain information about events generated by the graphical locator device. TloPoll, which returns a location object, is used only for simple event handling. For more information on manipulating location objects, including window system extension event handling, see VOlocation.

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	Tsc
Tdp	**Tlo**	Tvd
Tdr	Tob	Tvi

Tlo Functions

TloGetSelectedDrawport	Gets the drawport selected by the locator event.
TloGetSelectedObject	Gets the object selected by the locator event.
TloGetSelectedObjectName	Gets the name of the selected object.
TloGetSelectedSubObject	Gets the selected object or subobject in a subdrawing.
TloGetSelectedSubObjectName	Gets the name of selected object or subobject in a subdrawing.
TloPoll	Returns location object of next locator event in the event queue.
TloSetup	Sets up the values of a location object.
TloWinEventSetup	Sets up the values and *WINEVENT* structure of a location object.

TloGetSelectedDrawport

Tlo Functions T Routines

Gets the drawport selected by the locator event.

```
DRAWPORT
TloGetSelectedDrawport (
       OBJECT lo)
```

TloGetSelectedDrawport queries the location object, *lo*, returned by *VOloWinEventPoll*. Returns *NULL* if the cursor isn't in any drawport. Otherwise returns the drawport selected by the locator cursor.

TloGetSelectedObject

Tlo Functions T Routines

Gets the object selected by the locator event.

```
OBJECT
TloGetSelectedObject (
       OBJECT lo)
```

TloGetSelectedObject queries the location object, *lo*, returned by *VOloWinEventPoll*. Returns *NULL* if the cursor isn't pointing to any visible object. Otherwise returns the object selected by the locator cursor. If the pick is in a subdrawing, returns the subdrawing object

TloGetSelectedObjectName

Tlo Functions T Routines

Gets the name of the selected object.

```
char *
TloGetSelectedObjectName (
       OBJECT lo)
```

TloGetSelectedObjectName queries the location object, *lo*, returned by *VOloWinEventPoll*. Returns *NULL* if the cursor isn’t pointing to a visible named object. Otherwise returns the name of the object selected by the locator cursor.

This routine searches the drawing for the first named object at the cursor location. This object may be obscured by another object if the object in front is unnamed. Therefore, *TloGetSelectedObject* and *TloGetSelectedObjectName* may return different selected objects when called on the same location object.

TloGetSelectedSubObject

Tlo Functions T Routines

Gets selected object or subobject in a subdrawing.

```
OBJECT
TloGetSelectedSubObject (
       OBJECT lo)
```

TloGetSelectedSubObject works like *TloGetSelectedObject*, but for picks inside subdrawings, *TloGetSelectedSubObject* returns the selected object within the subdrawing. Nested subdrawings are traversed to the lowest level. Returns *NULL* if no visible object is selected.

TloGetSelectedSubObjectName

Tlo Functions T Routines

Gets name of selected object or subobject in a subdrawing.

```
char *
TloGetSelectedSubObjectName (
       OBJECT lo)
```

TloGetSelectedSubObjectName works like *TloGetSelectedObjectName*, but for picks inside subdrawings, *TloGetSelectedSubObjectName* returns the name of the selected object within the subdrawing. Nested subdrawings are traversed to the lowest level. Returns *NULL* if no visible object is selected.

TloPoll

Tlo Functions T Routines

Returns location object of next locator event in the event queue.

```
OBJECT
TloPoll (
       int poll_type)
```

TloPoll polls the locator device (mouse, tablet, etc.) attached to the current display device. Returns a corresponding location object which describes the position of the cursor and any key press that has occurred. For additional information on location objects, see *VOlocation*. The flag, *poll_type*, controls the type of polling. The possible values for the flag are:

LOC_POLL Returns the current location of the cursor and the last key press. If no selection was made, the last keypress is *NULL*. This flag makes *TloPoll* always return a valid *LOCATION*.

PICK_POLL Determines whether the user has made a selection. Returns a valid location object if one has been selected. Returns *NULL* if no selection was made.

WAIT_PICK Waits for the user to make a selection. Returns the current location of the cursor and the last keypress.

WAIT_CHANGE Waits for the user to move the cursor or make a selection. Returns the current location of the cursor and the last keypress.

Note that *TloPoll* is not appropriate for applications that require polling for a wider range of event types. For example, you cannot use *TloPoll* when you have button input objects, since they require button and key release events. For greater control over which events are polled, use *VOscWinEventMask* to set an event mask and *VOloWinEventPoll* or *VOscWinEventPoll* to poll. Using *TloPoll* after setting an event mask is not recommended since *TloPoll* resets the event mask internally.

TloSetup

Tlo Functions T Routines

Sets up the values of a location object.

```
BOOLPARAM
TloSetup (
       OBJECT lo,
       int key,
       DV_POINT *spt,
       OBJECT screen,
       DRAWPORT dp)
```

TloSetup sets a location object's key press, screen and drawport values, and location point in screen coordinates. This is used by the application program to create a location object as if it had been returned from *TloPoll*. If *dp* is *NULL*, it finds the top drawport that the screen point is in. Otherwise it associates the location object with the drawport. If *screen* is *NULL*, it assumes the screen is associated with the drawport. If both *screen* and *dp* are *NULL*, it assumes the current screen. Returns *DV_FAILURE* if *spt* is *NULL*. Otherwise returns *DV_SUCCESS*.

If your application runs on a window system, use *TloWinEventSetup* instead. It sets the key information accurately in cases where *key* and *keysym* values differ.

TloWinEventSetup

Tlo Functions T Routines

Sets up the values and *WINEVENT* structure of a location object.

```
BOOLPARAM
TloWinEventSetup (
       OBJECT lo,
       WINEVENT *we,
       OBJECT screen,
       DRAWPORT dp)
```

TloWinEventSetup sets a location object's *WINEVENT* structure, screen, and drawport values to those passed as parameters. It also calculates the key press, world coordinate, and screen coordinates based on the values in the fields of the *WINEVENT* structure passed in. You can use this routine to create a location object as though it had been returned from *VOscWinEventPoll* or *VOloWinEventPoll*. If *dp* is *NULL*, the routine finds the top drawport at the location given in the *WINEVENT* field *loc*. If *screen* is *NULL*, it assumes the screen associated with the drawport. If both *screen* and *dp* are *NULL*, it assumes the current screen. You must set the *type*, *loc*, and *button* or *firstchar* fields of the *WINEVENT* structure you pass in. If the event type is *V_BUTTONPRESS*, *V_BUTTONRELEASE*, *V_KEYPRESS*, or *V_KEYRELEASE*, the *key* field is set; otherwise the *key* field is not set. You can also set other fields of the *WINEVENT* structure. Currently always returns *DV_SUCCESS*.

Tob (Tobject)

Tob Functions T Routines

Access functions that work on objects that have subobjects. These include drawing objects, deque objects, and graphical objects. The *VOob* routines also act on general objects, and the *VO* routines act on specific objects.

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	Tsc
Tdp	Tlo	Tvd
Tdr	**Tob**	Tvi

Tob Functions

TobForEachSubobject	Traverses all subobjects in an object.
TobForEachVdp	Traverses all variable descriptors in an object.
TobWasSelected	Determines if an object was selected.

TobForEachSubobject

Tob Functions T Routines

Traverses all subobjects in an object.

```
ADDRESS
TobForEachSubobject (
       OBJECT object,
       TOBFOREACHSUBOBJFUNPTR fun,
       ADDRESS argblock)

   ADDRESS
   fun (
          OBJECT subobject,
          ADDRESS argblock)
```

TobForEachSubobject traverses all subobjects in the object and calls *fun* for each subobject. For example, if the object is a drawing, *fun* is called for each graphical object in the drawing. If the object is a graphical object such as an arc, *fun* is called for each control point. If the object is a subdrawing, *TobForEachSubobject* does not traverse objects in the subdrawing or any nested subdrawings. For a complete description of object subobjects, see the *VO Routines* chapter in this manual.

TobForEachSubobject continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the subobject being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each subobject in an object, or
2. to find a particular subobject.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the subobject is not found. Otherwise it should return the *ADDRESS* of the subobject.

Note: You should not alter the object by adding, deleting, or reordering its subobjects during traversal.

For an example of a typical function, see the example under *TdrForEachNamedObject*. Note that the example demonstrates the use of a function with three parameters, but *TobForEachSubobject* requires only two.

TobForEachVdp

Tob Functions T Routines

Traverses all variable descriptors in an object.

```
ADDRESS
TobForEachVdp (
       OBJECT object,
       TOBFOREACHVDPFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           OBJECT data_obj,
           VARDESC vdp,
           ADDRESS argblock)
```

TobForEachVdp traverses all variable descriptors in the object and calls *fun* for each variable descriptor pointer. If the object is a subdrawing, traverses all objects in the subdrawing and all levels of nested subdrawings for the variable descriptors of any embedded dynamics. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

For a description of *fun*, see *TobForEachSubobject*. Note that *TobForEachSubobject* traverses subobjects, passing two parameters to *fun*. *TobForEachVdp* traverses variable descriptors, passing three parameters to *fun*: the data object, the variable descriptor, and the argument block.

The *data_obj* parameter is the object that the variable descriptor belongs to. In the case of graphs or input objects, *data_obj* is the data group object (*dg*) or input object (*in*). In the case of dynamic control objects, *data_obj* is the threshold table object (*tt*) if there is one, or the variable descriptor object (*vd*) otherwise.

TobWasSelected

Tob Functions T Routines

Determines if an object was selected.

```
OBJECT
TobWasSelected (
       OBJECT object,
       OBJECT lo)
```

TobWasSelected determines if an object was selected by the location object, *lo*. Returns *object* if it was selected. Otherwise returns *NULL*. In some cases, an object drawn in an overlapping drawport might obscure the object you were initially selecting with *lo*. Therefore, the object being checked must have been drawn in the drawport returned by *TloGetSelectedDrawport*, or the function is not defined.

Tproto

Tproto Functions T Routines

Example

Displays prototypes created in DV-Draw.

These routines let you activate a prototype within a DV-Tools program. The prototype runs exactly as it does in the Prototype Menu of DV-Draw or when using *DVproto*, but you can control its environment.

To define a prototyping environment, you must specify the name of your top view, the screen you want to run the prototype in, and the drawport attributes for displaying the views. The drawport attributes include where on the screen you want to display the views and what portion of the views you want visible. They also include a stretch flag that controls whether TdpCreate or TdpCreateStretch is used to create the drawport. For more details, see *Tdrawport*.

You can invoke a prototype from DV-Tools in two ways:

TprotoRun invokes a prototype like using the *DVproto* script. You don’t return from this call until a quit rule or window quit event occurs. This method is useful when you want the prototype to be the only active function.

You can also call several *Tproto* functions within your application to invoke a prototype. This method gives you the most control; you can have several active prototypes, and you can do your own event polling and define your own update rates. When running a prototype this way, you must set up and save the prototype environment information in the *PROTO_ENV* private structure. Use the following steps:

To define a *PROTO_ENV* structure, call *TprotoInit*.

To process a location object, call TprotoHandleInput.

To update dynamics, call TprotoUpdate.

To stop the prototype arbitrarily or to clean up after a quit rule or window quit event, call TprotoCleanup.

TInit, TTerminate	Tds	**Tproto**
Tdl	Tdsv	Tsc
Tdp	Tlo	Tvd
Tdr	Tob	Tvi

Tproto Functions

TprotoCleanup	Cleans up after running a prototype.
TprotoHandleInput	Handles events for a prototype.
TprotoInit	Initializes the prototype environment.
TprotoRedraw	Redraws a prototype.
TprotoReset	Resets a prototype.
TprotoRun	Runs a prototype like using the *DVproto* script.
TprotoUpdate	Updates the dynamics for the prototype.

TprotoCleanup

Tproto Functions T Routines

Example

Cleans up after running a prototype.

```
void
TprotoCleanup (
        PROTO_ENV proto_env)
```

TprotoCleanup cleans up the prototyping environment. You must call *TprotoCleanup* to clean up if you called TprotoInit to start the prototype. For example, call *TprotoCleanup* after TprotoHandleInput returns *V_TPROTO_QUIT.*

TprotoHandleInput

Tproto Functions T Routines

Example

Handles events for a prototype.

```
int
TprotoHandleInput (
       PROTO_ENV proto_env,
       OBJECT location)
```

TprotoHandleInput handles events for the prototyping environment. *location* is the location object containing the event. You should determine that the location object is not associated with another screen or drawport before passing it.

Handles resize and expose events by calling TprotoReset and TprotoRedraw. Note that if the prototype screen contains other drawports, you should handle the event by calling TprotoReset and TprotoRedraw for the prototype, and TdpRedraw for each of the other drawports. Processes rules in the prototype and executes actions as specified by the event and condition. Also calls VUerHandleLocEvent internally to update input objects. Note that event requests posted by other parts of the application may be serviced when you call this routine because of the internal call to VUerHandleLocEvent.

Returns *DV_SUCCESS* if the location object was used by a rule, input object, or event request. Returns *V_TPROTO_QUIT* for a quit rule action or quit window event. Otherwise returns *DV_FAILURE*. You should check this return value to determine whether the location object was used; if not, you may have to handle the location object explicitly.

TprotoInit

Tproto Functions T Routines

Example

Initializes the prototype environment.

```
PROTO_ENV
TprotoInit (
       OBJECT screen,
       char *top_view,
       DRAWPORT_ATTRIBUTES *dp_atts)
```

TprotoInit initializes a prototyping environment and returns a *PROTO_ENV* structure. It also loads and displays the *top_view* into a drawport defined by *dp_atts*. It preloads views according to the *DVPRELOAD* configuration variable. This routine sets the cursor to the arrow cursor, *V_ACTIVE_CURSOR*. Returns *NULL* if the top view cannot be loaded.

TprotoRedraw

Tproto Functions T Routines

Example

Redraws a prototype.

```
void
TprotoRedraw (
       PROTO_ENV proto_env)
```

TprotoRedraw redraws the prototype. If you are handling your own window events and the screen contains more than one drawport, call this function after a *V_RESIZE* event and after calling TprotoReset, or after a *V_EXPOSE* event. Note that if the prototype screen contains other drawports, you should also call TdpRedraw for each of the other drawports. *TprotoRedraw* also calculates new rasters for popup and overlay objects. Redraws only the prototype drawport, not the whole screen. This function is called for you by TprotoHandleInput when its location object contains a *V_RESIZE* or *V_EXPOSE* event.

TprotoReset

Tproto Functions T Routines

Example

Resets a prototype.

```
void
TprotoReset (
       PROTO_ENV proto_env)
```

TprotoReset resets the prototype. Should be called after a *V_RESIZE* if the screen contains more than one drawport. This function is called for you by TprotoHandleInput when its location object contains a *V_RESIZE* event.

TprotoRun

Tproto Functions T Routines

Example

Runs a prototype like using the *DVproto* script.

```
void
TprotoRun (
       OBJECT screen,
       char *top_view,
       DRAWPORT_ATTRIBUTES *dp_atts)
```

TprotoRun runs a prototype just like using *DVproto*. It handles user events and updating the screen. This function doesn't return until a quit is generated through either a rule or a window event.

TprotoUpdate

Tproto Functions T Routines

Example

Updates the dynamics for the prototype.

```
void
TprotoUpdate (
        PROTO_ENV proto_env)
```

TprotoUpdate calls TdpDrawNext to update the visible objects in the prototype. This function does not update when a stop dynamics rule is active.

Tproto Example

The following code fragment, adapted from *proto_multi.c*, shows how to run two prototypes in two separate windows.

```
/* Initialize the window and prototype environments. */
for (i=0; i<MAXWINS; i++)
    {
    /* Create the windows and set up polling. */
    screen[i] = SetupScreen (i);

    /* Initialize the prototype environment to use a stretched drawport. */
    dp_atts.vvp = NULL;
    dp_atts.wvp = &whole_world;
    dp_atts.stretch_flag = (DV_BOOL)YES;
    proto_env[i] = TprotoInit (screen[i], view_name[i], &dp_atts);
    }

. . .

/* Main loop. Handle events and update dynamics. */
while (quit_status == NO)
    {
    /* Handle events. */
    if (location = VOloWinEventPoll (V_NO_WAIT))
        {
        VOscSelect (current_screen =
                              VOloScreen (location));
        i = (current_screen == screen[0]) ? 0 : 1;
        if (TprotoHandleInput (proto_env[i], location) == V_TPROTO_QUIT)
            quit_status = YES;
        }

    /* Update each prototype's dynamics if we didn't quit. */
    if (quit_status != NO)
        {
        for (i=0; i<MAXWINS; i++)
            TprotoUpdate (proto_env[i]);
        }
    }
/* End of main loop. */

. . .

/* Clean up. */
for (i=0; i<MAXWINS; i++)
    {
    VOscSelect (screen[i]);
    TprotoCleanup (proto_env[i]);
    TscClose (screen[i]);
    }
```

Tsc (Tscreen)

Tsc Functions T Routines

T level routines for managing screen objects (*sc*). These routines perform higher-level operations on screen objects than the *VOsc* routines. In particular, most of them take a screen object as a parameter rather than operating on the current screen. The screen object is the highest level object in the DV-Tools hierarchy of data structures. It represents the entire display device, or window in a windowing system, and maintains a list of the drawports (*dp*) it contains.

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	**Tsc**
Tdp	Tlo	Tvd
Tdr	Tob	Tvi

Tsc Functions

TscClose	Closes a screen object's associated display device.
TscCloseCurrentScreen	Closes the current display screen.
TscDefBackcolor	Sets the default background color for the screen.
TscDefForecolor	Sets the default foreground color for the screen.
TscDrawBackground	Repairs all or part of the screen by drawing with the background color.
TscErase	Erases the entire screen by drawing with the background color.
TscFindDrawport	Finds out which drawport a given point is in.
TscFlush	Flushes a screen object's associated display device.
TscFlushCurrentScreen	Flushes output to the screen.
TscOpen	Opens a device as a screen object.
TscOpenError	Checks for any case where TscOpen might return a NULL screen object.
TscOpenRemoteWindow	Specifies a remote display connection pointer.
TscOpenSet	Opens a device using specified attributes.
TscOpenWindow	Opens a window as a screen object.
TscPrintEnd	Ends printing on Microsoft Windows systems.
TscPrintSet	Sets up printer attributes on Microsoft Windows systems.
TscPrintStart	Starts printing on Microsoft Windows systems.
TscRedraw	Redraws all drawports in the screen.
TscReset	Resets all screen drawports after window resizing.
TscSetCurrentScreen	Sets currently active screen.

TscClose

Tsc Functions T Routines

Closes a screen object's associated display device.

```
BOOLPARAM
TscClose (
       OBJECT screen)
```

TscClose closes the display device associated with the given screen object, *screen*, and any attached drawports, freeing the device for later calls to TscOpen or TscOpenWindow. Returns *DV_FAILURE* if *screen* is *NULL*. Otherwise returns *DV_SUCCESS*.

TscCloseCurrentScreen

Tsc Functions T Routines

Closes the current display screen.

```
BOOLPARAM
TscCloseCurrentScreen (void)
```

TscCloseCurrentScreen flushes pending output to the currently active screen, closes polling, and closes the screen. Currently, this routine always returns *DV_SUCCESS*.

TscDefBackcolor

Tsc Functions T Routines

Sets the default background color for the screen.

```
OBJECT
TscDefBackcolor (
       OBJECT screen,
       OBJECT color)
```

TscDefBackcolor sets the screen object's default background color. Returns its original default background color. If *screen* is *NULL*, returns the current background color. The initial default background color of a screen is *NULL*.

TscDefForecolor

Tsc Functions T Routines

Sets the default foreground color for the screen.

```
OBJECT
TscDefForecolor (
       OBJECT screen,
       OBJECT color)
```

TscDefForecolor sets the screen object's default foreground color. Returns its original default foreground color. If *screen* is *NULL*, returns the current foreground color. The initial default foreground color of a screen is *NULL*.

TscDrawBackground

Tsc Functions T Routines

Repairs all or part of the screen by drawing with the background color.

```
BOOLPARAM
TscDrawBackground (
       OBJECT screen,
       RECTANGLE *svp)
```

TscDrawBackground draws over the portion of the screen specified by *svp* using the default background color. This has the effect of erasing the specified region. If *svp* is *NULL*, draws over the entire screen. Currently, this routine always returns *DV_SUCCESS*.

TscErase

Tsc Functions T Routines

Erases the entire screen by drawing with the background color.

```
BOOLPARAM
TscErase (
       OBJECT screen)
```

TscErase erases the screen by drawing over it using the default background color. If the screen's default background color is *NULL*, draws using color index 0. This color is usually black for color devices and white for black-and-white devices. Input objects are erased from the screen, but they remain active, responding to input, unless they are erased explicitly using TdpEraseObject. Currently, this routine always returns *DV_SUCCESS*.

TscFindDrawport

Tsc Functions T Routines

Finds out which drawport a given point is in.

```
DRAWPORT
TscFindDrawport (
       OBJECT screen,
       DV_POINT *spt)
```

TscFindDrawport returns the drawport containing a given screen coordinate point structure, *spt*. Returns *NULL* if the point is not in any drawport.

TscFlush

Tsc Functions T Routines

Flushes a screen object's associated display device.

```
BOOLPARAM
TscFlush (
       OBJECT screen)
```

TscFlush flushes any pending output to the given screen. Currently, this routine always returns *DV_SUCCESS*.

TscFlushCurrentScreen

Tsc Functions T Routines

Flushes output to the screen.

```
BOOLPARAM
TscFlushCurrentScreen (void)
```

TscFlushCurrentScreen flushes any pending output to the current or active screen. Currently, this routine always returns *DV_SUCCESS*.

TscOpen

Tsc Functions T Routines

Opens a device as a screen object.

```
OBJECT
TscOpen (
       char *device,
       char *clutfile)
```

TscOpen opens the device, *device*, giving it the specified color lookup table, *clutfile*, and returns its associated screen object. If *device* is *NULL*, the value (if set) of the configuration variable *DVDEVICE* is used. If *clutfile* is *NULL*, the value (if set) of the configuration variable *DVCOLORTABLE* is used. Otherwise, the default color lookup table is used. The *clutfile* format is a sequence of ASCII triples consisting of the red, green, and blue components of the color lookup table entries, with one line per entry in the table. The color components should be in the range [0,255]. A red component of -1 means that the entry should remain unchanged. Unspecified indices remain unchanged. Returns *DV_FAILURE* if it cannot open *device* or *clutfile*.

TscOpenError

Tsc Functions T Routines

Checks for any case where TscOpen might return a NULL screen object.

```
INT
TscOpenError (void)
```

TscOpenError checks for any case where *TscOpen* might return a *NULL* screen object. If *TscOpen* returns a *NULL* screen object, such as when the software protection check fails, *TscOpen* no longer returns a valid screen object. This means that the system may open the window, do the protection check, and then immediately close the window due to a failed check. (The device must be opened so that the floating license option can correctly identify the display.) *TscOpenError* returns an integer from 1 to 9 representing possible error causes. The following code fragment shows the use of this routine:

```
screen = TscOpen( device_name, NULL );
       if ( ! screen )
          error_code = TscOpenError();
```

The return value has the following meanings:

0	Screen was successfully opened - no error.
1	Unknown device name passed to TscOpen.
2	Could not find the specified color table file.
3	Could not open screen - driver level failure.
4	The DataViews logical device table is full.
5	Protection failure - couldn’t locate/decode license file.
6	Protection failure - failed basic protection check.
7	Protection failure - failed DataViews-specific protection check.
8	Protection failure - error involving HP ID module.
9	Protection failure - failure to acquire floating license.

TscOpenRemoteWindow

Tsc Functions T Routines

Specifies a remote display connection pointer.

```
OBJECT
TscOpenRemoteWindow (
       char *device,
       LONG display,
       LONG windowid,
       char *clutfile)
```

TscOpenRemoteWindow lets you specify a remote display connection pointer for opening windows on remote displays. *device* is the name of the device to open. *display* is a pointer to a remote display. *windowid* identifies a window system that has already been created. *clutfile* is the name of the file containing a color lookup table. If *device* is *NULL*, the value (if set) of the configuration variable *DVDEVICE* is used. If *clutfile* is *NULL*, the value (if set) of the configuration variable *DVCOLORTABLE* is used. Otherwise the default color lookup table is used. Returns *DV_FAILURE* if it cannot open *device* or *clutfile*. This routine is only useful with X11.

TscOpenSet

Tsc Functions   T Routines

Opens a device using specified attributes.

```
OBJECT
TscOpenSet (
       char *dev_name,
       char *clutfile,
           ULONG flag, <type> value,
           ULONG flag, <type> value,
           ...,
       V_END_OF_LIST)
```

TscOpenSet opens the device, *dev_name*, specifies the color lookup table, *clutfile*, sets device attributes, and returns a new screen object representing that device. Returns *NULL* if it cannot open the screen.

The device attributes are set using a variable length argument list of attribute/value pairs. Each pair of parameters starts with an attribute flag that specifies the particular attribute of the device to be set. The second argument sets the value of the attribute. The list must terminate with *V_END_OF_LIST* or *0*.

For example, to open a screen as an X11 window 800 pixels high by 600 pixels wide, with an upper left position of (100, 100) relative to the screen origin, you could call:

```
screen = TscOpenSet ("X", (char *) NULL,
                      V_WINDOW_X, 100, V_WINDOW_Y, 100,
                      V_WINDOW_WIDTH, 800, V_WINDOW_HEIGHT, 600,
                      V_END_OF_LIST);
```

To open a DataViews screen on an existing window, use the appropriate attribute flags to pass the window id and display id. For example:

```
screen = TscOpenSet (device, (char *) NULL,
                      V_DISPLAY, display, V_WINDOW_ID, window,
                      V_END_OF_LIST);
```

Attribute Flags

The attribute flags are optional; when attributes are not set, defaults are used. Not all attribute flags apply to all DataViews drivers since these attributes can only be set on certain devices. These flags are also used by Gropen_set, GRset, Vuopendev_set, VOscOpenClutSet, and VOscOpenSet, and are defined in the header file *dvGR.h*.

Attribute Flags	Description
V_WINDOW_WIDTH	Width of window in pixels. Takes an *int* argument.
V_WINDOW_HEIGHT	Height of window in pixels. Takes an *int* argument.
V_WINDOW_NAME	Title of window for window systems which have a title bar. Takes a *char ** argument.
V_WINDOW_X	The *x* coordinate position of the window's upper left corner relative to the parent window. Takes an *int* argument.
V_WINDOW_Y	The *y* coordinate position of the window's upper left corner relative to the parent window. Takes an *int* argument.
V_DRAW_FUNCTION	Drawing mode. Valid values are *V_COPY* (normal draw) and *V_XOR* (draw by reversing bits, applicable to rubberbanding). Takes a *LONG* argument.

Window System Data Structures

Flag	Description
V_WINDOW_ID	Identifier or "handle" for the window maintained by the current screen. Takes a *Window* argument for X11.
V_DISPLAY	The id or data structure for maintaining the network connection for window systems with network-based display (currently only X11). Takes a *Display* * argument.
V_ICON_NAME	Title of the icon for systems with an icon title bar. Takes a *char* * argument.
V_MOTION_COLLAPSE	Collapses all successive motion notify events to a single event. Default is *YES*. Takes a *BOOLPARAM* argument.
V_EXPOSE_COLLAPSE	Collapses all successive expose events to a single event. Default is *YES*. Takes a *BOOLPARAM* argument.

DataViews Pre-Defined Cursors

If using *WINEVENT* polling routines, DataViews cursors must be switched explicitly.

Flag	Description
V_ACTIVE_CURSOR	Sets the DataViews active cursor, the arrow. Doesn't take an argument.
V_INITIAL_CURSOR	Sets the DataViews initial cursor, the DV logo. Doesn't take an argument.

Microsoft Windows-Specific Data Flags:

These flags are also discussed in the *DataViews for DataViews Installation and System Administration Manual, Windows Version.*

Flag	Description
V_WIN32_WINDOW_HANDLE	Sets the window handle. Takes an *HWND* argument.
V_WIN32_NEWFONT	Sets the four DataViews hardware fonts. The fonts increase in size; the smallest is associated with *1*, the largest with *4*. Indices that are not set programmatically use the fonts specified in the *DV.INI* file if there is one. To maintain consistent sizes and styles, set all four fonts. Takes two arguments: an *int* specifying the index and an *HFONT*.
V_WIN32_DOUBLE_BUFFER	Double-buffering status of the window. Default is *YES*. Takes a *BOOLPARAM* argument.
V_WIN32_ICON_NAME	Identification of the icon. Takes a *char* * argument.
V_WIN32_XORFLAG	Win32 raster-operation code for XOR objects. Default is *R2_XORPEN*. Takes an *int* argument. For a list of valid values, see the Win32 documentation for *SetROP2*.
V_WIN32_HPALETTE	Handle to a logical palette. Lets you pass the Windows equivalent of a color table. The logical palette must have 256 colors or less. Takes an *HPALETTE* argument.

X11-Specific Data Structures

Some of these flags are discussed in more detail in the *DataViews and the View Widget in the X Environment Manual.*

Flag	Description
V_X_WINDOW_ID	Same as *V_WINDOW_ID*. Takes a *Window* argument.
V_X_DISPLAY	Same as *V_DISPLAY*. Takes a *Display* * argument.
V_X_DISPLAY_NAME	Character string giving the name of an X11 remote display, for opening an X11 window on a remote server. The string has the form: **UNIX:** *hostname:server.screen* **OpenVMS:** *hostname::server.screen* where *hostname* is the network name of the remote machine, *server* is the server number, and *screen* is the screen number on which to display the window. These last two numbers are usually zero. Takes a *char* *

	argument.
V_X_APPLIC_CONTEXT	The application context for the device. Ignored when widgets are passed. Within an application, all devices use the application context of the first device. Takes an *XtAppContext* argument.
V_X_DRAW_WIDGET	The widget passed to display DataViews. Can be a form widget or a widget of any other composite widget subclass. Takes a *Widget* argument.
V_X_CURSOR	X Window system representation of the current cursor. Takes a *Cursor* argument.
V_X_APPLIC_CLASS	The generic application class for this application. The application class of the first device is assigned to all subsequent devices. Takes a *char* * argument.
V_X_APPLIC_NAME	The specific application name for this device. Controls which set of defaults the window reads from the resource database and X defaults files. Takes a *char* * argument.

Flag	**Description**
V_X_ICON	X Window system representation for the current icon in the X bitmap format. Requires that you set *V_X_ICON_WIDTH* and *V_X_ICON_HEIGHT*. Takes a *char* * argument.
V_X_ICON_WIDTH	Width of the X icon. Takes an *int* argument.
V_X_ICON_HEIGHT	Height of the X icon. Takes an *int* argument.
V_X_ICON_X, V_X_ICON_Y	Control the *x* and *y* position of the iconified window, though the window manager may override the settings. Each flag takes an *int* argument.
V_X_ICONIC	Controls whether the window is drawn initially in an iconified state. Default is *NO*. Takes a *BOOLPARAM* argument.
V_X_EXPOSURE_BLOCK	Controls whether *TscOpenSet* blocks (waits for) the expose event before returning. Applies only to the initial expose event for internally created windows. If *YES*, the device is ready for drawing when *TscOpenSet* returns. If *NO*, your application should wait for an expose event before drawing on the device. Default is *NO*. Takes a *BOOLPARAM* argument.
V_X_RESIZE_BLOCK	Controls whether *GRset* blocks (waits for) the resize and expose events before returning after an explicit resize. If *YES*, your application should follow up immediately with calls to *TscReset* and *TscRedraw*. If *NO*, your application should wait for resize and expose events before drawing on the device. Default is *NO*. Takes a *BOOLPARAM* argument.

Flag	**Description**
V_X_FONTSTRUCT	Specifies the font corresponding to a 1-based index of fonts used for text. The fonts increase in size; the smallest is associated with *1*, the largest with *4*. Indices that are not set programmatically use the fonts specified in the *DVfonts* file if there is one, or in the resource file. To maintain consistent sizes and styles, set all four indices. Takes two arguments: an *int* argument specifying the index and an *XFontStruct* *. For example: *TscOpenSet (... V_X_FONTSTRUCT, 1, small_fontstr_ptr ...*
V_X_DOUBLE_BUFFER	If *YES*, graphics are written to an off-screen pixmap which is copied to the screen whenever *GRflush* is called. Reduces flicker but may slow down drawing speed. Default is *NO*. Takes a *BOOLPARAM* argument. If you are using double buffering with the OPEN LOOK server, you should also set *V_X_RAS_SYNC* to *YES*.
V_X_RAS_SYNC	If *YES*, forces an *XSync* call after every raster drawing. Ensures that all raster draws occur when many are done in rapid succession. Default is *NO*. Takes a *BOOLPARAM* argument.
V_X_POLY_HINT	Specifies the shape of polygons so the X driver can optimize its

performance. If all polygons in the application are non-self-intersecting, specify *Nonconvex* to achieve faster drawing. If all polygons are both non-self-intersecting and convex, specify *Convex* for even faster drawing. Default is *Complex*. Takes an *int* argument.

V_X_IMAGE_STRING	If *YES*, text is drawn on a filled rectangle drawn in the background color. If *NO*, the text is drawn directly on top of the existing graphics. Default is *YES*. Takes a *BOOLPARAM* argument.
V_X_DASH_STYLE	Specifies how gaps in a dashed line are drawn. Valid values are: *LineOnOffDash* (gaps are not drawn, so the underlying graphics are visible) or *LineDoubleDash* (the gaps are drawn using the current background color). Default is *LineOnOffDash*. Takes an *int* argument.
V_X_COLORMAP	The X colormap for the device. Lets you supply a shared colormap to avoid color swapping problems. Takes a *Colormap* argument.
V_X_PIXELS	Array of X pixels corresponding to the indices in the color table. Forces use of these pixels, taking precedence over any other method for setting colors. Takes two arguments: an *int* argument specifying the number of pixels and an *unsigned long[]*. For example: *TscOpenSet (... V_X_PIXELS, 128, pixels ...*
V_X_PLANES	Array of X plane masks corresponding to the color planes of the pixels. You must supply these masks if you are planemasking with pixels supplied using *V_X_PIXELS*. Takes two arguments: an *int* argument specifying the number of masks and an *unsigned long[]*. For example: *TscOpenSet (... V_X_PLANES, 7, masks ...*

V_X_COLORMAP, *V_X_PIXELS*, and *V_X_PLANES* give you more control over the color structures used by the X driver, but also require a deeper understanding of how X and DataViews work together. For a more detailed explanation, see the GRget description.

TscOpenWindow

Tsc Functions T Routines

Opens a window as a screen object.

```
OBJECT
TscOpenWindow (
       char *device,
       int windowid,
       char *clutfile)
```

TscOpenWindow opens the given window as a DV-Tools device, *device*, giving it the specified color lookup table, *clutfile*, and returns the screen object. If *device* is *NULL*, the value of the configuration variable *DVDEVICE* is used. If *clutfile* is *NULL*, the value of the configuration variable *DVCOLORTABLE* is used. Otherwise the default color lookup table is used. *windowid* is the handle used by the window system to refer to the window. The window must have been created by the application programmer using the local window system creation routines. The DataViews display device driver must be configured for multiple windows, or an error occurs. The DataViews driver is configured to allow a maximum number of 10 open windows. Exceeding this limit causes an error. Returns *DV_FAILURE* if it cannot open *device* or *clutfile*.

TscPrintEnd

Tsc Functions T Routines

Ends printing on Microsoft Windows systems.

```
void
TscPrintEnd(
       OBJECT screen)
```

TscPrintEnd stops the program from sending the graphics to the printer and resumes sending them to the monitor. Any subsequent calls, such as TscRedraw, are directed to the monitor.

TscPrintSet

Tsc Functions T Routines

Sets up printer attributes on Microsoft Windows systems.

```
ADDRESS
TscPrintSet(
          int flag, <type> value,
          int flag, <type> value,
          ...,
       V_END_OF_LIST)
```

TscPrintSet sets up a structure containing information for printing. To specify the information, pass flag-value pairs to *TscPrintSet,* then terminate the parameter list with *V_END_OF_LIST*. You do not have to set all the attributes since all attributes have system default values. You can also set attribute values in the *DV.INI* file instead of in your program. Values set in the *DV.INI* file override the system defaults.

TscPrintSet creates an internal structure and returns a pointer to this structure. The structure is destroyed when you call TscPrintStart.

The following table lists the flags and their definitions:

Flag	**Definition**
VP_PRINT_SCALE	Specifies the size of the printed image on the page. A value of 100 makes the image take up the full 8.5x11 page. The aspect ratio of the screen is maintained in the printed image. The origin for printing is the upper left corner. The value must be an integer. The default value is *100.*
VP_PRINT_ORIENTATION	Specifies the page direction. Valid values are *DV_LANDSCAPE* and *DV_PORTRAIT*. The default value is *DV_PORTRAIT.*
VP_PRINT_DRIVER	Specifies which printer driver is called. The value is type *char* *. The default value is the default driver for your system.
VP_PRINT_PORT	Specifies the I/O channel. The value is type *char* *. The default value is the default port for your system.
VP_PRINT_DEVICE	Specifies the printer name. The value is type *char* *. The default value is the default printer name for your system.
VP_PRINT_QUALITY	Specifies the quality used for printing the image. Valid values are *DV_DRAFT, DV_LOW, DV_MEDIUM,* and *DV_HIGH.* The default value is *DV_MEDIUM.*
VP_PRINT_NO_WARNING	Specifies whether or not to show warnings when an incorrect print setting is overruled in favor of a system default setting that works. The default value is *FALSE.*
VP_PRINT_DOCUMENT_NAME	Specifies name used for the print job. The value is type *char* *. The default value is the default job name for your system.

The *VP_PRINT** flags are defined in *dvGR.h*. The *DV_** flags are defined in *dvstd.h*.

TscPrintStart

Tsc Functions T Routines

Starts printing on Microsoft Windows systems.

```
void
TscPrintStart(
       OBJECT screen,
       ADDRESS pr_struct)
```

TscPrintStart starts the printing process for a screen. After this call, any calls that affect the graphics do not change the appearance on the monitor, but instead go to a printer. The printer is specified in *pr_struct*, a structure that you must create by using *TscPrintSet* before you call this routine. This call is normally followed by a call to TscRedraw, which sends the entire screen image to the printer. To end printing, call TscPrintEnd.

TscRedraw

Tsc Functions T Routines

Redraws all the drawports in the screen.

```
BOOLPARAM
TscRedraw (
       OBJECT screen,
       RECTANGLE *svp)
```

TscRedraw erases and then redraws the contents of all drawports in the given screen viewport rectangle, *svp*. If *svp* is *NULL*, the entire screen is redrawn. If screen is *NULL*, the current screen is used. The screen itself is erased by drawing the screen's default background color over the entire screen. If the value of the default background color is *NULL*, the screen is erased using color index zero. Drawports within the screen are erased using the background colors of their views. Objects that were drawn using TdpDrawObject are not redrawn. Currently, this routine always returns *DV_SUCCESS*.

TscReset

Tsc Functions T Routines

Resets all screen drawports after window resizing.

```
BOOLPARAM
TscReset (
       OBJECT screen)
```

TscReset recalculates the dimensions of each drawport in the screen after resizing the window in which the application is running. Since a drawport's screen viewport rectangle is specified in virtual coordinates, its physical dimensions and aspect ratios change in proportion to that of the window. Does not redraw the screen. Currently, this routine always returns *DV_SUCCESS*.

TscSetCurrentScreen

Tsc Functions T Routines

Sets currently active screen.

```
OBJECT
TscSetCurrentScreen (
       OBJECT screen)
```

TscSetCurrentScreen sets the currently active screen and returns the previously active screen. If *screen* is *NULL*, returns the object id of the currently active screen.

Tvd (Tvariabledescriptor)

Tvd Functions T Routines

Accesses the display variables associated with drawing objects.

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	Tsc
Tdp	Tlo	**Tvd**
Tdr	Tob	Tvi

Tvd Functions

TvdGetDataSourceVariable	Gets the data source variable.
TvdPutBuffer	Sets a new variable descriptor buffer.
TvdPutDataSourceVariable	Binds the display variable to a data source variable.

TvdGetDataSourceVariable

Tvd Functions T Routines

Gets the data source variable.

```
DSVAR
TvdGetDataSourceVariable (
       VARDESC vdp)
```

TvdGetDataSourceVariable queries the variable descriptor, *vdp*, to determine which data source variable it is linked to. Returns *DV_FAILURE* if *vdp* is invalid or not bound to a data source variable. Otherwise returns the data source variable.

TvdPutBuffer

Tvd Functions T Routines

Sets a new variable descriptor buffer.

```
ADDRESS
TvdPutBuffer (
       VARDESC vdp,
       ADDRESS newbuffer)
```

TvdPutBuffer sets the data buffer of the variable descriptor, *vdp*, to *newbuffer*. Rebinding must be done before the call to TdpDraw. Returns *DV_FAILURE* if *vdp* is invalid. Otherwise returns the *ADDRESS* of the previous buffer binding.

TvdPutDataSourceVariable

Tvd Functions T Routines

Binds the display variable to a data source variable.

```
DSVAR
TvdPutDataSourceVariable (
       VARDESC vdp,
       DSVAR dsvar)
```

TvdPutDataSourceVariable binds the variable descriptor, *vdp*, to the data source variable, *dsvar*. After this binding, the display variable gets its data from the new data source variable. Returns *DV_FAILURE* if the previous binding was not to a data source variable or if *vdp* or *dsvar* are invalid. Otherwise returns the previous binding.

Tvi (Tview)

Tvi Functions T Routines

View access functions. The view is composed of a drawing object and a data source list. The drawing contains all of the graphical objects that appear on the screen; the data source list contains the data sources that supply the data required to make the drawing dynamic. This module contains routines for getting, setting, and manipulating the view and its components.

The main functions for saving a view are TviSave, which saves a view in binary, and TviASCIISave, which saves a view in ASCII. The main function for loading, TviLoad, detects if the viewfile is binary or ASCII and loads it accordingly. Additional functions include TviFileSave and TviFileLoad which save or load a view from a view file that is already open and TviMemSave, TviASCIIMemSave, and TviMemLoad, which save or load a view from memory. Loading a view also recursively loads any views referenced by subdrawings in the view.

TInit, TTerminate	Tds	Tproto
Tdl	Tdsv	Tsc
Tdp	Tlo	Tvd
Tdr	Tob	**Tvi**

Tvi Functions

TviASCIIMemSave	Saves a view in ASCII format to a memory buffer.
TviASCIISave	Saves a view as an ASCII format file.
TviClone	Makes a deep copy of a view.
TviCloseData	Closes the data sources in a view.
TviConvertDynamics	Converts a view with pre-8.0 dynamics to use post-8.0 dynamics.
TviCreate	Creates a view.
TviDestroy	Destroys a view, freeing its memory.
TviExciseDrawing	Removes objects in a drawing from a view.
TviFileLoad	Loads a view from an open file.
TviFileSave	Saves a view to an open file.
TviForEachDataSource	Traverses the data sources of a view.
TviForEachVar	Traverses the data source variables of a view.
TviGetComment	Gets the comment field of the view.
TviGetDataSourceList	Gets a view's data source list.
TviGetDrawing	Gets a view's drawing object.
TviLoad	Loads a new view in from a file.
TviMemLoad	Loads a view from memory.
TviMemSave	Saves a view in binary format to a memory buffer.
TviMergeAddDataSources	Looks for data source list match and adds if necessary.
TviMergeDataSources	Looks for data source list match with no add option.
TviMergeDrawing	Merges a drawing's objects into a view.
TviOpenData	Opens the data sources of a view.
TviPutComment	Sets the comment field of the view.
TviPutDataSourceList	Replaces a view's data source list.
TviPutDrawing	Replaces a view's drawing.
TviReadData	Reads data from the data sources of a view.
TviSave	Saves a view as a binary format file.
TviTestDynamics	Tests a view for pre-8.0 dynamics.

TviASCIIMemSave

Tvi Functions T Routines

Saves a view in ASCII format to a memory buffer.

```
BOOLPARAM
TviASCIIMemSave (
        VIEW view,
        char **bufferpp,
        int *sizep)
```

TviASCIIMemSave stores a view in ASCII format into a memory buffer allocated by this function. *bufferpp* is a pointer to a character pointer which stores the location of the allocated buffer. *sizep* is a pointer to an integer which stores the size of the buffer. *TviASCIIMemSave* is useful in applications such as a network server or database where you might want to pass views in memory between applications. The user is responsible for freeing this buffer. Returns *DV_FAILURE* if passed an invalid view or cannot allocate enough memory. Otherwise returns *DV_SUCCESS*. See also TviMemSave and TviMemLoad.

TviASCIISave

Tvi Functions T Routines

Saves a view as an ASCII format file.

```
BOOLPARAM
TviASCIISave (
       VIEW view,
       char *filename)
```

TviASCIISave saves the view as an ASCII format file, *filename*. Returns *DV_FAILURE* if it cannot open the file for writing. Otherwise returns *DV_SUCCESS*.

TviClone

Tvi Functions T Routines

Creates and returns a deep copy of a view.

```
VIEW
TviClone (
       VIEW view)
```

TviCloseData

Tvi Functions T Routines

Closes the data sources in a view.

```
BOOLPARAM
TviCloseData (
       VIEW view)
```

TviCloseData closes the data source list of *view* and recursively closes the data source lists of any views referenced by enabled subdrawings contained in *view*. Returns *DV_FAILURE* if it is passed an invalid view or if an error occurs. Otherwise returns *DV_SUCCESS*.

TviConvertDynamics

Tvi Functions T Routines

Converts a view with pre-8.0 dynamics to use post-8.0 dynamics.

```
void
TviConvertDynamics (
       VIEW view)
```

TviConvertDynamics converts a view that uses pre-8.0 dynamics to use post-8.0 dynamics. *TviConvertDynamics* does this by creating a dynamic control object that emulates the functionality of the pre-8.0 dynamics. See also TviTestDynamics, VOuDyCoConvert, and VOuDySdConvert.

TviCreate

Tvi Functions T Routines

Creates and returns a view containing an empty drawing and an empty data source list.

```
VIEW
TviCreate (void)
```

TviDestroy

Tvi Functions T Routines

Destroys a view, freeing its memory.

```
BOOLPARAM
TviDestroy (
       VIEW view)
```

TviDestroy destroys the view, freeing its memory. The data source list and drawing that belong to the view are dereferenced. Returns *DV_FAILURE* if it is passed an invalid view.

TviExciseDrawing

Tvi Functions T Routines

Removes objects in a drawing from a view.

```
int
TviExciseDrawing (
       VIEW view,
       OBJECT drawing)
```

TviExciseDrawing removes each object contained in the given drawing object from the view. Typically called to remove objects added by a call to TviMergeDrawing. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns the number of objects removed from the view.

TviFileLoad

Tvi Functions T Routines

Loads a view from an open file.

```
VIEW
TviFileLoad (
       FILE *file_pointer)
```

TviFileLoad loads and returns a view from an open file. The call to this routine must be made in the same order as the corresponding call to TviFileSave. For example, if two strings are written to the file, followed by a call to TviFileSave, then you must read those two strings from the file before calling *TviFileLoad*. Returns *DV_FAILURE* if it cannot load a valid view. See also TviFileSave.

TviFileSave

Tvi Functions T Routines

Saves a view to an open file.

```
BOOLPARAM
TviFileSave (
       VIEW view,
       FILE *file,
       int access_mode)
```

TviFileSave saves a view to an open file using *access_mode*. *access_mode* should be *WRITE_EXPANDED* for ASCII write, or *WRITE_COMPACT* for binary write. Flag values are defined in *VOstd.h*. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns *DV_SUCCESS*.

TviForEachDataSource

Tvi Functions T Routines

Traverses the data sources of a view.

```
ADDRESS
TviForEachDataSource (
       VIEW view,
       ADDRFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           DATASOURCE ds,
           ADDRESS argblock)
```

TviForEachDataSource traverses all the data sources of *view* and recursively traverses the data sources of any views referenced by enabled subdrawings contained in *view*. Calls *fun* for each data source. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have two parameters: the data source being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1. to perform a particular operation on each data source, or
2. to find a particular data source.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return *V_CONTINUE_TRAVERSAL* for *ADDRESS* if the data source is not found. Otherwise it should return the data source for *ADDRESS*.

Note: You should not alter the view by adding, deleting, or reordering the data sources during traversal.

For an example of a typical function, see the example under TdrForEachNamedObject. Note that the example demonstrates the use of a function with three parameters, but *TviForEachDataSource* requires only two.

TviForEachVar

Tvi Functions T Routines

Traverses the data source variables of a view.

```
ADDRESS
TviForEachVar (
       VIEW view,
       ADDRFUNPTR fun,
       ADDRESS argblock)

    ADDRESS
    fun (
           DATASOURCE ds,
           DSVAR dsv,
           ADDRESS argblock)
```

TviForEachVar traverses all the data source variables of *view* and recursively traverses the data source variables of any views referenced by enabled subdrawings contained in *view*. Calls *fun* for each data source variable. Continues the traversal while *fun* returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*. For a description of *fun*, see TviForEachDataSource. Note that *TviForEachDataSource* traverses data sources, passing two parameters to *fun*. *TviForEachVar* traverses data source variables, passing three parameters to *fun*: the data source, the data source variable, and the argument block.

TviGetComment

Tvi Functions T Routines

Gets the comment field of the view.

```
char *
TviGetComment (
       VIEW *view)
```

TviGetDataSourceList

Tvi Functions T Routines

Gets a view’s data source list.

```
DATASOURCELIST
TviGetDataSourceList (
       VIEW view)
```

TviGetDataSourceList returns the data source list of the view. Returns *DV_FAILURE* if it is passed an invalid view.

TviGetDrawing

Tvi Functions T Routines

Gets a view's drawing object.

```
OBJECT
TviGetDrawing (
       VIEW view)
```

TviGetDrawing returns the drawing object of the view. Returns *DV_FAILURE* if it is passed an invalid view.

TviLoad

Tvi Functions T Routines

Loads a new view in from a file.

```
VIEW
TviLoad (
       char *filename)
```

TviLoad reads a view from the view file, *filename*, stored in either ASCII or binary format. If *filename* is *NULL*, the value (if set) of the configuration variable *DVVIEW* is used as the name of the file to load. The view file can be created with a call to TviSave or TviASCIISave, or by using the Save View command from DV-Draw. If the application has rebound any data source variables to user-defined data buffers, these data buffers must be recreated when restoring views that were saved with TviSave or TviASCIISave. Returns *DV_FAILURE* if filename does not contain a valid view. Otherwise returns the newly created view.

TviMemLoad

Tvi Functions T Routines

Loads a view from memory.

```
VIEW
TviMemLoad (
       char *bufferp,
       int size)
```

TviMemLoad reads a view from a previously allocated memory buffer. This memory buffer can be received from a network or copied from another process, but the original buffer must hold a view created by a call to TviMemSave or TviASCIIMemSave. *TviMemLoad* makes the appropriate translation from a binary or ASCII storage format. Returns *DV_FAILURE* if the buffer does not contain a valid view. Otherwise returns the newly created view. See also TviASCIIMemSave TviASCIIMemSave and *TviMemSave*.

TviMemSave

Tvi Functions T Routines

Saves a view in binary format to a memory buffer.

```
BOOLPARAM
TviMemSave (
       VIEW view,
       char **bufferpp,
       int *sizep)
```

TviMemSave stores a view in binary format into a memory buffer allocated by this function. *bufferpp* is a pointer to a character pointer into which the location of the allocated buffer is stored. *sizep* is a pointer to an integer which stores the size of the buffer. The user is responsible for freeing this buffer. Returns *DV_FAILURE* if passed an invalid view or cannot allocate enough memory. Otherwise returns *DV_SUCCESS*. See also TviASCIIMemSave and TviMemLoad.

TviMergeAddDataSources

Tvi Functions T Routines

Looks for data source list match and adds if necessary.

```
BOOLPARAM
TviMergeAddDataSources (
       VIEW view,
       DATASOURCELIST master_dsl,
       int matchflag)
```

TviMergeAddDataSources looks for a match between the views's data source list and the master data source list, *master_dsl*, using the *matchflag* parameter:

DS_EXACTMATCH	a data source in the view must exactly match one of the data sources in *master_dsl*.
DS_SUBSETMATCH	a data source in the view must be a subset of one of the data sources in *master_dsl*.
DS_NAMEMATCH	the name of a data source in the view must match the name of one of the data sources in *master_dsl*.

If a match is found, the view's data source variables are merged with the matching data sources in *master_dsl*, and the view's data source list is replaced with *master_dsl*. If no match is found, the view's data source is added to *master_dsl*. Returns *YES* if any data sources were added to *master_dsl.* Otherwise returns *NO*. Does nothing and returns *NO* if it is passed an invalid view or data source list.

TviMergeDataSources

Tvi Functions T Routines

Looks for data source list match with no add option.

```
DATASOURCELIST
TviMergeDataSources (
       VIEW view,
       DATASOURCELIST master_dsl,
       int matchflag)
```

TviMergeDataSources performs the same comparison and uses the same flags as TviMergeAddDataSources, but has no add feature. Instead, if no match occurs, returns a new data source list containing all non-matching data sources. Does nothing and returns *DV_FAILURE* if it is passed an invalid view or data source list.

TviMergeDrawing

Tvi Functions T Routines

Merges a drawing's objects into a view.

```
int
TviMergeDrawing (
       VIEW view,
       OBJECT drawing)
```

TviMergeDrawing adds all of the objects in the drawing to the view. Objects can be removed selectively using TviExciseDrawing. If the view contains dynamic objects, you should also call TviMergeAddDataSources to merge the data sources. If you have already drawn the view in a drawport, you must call *TdpDraw* to draw it again. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns the number of objects merged into the view.

TviOpenData

Tvi Functions T Routines

Opens the data sources of a view.

```
BOOLPARAM
TviOpenData (
       VIEW view)
```

TviOpenData opens the data source list of *view* and recursively opens the data source lists of any views referenced by enabled subdrawings contained in *view*. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns *DV_SUCCESS*.

TviPutComment

Tvi Functions T Routines

Sets the comment field of the view.

```
void
TviPutComment (
       VIEW view,
       char *comment)
```

TviPutDataSourceList

Tvi Functions T Routines

Replaces a view's data source list.

```
DATASOURCELIST
TviPutDataSourceList (
       VIEW view,
       DATASOURCELIST dsl)
```

TviPutDataSourceList replaces the data source list belonging to the view with a new one, specified in the parameter, *dsl*. If this parameter is *NULL*, an empty data source list is substituted. *NULL* typically occurs for a static drawing, or when the programmer is rebinding variable descriptors to an application program variable and wants to destroy *dsl* with *TdlDestroy*. Returns *DV_FAILURE* if it is passed an invalid view or *dsl*. Otherwise returns the old data source list belonging to the view.

TviPutDrawing

Tvi Functions T Routines

Replaces a view’s drawing.

```
OBJECT
TviPutDrawing (
       VIEW view,
       OBJECT drawing)
```

TviPutDrawing replaces the drawing object belonging to the view. Any drawports that use this view must be recreated in order for the changes to be seen. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns the old drawing object.

TviReadData

Tvi Functions T Routines

Reads data from the data sources of a view.

```
BOOLPARAM
TviReadData (
       VIEW view)
```

TviReadData reads one iteration of data from the data source list of *view* and recursively reads the data source lists of any views referenced by enabled subdrawings contained in *view*. Returns *DV_FAILURE* if it is passed an invalid view. Otherwise returns *DV_SUCCESS*.

TviSave

Tvi Functions T Routines

Saves a view as a binary format file.

```
BOOLPARAM
TviSave (
       VIEW view,
       char *filename)
```

TviSave saves the view as a binary format file, *filename*. Returns *DV_FAILURE* if it is passed an invalid view.

TviTestDynamics

Tvi Functions T Routines

Tests a view for pre-8.0 dynamics.

```
BOOLPARAM
TviTestDynamics (
       VIEW view)
```

TviTestDynamics tests a view for pre-8.0 dynamics. Pre-8.0 dynamics are subdrawing dynamics, which use a threshold table object, and color dynamics, which use the foreground color attribute field of an object to hold a variable descriptor object. Post-8.0 dynamics include subdrawing dynamics, color dynamics, and motion dynamics and are implemented using dynamic control objects. Returns *YES* if view has pre-8.0 dynamics. Otherwise returns *NO*. See also TviConvertDynamics.

VO Routines

Vo Routines

Routines for managing DataViews objects. Each *VOxx* module contains routines for creating and performing operations specific to an object, where *xx* is one of the DataViews object types. The *VOob* layer contains routines common to most objects types. Certain routines for the objects are located higher up in the *T* layer, in the *Tlocation*, *Tdrawing*, *Tobject*, and *Tscreen* modules. Because objects can be multiply referenced, there is no destroy operation. Instead, most object types maintain reference counts. When the reference count of an object reaches zero, DataViews deletes the object.

Objects are DataViews private types, declared as type *OBJECT*. You can use VOobType to determine the type of the object.

Objects fall into two categories: graphical objects, such as arc and line, and non-graphical objects, such as input technique object and location object. The drawing, deque, node, and edge are special non-graphical objects because they can contain graphical objects, which makes them displayable on the screen. For example, a drawing object is a list of the graphical objects in a view. The point object is also a special case because it appears as a small cross when it is not part of an object.

Each graphical object has attributes that it keeps track of with the DV-Tools public type *ATTRIBUTES*. The attribute structure contains all fields possible in an object. Only certain fields apply to a given object. To create a graphical object, determine which attribute fields are valid for that object by looking at the description of the *VOxxCreate* routine. Initialize an attribute structure with *VOuAtInit*, fill in the applicable attributes using *VOuAttr*, then pass the resulting attribute structure to the graphical object's create function. The *ATTRIBUTES* structure and flags are declared in *VOstd.h* and listed in the *Include Files* chapter of this manual.

Vo Routines

VO Modules

All modules in the *VO* layer require the following *#include* files:

```
#include "std.h"
#include "dvstd.h"
#include "dvtools.h"
#include "VOstd.h"
#include "VOfundecl.h"
```

Any additional *#include* files required by a particular *VOxx* module are listed in the synopsis section for that module.

VOob	A set of general operations that act on many different types of objects.
VOar	Manages arc objects (*ar*).
VOci	Manages circle objects (*ci*)
VOco	Manages color objects (*co*).
VOdbg	General debug and statistics routines.
VOdg	Manages data group objects (*dg*).
VOdq	Manages deque objects (*dq*).
VOdr	Manages drawing objects (*dr*).
VOdy	Manages dynamic control objects.
VOed	Manages edge objects (*dq*).
VOel	Manages ellipse objects (*el*).
VOg	Draws graphical objects on the screen using lower level routines.
VOic	Manages icon objects (*ic*).
VOim	Manages image objects (*im*).
VOin	Manages input objects (*in*).
VOit	Manages input technique objects (*it*).
VOln	Manages line objects (*ln*).
VOlo	Manages location objects (*lo*).
VOno	Manages node objects.
VOpm	Manages pixmap objects (*pm*).
VOpt	Manages point objects (*pt*).
VOpy	Manages polygon objects (*py*).
VOre	Manages rectangle objects (*re*).
VOru	Manages rule object.
VOsc	Manages screen objects (*sc*).
VOsd	Manages subdrawing objects (*sd*).
VOsf	Manages scalable font text objects (*sf*).
VOsk	Manages slotkey objects
VOtt	Manages threshold table objects (*tt*).
VOtx	Manages text objects (*tx*).
VOu	Utility routines for use with objects.
VOvd	Manages variable descriptor objects (*vd*).
VOvt	Manages vector text objects (*vt*).
VOxf	Manages transform objects (*xf*).

VOar (VOarc)

VOar Functions VO Routines

Manages arc objects (*ar*). An arc object is defined by three point subobjects: the first defines the start point, the second defines the center point, and the third defines the end point of the arc. Arc attributes are foreground color, background color, fill status, line type, line width, and arc draw direction. The arc is drawn from the start point until it meets the line defined by the center point and end point. The arc direction attribute determines whether the arc is drawn clockwise or counter-clockwise. The arc fill status can be *FILL*, *EDGE*, *EDGE_WITH_FILL*, *FILL_WITH_EDGE*, or *DV_TRANSPARENT*. When *EDGE* is used, the boundary is drawn using the line attributes. An arc using *DV_TRANSPARENT* fill looks identical to one with *EDGE* only, but you can select it with the cursor anywhere in the interior of the shape. A transparent arc does not visually obscure objects behind it, but they cannot be selected through it. Filled arcs resemble pie slices. When either *EDGE_WITH_FILL* or *FILL_WITH_EDGE* is used, the second feature listed in the fill status flag uses the background color attribute. The foreground color is used in all other cases.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

VOar Functions

VOarAtGet	See VOobAtGet.
VOarAtSet	See VOobAtSet.
VOarBox	See VOobBox.
VOarClone	See VOobClone.
VOarCreate	Creates an arc object.
VOarDereference	See VOobDereference.
VOarIntersect	See VOobIntersect.
VOarPtGet	See VOobPtGet.
VOarPtSet	See VOobPtSet.
VOarRefCount	See VOobRefCount.
VOarReference	See VOobReference.
VOarStatistic	Returns statistics about arcs.
VOarTraverse	See VOobTraverse.
VOarValid	See VOobValid.
VOarXfBox	See VOobXfBox.
VOarXformBox	See VOobXformBox.

A *VOar* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOar* routine to save the overhead of an additional routine call.

VOarCreate

VOar Functions VO Routines

Creates an arc object.

```
OBJECT
VOarCreate (
       OBJECT start,
       OBJECT center,
       OBJECT end,
       ATTRIBUTES *attributes;
```

VOarCreate creates and returns an arc object. Valid *attributes* field flags are:

FOREGROUND_COLOR *FILL_STATUS*
BACKGROUND_COLOR *LINE_TYPE*
ARC_DIRECTION *LINE_WIDTH*

If *attributes* is *NULL*, default values are used. Valid arc direction flags are *CLOCKWISE* and *COUNTER_CLOCKWISE*.

VOarStatistic

VOar Functions VO Routines

Returns statistics about arcs.

```
LONG
VOarStatistic (
       int flag)
```

VOarStatistic returns statistics about arcs, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of arcs.

Voob

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

A set of general operations that act on many different types of objects. Each *VOob* routine is listed with the objects for which it is defined in the *Domains* table. If a *VOob* routine is applied to an object for which it is not defined, there is no effect.

There are two categories of *VOob* routines: routines that serve as a layer over a specific routine in the *VO* layer and routines that extend object functionality. The first group works with corresponding routines in the *VO* layer. For example, the *VOobTraverse* function, which simply calls the appropriate *VOxxTraverse* function for the object being traversed. The second group has no corresponding routines in the *VO* layer. For example, the VOobDyUtil routines let you attach a dynamic control object to other objects, and the VOobSlotUtil routines let you attach general information to objects that support slots.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Voob Functions

VOobAtGet	Gets the current attributes of an object.
VOobAtSet	Sets new attributes of an object.
VOobBox	Gets an object's bounding box in world coordinates.
VOobClone	Makes a deep copy of an object.
VOobDeleteSlot	Deletes a slot from an object.
VOobDereference	Decrements the reference count of an object.
VOobDyDelete	Removes the dynamic control object from an object.
VOobDyGet	Returns the dynamic control object attached to the object.
VOobDySet	Associates a dynamic control object with a graphical object.
VOobGetSlot	Get a specified slot from the object.
VOobHasSlot	Determines if the object has the specified slot.
VOobIntersect	Determines if an object intersects the viewport.
VOobNumSlots	Gets the number of slots from an object.
VOobPtGet	Gets the *index*-th control point of an object.
VOobPtSet	Sets a new control point for an object.
VOobRefCount	Gets the reference count of an object.
VOobReference	Increments the reference count of an object.
VOobSetSlot	Sets a slot for an object.
VOobSupportsSlots	Determines if the object allows adding slots.
VOobTraverse	Applies a user-supplied function to subobjects.
VOobType	Returns the type flag of an object.
VOobValid	Determines if an object is valid.
VOobXfBox	Gets an object's bounding box in screen coordinates.
VOobXformBox	Gets the bounding box of a transformed object in screen coordinates.
VOobXformBoxPadded	Gets the bounding box of a transformed object in screen coordinates plus a specified amount of padding.

VOobAtGet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the current attributes of an object.

```
void
VOobAtGet (
       OBJECT object,
       ATTRIBUTES *attributes)
```

VOobAtGet sets the fields of the attributes structure to the attribute values of the current object. Fields that don't apply to the object are set to *EMPTY_FIELD* or *EMPTY_FLOAT_FIELD*, depending on the type of entry.

VOobAtSet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Sets new attributes in an object.

```
void
VOobAtSet (
       OBJECT object,
       ATTRIBUTES *attributes)
```

VOobAtSet sets the attributes of an object to the new values in the attributes structure. The attributes structure is a DataViews public type, which contains fields for all of the attributes of all the different graphical object types. It is used as an intermediate mechanism for manipulating the attributes of graphical objects. Each object copies only the fields for which it has attributes. If *attributes* contains fields with the value *EMPTY_FIELD* or *EMPTY_FLOAT_FIELD*, the original value of the field is retained. Otherwise it is replaced by the new value.

VOobBox

Routines for getting bounding boxes. Examples

Functions

VOobBox	Gets an object's bounding box in world coordinates.
VOobXfBox	Gets an object's bounding box in screen coordinates.
VOobXformBox	Gets the bounding box of a transformed object in screen coordinates.
VOobXformBoxPadded	Gets the bounding box of a transformed object in screen coordinates plus a specified amount of padding.

Examples

VOobBox

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets an object's bounding box in world coordinates.

```
void
VOobBox (
       OBJECT object,
       RECTANGLE *wvp,
       RECTANGLE *svp_delta)
```

VOobBox returns the world bounding box in *wvp*. The world bounding box calculation does not include device-dependent features such as wide lines, scalable font text size, or hardware text size. Instead, *VOobBox* provides a screen coordinate offset rectangle, *svp_delta*. This specifies the additional size in screen coordinates to allow for line thickness greater than one, scalable font text, and hardware text.

For vector text (*vt*), *svp_delta* is always zero. For hardware text (*tx*) and scalable font *(sf)* text, which are device-dependent, *wvp* is a dimensionless rectangle located at the text object's anchor point, and *svp_delta* specifies the size of the text object. Note: svp_delta is a best guess until the object is actually drawn.

VOobBox is the only way to get object size information before the drawport is created. After the drawport is created, you can get the bounding box in screen coordinates using *VOobXfBox*.

VOobXfBox

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets an object's bounding box in screen coordinates.

```
void
VOobXfBox (
       OBJECT object,
       OBJECT xform,
       RECTANGLE *svp)
```

VOobXfBox returns the screen bounding box in *svp*. *xform* is the drawing-to-screen transform of the object, which is available only after the drawport has been created. To get *xform*, call *TdpGetXform* with the *DR_TO_SCREEN* flag and the object's drawport. This routine is obsolete but maintained for compatibility with previous releases.

The bounding box is one pixel larger than the object appears in order to guarantee complete coverage of the object. On some objects, the bounding box may be several pixels larger. Calling this routine recursively can result in an accumulation of additional pixels. To get a true bounding box, use *VOobXformBox*. To get a true bounding box with a specified number of additional pixels, use *VOobXformBoxPadded*.

For objects such as drawings and subdrawings, the bounding box is the union of the bounding boxes of the subobjects. For node and edge objects, the bounding box is the bounding box of the associated geometry object.

This routine always returns a bounding box, even for objects with no dimensions such as empty text stings, empty subdrawing objects, or node or edge objects without geometry. For correct return values on such objects, use *VOobXformBox*.

Note that if your drawport pans or changes scale, the screen bounding box also changes. To get the new bounding box, you must first call *TdpGetXform* to get the new transformation, then call *VOobXfBox*.

To convert the screen coordinates to equivalent world coordinates, use *TdpScreenToWorld*.

VObXformBox

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the bounding box of a transformed object in screen coordinates.

```
BOOLPARAM
VOobXformBox (
       OBJECT object,
       OBJECT xform,
       RECTANGLE *svp)
```

VOobXformBox returns the true screen bounding box of a transformed object in *svp*. *xform* is the drawing-to-screen transform of the object, which is available only after the drawport has been created. To get *xform*, call *TdpGetXform* with the *DR_TO_SCREEN* flag and the object's drawport.

This routine returns a bounding box that encompasses the exact size of the object, without allowing for rounding in the calculations. To get a bounding box with a specified number of additional pixels, use *VOobXformBoxPadded*.

This routine returns a true bounding box even with rotational transformation. If the object has no dimensions, such as empty text stings, empty subdrawing objects, or node or edge objects without geometry, returns *NO*.

VOobXformBoxPadded

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the bounding box of a transformed object in screen coordinates plus a specified amount of padding.

```
BOOLPARAM
VOobXformBoxPadded (
       OBJECT object,
       OBJECT xform,
       RECTANGLE *svp,
       int padding)
```

VOobXformBoxPadded returns the true screen bounding box in *svp*, expanded by the number of pixels specified in *padding*. *xform* is the drawing-to-screen transform of the object, which is available only after the drawport has been created. To get *xform*, call *TdpGetXform* with the *DR_TO_SCREEN* flag and the object's drawport.

If the object has no dimensions, such as empty text stings, empty subdrawing objects, or node or edge objects without geometry, returns *NO*.

VOobBox Examples

Given a rectangle object, *re*, centered on the world coordinate origin, 200 world coordinate units per side, and with a line thickness of 4, use the following call:

```
OBJECT re;
RECTANGLE wvp, svp_delta;
VOobBox (re, &wvp, &svp_delta);
```

This results in the following values for the rectangles:

```
wvp = {-100, -100, 100, 100}
svp_delta = {-2, -2, 2, 2}
```

The following code fragment shows how to repair a portion of the drawport after explicitly erasing an object:

```
OBJECT xform;
RECTANGLE repair_vp;

/* Before erasing, determine the portion of the drawport to repair. */
xform = TdpGetXform (drawport, DR_TO_SCREEN);
VOobXfBox (object, xform, &repair_vp);

/* Erase the overlayed object. */
TdpEraseObject (drawport, object);
/* Repair the erased portion. */
TdpRedraw (drawport, &repair_vp, NO);
```

The following code fragment shows how to calculate a screen coordinate bounding box using VOobBox. This method was superseded with the introduction of VOobXfBox, but was a common method that your code may still be using. For the following objects, this method and *VOobXfBox* are equivalent:

dg, ic, im, in, tx

For these other objects, *VOobXfBox* gives more accurate results and should be used if possible. In particular, *VOobXfBox* is more accurate for drawing objects and when the drawport is created using *TdpCreateStretch*.

ar, ci, dr, ed, el, ln, no, py, re, sd, tt, *vt*

```
RECTANGLE wvp, svp_delta;
RECTANGLE combined;

VOobBox (object, &wvp, &svp_delta);
TdpWorldToScreen (drawport, &wvp.ll, &combined.ll);
combined.ll.x += svp_delta.ll.x;
combined.ll.y += svp_delta.ll.y;

TdpWorldToScreen (drawport, &wvp.ur, &combined.ur);
combined.ur.x += svp_delta.ur.x;
combined.ur.y += svp_delta.ur.y;
```

VOobClone

VOob Functions   VO Routines

VOob Modules: VOobDyUtil   VOobBox   VOobslotUtil

Makes a deep copy of an object.

```
OBJECT
VOobClone (
       OBJECT object)
```

VOobClone makes a deep copy of an object. A deep copy includes all of the object's subobjects. This makes a complete duplicate of the original object with no subobjects in common. There are some exceptions to this:

Subdrawing objects do not copy the drawings they contain.
Data group objects do not copy their attached data source variables.
Input objects do not copy their attached data source variables.
Input technique objects do not copy their template drawings.
Icon and *image objects* do not copy their associated pixmaps.

Returns a copy of the cloned object.

VOobDereference

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Decrements the reference count of an object.

```
void
VOobDereference (
       OBJECT object)
```

VOobDereference decrements the reference count of an object by one. If this results in a reference count of zero or less, DataViews destroys the object, frees the allocated memory, and dereferences its subobjects. The reference count is an integer stored within the object that records how many other objects reference it. To get the current reference count of an object, use *VOobRefCount*. For additional information on referencing objects, see *VOobReference*.

An object that was referenced by using *VOobReference* should be dereferenced by using *VOobDereference* when it is no longer needed.

Utility Vo dynamics Routines

Utility routines for getting, setting, and deleting dynamic control objects.

A dynamic control object is destroyed when it is no longer attached to any object, so it may be destroyed after a call to VOobDyDelete or VOobDySet. To prevent a dynamic control object from being destroyed, attach it to a dummy graphical object.

Functions

VOobDyDelete	Removes the dynamic control object from an object.
VOobDyGet	Returns the dynamic control object attached to the object.
VOobDySet	Associates a dynamic control object with a graphical object.

VOobdy Example

VOobDyDelete

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Removes the dynamic control object from the object,

```
void
VOobDyDelete (
       OBJECT object)
```

VOobDyGet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Returns the dynamic control object attached to the object.

```
OBJECT
VOobDyGet (
       OBJECT object)
```

VOobDySet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Attaches the dynamic control object to the object.

```
void
VOobDySet (
       OBJECT object,
       OBJECT dynamic)
```

See Also

VOdynamic

VOdy Example

The following code shows how to enable and disable dynamics for a rectangle given an existing rectangle, dynamic control object, and drawport:

```
OBJECT rectangle, dynamic;
DRAWPORT drawport;

/* enable dynamics for the rectangle */
VOobDySet (rectangle, dynamic);

/* display dynamic changes */
TdpDrawNext (drawport);

/* disable dynamics for the rectangle */
VOobDyDelete (rectangle);
```

VOobIntersect

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Determines if an object intersects the viewport.

```
BOOLPARAM
VOobIntersect (
       OBJECT object,
       OBJECT xform,
       RECTANGLE *vp)
```

VOobIntersect tests for the intersection of an object with the rectangle *vp*. The rectangle *vp* is normally specified in screen coordinates and *xform* is a transform object (*xf*) which specifies the world-to-screen coordinate transformation of the object. If *xform* is *NULL*, the rectangle *vp* is assumed to be in world coordinates.

Returns *YES* if intersecting, *NO* otherwise.

VOobPtGet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the *index*-th control point of an object.

```
OBJECT
VOobPtGet (
       OBJECT object,
       int index)
```

VOobPtGet gets a specific control point of an object. The point is specified by the integer *index*, where a value of 1 indicates the first point, a value of 2 the second point, etc.

If *index* is 0, returns the number of point objects contained in *object*.

If there is no *index*-th point, returns *NULL*.

VOobPtSet

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Sets a new control point for the object.

```
void
VOobPtSet (
       OBJECT object,
       int index,
       OBJECT new_point)
```

VOobPtSet replaces a specified control point of an object with a new control point, *new_point*. The control point to be replaced is specified by *index*, where a value of 1 indicates the first point, a value of 2 the second point, etc.

VOobRefCount

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the reference count of an object.

```
int
VOobRefCount (
       OBJECT object)
```

VOobRefCount returns the reference count of the object. The reference count is an integer stored within the object that records how many other objects reference it. This information is used to determine when it is safe for DataViews to destroy the object. To increment and decrement the reference count of the object, use *VOobReference* and *VOobDereference*.

VOobReference

Increments the reference count of an object.

```
OBJECT
VOobReference (
       OBJECT object)
```

VOobReference increments the reference count of an object by one. The reference count is an integer stored within the object which records how many other objects reference it. This information is used to determine when it is safe for DataViews to destroy the object. To get the current reference count of an object, use *VOobRefCount*.

Most objects, including all graphical objects, have reference counts. When an object is created, it has a reference count of zero. Every time a child object is added to a parent object such as a deque, drawing object, or drawing object's name list, the reference count of the child object is automatically incremented. When the parent object is dereferenced, the reference count of the child object is automatically decremented. The object is destroyed by DataViews if its reference count falls to or below zero. See also *VOobDereference*.

If you create an object to use only as a child object, do not reference it. The child object is then destroyed when its parent object is destroyed. If you create a child object that you want to retain after the destruction of its parent object, call *VOobReference* to reference it. The child object is then not destroyed when its parent object is destroyed. To destroy a parentless child object when you no longer want it, call *VOobDereference*.

Returns the object. This allows objects to be created and referenced with a single nested call, as shown below. If the object is invalid, returns the object.

Example

The following code fragment creates a permanent point by nesting the *VOptCreate* call in a *VOobReference* call. After this call, *pt1* has a reference count of 1.

```
pt1 = VOobReference (VOptCreate (WORLD_COORDINATES, -10000, 4000, (OBJECT)NULL));
```

The following code fragments show how the reference counts of point objects change as they are created, referenced, used in other objects, and dereferenced. In the first code fragment, two temporary point objects are created then destroyed by DataViews when the rectangle is destroyed.

```
/* RefCounts become: pt2 - 0, pt3 - 0. */
pt2 = VOptCreate (WORLD_COORDINATES, -9000, 3000, (OBJECT)NULL);
pt3 = VOptCreate (WORLD_COORDINATES, -8000, 2000, (OBJECT)NULL);

/* RefCounts become: pt2 - 1, pt3 - 1. */
rect1 = VOreCreate (pt2, pt3, (ATTRIBUTES *)NULL);

...

/* Destroy pt2 and pt3 now. */
VOobDereference (rect1);
```

In the following code fragment, two point objects are created and referenced. They are not destroyed by DataViews when the rectangle is destroyed, and should be dereferenced explicitly.

```
/* RefCounts become: pt4 - 0, pt5 - 0. */
pt4 = VOptCreate (WORLD_COORDINATES, -9000, 3000, (OBJECT)NULL);
pt5 = VOptCreate (WORLD_COORDINATES, -8000, 2000, (OBJECT)NULL);

/* RefCounts become: pt4 - 1, pt5 - 1. */
VOobReference (pt4);
VOobReference (pt5);

/* RefCounts become: pt4 - 2, pt5 - 2. */
rect2 = VOreCreate (pt4, pt5, (ATTRIBUTES *)NULL);

...

/* pt4 and pt5 are not destroyed now. */
VOobDereference (rect2);

/* Destroy pt4 and pt5 now. */
VOobDereference (pt4);
VOobDereference (pt5);
```

VOobSlotUtil

Utility routines for operating on slots. A slot is a means of attaching information to objects. If an object has more than one slot, you can think of these slots as being arranged in a table that can be accessed either using slotkey objects or indices. Slotkey objects associate a slot with the information describing what the slot contains. A slot can contain the following: an integer, an array of integers, a float, an array of floats, an object, or a pointer to a *NULL*-terminated string.

The routines provided in this module attach a slot to an object via a slotkey object or by getting a slot from an object. You can also verify that an object supports slots or has a particular slot. Deleting a slot from an object does not free the memory allocated to the slotkey object.

The *VOslotkey* module provides routines for declaring and getting information about slotkey objects.

The slotkey feature is intended for use by sophisticated DataViews users.

Functions

VOobDeleteSlot	Deletes a slot from an object.
VOobGetSlot	Gets a specified slot from the object.
VOobHasSlot	Determines if the object has the specified slot.
VOobNumSlots	Gets the number of slots from an object.
VOobSetSlot	Sets a slot for an object.
VOobSupportsSlots	Determines if the object allows adding slots.

VOobDeleteSlot

VOob Functions   VO Routines

VOob Modules: VOobDyUtil   VOobBox   VOobslotUtil

Deletes a slot from an object.

```
BOOLPARAM
VOobDeleteSlot (
       OBJECT object,
       OBJECT slotkey)
```

VOobDeleteSlot deletes a slot from an object as specified by *slotkey*. The parameter *slotkey* specifies the slot either as a slotkey object or as an index into the object's slot table. *VOobDeleteSlot* returns *DV_SUCCESS* if it finds and deletes the slot.

VOobGetSlot

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets a specified slot from the object.

```
LONG
VOobGetSlot (
       OBJECT object,
       OBJECT slotkey,
       LONG *value,
       ULONG *flags)
```

VOobGetSlot gets the slot specified by *slotkey* from the object and stores it in the parameter *value*. The parameter *slotkey* specifies the slot either as a slotkey object or as an index into the object's slot table. Use *VOobNumSlots* to get the number of slots in an object's slot table. Use the *flags* field to keep track of information about the value stored in the slot. For example, you can use the flag area to store access counts or semaphores or to keep track of whether the slot has been accessed, changed, or initialized. When a slot is created or loaded from a file, its flag field is set to 0. If *slotkey* is a slotkey object, *VOobGetSlot* returns the 1-based index of the slot found. If *slotkey* is an index, *VOobGetSlot* returns the slotkey object of the slot found. If the slot was not found, *VOobGetSlot* returns *0*.

VOobHasSlot

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Determines if the object has the specified slot.

```
int
VOobHasSlot (
       OBJECT object,
       OBJECT slotkey)
```

VOobHasSlot determines if the object has a slot for the given slotkey object. Returns the 1-based index of the slot if found. Otherwise returns *0*.

VOobNumSlots

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Gets the number of slots from an object.

```
int
VOobNumSlots (
       OBJECT object)
```

VOobNumSlots returns the number of slots of an object.

VOobSetSlot

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Sets a slot for an object.

```
int
VOobSetSlot (
       OBJECT object,
       OBJECT slotkey,
       LONG *value,
       ULONG *flags)
```

VOobSetSlot sets the object's slot specified by *slotkey* to *value*. The parameter *slotkey* must be a slotkey object, unlike the *slotkey* parameter of *VOobDeleteSlot* and *VOobGetSlot* which can also be an index. Use the *flags* field to keep track of information about the value stored in the slot. For example, you can use the flag area to store access counts or semaphores or to keep track of whether the slot has been accessed, changed, or initialized. When a slot is created or loaded from a file, its flag field is set to *0*. Returns the 1-based index of the slot if *VOobSetSlot* adds the slot. Otherwise returns *0*.

VOobSupportsSlots

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Determines if the object allows adding slots.

```
BOOLPARAM
VOobSupportsSlots (
       OBJECT object)
```

VOobSupportsSlots returns *YES* if the *object* allows adding slots. Otherwise returns *NO*.

See Also

VOslotkey module.

Example

The following code illustrates how to declare different types of slotkeys and attach them to a drawing object.

```
int intnum 1234;
int intarray[3] = {1,2,3};
float floatnum = 1.2345;
float floatarray[3] = {1.1, 2.2, 3.3};
OBJECT drawing, rectangle;
OBJECT intsk, intarraysk, namesk, objsk, floatsk, floatarraysk;

intsk = VOskDeclare ("int", VOSK_INT_TYPE);
intarraysk = VOskDeclare ("INT_ARRAY", VOSK_INT_ARRAY_TYPE, 3);
namesk = VOskDeclare ("STRING", VOSK_STRING_TYPE);
objsk = VOskDeclare ("OBJECT", VOSK_OBJECT_TYPE);
floatsk = VOskDeclare ("FLOAT", VOSK_FLOAT_TYPE);
floatarraysk = VOskDeclare ("FLOAT", VOSK_FLOAT_ARRAY_TYPE, 3);

Tinit ((char *) NULL, (char *) NULL);

view = TviCreate();
drawing = TviGetDrawing (view);

rectangle = VOreCreate (VOptCreate (WORLD_COORDINATES, -100, -100, 0),
             VOptCreate (WORLD_COORDINATES, -100, -100, 0),
             (ATTRIBUTES *)0);
VOobReference (rectangle);

VOobSetSlot (drawing, intsk, (LONG *)&intnum, (ULONG *)0);
VOobSetSlot (drawing, intarraysk, (LONG *)intarray, (ULONG *)0)
VOobSetSlot (drawing, namesk, (LONG *)"Hello World", (ULONG *)0)
VOobSetSlot (drawing, objsk, (LONG *)&rectangle, (ULONG *)0)
VOobSetSlot (drawing, floatsk, (LONG *)&floatnum, (ULONG *)0)
VOobSetSlot (drawing, floatarraysk, (LONG *)&floatarray, (ULONG *)0)
```

VOobTraverse

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Applies a user-supplied function to subobjects.

```
BOOLPARAM
VOobTraverse (
       OBJECT object,
       VOOBTRAVERSEFUNPTR test,
       ADDRESS testargs)
   BOOLPARAM
   test (
          OBJECT subobj,
          ADDRESS testargs)
```

VOobTraverse traverses all of the object's subobjects and calls *test (subobj, testargs)* for each subobject. Continues the traversal while *test* returns *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *test* returns *V_HALT_TRAVERSAL*.

test must be provided by the programmer to perform whatever operation is required. It should return a *BOOLPARAM*, and must have two parameters: the subobject being processed, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *test* function is typically used in one of two ways:

1. to perform a particular operation on each subobject, or
2. to find a particular subobject.

In the first case, *test* should be written so that it always returns *V_CONTINUE_TRAVERSAL*. In the second case, *test* should return *V_HALT_TRAVERSAL* if the subobject is found. Otherwise it should return *V_CONTINUE_TRAVERSAL*. See the example below. *VOobTraverse* returns the boolean value of the last call to the *test* function.

Note: You should not alter the object being traversed by adding, deleting, or reordering its subobjects during traversal.

Example

The following code fragment draws all of the objects in a deque, *dq*:

```
BOOLPARAM draw_func (
      OBJECT subobj,
      ADDRESS drawport)
{
   TdpDrawObject ((DRAWPORT) drawport, subobj);
   return V_CONTINUE_TRAVERSAL;
}

OBJECT dq;
DRAWPORT drawport;

VOobTraverse (dq, draw_func, (ADDRESS)drawport)
```

VOobType

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Returns the type flag of the object.

```
int
VOobType (
        OBJECT object)
```

VOobType returns the type flag of the object. The type flag can have one of the following values:

OT_ARC	arc object
OT_CIRCLE	circle object
OT_COLOR	color object in non-RGB format
OT_DEQUE	deque object
OT_DG	data group object
OT_DRAWING	drawing object
OT_DYNAMIC	dynamic control object
OT_EDGE	edge object
OT_ELLIPSE	ellipse object
OT_ICON	icon object
OT_IMAGE	image object
OT_INPUT	input object
OT_INPUT_TECHNIQUE	input technique object
OT_LINE	line object
OT_LOCATION	location object
OT_NODE	node object
OT_PIXMAP	pixmap object
OT_POINT	point object
OT_POLYGON	polygon object
OT_RECTANGLE	rectangle object
OT_REFCOLOR	color object that refers to another color object
OT_RGB	color object in RGB format: *COLOR_COMPONENT* or *COLOR_SPEC*
OT_RULE	rule object
OT_SCREEN	screen object
OT_SLOTKEY	slotkey object
OT_SUBDRAWING	subdrawing object
OT_TEXT	text object
OT_THRESHTABLE	threshold table object
OT_VD	variable descriptor object
OT_VTEXT	vector text object
OT_XFORM	transform object

Example

```
OBJECT location, object;
DRAWPORT dp;

object = TloGetSelectedObject (location);

if (VOobType (object) == OT_DG)
    {
        TdpDrawNextObject (dp, object);
    }
```

VOobValid

VOob Functions VO Routines

VOob Modules: VOobDyUtil VOobBox VOobslotUtil

Determines if an object is valid.

```
BOOLPARAM
VOobValid (
       OBJECT object)
```

VOobValid determines if an object is valid. A valid object is one that has been created properly and has not yet been destroyed using *VOobDereference*. Returns *YES* if valid, *NO* otherwise.

Introduction
Examples
Routines:
VOobBox
VOobXfBox
VOobXformBox
VOobXformBoxPadded

VOci (VOcircle)

VOci Functions　　VO Routines

Manages circle objects (*ci*). A circle object is defined by two point subobjects: a center point and a point on the circumference. Circle attributes are foreground color, background color, fill status, line type, and line width. The circle fill status can be *FILL*, *EDGE*, *EDGE_WITH_FILL*, *FILL_WITH_EDGE*, or *DV_TRANSPARENT*. When *EDGE* is used, the boundary is drawn using the line attributes. A circle using *DV_TRANSPARENT* fill looks identical to one with *EDGE* only, but you can select it with the cursor anywhere in the interior of the shape. A transparent circle does not visually obscure objects behind it, but they cannot be selected through it. When either *EDGE_WITH_FILL* or *FILL_WITH_EDGE* is used, the second feature listed in the fill status flag uses the background color attribute. The foreground color is used in all other cases.

VOci Functions

VOciAtGet	See VOobAtGet.
VOciAtSet	See VOobAtSet.
VOciBox	See VOobBox.
VOciClone	See VOobClone.
VOciCreate	Creates a circle object.
VOciDereference	See VOobDereference.
VOciIntersect	See VOobIntersect.
VOciPtGet	See VOobPtGet.
VOciPtSet	See VOobPtSet.
VOciRefCount	See VOobRefCount.
VOciReference	See VOobReference.
VOciStatistic	Returns statistics about circles.
VOciTraverse	See VOobTraverse.
VOciValid	See VOobValid.
VOciXfBox	See VOobXfBox.
VOciXformBox	See VOobXformBox.

A *VOci* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOci* routine to save the overhead of an additional routine call.

VOciCreate

VOci Functions VO Routines

Creates a circle object.

```
OBJECT
VOciCreate (
       OBJECT center,
       OBJECT radiuspt,
       ATTRIBUTES *attributes)
```

VOciCreate creates and returns a circle object. *radiuspt* is a point on the circumference of the circle, and *center* is the point around which the circle is drawn. Valid *attributes* field flags are:

FOREGROUND_COLOR	*FILL_STATUS*
BACKGROUND_COLOR	*LINE_TYPE*
LINE_WIDTH	

If *attributes* is *NULL*, default values are used.

VOciStatistic

VOci Functions VO Routines

Returns statistics about circles.

```
LONG
VOciStatistic (
       int flag)
```

VOciStatistic returns statistics about circles, depending on the value of *flag*. If *flag* is *OBJECT_COUNT*, returns the current number of circles. Valid flag values are defined in *VOstd.h.*

VOco (VOcolor)

VOco Functions VO Routines

Manages color objects (*co*) and describes the color of graphical objects. There are three types of color objects. One is represented by one byte of object type (*OT_RGB*) followed by three bytes of intensity in the range [0,255] (RGB format), where each intensity corresponds to one of the three additive primaries, red, green, and blue. The second is represented by one byte of object type (*OT_COLOR*) followed by a 24-bit integer representing the color in the device-dependent format. Usually this is an index into the device's color table, but it may be a true color if the device supports direct color. The last type is represented by one byte of object type (*OT_REFCOLOR*) followed by a 16-bit integer that is the offset into the object heap for the referenced color object.

VOco Functions

VOcoClone	See VOobClone.
VOcoCreate	Creates a color or RGB object.
VOcoCsGet	Gets color in the *COLOR_SPEC* structure format.
VOcoDereference	See VOobDereference.
VOcoIndex	Returns color index corresponding to the color.
VOcoNdxGet	Gets color in color index form for current screen.
VOcoRefCount	See VOobRefCount.
VOcoReference	See VOobReference.
VOcoRefSwitch	Switches current referenced color with new color.
VOcoRgbGet	Gets color in RGB form for current screen.
VOcoSubtype	Returns color object subtype.
VOcoValid	See VOobValid.

A *VOco* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOco* routine to save the overhead of an additional routine call.

VOcoCreate

VOco Functions VO Routines

Creates a color or RGB object.

```
OBJECT
VOcoCreate (
       int format,
           <type> arg1,
           ...,
           <type> argn)
```

VOcoCreate creates and returns a color object in index, RGB, or referenced format. Possible *format* values are:

COLOR_COMPONENTS	*Specifies color components.* arg1*,* arg2*,* arg3 *are the three primary color intensities in the range [0,255].*
COLOR_INDEX	*Specifies color index or device-dependent format.* arg1 *is the color (up to 24-bits).*
COLOR_NAME	*Specifies the name of a color.* arg1 *is a pointer to a character string name that names the color. Valid color name strings are:*

black	*blue*	*cyan*
gray	*green*	*grey*
magenta	*red*	*white*
yellow		

Note that on monochrome systems the color sense for black and white is the opposite of that on color systems. This means that the color object with the color name black appears as white on a monochrome system.

COLOR_REFERENCE	*Specifies a reference to a color.* arg1 *is a color object created by a previous call to* VOcoCreate*. This format lets several objects refer to the same color object.*
COLOR_STRUCTURE	*Specifies the* COLOR_SPEC *structure.* arg1 *is a pointer to a* COLOR_SPEC*. See the* COLOR_SPEC typedef *in the* Include Files *chapter.*

VOcoCsGet

VOco Functions VO Routines

Gets color in the *COLOR_SPEC* structure format.

```
void
VOcoCsGet (
       OBJECT color,
       COLOR_SPEC *color_spec)
```

VOcoCsGet gets the color in the *COLOR_SPEC* structure format, *color_spec*. See the *COLOR_SPEC typedef* in the *Include Files* chapter.

VOcoIndex

VOco Functions VO Routines

Returns the integer color index corresponding to the color.

```
LONG
VOcoIndex (
       OBJECT color)
```

VOcoNdxGet

VOco Functions VO Routines

Returns a color object in index format for the current screen.

```
OBJECT
VOcoNdxGet (
       OBJECT color)
```

VOcoRefSwitch

VOco Functions VO Routines

Switches current referenced color with new color.

```
BOOLPARAM
VOcoRefSwitch (
       OBJECT clr,
       OBJECT newclr)
```

VOcoRefSwitch switches the current referenced color of *clr*, with the new color, *newclr*. Returns *DV_SUCCESS* if the switch is successful. Returns *DV_FAILURE* only if *clr* is not created as a referencing color object.

VOcoRgbGet

VOco Functions VO Routines

Gets color in RGB form for current screen.

```
OBJECT
VOcoRgbGet (
       OBJECT color)
```

VOcoRgbGet returns a color object in RGB form. If the color object or the referenced color object is of type *OT_COLOR*, the index is converted to RGB values from the color table for the current screen.

VOcoSubtype

VOco Functions VO Routines

Returns color object subtype.

```
int
VOcoSubtype (
        OBJECT clr)
```

VOcoSubtype returns the subtype of the color object, *clr*. Returns *NULL* if *clr* is not a valid color object. Possible returned subtypes are:

COLOR_INDEX	*color table index or device-dependent format*
COLOR_COMPONENTS	*three color primaries in the range [0,255]*
COLOR_REFERENCE	*referenced color object*

If the color object was created with *COLOR_NAME*, *VOcoSubtype* returns *COLOR_COMPONENTS*.

If the color object was created with *COLOR_STRUCTURE* and *COLOR_SPEC* is RGB, *VOcoSubtype* returns *COLOR_COMPONENTS*.

If the color object was created with *COLOR_STRUCTURE* and *COLOR_SPEC* is *INDEX*, *VOcoSubtype* returns *COLOR_INDEX*.

VOdbg (VOdebug)

VOdbg Functions VO Routines

General debug and statistics routines. These routines can be called directly by the debugger on some systems and are therefore not located in the library, but occur as source modules in the *tooldebug* subdirectory of the *src* directory. Note that all references to “print” in the descriptions below refer to printing to the standard output.

Vodbg Functions

VOdbgAttr	Prints attributes data structure.
VOdbgCounts	Prints the numbers of each kind of *VO* object.
VOdbgDqList	Lists useful information about each object in the deque.
VOdbgOb	Prints statistics about a specified object.
VOdbgObPts	Prints the control points for a given object.

VOdbgAttr

VOdbg Functions VO Routines

Prints attributes data structure.

```
void
VOdbgAttr (
        ATTRIBUTES *attributes)
```

VOdbgAttr prints every non-empty field of *attributes*. A non-empty field is any field not set to *EMPTY_FIELD*. For example, fill status is reported as filled or non-filled, text direction as vertical or horizontal. Other information, such as objects and dimensional or structural information, is given in hexadecimal or decimal form respectively.

VOdbgCounts

VOdbg Functions VO Routines

Prints the numbers of each kind of *VO* object.

```
void
VOdbgCounts (void)
```

VOdbgCounts counts and returns the number of VO objects allocated. Also gives the number of changes that have occurred, if any, since the last time *VOdbgCounts* was called. Information is given in the following form:

`bb : nn -cc` or `bb : nn +cc`

where bb stands for the object (*ar* = arc, *ci* = circle, etc.), nn = how many objects are currently allocated, and (-)(+) cc = is the change in the number of objects since the last call.

VOdbgDqList

VOdbg Functions VO Routines

Lists useful information about each object in the deque.

```
int
VOdbgDqList (
       OBJECT deque)
```

VOdbgDqList calls VOdbgOb for all of the objects in a deque. If the deque is valid, information is printed about every object in the deque. A non-valid deque prints nothing and returns a value of *-1*.

VOdbgOb

VOdbg Functions VO Routines

Prints statistics about a specified object.

```
int
VOdbgOb (
       OBJECT object)
```

VOdbgOb prints information about the object, including its internal representation in hexadecimal, its type, its attributes, and if valid, object-specific information.

VOdbgObPts

VOdbg Functions VO Routines

Prints the control points for a given object.

```
int
VOdbgObPts (
       OBJECT object)
```

VOdbgObPts prints the world coordinate values of every control point of the object. Coordinates are printed as (x,y) pairs. The routine also returns the number of control points if valid. Otherwise returns zero.

VOdg (VOdatagroup)

VOdg Functions VO Routines

Manages data group objects (*dg*). Data group objects, which are also called graphs, manage lower level data structures known as data groups (*dgp*). Data groups contain variable descriptors (*vdp*) and one display formatter (*df*), and are manipulated with the *VPdg* and *VGdg* routines. The variable descriptors supply the data group with data and the display formatter describes how this data is to be displayed on the screen.

If a data group object is too large for its drawport, it is clipped to fit within the drawport boundary. A data group object also gets clipped if it is obscured by another drawport.

Data group objects use foreground and background color attributes, and inherit foreground and background colors. When they do not inherit foreground and background colors, the default colors are a white foreground on a black background. Note that on monochrome systems the color sense for black and white is the opposite of the color sense of black and white on color systems.

Data groups cannot be multiply referenced.

Vodg Functions

VOdgAtGet	See VOobAtGet.
VOdgAtSet	See VOobAtSet.
VOdgBox	See VOobBox.
VOdgClone	See VOobClone.
VOdgCreate	Creates a data group object.
VOdgDereference	See VOobDereference.
VOdgGetDgp	Returns the pointer to an object's data group structure.
VOdgIntersect	See VOobIntersect.
VOdgIsDrawable	Determines if the data group is drawable.
VOdgIsDrawn	Determines if the data group has been drawn.
VOdgPtGet	See VOobPtGet.
VOdgPtSet	See VOdbPtSet
VOdgRefCount	See VOobRefCount.
VOdgReference	See VOobReference.
VOdgReset	Resets the data group object to start at beginning.
VOdgStatistic	Returns statistics about data group objects.
VOdgTraverse	See VOobTraverse.
VOdgValid	See VOobValid.
VOdgXfBox	See VOobXfBox.
VOdgXformBox	See VOobXformBox.

A *VOdg* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOdg* routine to save the overhead of an additional routine call.

VOdgCreate

VOdg Functions VO Routines

Creates a data group object.

```
OBJECT
VOdgCreate (
       DATAGROUP dgp,
       OBJECT ll,
       OBJECT ur,
       ATTRIBUTES *attributes)
```

VOdgCreate creates and returns a data group object defined by the lower left (*ll*) and upper right (*ur*) point subobjects. If the data group structure, *dgp*, does not already exist, the routine creates a data group structure with a default display formatter, *VDbar*, and creates and attaches one variable descriptor. Note that if you pass in a data group structure, it is destroyed when the data group object is destroyed. Valid *attributes* field flags are:

FOREGROUND_COLOR
BACKGROUND_COLOR

VOdgGetDgp

VOdg Functions VO Routines

Returns the pointer to an object's data group structure.

```
DATAGROUP
VOdgGetDgp (
       OBJECT dg)
```

VOdgGetDgp returns the pointer to the data group structure being managed by the data group object, *dg*.

VOdgIsDrawable

VOdg Functions VO Routines

Determines if the data group is drawable.

```
BOOLPARAM
VOdgIsDrawable (
       OBJECT dg,
       OBJECT xform)
```

VOdgIsDrawable determines if the data group, *dg*, is drawable: that is, whether it can be rendered correctly with the specified Xform, without errors such as “Viewport too small.” Drawability depends on constraints of the attached display formatter and context flags set for the data group. *VOdgIsDrawable* checks drawability by cloning the data group and passing the clone to *VPdgsetup*. After testing, destroys the clone. Returns *DV_SUCCESS* if the data group is drawable. Otherwise, returns *DV_FAILURE*.

VOdgIsDrawable is not intended as a validity check and may give unpredictable results if you pass it an invalid data group object. To check validity, use *VOdgValid.*

VOdgIsDrawn

VOdg Functions VO Routines

Determines if the data group has been drawn.

```
BOOLPARAM
VOdgIsDrawn (
       OBJECT dg)
```

VOdgIsDrawn determines if the display formatter associated with the data group, *dg*, has been drawn. If the display formatter has been set up and the context has been drawn, the display formatter is considered to be drawn. Returns *YES* if the display formatter is drawn. Otherwise, returns *NO*.

VOdgReset

VOdg Functions VO Routines

Resets the data group object to start at beginning.

```
void
VOdgReset (
       OBJECT dg)
```

VOdgReset resets the data group object, *dg*, to its initial state. The next time the data group is drawn, the graph's context is redrawn. This also frees temporary storage allocated the last time the graph ran.

VOdgStatistic

VOdg Functions VO Routines

Returns statistics about data group objects.

```
LONG
VOdgStatistic (
       int flag)
```

VOdgStatistic returns statistics about data groups, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, *VOdgStatistic* returns the current number of data groups.

VOdq (VOdeque)

VOdq Functions VO Routines

Manages deque objects (*dq*). Deques are used to manage lists of objects. For example, drawing objects maintain their contents by using deques of graphical objects. Deques can also be used to manage lists of non-objects that fit into a *LONG*. For lists of non-objects, use VOdqCreateGeneric to create a deque of non-objects. Then use the other routines normally.

Objects can be inserted at the top or bottom of the deque or at a specific index position in the deque. Objects can be deleted by their object id or by their position in the deque. You can also insert and delete deques of objects. Objects anywhere in the list can be accessed by their index value in the deque. The index starts at 1 on the bottom of the deque and increases to the maximum index at the top of the deque. As with all subobjects, the items in the deque can be shared with other deques.

When objects are added to or deleted from deques using these routines, reference counts for the objects are handled automatically.

The deque should more accurately be called a list manager; however, the name deque is retained for historical purposes.

Vodq Functions

VOdqAdd	Adds an object at the top or the bottom of a deque.
VOdqAddDq	Adds a deque of objects to the top or the bottom of a deque.
VOdqAddDqIndexed	Adds a deque of objects after the given index.
VOdqAddIndexed	Adds an object after the given index.
VOdqClone	See VOobClone.
VOdqCreate	Creates a deque of objects.
VOdqCreateGeneric	Creates a deque of non-objects.
VOdqDelete	Deletes an object from the deque.
VOdqDeleteAll	Deletes all entries from the deque.
VOdqDeleteDq	Deletes a deque of objects from the deque.
VOdqDeleteIndexed	Deletes the object at a given index.
VOdqDereference	See VOobDereference.
VOdqGetEntry	Returns the object at a given index position in the deque.
VOdqHasEntry	Determines if the object is in the deque and returns its index.
VOdqRefCount	See VOobRefCount.
VOdqReference	See VOobReference.
VOdqReplaceEntry	Replaces one object in the deque with another.
VOdqSize	Gets the number of entries in the deque.
VOdqSort	Sorts the deque using a user-supplied comparison.
VOdqStatistic	Returns statistics about deques.
VOdqSwapEntries	Swaps two entries in the table.
VOdqTraverse	See VOobTraverse.
VOdqValid	See VOobValid.
VOdqVersion	Gets the version number of the deque.

A *VOdq* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOdq* routine to save the overhead of an additional routine call.

VOdqAdd

VOdq Functions VO Routines

Adds an object at the top or the bottom of a deque.

```
void
VOdqAdd (
       OBJECT deque,
       int position,
       OBJECT object)
```

VOdqAdd adds the object to the top or the bottom of *deque* as specified by *position*. *position* can be either *TOP*, for the top of the list, or *BOTTOM*, for the bottom of the list.

VOdqAddDq

VOdq Functions VO Routines

Adds a deque of objects to the top or the bottom of a deque.

```
void
VOdqAddDq (
       OBJECT deque,
       int position,
       OBJECT obdeque)
```

VOdqAddDq adds a deque of objects to the top or the bottom of *deque* as specified by *position*. *position* can be either *TOP* for the top of the list, or *BOTTOM* for the bottom of the list.

VOdqAddDqIndexed

VOdq Functions VO Routines

Adds a deque of objects after the given index.

```
void
VOdqAddDqIndexed (
       OBJECT deque,
       int index,
       OBJECT obdeque)
```

VOdqAddDqIndexed adds a deque of objects to the *deque* after the given *index* position. Because the *index* values are 1-based, an *index* of 0 means to add the object to the beginning.

VOdqAddIndexed

VOdq Functions VO Routines

Adds an object after the given index.

```
void
VOdqAddIndexed (
       OBJECT deque,
       int index,
       OBJECT object)
```

VOdqAddIndexed adds an *object* to the *deque* after the given *index* position. Because the index values are 1-based, an index of 0 means to add the object to the beginning.

VOdqCreate

VOdq Functions VO Routines

Creates a deque of objects.

```
OBJECT
VOdqCreate (
       int initial_size)
```

VOdqCreate creates and returns an empty deque object. *initial_size* specifies the initial memory to allocate for storing the contents of the deque. Allocating initial memory is only an efficiency measure since the deque object allocates new memory if it grows beyond this size. If *initial_size* is *NULL*, a default of 10 is used.

VOdqCreateGeneric

VOdq Functions VO Routines

Creates a deque of non-objects.

```
OBJECT
VOdqCreateGeneric (
       int initial_size,
       VODQADDFUNPTR addfun,
       VODQDELFUNPTR delfun,
       VODQEQUALFUNPTR is_equalfun)

   OBJECT
   addfun (
         OBJECT entity)

   void
   delfun (
         OBJECT entity)

   BOOLPARAM
   is_equalfun (
         OBJECT entity1,
         OBJECT entity2)
```

VOdqCreateGeneric creates a deque object that contains non-objects. You can specify functions to be called before the entity is added to the list and before it is deleted from the list. The entity that is added or deleted must fit into an *OBJECT*, which is type *LONG*. *addfun* should be defined to take an *entity* and return the *entity_to_be_added*. *delfun* should be defined to free or decrement the reference count of *entity*. *is_equalfun* should be defined to take *entity1* and *entity2* and return *YES* if they are equal. Otherwise, should return *NO*.

VOdqDelete

VOdq Functions VO Routines

Deletes an *object* from the *deque*.

```
void
VOdqDelete (
       OBJECT deque,
       OBJECT object)
```

VOdqDeleteAll

VOdq Functions VO Routines

Deletes all entries from the deque.

```
void
VOdqDeleteAll (
       OBJECT deque)
```

VOdqDeleteAll removes all entries from the *deque*. This routine sets the empty slots to *NULL*.

VOdqDeleteDq

VOdq Functions VO Routines

Deletes a deque of objects from the deque.

```
void
VOdqDeleteDq (
       OBJECT deque,
       OBJECT obdeque)
```

VOdqDeleteDq removes a deque of objects from the *deque*. Any *obdeque* objects that are in *deque* are removed from deque.

VOdqDeleteIndexed

VOdq Functions VO Routines

Deletes the object at a given index.

```
void
VOdqDeleteIndexed (
       OBJECT deque,
       int position)
```

VOdqDeleteIndexed deletes the object at the specified *position* in the *deque*. *position* is the 1-based index of the entry in the deque.

VOdqGetEntry

VOdq Functions VO Routines

Returns the object at a given index position in the deque.

```
OBJECT
VOdqGetEntry (
       OBJECT deque,
       int index)
```

VOdqGetEntry searches the deque for the object specified by the index and returns the object. An index of 1 refers to the bottom of the list.

VOdqHasEntry

VOdq Functions VO Routines

Determines if the object is in the deque and returns its index.

```
int
VOdqHasEntry (
       OBJECT deque,
       OBJECT object)
```

VOdqHasEntry searches the deque for the object. Returns the object’s index if the object is found. Otherwise returns zero. An index of 1 refers to the bottom of the list.

VOdqReplaceEntry

VOdq Functions VO Routines

Replaces one object in the deque with another.

```
void VOdqReplaceEntry (
       OBJECT deque,
       int position,
       OBJECT object)
```

VOdqReplaceEntry replaces an indexed object in the deque with another object. Use *VOdqHasEntry* to determine the index *position* of the object.

VOdqSize

VOdq Functions VO Routines

Returns the number of entries in the deque.

```
int
VOdqSize (
	OBJECT deque)
```

VOdqSort

VOdq Functions VO Routines

Sorts the deque using a user-supplied comparison.

```
void
VOdqSort (
       OBJECT deque,
       VODQCOMPAREFUNPTR compare_fun)

    int
    compare_fun (
           OBJECT entry1,
           OBJECT entry2)
```

VOdqSort sorts the *deque* according to the caller-supplied comparison function, *compare_fun*. *compare_fun* should be defined to return the following values:

-1	*if* entry1 < entry2
0	*if* entry1 == entry2
+1	*if* entry1 > entry2

VOdqStatistic

VOdq Functions VO Routines

Returns statistics about deques.

```
LONG
VOdqStatistic (
       int flag)
```

VOdqStatistic returns statistics about deques, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of deques.

VOdqSwapEntries

VOdq Functions VO Routines

Swaps two entries in the table.

```
void
VOdqSwapEntries (
       OBJECT deque,
       int position1,
       int position2)
```

VOdqSwapEntries swaps the entry in *position1* with the entry in *position2* in the specified *deque*.

VOdqVersion

VOdq Functions VO Routines

Gets the version number of the deque.

```
LONG
VOdqVersion (
       OBJECT deque)
```

VOdqVersion returns the version number of the specified *deque*. The version number of a deque starts at zero and is incremented every time the deque contents are changed by adding, deleting, replacing, sorting, or swapping entries.

VOdr (VOdrawing)

VOdr Functions   VO Routines

Manages drawing objects (*dr*). A drawing object contains a deque of graphical objects and an associated name list for named objects. It also contains a foreground color, which is used to draw objects that have no foreground color of their own, and a background color, which is used to erase objects in the drawing. A drawing can be viewed in one or more drawports and it can contain any of the graphical objects. Many of the operations on drawings can be handled at the T level by the Tdr, Tdp, and Tvi routines.

Vodr Functions

VOdrAddName	Names an object in the drawing.
VOdrBackcolor	Sets the drawing's background color.
VOdrBounds	Gets drawing boundary given a transformation.
VOdrBox	See VOobBox.
VOdrClone	See VOobClone.
VOdrCreate	Creates a drawing object.
VOdrDeleteName	Deletes the name of an object in the drawing.
VOdrDereference	See VOobDereference.
VOdrForecolor	Sets the drawing's foreground color.
VOdrGetName	Gets the name of an object.
VOdrGetNamedObject	Gets the object with a name.
VOdrGetObjectDeque	Gets the deque object containing the drawing's objects.
VOdrGetScale	Gets the default scale for a drawing.
VOdrIntersect	See VOobIntersect.
VOdrNameTraverse	Traverses the drawing's name list.
VOdrObAdd	Adds an object to the drawing.
VOdrObAddNamed	Adds a named object to the drawing.
VOdrObBottom	Moves an object to the bottom of the drawing.
VOdrObDelete	Deletes an object from the drawing.
VOdrObReplace	Replaces the current object with a new object.
VOdrObTop	Moves an object to the top of the drawing.
VOdrOffcolor	Sets the color of the off-drawing region.
VOdrRefCount	See VOobRefCount.
VOdrReference	See VOobReference.
VOdrSetScale	Sets the default scale for a drawing.
VOdrStatistic	Returns statistics about drawings.
VOdrTraverse	See VOobTraverse.
VOdrValid	See VOobValid.
VOdrXfBox	See VOobXfBox.
VOdrXformBox	See VOobXformBox.

A *VOdr* routine that refers to a VOob routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOdr* routine to save the overhead of an additional routine call.

VOdrAddName

VOdr Functions VO Routines

Names an object in the drawing.

```
void
VOdrAddName (
       OBJECT drawing,
       OBJECT object,
       char *name)
```

VOdrAddName assigns a name string, *name*, to the object in the drawing. Does nothing if the object is not in the drawing.

VOdrBackColor

VOdr Functions VO Routines

Sets the drawing's background color.

```
OBJECT
VOdrBackcolor (
       OBJECT drawing,
       OBJECT color)
```

VOdrBackcolor sets the drawing's background color. Returns the old color. Special values of *color* have the following meanings:

NULL	*The drawing's background inherits the screen background color.*
NO_BACKGROUND	*The background is to be transparent.*
DONT_SET_THE_VALUE	*The color remains unchanged. Returns the current color.*

VOdrBounds

VOdr Functions VO Routines

Gets drawing boundary given a transformation.

```
void
VOdrBounds (
       OBJECT xform,
       RECTANGLE *bounds)
```

VOdrBounds gets the boundary, *bounds*, of the whole world coordinate space, expressed in screen coordinates, after being transformed by the transformation *xform*.

VOdrCreate

VOdr Functions VO Routines

Creates a drawing object.

```
OBJECT
VOdrCreate (void)
```

VOdrCreate creates and returns a drawing object. A drawing uses foreground and background color attributes.

VOdrDeleteName

VOdr Functions VO Routines

Deletes the name of an object in the drawing.

```
void
VOdrDeleteName (
       OBJECT drawing,
       OBJECT object)
```

VOdrDeleteName deletes the name of *object* in *drawing*. Does nothing if the object is not in the drawing.

VOdrForecolor

VOdr Functions VO Routines

Sets the drawing's foreground color.

```
OBJECT
VOdrForecolor (
       OBJECT drawing,
       OBJECT color)
```

VOdrForecolor sets the drawing's foreground color. Returns the old color. If the color flag is *NULL*, the drawing inherits the screen foreground color. If the color flag is *DONT_SET_THE_VALUE*, the color remains unchanged and the routine returns the current color.

VOdrGetName

VOdr Functions VO Routines

Gets the name of an object.

```
char *
VOdrGetName (
       OBJECT drawing,
       OBJECT object)
```

VOdrGetName returns the name of *object* in *drawing*. Returns a pointer to an internal string which should not be modified.

VOdrGetNamedObject

VOdr Functions VO Routines

Gets the object with a name.

```
OBJECT
VOdrGetNamedObject (
       OBJECT drawing,
       char *name)
```

VOdrGetNamedObject searches *drawing* for the first object with the name, *name*. Returns the object if successful. Otherwise returns *NULL*.

VOdrGetObjectDeque

VOdr Functions VO Routines

Gets the deque object containing the drawing’s objects.

```
OBJECT
VOdrGetObjectDeque (
       OBJECT drawing)
```

VOdrGetObjectDeque returns the deque object containing all the objects in *drawing*. This deque is an internal structure that should be modified with care. For most actions such as adding, deleting, or reordering objects, you should operate on the drawing object using VOdr routines instead of operating on the deque.

VOdrGetScale

VOdr Functions VO Routines

Gets the default scale for a drawing.

```
double
VOdrGetScale (
       OBJECT drawing)
```

VOdrGetScale returns the default scale factor associated with the drawing. If the drawing has no default scale factor, this routine returns 0, which is an invalid scale factor.

VOdrNameTraverse

VOdr Functions VO Routines

Traverses the drawing's name list.

```
ADDRESS
VOdrNameTraverse (
       OBJECT drawing,
       VODRNAMETRVRSFUNPTR fun,
       ADDRESS args)

    ADDRESS
    fun (
           OBJECT object,
           char *object_name,
           ADDRESS args)
```

VOdrNameTraverse traverses all the named objects in the drawing and calls *fun (object, object_name, args)* for each named object. Continues traversal while fun returns *NULL* or *V_CONTINUE_TRAVERSAL*. Aborts the traversal when *fun* returns a non-*NULL ADDRESS* or *V_HALT_TRAVERSAL*. The return value of the traversal is the return value of the last call to *fun*.

fun must be provided by the programmer to perform whatever operation is required. It should return an *ADDRESS*, and must have three parameters: the object being processed, the name of the object, and the argument or argument block required by the function. The argument can be *NULL*. If more than one argument is required, the argument block should be a pointer to a structure that holds the arguments or addresses of the arguments required.

The *fun* function is typically used in one of two ways:

1) to perform a particular operation on each named object in the drawing, or
2) to find a particular object with a given name.

In the first case, *fun* should be written so that it always returns *V_CONTINUE_TRAVERSAL* or *NULL* for *ADDRESS*. In the second case, *fun* should return a *NULL* value for *ADDRESS* if the object is not found. Otherwise it should return the *ADDRESS* of the object.

Note: You should not alter the drawing by adding, deleting, or reordering the named objects during traversal.

For an example of a typical function, see the example under TdrForEachNamedObject.

VOdrObAdd

VOdr Functions VO Routines

Adds an object to the drawing.

```
BOOLPARAM
VOdrObAdd (
       OBJECT drawing,
       OBJECT object)
```

VOdrObAdd adds the object to the top of the drawing deque. When drawn, the added object is drawn last, in front of the other objects in the drawing. Returns *YES* if successful. Otherwise returns *NO*.

VOdrObAddNamed

VOdr Functions   VO Routines

Adds a named object to the drawing.

```
BOOLPARAM
VOdrObAddNamed (
       OBJECT drawing,
       OBJECT obj,
       char *name;
```

VOdrObAddNamed adds the named object to the top of the drawing queue. It combines the features of VOdrObAdd and VOdrAddName. When drawn, the added object is drawn last, in front of the other objects in the drawing. Returns *YES* if successful. Otherwise returns *NO*.

VOdrObBottom

VOdr Functions VO Routines

Moves an object to the bottom of the drawing.

```
void
VOdrObBottom (
       OBJECT drawing,
       OBJECT object)
```

VOdrObBottom moves the object to the bottom of the drawing. When drawn, the object is drawn first, behind the other objects in the drawing.

VOdrObDelete

VOdr Functions VO Routines

Deletes an object from the drawing.

```
BOOLPARAM
VOdrObDelete (
       OBJECT drawing,
       OBJECT object)
```

VOdrObDelete deletes the object from the drawing. Returns *YES* if successful. Otherwise returns *NO*.

VOdrObReplace

VOdr Functions VO Routines

Replaces the current object with a new object.

```
BOOLPARAM
VOdrObReplace (
       OBJECT drawing,
       OBJECT currobj,
       OBJECT newobj)
```

VOdrObReplace replaces the current object with a new object. This routine ensures that when a named object is replaced, the new object receives the name of the replaced object. The replaced object is dereferenced. Returns *NO* if one or both objects do not exist.

VOdrObTop

VOdr Functions VO Routines

Moves an object to the top of the drawing.

```
void
VOdrObTop (
       OBJECT drawing,
       OBJECT object)
```

VOdrObTop moves the object to the top of the drawing. When drawn, the object is drawn last, in front of the other objects in the drawing.

VOdrOffcolor

VOdr Functions VO Routines

Sets the color of the off-drawing region.

```
OBJECT
VOdrOffcolor (
       OBJECT color)
```

VOdrOffcolor sets the color object, *color*, to be used when drawing the region beyond the drawing's coordinates. This routine sets a global variable, used for all drawings, and is not associated with a particular drawing object. If the color parameter has the value *DONT_SET_THE_VALUE*, the current off-drawing color is returned. If the *color* parameter has the value *NO_OFF_DRAWING_COLOR*, then the off-drawing region is not drawn and appears transparent. The default off-drawing region color is the background color of the drawing object.

VOdrSetScale

VOdr Functions VO Routines

Sets the default scale for a drawing.

```
void
VOdrSetScale (
       OBJECT drawing,
       double scale)
```

VOdrSetScale sets the default scale factor for the drawing. A *scale* value of zero means to delete the current scale factor. Zero is an invalid scale factor.

VOdrStatistic

VOdr Functions VO Routines

Returns statistics about drawings.

```
LONG
VOdrStatistic (
       int flag)
```

VOdrStatistic returns statistics about drawings, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of drawing objects.

VOdy (VOdynamic)

VOdy Functions VO Routines

Manages dynamic control objects. A dynamic control object is used to describe and control the dynamic behavior of associated graphical objects.

The VOobDyUtil module contains routines that manage the connection between dynamic control objects and graphical objects. To access a dynamic control object using the name assigned in DV-Draw, see *VOuObMatchNameSlots*.

Vody Functions

VOdyAttachData	Attaches a data object to a dynamic control object.
VOdyChanged	Determines if a graphical object's dynamic control object has changed.
VOdyClone	See VOobClone.
VOdyCreate	Creates a dynamic control object.
VOdyDetachData	Detaches a data object from the dynamic control object.
VOdyGetDataObj	Returns the *index*-th data object attached to dynamic control object.
VOdyGetEraseColor	Gets the erase color for a dynamic control object.
VOdyGetEraseMethod	Gets the erase method for a dynamic control object.
VOdyGetPath	Gets the polygon path for a dynamic action.
VOdyGetRange	Gets the range for a specific dynamic action.
VOdyGetRefPoint	Gets the reference point of a dynamic action that uses a reference point.
VOdyGetTextFormat	Gets the text format for text dynamics.
VOdyReset	Returns a graphical object to its original state before dynamics were applied.
VOdySetEraseColor	Sets the erase color for a dynamic control object.
VOdySetEraseMethod	Sets the erase method for a dynamic control object.
VOdySetPath	Sets the polygon path for a dynamic action.
VOdySetRange	Sets the range for a specific dynamic action.
VOdySetRefPoint	Sets the reference point of a dynamic action that uses a reference point.
VOdySetState	Sets the state of an object's dynamic control object to *YES* or *NO*.
VOdySetTextFormat	Sets the text format for text dynamics.
VOdyTraverse	See VOobTraverse.
VOdyUpdate	Updates the current dynamics for a given object.
VOdyValid	See VOobValid.

A *VOdy* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOdy* routine to save the overhead of an additional routine call.

VOdyAttachData

VOdy Functions VO Routines

Attaches a data object to a dynamic control object.

```
BOOLPARAM
VOdyAttachData (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj,
       float *low_range,
       float *high_range,
       OBJECT ref_point,
       OBJECT polygon_path)
```

VOdyAttachData activates a dynamic action and attaches a data object to the dynamic control object, *dycontrol_obj*. *dyn_action_flag* indicates the action that becomes dynamic. *data_obj* provides the changing data for the dynamic action. A data object can be either a threshold table or a variable descriptor object. The other parameters, *low_range*, *high_range*, *ref_point*, and *polygon_path* are used only for specific dynamic actions. The range parameters are used only if the data object is a variable descriptor object. The reference point can be used for certain transformation dynamics. The polygon path is used for the dynamic action for movement along a path.

Visibility Dynamics: Changes whether or not a graphical object is visible. This dynamic action is defined by the dynamic action flag, *V_DYN_VISIBILITY*, and a threshold table. The output values in the threshold table must be *YES* or *NO*. When the object is not visible, its other dynamic actions are still updated. When the object becomes visible, it reflects the current state of these dynamic actions. An object must be visible to be pickable using *TloGetSelectedObject* or *TloGetSelectedObjectName*, or to have its event requests serviced. The *V_DYN_ERASE_XOR* and *V_DYN_ERASE_NONE* erase methods are not useful for visibility dynamics.

Transformation Dynamics: The following table shows the action flags for transformation dynamics and the parameters that each uses. All transformation dynamic actions use the *low_range* and *high_range* parameters. The data object should be a variable descriptor object.

dynamic action flag	**reference points**	**polygon path**
V_DYN_ROTATE	optional	
V_DYN_PATH_MOVE	optional	yes
V_DYN_REL_MOVE_X	no	
V_DYN_REL_MOVE_Y	no	
V_DYN_ABS_MOVE_X	optional	
V_DYN_ABS_MOVE_Y	optional	
V_DYN_SCALE	optional	
V_DYN_SCALE_X	optional	
V_DYN_SCALE_Y	optional	

The parameters *low_range* and *high_range* ensure that the graphical object receives valid data from a variable descriptor object. The incoming data range is determined by the data object's variable descriptor. The outgoing data range is set using *low_range* and *high_range*. The incoming data range is mapped to the outgoing data range. For example, rotation might have an incoming range of [0,1] mapped to an outgoing data range of [0,360].

A reference point can be specified for dynamic actions that involve rotation, absolute movement, scaling, and movement along a path. If the value of *ref_pt* is *NULL*, the graphical object's center is used as the reference point for transformation dynamics.

Certain transformation dynamics can be defined more than once using different data objects. It is also effective to use different reference points. The dynamic action flags that can be used more than once in a dynamic control object are *V_DYN_ROTATE*, *V_DYN_REL_MOVE_X*, *V_DYN_REL_MOVE_Y*, *V_DYN_SCALE*, *V_DYN_SCALE_X*, and *V_DYN_SCALE_Y*.

Attribute Dynamics: Dynamic actions that change object attributes are defined by an action flag and a data object. No additional parameters are required. While some attribute dynamics can use a variable descriptor object directly, using a threshold table is recommended. Valid attribute action flags are:

FOREGROUND_COLOR	*BACKGROUND_COLOR*
FILL_STATUS	*LINE_TYPE*
LINE_WIDTH	*TEXT_DIRECTION*
ARC_DIRECTION	*CURVE_TYPE*
TEXT_FONTNAME	*TEXT_POSITION*
TEXT_FONT	*TEXT_SIZE*
TEXT_WIDTH	*TEXT_HEIGHT*
TEXT_ANGLE	*TEXT_SLANT*
TEXT_CHARSPACE	*TEXT_LINESPACE*
TEXT_NAME	

Proportional Fill: Proportional Fill is a special case of attribute dynamics that works only on objects that have the fill status attribute set to *FILL*, *EDGE_WITH_FILL*, or *FILL_WITH_EDGE*. The data object value is mapped to the percentage of the object to be filled, which is set using *high_range* and *low_range*. A variable descriptor object is recommended as the data object. The direction of the proportional fill is specified by one of the following dynamic action flags: *V_DYN_FILL_RIGHT*, *V_DYN_FILL_UP*, *V_DYN_FILL_LEFT*, *V_DYN_FILL_DOWN*.

Text Dynamics: Displays the formatted variable value. The variable can be embedded in a text string. The dynamic action flag is *V_DYN_TEXT* and works only on text or vector text objects. The data object should be a variable descriptor object. *VOdySetTextFormat* sets the format string. The *V_DYN_ERASE_XOR* and *V_DYN_ERASE_NONE* erase methods are not useful for text dynamics.

Subdrawing Dynamics: The subdrawing dynamic action is defined by the dynamic action flag, *V_DYN_SUBDRAWING*, and a threshold table. Each element of the threshold table is associated with a subdrawing object.

Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdyChanged

VOdy Functions VO Routines

Determines if a graphical object's dynamic control object has changed.

```
BOOLPARAM
VOdyChanged (
       OBJECT dycontrol_obj,
       OBJECT graphical_obj)
```

VOdyChanged determines if *dycontrol_obj*, associated with *graphical_obj*, has changed since the last update of the data. Returns *YES* if the dynamic control object has changed. Otherwise returns *NO*.

VOdyCreate

VOdy Functions VO Routines

Creates a dynamic control object.

```
OBJECT
VOdyCreate ( void)
```

VOdyCreate creates a dynamic control object with no associated dynamic actions or graphical objects. Use *VOdyAttachData* to add dynamic actions to the dynamic control object and *VOobDySet to associate* the dynamic control object with a graphical object. If successful, returns a new dynamic control object. Otherwise returns *NULL*.

VOdyDetachData

VOdy Functions VO Routines

Detaches a data object from the dynamic control object.

```
void
VOdyDetachData (
       OBJECT dycontrol_obj,
       int dyn_action_flag)
```

VOdyDetachData removes the dynamic action associated with *dyn_action_flag* from *dycontrol_obj*. Passing a *NULL data_obj* causes the first dynamic action with *dyn_action_flag* to be removed. See also *VOdyAttachData*.

VOdyGetDataObj

VOdy Functions   VO Routines

Returns the *index*-th data object attached to a dynamic control object.

```
OBJECT
VOdyGetDataObj (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       int index)
```

VOdyGetDataObj returns the data object attached to *dycontrol_obj* that is associated with *dyn_action_flag*. The data object can be either a threshold table object or a variable descriptor object. Most dynamic action flags can only have one data object in a particular dynamic control object. Only the absolute movement, scaling, and rotation actions can be defined with more than one data object. To distinguish between multiple data objects for the same dynamic action, use *index*, where the value of *index* can range from 1 to the total number of data objects. To determine the number of data objects for a particular dynamic action, set *index* to *0* and this routine returns the number as the return value. If only one data object supplies data for a dynamic action, set *index* to *1*. *VOdyGetDataObj* returns *NULL* if *index* is greater than the total number of data objects for a given dynamic action.

VOdyGetEraseColor

VOdy Functions VO Routines

Returns the erase color for a dynamic control object.

```
OBJECT
VOdyGetEraseColor (
       OBJECT dycontrol_obj)
```

Returns the erase color object for a dynamic control object, *dycontrol_obj*, or returns *V_NO_COLOR*.

VOdyGetEraseMethod

VOdy Functions VO Routines

Gets the erase method for a dynamic control object.

```
int
VOdyGetEraseMethod (
       OBJECT dycontrol_obj)
```

VOdyGetEraseMethod returns the erase method for *dycontrol_obj*. See *VOdySetEraseMethod* for a list of valid erase method flags.

VOdyGetPath

VOdy Functions VO Routines

Gets the polygon path for a dynamic action.

```
OBJECT
VOdyGetPath (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj)
```

VOdyGetPath returns the polygon path for the dynamic action defined with the flag *V_DYN_PATH_MOVE*. Returns *NULL* if no polygon path is defined.

VOdyGetRange

VOdy Functions VO Routines

Gets the range for a specific dynamic action.

```
BOOLPARAM
VOdyGetRange (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj,
       float *low_rangep,
       float *high_rangep)
```

VOdyGetRange gets the range for the dynamic action specified by *dyn_action_flag* and *data_obj*. The range is passed back in *low_rangep* and *high_rangep*. Some transformation dynamic actions can receive data from more than one data object; in this case, use *data_obj* to distinguish between them. If *data_obj* is *NULL*, gets the range corresponding to the first data object for the specified dynamic action. Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdyGetRefPoint

VOdy Functions VO Routines

Gets the reference point of a dynamic action that uses a reference point.

```
OBJECT
VOdyGetRefPoint (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj)
```

VOdyGetRefPoint returns the reference point for a dynamic action that uses a reference point. The dynamic actions rotation, scaling, absolute move, and movement along a path use reference points. Rotation and scaling actions can receive data from more than one data object; in this case, use *data_obj* to distinguish between them. If *data_obj* is *NULL*, returns the reference point corresponding to the first data object for the specified dynamic action.

VOdyGetTextFormat

VOdy Functions VO Routines

Gets the text format for text dynamics.

```
char *
VOdyGetTextFormat (
       OBJECT dycontrol_obj,
       int dyn_action_flag)
```

VOdyGetTextFormat returns the string used to format the variable value associated with the text dynamics action. Use *V_DYN_TEXT* for *dyn_action_flag*. *dycontrol_obj* is the dynamic control object. Returns *NULL* if the action isn’t valid for the dynamic object.

VOdyReset

VOdy Functions VO Routines

Returns a graphical object to its original state before dynamics were applied.

```
void
VOdyReset (
       OBJECT dycontrol_obj,
       OBJECT graphical_obj)
```

VOdyReset resets *graphical_obj* to its original state. Its original state consists of the graphical object's original points and attributes before any dynamics were applied. If *graphical_obj* parameter is *NULL*, resets all the graphical objects associated with *dycontrol_obj*.

VOdySetEraseColor

VOdy Functions VO Routines

Sets the erase color for a dynamic object.

```
BOOLPARAM
VOdySetEraseColor (
       OBJECT dycontrol_obj,
       OBJECT color)
```

VOdySetEraseColor sets the erase color for a dynamic control object, *dycontrol_obj*, that uses the *V_DYN_ERASE_BOX* or *V_DYN_ERASE_OBJECT* erase method. The color can be set at any time, regardless of the current erase method setting. The setting is initialized to *V_NO_COLOR*, and you can clear a color setting by setting the color to *V_NO_COLOR*. If *V_NO_COLOR*, the drawing's background color is used. Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdySetEraseMethod

VOdy Functions VO Routines

Sets the erase method for a dynamic control object.

```
BOOLPARAM
VOdySetEraseMethod (
       OBJECT dycontrol_obj,
       int erase_method)
```

VOdySetEraseMethod specifies the erase method for *dycontrol_obj*. Valid erase method flags are:

- V_DYN_ERASE_REDRAW_IMMEDIATE - *Redraws the objects that were obscured by and obscuring the dynamic object immediately after this object has moved.*
- V_DYN_ERASE_REDRAW_DELAY - *Redraws the objects that were obscured by and obscuring the dynamic object after all dynamic objects have moved.*
- V_DYN_ERASE_RASTER - *Redraws the affected portion of the screen using the raster information saved before drawing the dynamic object in its new position. Not supported on all systems.*
- V_DYN_ERASE_BOX - *Erases the dynamic object by redrawing the area inside the dynamic object's bounding box either in the drawing's background color or in a color specified by* VOdySetEraseColor.
- V_DYN_ERASE_OBJECT - *Erases the dynamic object by redrawing the dynamic object either in the drawing's background color or in a color specified by* VOdySetEraseColor.
- V_DYN_ERASE_XOR - *Erases the object by XORing the object's bits. Not supported on all systems. Not useful for visibility or text dynamics.*
- V_DYN_ERASE_NONE - *No erase occurs. Leaves all versions of the dynamic object on the screen until a subsequent action draws over them. Not useful for visibility or text dynamics.*

Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdySetPath

VOdy Functions VO Routines

Sets the polygon path for a dynamic action.

```
BOOLPARAM
VOdySetPath (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj,
       OBJECT polygon_path)
```

VOdySetPath sets the polygon path for a dynamic action that is defined using the *V_DYN_PATH_MOVE* dynamic action flag.

VOdySetRange

VOdy Functions VO Routines

Sets the range for a specific dynamic action.

```
BOOLPARAM
VOdySetRange (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj,
       float *low_range,
       float *high_range)
```

VOdySetRange sets the ranges for a specific dynamic action affecting the dynamic control object to values pointed to by *low_range* and *high_range*. If either of the pointers holding the ranges is *NULL*, the dynamic action is reset to indicate that there is no range. Some transformation dynamic actions can receive data from more than one data object; in this case, use *data_obj* to distinguish between them. If *data_obj* is *NULL*, sets the range corresponding to the first data object for the specified dynamic action. Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdySetRefPoint

VOdy Functions VO Routines

Sets the reference point of a dynamic action that uses a reference point.

```
BOOLPARAM
VOdySetRefPoint (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       OBJECT data_obj,
       OBJECT point)
```

VOdySetRefPoint sets the reference point for a dynamic action that uses a reference point. The dynamic actions rotation, scaling, absolute move, and movement along a path can use reference points. Rotation and scaling actions can receive data from more than one data object; in this case, use *data_obj* to distinguish between them. If *data_obj* is *NULL*, sets the reference point corresponding to the first data object for the specified dynamic action. Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdySetState

VOdy Functions VO Routines

Sets the state of an object's dynamic control object to *YES* or *NO*.

```
BOOLPARAM
VOdySetState (
       OBJECT dycontrol_obj,
       OBJECT graphical_obj,
       int state)
```

VOdySetState sets the dynamic state of *graphical_obj* to be on (*YES*) or off (*NO*). When *state* is *YES*, which is the default, dynamic changes occur every time *VOdyUpdate* is called. When *state* is *NO*, no dynamic changes take place. If *graphical_obj* is *NULL*, *VOdySetState* sets the state for all graphical objects associated with the dynamic control object.

The state is not normally saved when you save a view containing the dynamic control object, or restored when you load a view file. To save or restore the state, set the configuration variable *DVSAVEDYNSTATE* to *yes*. Note that setting the state to *NO* and saving does not save a graphical object in its current state. The only additional information saved is whether or not the graphical object is to be updated.

Returns *DV_SUCCESS* or *DV_FAILURE*.

VOdySetTextFormat

VOdy Functions VO Routines

Sets the text format for text dynamics.

```
BOOLPARAM
VOdySetTextFormat (
       OBJECT dycontrol_obj,
       int dyn_action_flag,
       char *format)
```

VOdySetTextFormat specifies the format string to be used with the *V_DYN_TEXT dyn_action_flag*. You can specify the format using the following C *printf()* conversion characters:

Variable Type	**Conversion Characters**
V_T_TYPE	s
*V_D_TYPE_* and *V_F_TYPE*	f, e, E, g, G
all others	d, i, o, u, x, X

The conversion character can be embedded in a text string. For example: *"volume=%6.2f"*, *"score=%d%%"*, *"account name:%s"*

The default formats are *"%s"*, *"%f"*, *"%d"*. If the format is *NULL*, a default (*%s*, *%f*, or *%d*) corresponding to the type of data is assigned. Returns *DV_FAILURE* if the dynamic action is not activated. Otherwise returns *DV_SUCCESS*.

VOdyUpdate

VOdy Functions VO Routines

Updates the current dynamics for a given object.

```
void
VOdyUpdate (
       OBJECT dycontrol_obj,
       OBJECT graphical_obj)
```

VOdyUpdate updates *dycontrol_obj* for *graphical_obj*. This function affects the attributes and points of the graphical object by looping through each dynamic action and reading the data from the data object to create a new attribute structure or set of points or both. This function also saves the original points and attributes so the application can use *VOdyReset* to restore the object to the state it was in before any dynamic change was made. If *VOdyReset* is not called, the original attributes are not used after they are saved. The original points, however, are used with each update since dynamic changes transform the original points to create a new set of points. If dynamics are turned off, any change resulting from a call to *VOdyUpdate* is ignored and the object remains unchanged.

Examples

The following code fragment, adapted from *dynamics.c*, creates a dynamic control object and attaches data objects to it.

```
/* Create a dynamic control object. */
dyn_control = VOdyCreate ();

/* Attach the two data objects to the dynamic control object. */
VOdyAttachData (dyn_control, V_DYN_ROTATE, rotation_vd, &low_range, &high_range,
             center_point, (OBJECT)NULL);
VOdyAttachData (dyn_control, FOREGROUND_COLOR, color_tt, (float *) NULL, (float *)
             NULL, (OBJECT) NULL, (OBJECT) NULL);

/* Attach the dynamic control object to the dial object. */
VOobDySet (dial, dyn_control);

/* Draw the dial and update the display as the data changes. */
TdpDraw (drawport);
for (n = 0; n < 2000; n++)
{
   dial_input = sin ((double) (n / 15.0)) - cos ((double) (n / 25.0));
   TdpDrawNext (drawport);
```

VOed (VOedge)

VOed Functions VO Routines

Manages edge objects. Edge objects, together with node objects, are used to construct abstract graphs. Graphs are data structures that represent relationships between data. Edges and nodes let you show hierarchical relationships between data. Node objects represent data and edge objects provide the connections between nodes. Some example ways of using this kind of graph are finding the shortest routes between objects, project planning, and electrical circuit analysis. Edge and node objects are provided as application modelling tools for the DataViews environment. For a description of graphs, see any computer science textbook on data structures.

Each edge object is specified by up to two node objects connected by the edge object. The edge direction is defined by the order that the nodes are given to *VOedCreate*. An edge object can have an optional geometry object that graphically represents the edge object. The geometry object must be a graphical object or a deque of graphical objects. If a geometry object is used, it is drawn when the edge object is drawn.

An edge object can have an arbitrary number of slots attached to it that contain user-defined data. Use the *VOslotkey* routines to create and initialize a slot, then use the *VOobSlotUtil* routines to attach the slot to the edge object.

See Also

VOnode module

Voed Functions

VOedAtGet	See VOobAtGet.
VOedAtSet	See VOobAtSet.
VOedBox	See VOobBox.
VOedClearMark	Clears the mark bits of all edge objects.
VOedClearVisit	Clears the visit counts of all edge objects.
VOedClone	See VOobClone.
VOedCreate	Creates an edge object.
VOedDereference	See VOobDereference.
VOedGetGeometry	Gets the geometry object of the edge object.
VOedGetMark	Gets the mark bit of the edge object.
VOedGetNode	Gets a node of the edge object.
VOedGetVisit	Gets the visit count of the edge object.
VOedIntersect	See VOobIntersect.
VOedPtGet	See VOobPtGet.
VOedPtSet	See VOobPtSet.
VOedRefCount	See VOobRefCount.
VOedReference	See VOobReference.
VOedSetGeometry	Sets the geometry object of the edge object.
VOedSetMark	Sets the mark bit of the edge object.
VOedSetNode	Sets a node of the edge object.
VOedSetVisit	Sets the visit count of the edge object.
VOedStatistic	Returns statistics about edge objects.
VOedTraverse	See VOobTraverse.
VOedValid	See VOobValid.
VOedXfBox	See VOobXfBox.
VOedXformBox	See VOobXformBox.

A *VOed* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOed* routine to save the overhead of an additional routine call.

VOedClearMark

VOed Functions VO Routines

Clears the mark bits of all edge objects.

```
void
VOedClearMark (void)
```

VOedClearMark clears the mark bit of all edge objects.

VOedClearVisit

VOed Functions VO Routines

Clears the visit counts of all edge objects.

```
void
VOedClearVisit (void)
```

VOedClearVisit clears the visit counts of all edge objects.

VOedCreate

VOed Functions VO Routines

Creates an edge object.

```
OBJECT
VOedCreate (
       OBJECT Node1,
       OBJECT Node2,
       OBJECT Geometry,
       ATTRIBUTES *attributes)
```

VOedCreate creates and returns an edge object. If the values of *Node1* and *Node2* are not *NULL*, they are added to the edge object in order. *VOedTraverse* visits *Node1* first then *Node2*. If the value of the *Geometry* object is not *NULL*, it can be one of the following: *ar*, *ci*, *dg*, *dq*, *el*, *in*, *ln*, *pt*, *py*, *re*, *sd*, *tx*, *vt*.

VOedGetGeometry

VOed Functions VO Routines

Gets the geometry object of the edge object.

```
OBJECT
VOedGetGeometry (
       OBJECT edge)
```

VOedGetGeometry returns the geometry object of the edge object.

VOedGetMark

VOed Functions VO Routines

Gets the mark bit of the edge object.

```
BOOLPARAM
VOedGetMark (
       OBJECT edge)
```

VOedGetMark returns the mark bit of the edge object.

VOedGetNode

VOed Functions VO Routines

Gets a node of the edge object.

```
OBJECT
VOedGetNode (
       OBJECT edge,
       int index)
```

VOedGetNode returns a node at the *index*-th position of the edge object. If *index* is zero, returns the number of nodes attached to the edge object, which is always 2. Returns *NULL* if passed an invalid *index*.

VOedGetVisit

VOed Functions VO Routines

Gets the visit count of the edge object.

```
LONG
VOedGetVisit (
       OBJECT edge)
```

VOedGetVisit returns the visit count of the edge object.

VOedSetGeometry

VOed Functions VO Routines

Sets the geometry object of the edge object.

```
OBJECT
VOedSetGeometry (
       OBJECT edge,
       OBJECT NewGeometry)
```

VOedSetGeometry sets the geometry of *edge* to *NewGeometry*. For a list of valid geometry objects, see *VOedCreate*. Returns the value of the old geometry object.

VOedSetMark

VOed Functions VO Routines

Sets the mark bit of the edge object.

```
BOOLPARAM
VOedSetMark (
       OBJECT edge,
       BOOLPARAM NewMark)
```

VOedSetMark sets the mark bit of the edge object to *NewMark*. Returns the old value of the mark bit.

VOedSetNode

VOed Functions VO Routines

Sets a node of the edge object.

```
OBJECT
VOedSetNode (
       OBJECT edge,
       int index,
       OBJECT NewNode)
```

VOedSetNode sets a node at the *index*-th position in *edge* to *NewNode*. The value of *index* can be 1, or 2. Returns the old value of the node. Returns *NULL* if passed an invalid index.

VOedSetVisit

VOed Functions   VO Routines

Sets the visit count of the edge object.

```
LONG
VOedSetVisit (
        OBJECT edge,
        LONG NewCount)
```

VOedSetVisit sets the visit count of the *edge* object to *NewCount*. Returns the old value of the visit count.

VOedStatistic

VOed Functions VO Routines

Returns statistics.

```
LONG
VOedStatistic (
       int Flag)
```

VOedStatistic returns statistics. Valid flag values are defined in *VOstd.h*. If the flag is *OBJECT_COUNT*, returns the current number of edges.

VOel (VOellipse)

VOel Functions VO Routines

Manages ellipse objects (*el*). An ellipse is defined by three point subobjects that define the major and minor axis of the ellipse. In DataViews an ellipse object is a generalized implementation of an ellipse where the major axis and minor axes do not have to be perpendicular. Ellipse attributes are foreground color, background color, line type, line width, and fill status.

The ellipse fill status can be *FILL*, *EDGE*, *EDGE_WITH_FILL*, *FILL_WITH_EDGE*, or *DV_TRANSPARENT*. When *EDGE* is used, the boundary is drawn using the line attributes. An ellipse using *DV_TRANSPARENT* fill looks identical to one with *EDGE* only, but you can select it with the cursor anywhere in the interior of the shape. A transparent ellipse does not visually obscure objects behind it, but they cannot be selected through it. When either *EDGE_WITH_FILL* or *FILL_WITH_EDGE* is used, the second feature listed in the fill status flag uses the background color attribute. The foreground color is used in all other cases.

Voel Functions

VOelAtGet	See VOobAtGet.
VOelAtSet	See VOobAtSet.
VOelBox	See VOobBox.
VOelClone	See VOobClone.
VOelCreate	Creates an ellipse object.
VOelDereference	See VOobDereference.
VOelIntersect	See VOobIntersect.
VOelPtGet	See VOobPtGet.
VOelPtSet	See VOobPtSet.
VOelRefCount	See VOobRefCount.
VOelReference	See VOobReference.
VOelStatistic	Returns statistics about ellipses.
VOelTraverse	See VOobTraverse.
VOelValid	See VOobValid.
VOelXfBox	See VOobXfBox.
VOelXformBox	See VOobXformBox.

A *VOel* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOel* routine to save the overhead of an additional routine call.

VOelCreate

VOel Functions   VO Routines

Creates an ellipse object.

```
OBJECT
VOelCreate (
       OBJECT pt1,
       OBJECT pt2,
       OBJECT pt3,
       ATTRIBUTES *attributes)
```

VOelCreate creates and returns an ellipse object. The points *p1*, *p2*, and *p3* define the major and minor axis with *p2* as the center. Valid *attributes* field flags are:

FOREGROUND_COLOR   *FILL_STATUS*
BACKGROUND_COLOR   *LINE_TYPE*
LINE_WIDTH

VOelStatistic

VOel Functions VO Routines

Returns statistics about ellipses.

```
LONG
VOelStatistic (
       int Flag)
```

VOelStatistic returns statistics about ellipses, depending on the value of the flag. Valid flag values are defined in *VOstd.h*. If flag is *OBJECT_COUNT*, *VOarStatistic* returns the current number of ellipses.

VOg (VOgraphics)

VOg Functions VO Routines

Draws graphical objects on the screen using lower level routines. This module can be thought of as a layer that sits on top of the *GR* routines and augments them by allowing clipping to overlapping drawports. The conceptual model for the system resembles that of the *GR* routines, except that all graphical output is clipped to a specified boundary. These routines expect screen coordinates, which are device-dependent. To make a routine device-independent, you can use *GRvcs_to_scs* to convert virtual coordinates to screen coordinates.

These routines can be used to improve drawing speed; they are not recommended for typical DV-Tools applications.

All routines that use the *invp* and *outvps* parameters interpret them as defined below.

invp *The clipping viewport.* invp *is a pointer to a* RECTANGLE *structure that specifies a viewport in screen coordinates. The graphical object (circle, line, etc.) is clipped to this viewport. This parameter must be specified.*

outvps *The obscuring viewports.* outvp *is a pointer to a* NULL*-terminated array of* RECTANGLE *structures specifying viewports in screen coordinates that obscure the graphical object. If* NULL*, clipping to obscuring viewports is not required.*

The *RECTANGLE* structures used for *invp* and *outvps* must contain the **lower left** and **upper right** points. If the *RECTANGLE* structures contain upper left and lower right points, the routines will not work correctly. To sort coordinates in a *RECTANGLE*, call *VOuVpSort*. This routine switches the coordinates if required to ensure that the lower left point is actually below and to the left of the upper right point.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	**VOg**	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Vog Functions

VOgArc	Draws a filled or unfilled arc.
VOgChSize	Sets the character size.
VOgCircle	Draws a filled or unfilled circle.
VOgCubic	Draws a cubic curve.
VOgDot	Draws a dot.
VOgFrame	Draws an unfilled rectangle.
VOgGenericDraw	Draws an object of unknown geometry.
VOgIsVisible	Determines if any part of the object's viewport is visible.
VOgLine	Draws a line.
VOgMultiline	Draws an array of connected lines.
VOgPolygon	Draws a filled polygon.
VOgRaster	Draws a raster image.
VOgRect	Draws a filled rectangle.
VOgReErase	Erases a rectangular area of the screen.
VOgText	Draws text.
VOgTextsize	Gets the size of a text string.
VOgvText	Draws vector text string.
VOgvTextsize	Calculates vector text bounding box.

VOgArc

VOg Functions   VO Routines

Draws a filled or unfilled arc.

```
void
VOgArc (
        DV_POINT *center,
        int radius,
        int start,
        int delta,
        int filled,
        RECTANGLE *invp,
        RECTANGLE **outvps)
```

VOgArc draws a filled or unfilled arc. *start* is the starting angle of the arc in degrees. *delta* is the angle in degrees subtended by the arc. *filled* is a flag indicating whether the arc is filled (*YES*) or not (*NO*). The *invp* and *outvps* parameters are defined above.

VOgChSize

VOg Functions VO Routines

Sets the character size.

```
int
VOgChSize (
       int newsize)
```

VOgChSize sets the size of text to be drawn to *newsize*, and returns the old text size. If *newsize* is *0* (zero), returns the current size of the text, but does not set the size of the text to be drawn.

VOgCircle

VOg Functions VO Routines

Draws a filled or unfilled circle.

```
void
VOgCircle (
       DV_POINT *center,
       int radius,
       int filled,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgCircle draws a filled or unfilled circle. *filled* is a flag indicating whether the circle is filled (*YES*) or not (*NO*). The *invp* and *outvps* parameters are defined above.

VOgCubic

VOg Functions VO Routines

Draws a cubic curve.

```
void
VOgCubic (
       DV_POINT *pts,
       int pattern,
       int width,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgCubic draws a cubic curve. *pts* is a pointer to four coefficient pairs for the parametric equations of the curve. The curve is defined as:

```
x(t) = SUM(i from 0 to 3)
   a[i].x *t ^ (3-i)

y(t) = SUM(i from 0 to 3)
   a[i].y *t ^ (3-i)
```

where ^ means “raised to the power of.” *pattern* is the index of the line pattern. *width* is the width of the line in pixels. The *invp* and *outvps* parameters are defined above.

VOgDot

VOg Functions VO Routines

Draws a dot.

```
void
VOgDot (
       DV_POINT *p,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgDot draws a single pixel. The *invp* and *outvps* parameters are defined above.

VOgFrame

VOg Functions VO Routines

Draws an unfilled rectangle.

```
void
VOgFrame (
       DV_POINT *p1,
       DV_POINT *p2,
       int pattern,
       int width,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgFrame draws an unfilled rectangle. *p1* and *p2* are the diagonally opposite corner points of the rectangle. *pattern* is the index of the line pattern used to draw the rectangle border. *width* is the width of the rectangle outline in pixels. The *invp* and *outvps* parameters are defined above.

VOgGenericDraw

VOg Functions VO Routines

Draws an object of unknown geometry.

```
void
VOgGenericDraw (
       VOGDRAWFUNPTR drawfunction,
       ADDRESS drawargs,
       RECTANGLE *objvp,
       RECTANGLE *invp,
       RECTANGLE **outvps)

   void
   drawfunction(
          ADDRESS drawargs)
```

VOgGenericDraw draws an object of arbitrary geometry using a user-defined drawing function, *drawfunction*. Clipping is provided even if the drawing function does not have clipping capabilities. *drawargs* is a pointer to arguments for use by the drawing function. *objvp* is the smallest viewport that contains the object. The *RECTANGLE* structure used for *objvp* must contain the actual upper right and lower left points. *invp* is the viewport that contains the graphical object. If *invp* is *NULL*, the object is drawn completely within *objvp*. *outvps* is the *NULL*-terminated list of pointers to obscuring viewports. If *outvps* is *NULL*, no viewports obscure *invp*. For a code fragment, see the examples at the end of this section.

VOgIsVisible

VOg Functions VO Routines

Determines if any part of the object's viewport is visible.

```
BOOLPARAM
VOgIsVisible (
       RECTANGLE *objvp,
       RECTANGLE *invp,
       RECTANGLE **outvps,
       DV_BOOL *all_in,
       DV_BOOL *covered)
```

VOgIsVisible determines if any part of the object viewport, *objvp*, is visible. The *RECTANGLE* structure used for *objvp* must contain the actual upper right and lower left points. The object viewport is visible if part of it is in the clipping viewport, *invp*, and part is uncovered by the obscuring viewport list, *outvps*. Further information is available in the parameters *all_in* and *covered*. *YES* is passed back in the parameter *all_in* if the object viewport is entirely within the clipping viewport. *YES* is passed back in the parameter *covered* if any part of the object viewport intersected by the clipping viewport is partially covered by a rectangle in the obscuring viewport list. *VOgIsVisible* returns *YES* if any part of the object viewport, *objvp*, is visible. Otherwise returns *NO*.

VOgLine

VOg Functions VO Routines

Draws a line.

```
void
VOgLine (
       DV_POINT *p1,
       DV_POINT *p2,
       int pattern,
       int width,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgLine draws a line. *p1* and *p2* are the start and end points of the line. *pattern* is the index of the line type; *width* is the width of the line in pixels. The *invp* and *outvps* parameters are defined above.

VOgMultiline

VOg Functions VO Routines

Draws an array of connected lines.

```
void
VOgMultiline (
       DV_POINT *pts,
       int numpts,
       int pattern,
       int width,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgMultiline draws a series of connected lines. *pts* is an array of *DV_POINT* structures containing the points to connect by the multiple line. *numpts* gives the number of points in the array. *pattern* is the index of the line type. *width* is the width of the line in pixels. The *invp* and *outvps* parameters are defined above.

VOgPolygon

VOg Functions VO Routines

Draws a filled polygon.

```
void
VOgPolygon (
       DV_POINT *pts,
       int numpts,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgPolygon draws a filled polygon. The last point is automatically connected back to the first point. The *invp* and *outvps* parameters are defined above. To draw an unfilled polygon, use *VOgFrame*.

VOgRaster

VOg Functions VO Routines

Draws a raster image.

```
void
VOgRaster (
       ADDRESS raster,
       DV_POINT *ll,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgRaster draws a raster image. *ll* indicates where to draw the origin (lower left corner) of the raster. The *invp* and *outvps* parameters are defined above.

VOgRect

VOg Functions VO Routines

Draws a filled rectangle.

```
void
VOgRect (
       DV_POINT *p1,
       DV_POINT *p2,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgRect draws a filled rectangle. *p1* and *p2* are opposite corners of the rectangle. The *invp* and *outvps* parameters are defined above. To draw an unfilled rectangle, use *VOgFrame*.

VOgReErase

VOg Functions VO Routines

Erases a rectangular area of the screen.

```
void
VOgReErase (
       DV_POINT *p1,
       DV_POINT *p2,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgReErase erases a rectangular area of the screen. This is done by drawing a filled rectangle in the current background color. *p1* and *p2* are opposite corners of the rectangle. The *invp* and *outvps* parameters are defined above.

VOgText

VOg Functions VO Routines

Draws text.

```
void
VOgText (
       char *string,
       DV_POINT *spt,
       int direction,
       int position,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgText draws a text string, *string*. The string can contain embedded carriage returns. The location of the string on the screen is specified by the anchor point parameter, *spt*, in screen coordinates. *direction* controls the direction of the text on the screen. Valid values are *HORIZONTAL_TEXT* or *VERTICAL_TEXT*. *position* defines how the text is justified with respect to the anchor point position. There are nine possible positions and they can be defined by bitwise ORing together one flag from each of these two groups:

AT_TOP_EDGE	*AT_LEFT_EDGE*
CENTERED	*CENTERED*
AT_BOTTOM_EDGE	*AT_RIGHT_EDGE*

The *invp* and *outvps* parameters are defined above.

VOgTextsize

VOg Functions VO Routines

Gets the size of a text string.

```
void
VOgTextsize (
       char *string,
       int text_direction,
       int *width,
       int *height)
```

VOgTextsize calculates the size, in screen coordinates (pixels), of a text string with embedded carriage returns.

VOgvText

VOg Functions VO Routines

Draws a vector text string.

```
void
VOgvText (
       char *string,
       DV_POINT *p,
       int direction,
       int position,
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VOgvText draws a text block to the current viewport using a vector font. The text block is passed as a string with embedded carriage returns for line breaks. The parameters are:

string *the text block to be drawn.*

p *the anchor or reference point.*

direction *indicates whether the text is to be drawn from left-to-right (*HORIZONTAL_TEXT*) or top-to-bottom (*VERTICAL_TEXT*).*

position *defines how the text is justified with respect to the anchor point,* p. *There are nine possible positions which can be defined by bitwise ORing together one flag from each of these two groups:*

AT_TOP_EDGE	*AT_LEFT_EDGE*
CENTERED	*CENTERED*
AT_BOTTOM_EDGE	*AT_RIGHT_EDGE*

These flag values are defined in *VOstd.h*. The *invp* and *outvps* parameters are defined at the beginning of this section.

VOgvTextsize

VOg Functions VO Routines

Calculates vector text bounding box.

```
void
VOgvTextsize (
       char *string,
       DV_POINT *p,
       int direction,
       int position,
       RECTANGLE *bound)
```

VOgvTextsize calculates the size in screen coordinates of the bounding box of a multiple-line text, passed as string with embedded carriage returns. For rotated text, this is the tightest enclosing rectangle. The parameters are:

string *the text block to be drawn.*

p *the anchor or reference point.*

direction *indicates whether the text is to be drawn from left-to-right (*HORIZONTAL_TEXT*) or top-to-bottom (*VERTICAL_TEXT*).*

position *defines how the text is justified with respect to the anchor point,* p*. There are nine possible positions which can be defined by bitwise ORing together one flag from each of these two groups:*

AT_TOP_EDGE *AT_LEFT_EDGE*

CENTERED *CENTERED*

AT_BOTTOM_EDGE *AT_RIGHT_EDGE*

bound *returns the vector text boundary.*

These flag values are defined in *VOstd.h*.

Examples

The following code shows an example draw function for *VOgGenericDraw*. Arguments are passed to this routine using the static local variables below.

```
typedef struct
   {
   DV_POINT *start;
   DV_POINT *end;
   int pattern;
   int width;
   }  LINE_ARGS;

LOCAL void drawline (argsa)
   ADDRESS argsa;
{
   LINE_ARGS *args = (LINE_ARGS *)argsa;
   GRmv_and_line (args->start, args->end, args->pattern, args->width);
}
```

The following code fragment sets the argument block and calls *VOgGenericDraw* with the local drawing function defined above. *linebox* is the smallest viewport containing the object. *invp* is the viewport in which the object is to be displayed. *outvps* is a *NULL*-terminated list of obscuring viewports.

```
RECTANGLE linebox;
LINE_ARGS args;

/* Get the viewport containing the line. */
if (p1->x < p2->x)
   { linebox.ll.x = p1->x; linebox.ur.x = p2->x; }
else
   { linebox.ll.x = p2->x; linebox.ur.x = p1->x; }

if (p1->y < p2->y)
   { linebox.ll.y = p1->y; linebox.ur.y = p2->y; }
else
   { linebox.ll.y = p2->y; linebox.ur.y = p1->y; }

args.start = p1;
args.end = p2;
args.pattern = pattern;
args.width = width;
VOgGenericDraw (drawline, (ADDRESS) &args, &linebox, invp, outvps);
```

VOic (VOicon)

VOic Functions VO Routines

Manages icon objects (*ic*). An icon object displays the bit-mapped graphic information contained in a pixmap (*pm*).

The size of an icon object depends on the screen resolution, since it is based on the number of pixels in the pixmap. Icons do not automatically resize when a view is zoomed. However, their height and width can be explicitly set to any size.

Icons can have a writemask and color transform for masking. When an icon is drawn, the only pixels drawn are those whose corresponding pixels in the mask are greater than 0. The color transform changes how the writemask is interpreted. The writemask and color transform let you make icons with “transparent” portions that are pixel-based, color-based, or both.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	**VOic**	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Voic Functions

VOicAtGet	See VOobAtGet.
VOicAtSet	See VOobAtSet.
VOicBox	See VOobBox.
VOicClone	See VOobClone.
VOicCreate	Creates an icon from a pixmap.
VOicDereference	See VOobDereference.
VOicGet	Gets information about an icon.
VOicIntersect	See VOobIntersect.
VOicPtGet	See VOobPtGet.
VOicPtSet	See VOobPtSet.
VOicRefCount	See VOobRefCount.
VOicReference	See VOobReference.
VOicSet	Sets characteristics for an icon.
VOicStatistic	Returns statistics about icons.
VOicTraverse	See VOobTraverse.
VOicValid	See VOobValid.
VOicXfBox	See VOobXfBox.
VOicXformBox	See VOobXformBox.

A *VOic* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOic* routine to save the overhead of an additional routine call.

VOicCreate

VOic Functions VO Routines

Creates an icon from a pixmap.

```
OBJECT
VOicCreate (
       OBJECT pixmap,
       OBJECT anchor_pt,
       ATTRIBUTES *attributes,
          V_IC_ATTR_ENUM flag, <type> value,
          V_IC_ATTR_ENUM flag, <type> value,
          ...,
       V_IC_ATTR_ARGEND)
```

VOicCreate creates an icon from *pixmap*. The anchor point, *anchor_pt,* is the point object in the drawing where the icon is attached. Valid *attributes* field flags are:

FOREGROUND_COLOR *BACKGROUND_COLOR*
TEXT_POSITION

The *TEXT_POSITION* attribute determines the position of the icon with respect to the anchor point. For example, if *TEXT_POSITION* is *CENTERED*, the anchor point is at the center of the icon. If *attributes* is *NULL*, default values are used.

Mask, color mapping, and size characteristics are specified using a variable length argument list of flag/value pairs. The type of characteristic to be set is specified using a variable length argument list of flag/value pairs. *flag* specifies the characteristic to be set. *value* specifies the new value for the characteristic. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are listed in *VOicGet*. To set the value rather than get it, remove one pointer from the value type listed. For example, to set the mask pixmap, declare the value as *OBJECT* instead of *OBJECT* *. Use the parameter *pixmap*, not the *V_IC_PIXMAP* flag, to set the pixmap.

If you do not specify a color transform for the pixmap using the *V_IC_PIXMAP_XFORM* flag, and the *DVMATCH_COLORS* variable in your configuration file is set to *YES*, DataViews creates a color transform that makes the best match from the pixmap colors to the screen's color table. If *DVMATCH_COLORS* is set to *NO*, no color transform is used and the icon is drawn in the colors of the screen's color table that have the same index as the colors in the pixmap's color table. For more information on *DVMATCH_COLORS*, refer to the *Setting the DataViews Environment* appendix of the *DV-Draw User's Guide*. Returns the icon if successful. Otherwise returns *NULL*.

VOicGet

VOic Functions VO Routines

Gets information about an icon.

```
void
VOicGet (
       OBJECT icon,
           V_IC_ATTR_ENUM flag, <type> *valuep,
           V_IC_ATTR_ENUM flag, <type> *valuep,
           ...,
       V_IC_ATTR_ARGEND)
```

VOicGet gets information about *icon*. The type of information to be returned is specified using a variable length argument list of flag/value pairs. *flag* specifies the kind of information to be passed. *valuep* specifies the location to write the information. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are:

Flags	**Value Type**	**Description**
V_IC_PIXMAP	OBJECT *	Pixmap that the icon is based on.
V_IC_MASK_PIXMAP	OBJECT *	Pixmap used as the writemask.
V_IC_HEIGHT	int *	Height of the icon in screen coordinates.
V_IC_WIDTH	int *	Width of the icon in screen coordinates.
V_IC_PIXMAP_XFORM	COLOR_XFORM **	Mapping of the pixmap's color indices to the screen's color indices.
V_IC_MASK_PIXMAP_XFORM	COLOR_XFORM **	Color transform used to interpret the writemask.
V_IC_RASTER	ADDRESS *	Raster drawn on the screen. Can be manipulated using *GR* routines (get only).

VOicSet

VOic Functions VO Routines

Sets characteristics for an icon.

```
void
VOicSet (
       OBJECT icon,
           V_IC_ATTR_ENUM flag, <type> value,
           V_IC_ATTR_ENUM flag, <type> value,
           ...,
       V_IC_ATTR_ARGEND)
```

VOicSet sets characteristics for *icon*. The type of characteristic to be set is specified using a variable length argument list of flag/value pairs. *flag* specifies the characteristic to be set. *value* specifies the new characteristic value. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are listed in *VOicGet*. To set the value rather than get it, remove one pointer from the value type listed. For example, to set the pixmap, declare the value as *OBJECT* instead of *OBJECT **.

If you change the pixmap using the *V_IC_PIXMAP* flag, but do not specify a new color transform using the *V_IC_PIXMAP_XFORM* flag, and *DVMATCH_COLORS* is set to *YES*, DataViews creates a new "best match" color transform. Otherwise it uses the old color transform, if any, and the colors in the icon may look arbitrary.

VOicStatistic

VOic Functions VO Routines

Returns statistics about icons.

```
LONG
VOicStatistic (
       int flag)
```

VOicStatistic returns statistics about icons, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of icons.

VOim (VOimage)

VOim Functions VO Routines

Manages image objects (*im*). An image object displays the bit-mapped graphic information contained in a pixmap (*pm*).

The size of an image object depends on the positions of its control points. Images automatically resize when a view is zoomed. Pixels are automatically added or deleted as required to fill the area defined by the control points.

Images can have a writemask and color transform for masking. When an image is drawn, only the pixels whose corresponding pixels in the mask are greater than 0 are drawn. The color transform changes how the writemask is interpreted. The writemask and color transform let you make images with “transparent” portions that are pixel-based, color-based, or both.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	**VOim**	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

***Voim* Functions**

VOimAtGet	See VOobAtGet.
VOimAtSet	See VOobAtSet.
VOimBox	See VOobBox.
VOimClone	See VOobClone.
VOimCreate	Creates an image from a pixmap.
VOimDereference	See VOobDereference.
VOimGet	Gets information about an image.
VOimIntersect	See VOobIntersect.
VOimPtGet	See VOobPtGet.
VOimPtSet	See VOobPtSet.
VOimRefCount	See VOobRefCount.
VOimReference	See VOobReference.
VOimScalePixmap	Displays an image at an exact scale factor relative to the pixmap size.
VOimSet	Sets characteristics for an image.
VOimStatistic	Returns statistics about images.
VOimTraverse	See VOobTraverse.
VOimValid	See VOobValid.
VOimXfBox	See VOobXfBox.
VOimXformBox	See VOobXformBox.

A *VOim* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOim* routine to save the overhead of an additional routine call.

VOimCreate

VOim Functions VO Routines

Creates an image from a pixmap.

```
OBJECT
VOimCreate (
       OBJECT pixmap,
       OBJECT p1,
       OBJECT p2,
       ATTRIBUTES *attributes,
          V_IM_ATTR_ENUM flag, <type> value,
          V_IM_ATTR_ENUM flag, <type> value,
          ...,
       V_IM_ATTR_ARGEND)
```

VOimCreate creates an image from *pixmap*. The image is bounded by the anchor points, *p1* and *p2*. Valid *attributes* field flags are:

FOREGROUND_COLOR *BACKGROUND_COLOR*
TEXT_POSITION

If *attributes* is *NULL*, default values are used. Mask and color mapping characteristics are specified using a variable length argument list of flag/value pairs. The type of characteristic to be set is specified using a variable length argument list of flag/value pairs. *flag* specifies the characteristic to be set. *value* specifies the new value for the characteristic. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are listed in *VOimGet*. To set the value rather than get it, remove one pointer from the value type listed. For example, to set the mask pixmap, declare the value as *OBJECT* instead of *OBJECT **. Use the parameter *pixmap*, not the *V_IC_PIXMAP* flag, to set the pixmap.

If you do not specify a color transform for the pixmap using the *V_IM_PIXMAP_XFORM* flag, and the *DVMATCH_COLORS* variable in your configuration file is set to *YES*, DataViews creates a color transform that makes the best match from the pixmap colors to the screen's color table. If *DVMATCH_COLORS* is set to *NO*, no color transform is used and the image is drawn in the colors of the screen's color table that have the same index as the colors in the pixmap's color table. For more information on *DVMATCH_COLORS*, refer to the *Setting the DataViews Environment* appendix of the *DV-Draw User's Guide*. Returns the image object if successful. Otherwise returns *NULL*.

VOimGet

VOim Functions VO Routines

Gets information about an image.

```
void
VOimGet (
       OBJECT image,
          V_IM_ATTR_ENUM flag, <type> *valuep,
          V_IM_ATTR_ENUM flag, <type> *valuep,
          ...,
       V_IM_ATTR_ARGEND)
```

VOimGet gets information about *image*. The type of information to be returned is specified using a variable length argument list of flag/value pairs. *flag* specifies the kind of information to be passed. *valuep* specifies the location to write the information. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are:

Flags	**Value Type**	**Description**
V_IM_PIXMAP	OBJECT *	Pixmap that image is based on.
V_IM_MASK_PIXMAP	OBJECT *	Pixmap used as the writemask.
V_IM_PIXMAP_XFORM	COLOR_XFORM **	Mapping of the pixmap's color indices to the screen's color indices.
V_IM_MASK_PIXMAP_XFORM	COLOR_XFORM **	Color transform used to interpret the writemask.
V_IM_RASTER	ADDRESS *	Raster drawn on the screen. Can be manipulated using GR routines (get only).

VOimScalePixmap

VOim Functions VO Routines

Displays an image at an exact scale factor relative to the pixmap size.

```
void
VOimScalePixmap (
       OBJECT image,
       OBJECT xform,
       double xscale,
       double yscale)
```

VOimScalePixmap adjusts the control points of *image* so that its screen coordinate size is exactly the scale factor, *xscale* and *yscale*, times the size in pixels of the pixmap. For example, if *xscale* and *yscale* both equal 1.0, the image is adjusted so that each pixel in the pixmap is exactly one pixel on the screen. The control points are adjusted only to the edge of the world coordinate system. If the window is small or the scale factor large, you may not get the requested scale. The *TEXT_POSITION* attribute of the image determines the direction of the adjustment. For example, if *TEXT_POSITION* is *CENTERED*, the center of the image remains stationary while both control points are adjusted. *xform* specifies the world to screen transform used to determine the coordinates of the points.

VOimSet

VOim Functions VO Routines

Sets characteristics for an image.

```
void
VOimSet (
       OBJECT image,
           V_IM_ATTR_ENUM flag, <type> value,
           V_IM_ATTR_ENUM flag, <type> value,
           ...,
       V_IM_ATTR_ARGEND)
```

VOimSet sets characteristics for *image*. The type of characteristic to be set is specified using a variable length argument list of flag/value pairs. *flag* specifies the characteristic to be set. *value* specifies the new value for the characteristic. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs listed in *VOimGet*. To set the value rather than get it, remove one pointer from the value type listed. For example, to set the pixmap, declare the value as *OBJECT* instead of *OBJECT **.

If you change the pixmap using the *V_IM_PIXMAP* flag, but do not specify a new color transform using the *V_IM_PIXMAP_XFORM* flag, and *DVMATCH_COLORS* is set to *YES*, DataViews creates a new “best match” color transform. Otherwise it uses the old color transform, if any, so the colors in the image may look arbitrary.

VOimStatistic

VOim Functions VO Routines

Returns statistics about images.

```
LONG
VOimStatistic (
       int flag)
```

VOimStatistic returns statistics about images, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of images.

VOin (VOinput)

VOin Functions VO Routines

Manages input objects (*in*). Input objects are graphical objects used to get data interactively from the user and modify the associated variable descriptors accordingly. Input objects are the functional counterpart to data groups or graph objects, which can be thought of as output objects. An input object contains an input technique object and a list of variable descriptors (*vdp*). The input technique object maintains the details of how the input object interacts with the user, and the list of variable descriptors (*vdp*) stores the data resulting from the interaction. Input objects can be multiply referenced, but they cannot be multiply displayed. Input objects work closely with the event handler.

Input objects use only the foreground and background color attributes. Unlike most graphical objects, input objects cannot inherit foreground and background color attributes. Therefore, setting those attributes to *NULL* means that they will get set to some default values, namely, white foreground on a black background.

Applications using the *VOin* routines must *#include* the header file *dvinteract.h*. See also the *VOit* routines, the *VUer* routines, and the *Interaction Handlers* chapter for more information about input objects.

Voin Functions

VOinAtGet	See VOobAtGet.
VOinAtSet	See VOobAtSet.
VOinBox	See VOobBox.
VOinClone	See VOobClone.
VOinCreate	Creates and returns an input object.
VOinDereference	See VOobDereference.
VOinGetFlag	Returns the current value of a flag.
VOinGetInternal	Retrieves an input object's internal components.
VOinGetVarList	Gets the variable descriptor list of an input object.
VOinIntersect	See VOobIntersect.
VOinIsDrawn	Determines if the input object is currently drawn.
VOinPtGet	See VOobPtGet.
VOinPtSet	See VOobPtSet.
VOinPutFlag	Sets a flag in the input object.
VOinPutVarList	Sets the variable descriptor list of the input object.
VOinRefCount	See VOobRefCount.
VOinReference	See VOobReference.
VOinReset	Restores an input object to its initial state.
VOinState	Queries or sets the input object activation state.
VOinStatistic	Returns statistics about input objects.
VOinTechnique	Gets and sets the input technique of the input object.
VOinTraverse	See VOobTraverse.
VOinValid	See VOobValid.
VOinXfBox	See VOobXfBox.
VOinXformBox	See VOobXformBox.

A *VOin* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOin* routine to save the overhead of an additional routine call.

VOinCreate

VOin Functions VO Routines

Creates and returns an input object.

```
OBJECT
VOinCreate (
       OBJECT p1,
       OBJECT p2,
       ATTRIBUTES *attributes)
```

VOinCreate creates and returns an input object. *pt1* and *pt2* are control points that define opposite corners of the input object. Valid *attributes* field flags are:

FOREGROUND_COLOR *BACKGROUND_COLOR*

If *attributes* is *NULL*, default values are used.

VOinGetFlag

VOin Functions VO Routines

Returns the current value of a flag.

```
int
VOinGetFlag (
       OBJECT Input,
       int FlagName)
```

VOinGetFlag returns the current value of the flag, *FlagName*, from the input object. Valid values for this flag are listed under *VOinPutFlag* below.

VOinGetInternal

VOin Functions VO Routines

Retrieves an input object's internal components.

```
ADDRESS
VOinGetInternal (
       OBJECT Input,
       int InternalObj)
```

VOinGetInternal returns a pointer to an input object's internal components. The input object must be drawn. This routine is intended for use by sophisticated users. The following flags are valid values for *InternalObj*:

TRANSFORM	*Transformation object used by all input objects to map from the layout to the screen.*
ECHO_VIEWPORT	*Screen coordinates of the primary echo area (in the form of a* RECTANGLE*) for the input object, such as the slider area for* VNslider *and the text echo area for* VNtext.
AREA_DEQUE	*Deque of pickable menu area objects used by* VNmenu *and* VNmultiplexor *to highlight menu items; or a deque of embedded object areas for* VNcombiner.
OBJECT_TRANS	*Transform object used by* VNcombiner *and* VNmultiplexor *to draw embedded input objects.*
INOBJS_DEQUE	*Deque of input objects embedded in* VNcombiner *and* VNmultiplexor.
OBJECT_DEQUE	*Deque of object choices used in* VNmenu, VNmultiplexor, *and* VNtoggle*; or a deque of pickable objects for* VNchecklist.
ITEM_DEQUE	*Deque of menu text objects used by* VNmenu *and* VNmultiplexor *for text menus.*
INITIAL_VALUE	*A pointer to the original value of the variable descriptor used by* VNmenu, VNtoggle, VNslider, VNpalette, *and* VNmultiplexor. *For example, this flag allows updating the initial value to reflect a new value supplied by the user. This new value would then be used in the case of a* CANCEL *or* RESTORE *event.*
INITIAL_XVALUE	*A pointer to the original value of the x variable descriptor used by* VNslider2D.
INITIAL_YVALUE	*A pointer to the original value of the y variable descriptor used by* VNslider2D.

VOinGetVarList

VOin Functions VO Routines

Gets the variable descriptor list of an input object.

```
void
VOinGetVarList (
       OBJECT Input,
       ADDRESS **VarList,
       int *NumVars)
```

VOinGetVarList gets the variable descriptor list. *VarList* is the address of a pointer to a variable descriptor array. This is an internal data structure and should not be modified.

VOinIsDrawn

VOin Functions VO Routines

Determines if the input object is currently drawn.

```
BOOLPARAM
VOinIsDrawn (
       OBJECT Input)
```

VOinIsDrawn queries the input object to determine whether it is currently drawn, or if it has been successfully drawn by *TdpDrawObject*. Returns *YES* if the input object is drawn. Otherwise returns *NO*.

VOinPutFlag

VOin Functions VO Routines

Sets a flag in the input object.

```
void
VOinPutFlag (
       OBJECT Input,
       int FlagName,
       int FlagValue)
```

VOinPutFlag sets the current value of the flag, *FlagName*, to the value specified in *FlagValue* for the input object. These flags are used to control certain aspects of how the input object is drawn and erased. Possible values for these flags are:

FlagName	**FlagValue**	**Action**
DRAW_LAYOUT_BOUND	YES/NO	Draws layout viewport boundary.
DRAW_ECHO_BOUND	YES/NO	Draws echo viewport boundary.
SAVE_RASTER	YES/NO	Saves raster of overwritten background.
REDRAW_ON_UPDATE	YES/NO	Redraws any obscuring objects damaged by the input object update.
ERASE_METHOD	RESTORE_RASTER	Restores background from saved raster.
CALL_REDRAW		Redraws background by calling *VOscRedraw.*
ERASE_RECTANGLE		Draws a rectangle in the background color.
NO_ERASE		Does not erase input object image.

This routine queries the device to determine if it supports raster operations. If they are supported, the default erase method is *RESTORE_RASTER*; otherwise the default is *CALL_REDRAW*. The defaults of the *REDRAW_ON_UPDATE* flag is *NO;* the defaults of the other flags are *YES*.

Setting the *REDRAW_ON_UPDATE* flag to *YES* prevents input objects from “bleeding through” other objects, but can slow your application’s performance. For best results, set this flag to *YES* only for input objects that may be obscured by other objects in the drawports. To set this flag to *YES* for all input objects in a view, use the *SetInputFlag* utility. This flag cannot be set in DV-Draw.

VOinPutVarList

VOin Functions VO Routines

Sets the variable descriptor list of the input object.

```
void
VOinPutVarList (
       OBJECT Input,
       ADDRESS *VarList,
       int NumVars)
```

VOinPutVarList sets the variable descriptor. *VarList* is the address of a variable descriptor list. *NumVars* is the number of variable descriptors assigned to the input object.

VOinReset

VOin Functions VO Routines

Restores an input object to its initial state.

```
void
VOinReset (
       OBJECT Input)
```

VOinReset restores the input object to its initial state after it has been drawn. This routine should be called if the input object has been erased in some unusual way. For example, when the input object has been drawn and then the screen is erased by calling *TscErase*. If *VOinReset* is not called at this point, the input object continues to be active even though it is not visible. Redrawing the input object implicitly resets it.

VOinState

VOin Functions VO Routines

Queries or sets the input object activation state.

```
int
VOinState (
       OBJECT Input,
       int State)
```

VOinState queries or sets the input object activation state. Input objects are *ACTIVE* or *INACTIVE.* Returns the state of the input object at entry. If *State* is not *NULL*, the input object is changed to the new activation state. If the input object is drawn, its associated events are activated or deactivated, depending on the setting of *State.*

VOinStatistic

VOin Functions VO Routines

Returns statistics about input objects.

```
LONG
VOinStatistic (
       int flag)
```

VOinStatistic returns statistics about input objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of input objects.

VOinTechnique

VOin Functions VO Routines

Gets and sets the input technique of the input object.

```
OBJECT
VOinTechnique (
       OBJECT Input,
       OBJECT Technique)
```

VOinTechnique associates the input technique object with the input object. Sets the input object's input technique object to *technique* and returns the old value. If the *technique* parameter has the value *DONT_SET_THE_VALUE*, the current input technique object is returned without change.

VOit (VOintech)

VOit Functions VO Routines

Manages input technique objects (*it*). Input technique objects are non-graphical objects that represent methods of acquiring data from users for use by input objects (*in*). Although input technique objects have reference counts and can be multiply referenced, they can only be attached to one input object at a time.

An input technique object contains an interaction handler (*ih*), which defines a specific method of interaction, and a template drawing object, which defines the physical layout of the user interaction on the screen. An interaction technique object also contains a list of pickable items and their associated values. The list is used for interaction handlers such as *VNmenu*, which can have pickable items. An interaction technique objects can also contain information about key-action bindings and a pointer to an echo function which is called every time the input object is drawn, erased, selected, or accepts input.

Applications using these routines must *#include* the header file *dvinteract.h*. Interaction handlers are DV-Tools global variables and must be globally referenced using *GLOBALREF*. For more information on specific interaction handlers and their template drawings, see the *Interaction Handlers* chapter. For more information about input objects, see the *VOinput* section.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	**VOit**	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Voit Functions

VOitClone	See VOobClone.
VOitCreate	Creates and returns an input technique object.
VOitDereference	See VOobDereference.
VOitGetEchoFunction	Gets the Echo Function from the input technique.
VOitGetInteraction	Returns the input technique's interaction handler.
VOitGetKeys	Returns bindings from keys to actions.
VOitGetList	Gets the list of pickable items.
VOitGetListValues	Gets the list of values for pickable items.
VOitGetTemplate	Returns the template drawing.
VOitGetTemplateName	Gets the filename associated with the template.
VOitKeyOrigin	Sets the origin of the keys.
VOitListStart	Gets and sets the starting index for list.
VOitPutEchoFunction	Sets the Echo Function for the input technique.
VOitPutInteraction	Sets the interaction handler.
VOitPutKeys	Sets bindings from keys to actions.
VOitPutList	Sets the list of pickable items.
VOitPutListValues	Sets the list of values for pickable items.
VOitPutTemplate	Sets the template.
VOitPutTemplateName	Sets the filename associated with the template.
VOitRefCount	See VOobRefCount.
VOitReference	See VOobReference.
VOitStatistic	Returns statistics about input techniques.
VOitTraverse	See VOobTraverse. The only subobject for *it* objects is the template drawing.
VOitValid	See VOobValid.

A *VOit* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOit* routine to save the overhead of an additional routine call.

VOitCreate

VOit Functions VO Routines

Creates and returns an input technique object.

```
OBJECT
VOitCreate (
       INHANDLER ih,
       OBJECT template)
```

VOitCreate creates an input technique object. *ih* specifies the interaction handler to be associated with the input technique object. Interaction handlers are DV-Tools global variables and must be globally referenced in the application program using the *GLOBALREF* or external declaration. They are directly analogous to display formatters and their names begin with the *VN* prefix. *template* specifies a drawing object template that provides the format layout and other graphical parameters of the input technique object. For some interaction handlers, if the template is *NULL*, a default layout is used. For more information, see the *Interaction Handlers* chapter.

VOitGetEchoFunction

VOit Functions VO Routines

Gets the Echo Function from the input technique.

```
VOITECHOFUNPTR
VOitGetEchoFunction (
       OBJECT InputTechnique,
       ADDRESS *Args,
       int *Argbytes)
```

VOitGetEchoFunction returns a pointer to the echo function belonging to *InputTechnique*.

VOitGetInteraction

VOit Functions VO Routines

Returns the interaction handler belonging to *InputTechnique*.

```
INHANDLER
VOitGetInteraction (
       OBJECT InputTechnique)
```

VOitGetKeys

VOit Functions VO Routines

Returns bindings from keys to actions.

```
char *
VOitGetKeys (
       OBJECT InputTechnique,
       int ActionType)
```

VOitGetKeys returns a character string representing the key bindings for a specific action type. *InputTechnique* is the input technique object supplying the key bindings. *ActionType* specifies the action type. Valid action type flags are *DONE_KEYS*, *CANCEL_KEYS*, *SELECT_KEYS*, *RESTORE_KEYS*, *CLEAR_KEYS*, or *TOGGLE_POLLING_KEYS*. For more information about these flags, see the *Interaction Handlers* chapter. To assign key-action bindings to input technique objects, use the *VOitPutKeys* routine.

VOitGetList

VOit Functions VO Routines

Gets the list of pickable items.

```
void
VOitGetList (
       OBJECT InputTechnique,
       int *ListType,
       ADDRESS *list,
       int *NumItems)
```

VOitGetList gets the input technique object's list of pickable items. The type of list is returned in the *ListType* flag. These values have the following meanings:

TEXT_LIST	*The list contains a pointer to an array of text string pointers.*
OBJECT_LIST	*The list contains a pointer to an array of object ids.*
NO_LIST	*No pickable items list exists for the input technique object.*

NumItems specifies the number of items in the list. *list* contains a pointer to an internal buffer and should be modified with care. See also *VOitPutList*.

VOitGetListValues

VOit Functions VO Routines

Gets the list of values for pickable items.

```
void
VOitGetListValues (
       OBJECT InputTechnique,
       float **values,
       int *NumValues)
```

VOitGetListValues gets the list of values for pickable items. This sets a pointer to an array of float numbers, which are associated with the pickable items. If this array exists and an item is picked, the input variable is set to the float value associated with the item. If the array is *NULL* and an item is picked, the input variable is set to the 1-based index of the item. The number of values should equal the number of pickable items. Note that *values* contains a pointer to an internal array buffer of floats and should be modified with care. See also *VOitPutListValues*.

VOitGetTemplate

VOit Functions VO Routines

Returns the template drawing.

```
OBJECT
VOitGetTemplate (
       OBJECT InputTechnique)
```

VOitGetTemplate returns the template drawing object belonging to *InputTechnique*.

VOitGetTemplateName

VOit Functions VO Routines

Gets the filename associated with the template.

```
char *
VOitGetTemplateName (
       OBJECT InputTechnique)
```

VOitGetTemplateName returns the filename of the template belonging to *InputTechnique.*

VOitKeyOrigin

VOit Functions VO Routines

Sets the origin of the keys.

```
int
VOitKeyOrigin (
       OBJECT InputTechnique,
       int ActionType,
       int Origin)
```

VOitKeyOrigin defines which set of key-action bindings is to be bound to the specified input technique, *InputTechnique*, when its associated input object is drawn. *ActionType* specifies which key-action binding is being referenced (*DONE_KEYS*, *CANCEL_KEYS*, *SELECT_KEYS*, *RESTORE_KEYS*, *CLEAR_KEYS*, or *TOGGLE_POLLING_KEYS*), and *Origin* specifies whether to use the local (*LOCAL_KEYS*) or global (*GLOBAL_KEYS*) bindings for that particular action type. Local key-action bindings are set for each individual input technique using the *VOitPutKeys* routine; global bindings are set for all input objects using the *VUerPutKeys* routine. If the key origin is set to *LOCAL_KEYS* but no local keys are defined, the global keys are used. *VOitKeyOrigin* returns the previous key-action origin. If *Origin* is set to *DONT_SET_THE_VALUE*, the current key origin is returned and left unchanged.

VOitListStart

VOit Functions VO Routines

Gets and sets the starting index for list.

```
int
VOitListStart (
       OBJECT InputTechnique,
       int StartIndex)
```

VOitListStart defines the beginning of a text menu list that allows scrolling for efficient display update. The routine gets and sets the starting index for the input technique object's list of pickable items. The list start index is 1-based, meaning the first item has an index of 1, and indicates which pickable item goes in the first slot of a menu. This allows paging for menus that don't have enough room for all of the pickable items. This routine always returns the old value. If the new value is invalid, e.g. zero or *DONT_SET_THE_VALUE*, the routine returns the value with no change.

VOitPutEchoFunction

VOit Functions VO Routines

Sets the Echo Function for the input technique.

```
void
VOitPutEchoFunction (
       OBJECT InputTechnique,
       VOITECHOFUNPTR echo_fcn,
       ADDRESS Args,
       int Argbytes)

   void
   echo_fcn (
          OBJECT Input,
          int Origin,
          int State,
          double *Value,
          VARDESC Vdp,
          RECTANGLE *EchoVP,
          ADDRESS args)
```

VOitPutEchoFunction sets the echo function, *echo_fcn*, for the input technique. The echo function is a user-supplied routine that is called by the interaction handler after one of its internal interaction routines has been called. The echo function is called with the current values of the variables, the address of its variable descriptors, the echo viewport, a programmer-supplied argument structure, and the size of the structure in bytes. *Args* and *Argbytes* define the contents and size of this structure to be passed to the echo function. The form of the echo function varies slightly for each type of interaction handler. For the exact syntax of the echo function for a specific interaction handler, see the *Interaction Handlers* chapter. The following echo function for *VNtext* shows a slight variation in the parameters:

```
   void
   echo_fcn (
          OBJECT Input,
          int Origin,
          int State,
          char **Value,
          VARDESC Vdp,
          RECTANGLE *EchoVP,
          ADDRESS args)
```

Input is the invoking input object. *Origin* specifies the action that originated the call to the echo function (*INITIAL_DRAW*, *TAKE_INPUT*, *UPDATE_DRAW*, *CONTEXT_REDRAW*, *ERASE*) or the sub-actions *(SETUP_FOR_DRAW*, *CONTEXT_DRAW*, *CLEANUP_DATA*, *DATA_RESET*). *State* indicates which type of return value action caused the call to the interaction routine (*INPUT_ACCEPT*, *INPUT_DONE*, *INPUT_CANCEL*, *INPUT_USED*, *INPUT_UNUSED*). *Value* and *Vdp* provide the variable descriptor of the input object and its current value. *EchoVP* is a screen coordinate viewport rectangle indicating where the echo area is placed on the screen. *args* is a pointer to the programmer-specified argument structure.

The echo function receives valid parameters when called from all origins except *ERASE*. When the origin is *ERASE*, the parameters *Vdp* and *Value* may be *NULL* or invalid. To ensure that your echo function does not process invalid parameters, check either the *Origin* or the validity of *Vdp* and *Value* within the echo function.

VOitPutInteraction

VOit Functions VO Routines

Sets the interaction handler.

```
INHANDLER
VOitPutInteraction (
       OBJECT InputTechnique,
       INHANDLER Format)
```

VOitPutInteraction replaces the interaction handler belonging to the input technique object with *Format*. Returns *ADDRESS* of the old interaction handler.

VOitPutKeys

VOit Functions VO Routines

Sets bindings from keys to actions.

```
void
VOitPutKeys (
       OBJECT InputTechnique,
       int ActionType,
       char *Keys)
```

VOitPutKeys defines a set of local key-action bindings for the input technique object given by *InputTechnique*. *ActionType* specifies the desired action type. Valid action type flags are: *DONE_KEYS*, *CANCEL_KEYS*, *SELECT_KEYS*, *RESTORE_KEYS*, *CLEAR_KEYS*, or *TOGGLE_POLLING_KEYS*. For additional information, see *VUerPutKeys*. *Keys* should be a character string containing the characters for all the keys to be bound to that action. Note that *VUerPutKeys* defines a global set of key-action bindings. These global bindings are used when any of the following conditions apply:

- *No key-action bindings have been given to the particular input technique object using* VOitPutKeys*.*
- *The key origin has not been set to* LOCAL_KEYS *using* VOitKeyOrigin*.*
- *The key origin has been set to* GLOBAL_KEYS *using* VOitKeyOrigin*.*

VOitPutList

VOit Functions VO Routines

Sets the list of pickable items.

```
void
VOitPutList (
       OBJECT InputTechnique,
       int ListType,
       ADDRESS list,
       int NumItems)
```

VOitPutList sets the input technique object's list of pickable items. The type of list is specified by *ListType*. If this has the value *TEXT_LIST*, *list* should be an array of text string pointers; if it has the value *OBJECT_LIST*, *list* should be an array of graphical object ids. *NumItems* specifies the number of items in the list. This list is not used by all interaction handlers. To determine whether a specific interaction handler uses a pickable list, see the description of the particular interaction handler in the *Interaction Handlers* chapter.

VOitPutListValues

VOit Functions VO Routines

Sets the list of values for pickable items.

```
void
VOitPutListValues (
       OBJECT InputTechnique,
       float *values,
       int NumValues)
```

VOitPutListValues sets the list of values for pickable items. This sets a pointer to an array of float numbers, which are associated with the pickable items. If this array exists and an item is picked, the input variable is set to the float value associated with the item. If the array is *NULL* and an item is picked, the input variable is set to the 1-based index of the item. The number of values should equal the number of pickable items. If *values* is *NULL*, *NumValues* should be *0*. This list is not used by all interaction handlers. To see if it is used by a specific interaction handler, see the description in the *Interaction Handlers* chapter.

VOitPutTemplate

VOit Functions VO Routines

Sets the template.

```
OBJECT
VOitPutTemplate (
       OBJECT InputTechnique,
       OBJECT Template)
```

VOitPutTemplate replaces the template drawing object belonging to *InputTechnique* with *Template*. Returns the old template.

VOitPutTemplateName

VOit Functions VO Routines

Sets the filename associated with the template.

```
void
VOitPutTemplateName (
       OBJECT InputTechnique,
       char *FileName)
```

VOitPutTemplateName sets the filename associated with the template belonging to *InputTechnique.* Should be called in addition to *VOitPutTemplate* for the correct filename to appear in DV-Draw.

VOitStatistic

VOit Functions VO Routines

Returns statistics about input techniques.

```
LONG
VOitStatistic (
       int flag)
```

VOitStatistic returns statistics about input technique objects, depending on the value of *flag.* Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of input technique objects.

VOln (VOline)

VOln Functions VO Routines

Manages line objects (*ln*). A line object is defined by two point subobjects which specify its end points. A line object uses foreground color, line width, and line type attributes.

VoIn Functions

VOlnAtGet	See VOobAtGet.
VOlnAtSet	See VOobAtSet.
VOlnBox	See VOobBox.
VOlnClone	See VOobClone.
VOlnCreate	Creates and returns a line object.
VOlnDereference	See VOobDereference.
VOlnIntersect	See VOobIntersect.
VOlnPtGet	See VOobPtGet.
VOlnPtSet	See VOobPtSet.
VOlnRefCount	See VOobRefCount.
VOlnReference	See VOobReference.
VOlnStatistic	Returns statistics about line objects.
VOlnTraverse	See VOobTraverse.
VOlnValid	See VOobValid.
VOlnXfBox	See VOobXfBox.
VOlnXformBox	See VOobXformBox.

A *VOln* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOln* routine to save the overhead of an additional routine call.

VOlnCreate

VOln Functions VO Routines

Creates and returns a line object.

```
OBJECT
VOlnCreate (
       OBJECT pt1,
       OBJECT pt2,
       ATTRIBUTES *attributes)
```

VOlnCreate creates and returns a line object. The two point subobjects, *pt1* and *pt2*, define the end-points of the line object. Valid *attributes* field flags are:

FOREGROUND_COLOR *LINE_WIDTH*
LINE_TYPE

If *attributes* is *NULL*, default values are used.

VOlnStatistic

VOln Functions VO Routines

Returns statistics about line objects.

```
LONG
VOlnStatistic (
       int flag)
```

VOlnStatistic returns statistics about lines, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of line objects.

VOlo (VOlocation)

VOlo Functions VO Routines

Manages location objects (*lo*), which contain information about the last locator or window event. Typically, the location object is obtained by calling a polling routine: *TloPoll* for simple polling and *VOscWinEventPoll* or *VOloWinEventPoll* for using window extensions. These two types of polling return location objects that are not equivalent. The location objects contain different information and are compatible with different routines. Simple polling returns a location object with key press, position, and screen origination information, and *NULL* values for the *WINEVENT* structure. The location object returned by window event polling routines contains all the information listed above and the additional information contained in the *WINEVENT* structure, such as keyboard state and event type. For the *WINEVENT* typedef, see the *Include Files* chapter. The following table shows which routines support each type of polling.

Window Event Polling:	**Simple Polling:**	**Both:**
VOscWinEventPoll or VOloWinEventPoll	TloPoll or VOscPoll	TloGetSelectedDrawport
		TloGetSelectedObject
TloWinEventSetup	TloSetup	TloGetSelectedObjectName
VOloButton	VOscClosePoll	VOloCreate
VOloKeyString	VOscLocate	VOloKey
VOloKeySym	VOscLoSet	VOloScpGet
VOloMaxPoint	VOscOpenPoll	VOloScreen
VOloRegion	VOscUnlocate	VOloStatistic
VOloState		VOloValid
VOloType		VOloWcpGet
VOloWinEventGet		VOloDereference
VOscWinEventMask		VOloRefCount
VUerWinEventPost		VOloReference

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	**VOlo**	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Volo Functions

VOloButton	Returns the button that was pressed.
VOloCreate	Creates and returns a location object.
VOloDereference	See VOobDereference.
VOloKey	Returns the key that was pressed.
VOloKeyString	Returns the keystring value of the location object.
VOloKeySym	Returns the key symbol value of the location object.
VOloMaxPoint	Returns a point representing the maximum point on the screen.
VOloRefCount	See VOobRefCount.
VOloReference	See VOobReference.
VOloRegion	Returns a rectangle representing the exposed region on the screen.
VOloScpGet	Returns location in screen coordinates.
VOloScreen	Gets the location object's screen object.
VOloState	Returns an unsigned long representing the state of the buttons and modifier keys.
VOloStatistic	Returns statistics about location objects.

VOloType	Returns the type of event.
VOloValid	See VOobValid.
VOloWcpGet	Returns the location object in drawing's world coordinates.
VOloWinEventGet	Returns the window event structure of the location object.
VOloWinEventPoll	Polls for the next window event.

A *VOlo* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOlo* routine to save the overhead of an additional routine call.

VOloButton

VOlo Functions VO Routines

Returns the button that was pressed.

```
int
VOloButton (
       OBJECT location)
```

VOloButton returns an integer indicating which mouse button was pressed, starting with the left button as number 1. This routine must be preceded by a call to a *WINEVENT* polling routine.

VOloCreate

VOlo Functions VO Routines

Creates and returns a location object.

```
OBJECT
VOloCreate ( void)
```

VOloCreate creates and returns a location object. This routine can be used to create a location object without calling a polling routine.

VOloKey

VOlo Functions VO Routines

Returns the key that was pressed.

```
int
VOloKey (
       OBJECT location)
```

VOloKey returns the ASCII code of the key that was pressed. Mouse buttons are returned as 1, 2, and 3 for the left, middle, and right buttons respectively.

VOloKeyString

VOlo Functions VO Routines

Returns the keystring value of the location object.

```
char *
VOloKeyString (
       OBJECT location)
```

VOloKeyString returns the keystring value of the location object. The keystring is a character string associated with the particular key symbol. Normally, its length is 1 and it is the ASCII character associated with the particular key symbol. Function and other keys can be rebound to arbitrary strings of any length. Returns a pointer to an internal character string which should not be modified. This routine must be preceded by a call to a *WINEVENT* polling routine.

VOloKeySym

VOlo Functions VO Routines

Returns the key symbol value of the location object.

```
ULONG
VOloKeySym (
       OBJECT location)
```

VOloKeySym returns the key symbol (*keysym*) value of the location object. The key symbol is an integer representing the symbol on the key that was pressed, taking into account the effect of modifier keys such as Shift and Control. For key symbols that are ASCII characters, and for ASCII meta characters, the key symbol has the same value as the ASCII code. For other keys, such as function keys and modifier keys, the key symbol has a value larger than 255. The key symbol values are identical to the key symbol values in X11. Constants representing these values are defined in the *#include* files *GRkeysym.h* and *GRkeysymdef.h*, which are adapted from the standard Xlib *#include* files. *VOloKeySym* requires a prior call to a *WINEVENT* polling routine.

VOloMaxPoint

VOlo Functions VO Routines

Returns a point representing the maximum point on the screen.

```
DV_POINT *
VOloMaxPoint (
       OBJECT location)
```

VOloMaxPoint returns the maximum point on the screen, which is the point with the largest possible x and y coordinates. Returns a pointer to internal point structure which should not be modified. This routine must be preceded by a call to a *WINEVENT* polling routine.

VOloRegion

VOlo Functions VO Routines

Returns a rectangle representing the exposed region on the screen.

```
RECTANGLE *
VOloRegion (
       OBJECT location)
```

VOloRegion returns a pointer to a rectangle representing the exposed region on the screen. The pointer points to an internal rectangle structure which should not be modified. When the event exposes several regions, the union of these regions is returned. To access an array of the individual regions, call *VOloWinEventGet* to get the *WINEVENT* structure. The *rectlist* field of the *WINEVENT* structure contains a pointer to an array of the exposed rectangular regions, but is currently only implemented for X. The rectangle has a value of (0,0, 0,0) for events other than *type V_EXPOSE*. This routine must be preceded by a call to a *WINEVENT* polling routine.

VOloScpGet

VOlo Functions VO Routines

Returns location in screen coordinates.

```
DV_POINT *
VOloScpGet (
       OBJECT location)
```

VOloScpGet returns the locator position in screen coordinates. The routine returns a pointer to an internal point structure which should not be modified.

VOloScreen

VOlo Functions VO Routines

Returns the location object's screen object.

```
OBJECT
VOloScreen (
       OBJECT location)
```

VOloState

VOlo Functions VO Routines

Returns an unsigned long representing the state of the buttons and modifier keys.

```
ULONG
VOloState (
       OBJECT location)
```

VOloState returns an unsigned long representing the state of buttons and modifier keys prior to the reported event. Each button or modifier key is represented by a bit in the returned value. If the bit is set to 1, the corresponding key or button has been pressed. The bit mask for each button and modifier is specified in constants defined in *dvGR.h*. The state can be interpreted using the following list of modifier keys and mouse buttons state flags, which are ORed together to reflect the combination of modifier keys and mouse buttons. This routine must be preceded by a call to a *WINEVENT* polling routine.

V_STATE_SHIFT	A shift key is down.
V_STATE_LOCK	The caps lock key has been pressed.
V_STATE_CONTROL	The control key is down.
V_STATE_MOD1	The meta key is down.
V_STATE_MOD2, V_STATE_MOD3, V_STATE_MOD4, V_STATE_MOD5	Additional meta keys are down. If your device has additional meta keys, they can be mapped to these flags.
V_STATE_BUTTON1	Left mouse button is down.
V_STATE_BUTTON2	Middle mouse button is down.
V_STATE_BUTTON3	Right mouse button is down.
V_STATE_BUTTON4, V_STATE_BUTTON5	Additional mouse buttons are down. If your device has additional mouse buttons, they can be mapped to these flags.

VOloStatistic

VOlo Functions VO Routines

Returns statistics about location objects.

```
LONG
VOloStatistic (
       int flag)
```

VOloStatistic returns statistics about location objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of location objects.

VOloType

VOlo Functions VO Routines

Returns the type of event.

```
ULONG
VOloType (
        OBJECT location)
```

VOloType returns the type of event. These types are identical to the event types specified in *VOscWinEventMask* and are represented by a set of constants defined in *dvGR.h*. This routine must be preceded by a call to a *WINEVENT* polling routine. These are the valid flags that can be returned:

V_KEYPRESS	*A key was pressed. Keys include modifier keys (*<Shift>*,* <Control>*, etc.) and function keys. Extract the key information from the location object using* VOloKey*,* VOloKeyString*, or* VOloKeySym.
V_KEYRELEASE	*A key was released. Keys include modifier keys (*<Shift>*,* <Control>*, etc.) and function keys. Extract the key information from the location object using* VOloKey*,* VOloKeyString*, or* VOloKeySym.
V_BUTTONPRESS	*A mouse button was pressed. Extract the mouse button information from the location object using* VOloButton.
V_BUTTONRELEASE	*A mouse button was released. Extract the mouse button information from the location object using* VOloButton.
V_MOTIONNOTIFY	*Any motion of the mouse, with or without the mouse buttons down. Extract the position information from the location object using* VOloScpGet *or* VOloWcpGet.
V_ENTERNOTIFY	*The mouse has entered the window.*
V_LEAVENOTIFY	*The mouse has left the window.*
V_WINDOW_ICONIFY	*The user iconifies the window.*
V_EXPOSE	*Some portion of the window has been exposed and may need to be redrawn. Extract the region information from the location object using* VOloRegion.
V_RESIZE	*The window size has changed. Extract size information from the location object using* VOloMaxPoint.
V_WINDOW_QUIT	*The user requested a window quit.*
V_NON_STANDARD_EVENT	*An event specified in* altmask *occurred. Extract the event data structure from the location object using* VOloWinEventGet*. The event data structure is in the* eventdata *field.*
V_NON_DV_WINDOW_EVENT	*An event occurred in a window not explicitly opened as a screen, such as a widget. Extract the event data structure from the location object using* VOloWinEventGet*. The event data structure is in the* eventdata *field.*

VOloWcpGet

VOlo Functions VO Routines

Returns the location object in drawing's world coordinates.

```
DV_POINT *
VOloWcpGet (
       OBJECT location)
```

VOloWcpGet returns the locator position in a drawing's world coordinates. This routine returns a pointer to an internal point structure which should not be modified. If the locator is not within a drawport, returns *NULL*.

VOloWinEventGet

VOlo Functions VO Routines

Returns the window event structure of the location object.

```
WINEVENT *
VOloWinEventGet (
       OBJECT location)
```

VOloWinEventGet returns the window event structure of the location object. Returns a pointer to the internal *WINEVENT* structure which should not be modified. This routine must be preceded by a call to a *WINEVENT* polling routine.

VOloWinEventPoll

VOlo Functions VO Routines

Polls for the next window event.

```
OBJECT
VOloWinEventPoll (
       int mode)
```

VOloWinEventPoll returns a location object representing the next window event on the event queue. Only event types passed by the mask, either the default mask or one set by *VOscWinEventMask*, are returned. If no mask was set, the default mask passes the following events to the event queue: key press, key release, button press, button release, motion notify, window quit, enter notify, leave notify, iconify, expose, and resize.

The event queue can contain events from more than one window on systems where windows of the same device type share a single event queue. When the event queue is shared, the screen to which the location object belongs can be identified using *VOloScreen.* When only events from a specific window are desired, use *VOscWinEventPoll* with the specific window selected as the current screen.

If the DataViews windows contain widgets or if the application includes non-DataViews windows, the event queue may contain non-DataViews events. These events are always passed onto the queue, regardless of the event mask.

mode specifies which type of polling mode to use. If the event queue is empty and *mode* is *V_WAIT*, *VOloWinEventPoll* does not return until an event specified by *mask* or *altmask* is generated. If *mode* is *V_NO_WAIT*, *VOloWinEventPoll* does not wait until an event is generated, but returns *NULL* instead of the location object.

VOno (VOnode)

VOno Functions VO Routines

Manages node objects. Node objects, together with edge objects, are used to construct abstract graphs. Graphs are data structures that represent relationships between data. Edges and nodes let you show hierarchical relationships between data. Node objects represent data and edge objects provide the connections between nodes. Some example ways of using this kind of graph are finding the shortest routes between objects, project planning, and electrical circuit analysis. Edge and node objects are provided as application modelling tools for the DataViews environment. For a description of graphs, see any computer science textbook on data structures.

Each node can have any number of edge objects. A node object can have an optional geometry object that graphically represents the node. The geometry object must be a graphical object or a deque of graphical objects. The geometry object is drawn when the node object is drawn.

A node object can have an arbitrary number of slots attached to it that contain user-defined data. Use the *VOslotkey* routines to create and initialize a slot, then use the *VOobSlotUtil* routines to attach the slot to the edge object.

See Also

VOedge module

Vono Functions

VOnoAddEdge	Adds an edge to the node object.
VOnoAtGet	See VOobAtGet.
VOnoAtSet	See VOobAtSet.
VOnoBox	See VOobBox.
VOnoClearMark	Clears mark bits of all node objects.
VOnoClearVisit	Clears visit counts of all node objects.
VOnoClone	See VOobClone.
VOnoCreate	Creates a node object.
VOnoDelEdge	Deletes an edge from the node object.
VOnoDereference	See VOobDereference.
VOnoGetEdge	Gets an edge of the node object.
VOnoGetGeometry	Gets the geometry object of the node object.
VOnoGetMark	Gets the mark bit of the node object.
VOnoGetVisit	Gets the visit count of the node object.
VOnoIntersect	See VoobIntersect.
VOnoPtGet	See VOobPtGet.
VOnoPtSet	See VOobPtSet.
VOnoRefCount	See VOobRefCount.
VOnoReference	See VOobReference.
VOnoSetEdge	Sets a edge of the node object.
VOnoSetGeometry	Sets the geometry object of the node object.
VOnoSetMark	Sets the mark bit of the node object.
VOnoSetVisit	Sets the visit count of the node object.
VOnoStatistic	Returns statistics about nodes.
VOnoTraverse	See VOobTraverse.
VOnoValid	See VOobValid.
VOnoXfBox	See VOobXfBox.
VOnoXformBox	See VOobXformBox.

A *VOno* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOno* routine to save the overhead of an additional routine call.

VOnoAddEdge

VOno Functions VO Routines

Adds an edge to the node object.

```
OBJECT
VOnoAddEdge (
       OBJECT node,
       LONG index,
       OBJECT edge)
```

VOnoAddEdge adds an edge object to the node object. The routine adds *edge* after the *index*-th edge in *node*. To add an edge to the beginning of the node object's edge list, set *index* to zero. To add *edge* to the end of the node object's edge list set *index* equal to the number of edges as shown in the following code fragment:

```
VOnoAddEdge (node, (LONG)VOnoGetEdge (node, 0), edge);
```

If there is no *index*-th edge, the routine does nothing.

VOnoClearMark

VOno Functions VO Routines

Clears mark bits of all node objects.

```
void
VOnoClearMark (void)
```

VOnoClearVisit

VOno Functions VO Routines

Clears visit counts of all node objects.

```
void
VOnoClearVisit (void)
```

VOnoCreate

VOno Functions VO Routines

Creates a node object.

```
OBJECT
VOnoCreate (
       OBJECT Edge1,
       OBJECT Edge2,
       OBJECT Geometry,
       ATTRIBUTES *attributes)
```

VOnoCreate creates and returns a node object. The parameters *Edge1*, *Edge2*, and *Geometry* are optional. If *Edge1* and *Edge2* are specified, a node is created with *Edge1* in the first indexed position and *Edge2* in the second. Use *VOnoAddEdge* to add more edge objects. Use *VOnoSetGeometry* to change the geometry object.

VOnoDelEdge

VOno Functions VO Routines

Deletes an edge from the node object.

```
void
VOnoDelEdge (
       OBJECT node,
       LONG index)
```

VOnoDelEdge deletes an *edge* from the *node*. The routine deletes the edge object at the *index*-th position in the node object's edge list. To delete an edge at the end of the nodes's edge list, set index equal to the number of edges as shown in the following code fragment:

```
VOnoDelEdge (node, (LONG)VOnoGetEdge (node, 0));
```

If there is no *index*-th *edge* the routine does nothing.

VOnoGetEdge

VOno Functions VO Routines

Gets an edge of the node object.

```
OBJECT
VOnoGetEdge (
       OBJECT node,
       LONG index)
```

VOnoGetEdge returns the edge at the *index*-th position in the node's edge list. If *index* is zero, returns the number of edges that the node object contains.

VOnoGetGeometry

VOno Functions VO Routines

Returns the geometry object of the node object.

```
OBJECT
VOnoGetGeometry (
       OBJECT node)
```

VOnoGetMark

VOno Functions VO Routines

Returns the mark bit of the node object.

```
BOOLPARAM
VOnoGetMark (
       OBJECT node)
```

VOnoGetVisit

VOno Functions VO Routines

Returns the visit count of the node object.

```
LONG
VOnoGetVisit (
       OBJECT node)
```

VOnoSetEdge

VOno Functions VO Routines

Sets a edge of the node object.

```
OBJECT
VOnoSetEdge (
       OBJECT node,
       LONG index,
       OBJECT NewEdge)
```

VOnoSetEdge sets a edge at the *index*-th position of *node* to *NewEdge*. Returns the old value of *edge*.

VOnoSetGeometry

VOno Functions VO Routines

Sets the geometry object of the node object.

```
OBJECT
VOnoSetGeometry (
       OBJECT node,
       OBJECT NewGeometry)
```

VOnoSetGeometry sets the geometry object of the node object to *NewGeometry*. Returns the old geometry object.

VOnoSetMark

VOno Functions VO Routines

Sets the mark bit of the node object.

```
BOOLPARAM
VOnoSetMark (
       OBJECT node,
       BOOLPARAM NewMark)
```

VOnoSetMark sets the mark bit of *node* to *NewMark*. Returns the value of the old mark bit.

VOnoSetVisit

VOno Functions VO Routines

Sets the visit count of the node object.

```
LONG
VOnoSetVisit (
       OBJECT node,
       LONG NewCount)
```

VOnoSetVisit sets the visit count of the node object to *NewCount*. Returns the old value of the visit count.

VOnoStatistic

VOno Functions VO Routines

Returns statistics about nodes.

```
LONG
VOnoStatistic (
       int Flag)
```

VOnoStatistic returns statistics about nodes, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If the flag is *OBJECT_COUNT*, *VOnoStatistic* returns the current number of nodes.

VOpm (VOpixmap)

VOpm Functions VO Routines

Manages pixmap objects (*pm*). A pixmap object is a pixel-based object used by image and icon objects. It consists of a stream of data representing the actual pixel values and information about the height, width, depth, and colors used by the pixmap. The origin of a pixmap is the lower left corner. Pixel positions are determined in relation to this origin.

Pixmaps can be created from files or in-memory data. The files must be in a compatible pixel format. The in-memory data must contain a raster created using *GRraster* routines or data in *GIF*, *PPM*, *TIFF*, raster, or pixrep format.

Compatible pixel formats include the GIF format of Compuserve Corporation, the PPM format of Jef Poskanzer, and the TIFF format of Aldus/Microsoft. The following TIFF classes are supported by DataViews:

TIFF Class	**Image Type**
Class B	1-bit black-and-white images
Class G	grayscale images
Class P	color images using color tables
Class R	color images using RGB values

If your TIFF file does not work with DataViews, you may have an incompatible TIFF file.

Sample pixel files are included with your DataViews release. To use your own pixel files, they must be converted to one of the compatible formats.

Pixmaps can also be written out to files in *GIF*, *PPM*, or *TIFF* format. You can then convert these files to device-dependent formats for use with non-DataViews graphic tools.

Pixmaps are either referenced or included. A **referenced** pixmaps stores the name of the file containing the graphics information. An **included** pixmap stores the graphics information directly. Pixmaps created from in-memory data are always included. Pixmaps created from files are initially referenced, but you can set them to be included.

If a pixmap is referenced, any changes in the pixmap are lost when you reload the view containing the pixmap. To save changes in a pixmap, set the pixmap to included or write the pixmap out to a file and create a new pixmap that references that file.

When a pixmap based on a file is created or loaded as part of a view, it is added to a cache of pixmaps. The cache contains a one-to-one mapping of filenames to pixmaps. If there is already a pixmap in the cache that represents a file, no other pixmaps based on that file are put in the cache. The cache serves as a library of existing pixmaps to help you avoid creating duplicates.

Vopm Functions

VOpmBestColors	Creates a color table that best matches the pixmaps.
VOpmCacheFind	Finds a pixmap in the cache.
VOpmCacheRemove	Removes a pixmap from the cache.
VOpmCacheRemoveAll	Removes all pixmaps from the cache.
VOpmClip	Clips an existing pixmap.
VOpmClone	See VOobClone.
VOpmCreate	Creates and returns a pixmap.
VOpmDereference	See VOobDereference.
VOpmFlip	Flips a pixmap.
VOpmGet	Gets information about a pixmap.
VOpmGetPixel	Gets the color index of a pixel in a pixmap.
VOpmHasDummyPixels	Returns the status of the drawing contained in the pixmap.
VOpmMerge	Merges two pixmaps.
VOpmNewColorTable	Maps a pixmap’s colors to a new color table.
VOpmRefCount	See VOobRefCount.
VOpmReference	See VOobReference.
VOpmResize	Resizes a pixmap to a given height and width.
VOpmRotate	Rotates a pixmap.
VOpmSet	Sets characteristics for a pixmap.
VOpmSetPixel	Sets the color index of a pixel in a pixmap.
VOpmSetRasterMask	Creates a writemask for a raster using a pixmap.
VOpmStatistic	Returns statistics about pixmaps.
VOpmToRaster	Creates a raster from a pixmap.
VOpmValid	See VOobValid.
VOpmWrite	Writes a pixmap to an external file.

A *VOpm* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOpm* routine to save the overhead of an additional routine call.

VOpmBestColors

VOpm Functions VO Routines

Creates a color table that best matches the pixmaps.

```
BOOLPARAM
VOpmBestColors (
       OBJECT *pixmaps,
       int new_size,
       COLOR_TABLE *clutp)
```

VOpmBestColors reduces the number of colors used by pixmaps to a set that best represents the original colors. *pixmaps* can be either a *NULL*-terminated array of pixmaps or a pointer to a deque of pixmaps. *new_size* specifies the size of the new set and must be between 1 and 256. Returns the reduced set of colors in *clutp*. Returns *DV_SUCCESS* or *DV_FAILURE*.

VOpmCacheFind

VOpm Functions VO Routines

Finds a pixmap in the cache.

```
OBJECT
VOpmCacheFind (
       char *file_name)
```

VOpmCacheFind searches the cache for the pixmap based on *file_name*. Returns the pixmap if found in the cache. Otherwise returns *0*. If a pixmap based on the file already exists, returns the existing pixmap instead of creating a duplicate.

VOpmCacheRemove

VOpm Functions VO Routines

Removes a pixmap from the cache.

```
void
VOpmCacheRemove (
       char *file_name)
```

VOpmCacheRemove removes the pixmap based on *file_name* from the cache. Does nothing if there is no such pixmap in the cache. To replace a pixmap in the cache, you must first call this routine to remove the existing pixmap. For example, if you change the file that the pixmap references, you can remove the existing pixmap then call *VOpmCreate* to create a new pixmap and add it to the cache.

VOpmCacheRemoveAll

VOpm Functions VO Routines

Removes all pixmaps from the cache.

```
void
VOpmCacheRemoveAll (void);
```

VOpmClip

VOpm Functions VO Routines

Clips an existing pixmap.

```
OBJECT
VOpmClip (
       OBJECT pixmap,
       RECTANGLE *rectp)
```

VOpmClip clips a pixmap to contain only the pixels within the rectangle *rectp*. The remaining pixels are discarded. If the rectangle is 10x20, the pixmap size changes to 10x20. Returns the clipped pixmap if successful. Otherwise returns *NULL*.

VOpmCreate

VOpm Functions VO Routines

Creates and returns a pixmap.

```
OBJECT
VOpmCreate (
       char *file_name,
       ADDRESS data)
```

VOpmCreate creates a pixmap from a file or in-memory data variable. Either *file_name* or *data* must be valid. If *file_name* is valid, the pixmap defaults to referenced, and the graphics contents of the pixmap are not saved when the pixmap is saved. If *data* is valid, the pixmap defaults to included, and the graphic contents are saved with the pixmap. If the pixmap is created from a file, this routine adds the pixmap to the cache unless it duplicates a pixmap already in the cache. See also *VOpmCacheFind*. Valid formats for files and *data* are listed in the introduction to this module. Returns the pixmap object if successful. Otherwise returns *NULL*.

VOpmFlip

VOpm Functions VO Routines

Flips a pixmap.

```
OBJECT
VOpmFlip (
       OBJECT pixmap,
       V_PM_FLIP_ENUM direction)
```

VOpmFlip flips *pixmap*. If *direction* is *V_PM_HORIZONTAL*, flips the pixmap along the horizontal axis; if *direction* is *V_PM_VERTICAL*, flips the pixmap along the vertical axis. Returns the flipped pixmap if successful. Otherwise returns *NULL*.

VOpmGet

VOpm Functions VO Routines

Gets information about a pixmap.

```
void
VOpmGet (
       OBJECT pixmap,
           V_PM_ATTR_ENUM flag, <type> *valuep,
           V_PM_ATTR_ENUM flag, <type> *valuep,
           ...,
       V_PM_ATTR_ARGEND)
```

VOpmGet gets information about *pixmap*. The type of information to be returned is specified using a variable length argument list of flag/value-pointer pairs. *flag* specifies the kind of information to be passed. *valuep* specifies the location to write the information. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value-pointer pairs are:

Flags	**Value Type**	**Description**
V_PM_HEIGHT	int *	Height in pixels.
V_PM_WIDTH	*int* *	Width in pixels.
V_PM_DEPTH	*int* *	Color depth.
V_PM_COLOR_TABLE	*COLOR_TABLE* **	Colors used by pixmap.
V_PM_FILENAME	*char* **	File that the pixmap is based on.
V_PM_INCLUDE_PIXELS	int *	*TRUE* for included pixmaps; *FALSE* for referenced pixmaps.
V_PM_VERSION	*int* *	Version count incremented whenever the pixmap is changed.
V_PM_PIXREP_DATA	PIXREP *	The pixrep used by the pixmap.

VOpmGetPixel

VOpm Functions VO Routines

Gets the color index of a pixel in a pixmap.

```
int
VOpmGetPixel (
       OBJECT pixmap,
       DV_POINT *pointp)
```

VOpmGetPixel gets the color index of a specified pixel in *pixmap*. *pointp* specifies the position of the pixel in the raster array. Returns the color index of the pixel if successful. Otherwise returns a negative number.

VOpmHasDummyPixels

VOpm Functions VO Routines

Returns the status of the drawing contained in the pixmap.

```
BOOLPARAM
VOpmHasDummyPixels (
       OBJECT pixmap)
```

VOpmHasDummyPixels determines whether the external file the pixmap points to was available when it was created. *VOpmHasDummyPixels* returns TRUE if the external file was not available. (The user sees a question mark in place of the actual pixmap.) Returns FALSE if the correct pixmap is being displayed.

VOpmMerge

VOpm Functions VO Routines

Merges two pixmaps.

```
OBJECT
VOpmMerge (
       OBJECT source,
       RECTANGLE *rectp,
       OBJECT dest,
       DV_POINT *llp,
       V_PM_MERGEMODE_ENUM mode,
       OBJECT mask,
       COLOR_XFORM *mask_transform)
```

VOpmMerge modifies the destination pixmap, *dest*, by merging data from the source pixmap, *source*, into it. *rect* is the portion from the source pixmap to merge. *llp* indicates where to place the lower left corner of the source portion within the destination pixmap. *mode* indicates the method for merging the source and destination. Valid flags for *mode* are:

V_PM_COPY	*Replace the* destination *portion with the* source *portion.*
V_PM_AND	*Bit-wise AND the destination and source portions.*
V_PM_OR	*Bit-wise OR the destination and source portions.*
V_PM_XOR	*Bit-wise XOR the destination and source portions.*

The merged pixmap uses the color table of the destination pixmap; if the destination and source pixmaps have different color tables, the results may not be what you expect. The AND, OR, and XOR modes combine the color index of a source pixel with the color index of the corresponding pixel in the destination pixmap. For good results, you must set up the color table of the destination pixmap, especially for the merge mode. For information on setting up the color table, see the *Plane Masking* technical note.

If *mask* is specified, only the pixels in the destination pixmap whose corresponding pixels in *mask* have an index greater than 0 are actually merged with the source portion. All others are unchanged. *mask_transform* specifies a color transform that changes the interpretation of *mask*. When *mask* is the destination or source pixmap, you can only use *mask_transform* to merge certain colors in either the source or destination. If *mask_transform* is *NULL*, the mask is used directly.

Returns the modified pixmap if successful. Otherwise returns *NULL*.

VOpmNewColorTable

VOpm Functions VO Routines

Maps a pixmap's colors to a new color table.

```
OBJECT
VOpmNewColorTable (
       OBJECT pixmap,
       COLOR_TABLE *color_table,
       BOOLPARAM dither)
```

VOpmNewColorTable replaces the color table of *pixmap* with a new color table, *color_table*. If a color in *pixmap* does not have an exact match in the new color table, the closest match is used. If *dither* is *TRUE* a Floyd-Steinberg dither is applied when matching colors. Returns the changed pixmap if successful. Otherwise returns *NULL*.

VOpmResize

VOpm Functions VO Routines

Resizes a pixmap to a given height and width.

```
OBJECT
VOpmResize (
       OBJECT pixmap,
       int new_height,
       int new_width)
```

VOpmResize resizes *pixmap* to *new_height* and *new_width*. If either *new_height* or *new_width* is a negative number, the corresponding dimension is not changed. Returns the resized pixmap if successful. Otherwise returns *NULL*.

VOpmRotate

VOpm Functions VO Routines

Rotates a pixmap.

```
OBJECT
VOpmRotate (
       OBJECT pixmap,
       int amount)
```

VOpmRotate rotates *pixmap*. *amount* specifies the number of degrees of rotation. Rotation is clockwise and rounded down to the nearest multiple of 90 degrees. Returns the rotated pixmap if successful. Otherwise returns *NULL*.

VOpmSet

VOpm Functions VO Routines

Sets characteristics for a pixmap.

```
void
VOpmSet (
       OBJECT pixmap,
          V_PM_ATTR_ENUM flag, <type> value,
          V_PM_ATTR_ENUM flag, <type> value,
          ...,
       V_PM_ATTR_ARGEND)
```

VOpmSet sets characteristics for *pixmap*. The type of characteristic to be set is specified using a variable length argument list of flag/value pairs. *flag* specifies the characteristic to be set. *value* specifies the new value for the characteristic. The list must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are:

Flags	**Value Type**	**Description**
V_PM_FILENAME	char *	File that the pixmap is based on.
V_PM_RAW_DATA	*ADDRESS*	Graphics data that the pixmap is based on.
V_PM_INCLUDE_PIXELS	*int*	*TRUE* for included pixmaps; *FALSE* for referenced pixmaps.

VOpmSetPixel

VOpm Functions VO Routines

Sets the color index of a pixel in a pixmap.

```
BOOLPARAM
VOpmSetPixel (
       OBJECT pixmap,
       DV_POINT *pointp,
       int value)
```

VOpmSetPixel sets the color of a specified pixel in *pixmap*. *pointp* specifies the position of the pixel in the raster array. *value* is the new color index for the pixel. Returns *DV_SUCCESS* if successful. Returns *DV_FAILURE* if the position is outside the raster array.

VOpmSetRasterMask

VOpm Functions VO Routines

Creates a writemask for a raster using a pixmap.

```
ADDRESS
VOpmSetRasterMask (
        OBJECT pixmap,
        ADDRESS raster,
            V_PM_ATTR_ENUM flag, <type> value,
            V_PM_ATTR_ENUM flag, <type> value,
            ...,
        V_PM_ATTR_ARGEND)
```

VOpmSetRasterMask uses *pixmap* to create a writemask for a raster. The raster can be displayed and manipulated using *GR* routines. The flag-value pairs specify how to manipulate the pixel information to make the writemask. The list of flag-value pairs must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are:

Flags	**Value Type**	**Description**
V_PM_BOUNDS	RECTANGLE *	Use the pixels within this rectangle. Pixels are added or deleted to match the size of the raster.
V_PM_COLOR_XFORM	COLOR_XFORM *	Convert the pixel color indices using this color transform.

Returns the raster with its new write mask if successful. Otherwise returns *NULL*.

VOpmStatistic

VOpm Functions VO Routines

Returns statistics about pixmaps.

```
LONG
VOpmStatistic (
       int flag)
```

VOpmStatistic returns statistics about pixmaps, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of pixmaps.

VOpmToRaster

VOpm Functions VO Routines

Creates a raster from a pixmap.

```
ADDRESS
VOpmToRaster (
       OBJECT pixmap,
           V_PM_ATTR_ENUM flag, <type> value,
           V_PM_ATTR_ENUM flag, <type> value,
           ...,
       V_PM_ATTR_ARGEND)
```

VOpmToRaster creates a raster from *pixmap*. The raster can be displayed and manipulated using *GR* routines. The flag-value pairs specify how to manipulate the pixel information to make the raster. The list of flag-value pairs must terminate with *V_PM_ATTR_ARGEND*. Valid flag/value pairs are:

Flags	Value Type	Description
V_PM_BOUNDS	RECTANGLE *	Use the pixels within this rectangle.
V_PM_HEIGHT	int	Add or delete pixels to attain this height.
V_PM_WIDTH	int	Add or delete pixels to attain this width.
V_PM_COLOR_XFORM	COLOR_XFORM *	Convert the pixel color indices using this color transform.

Returns the raster if successful. Otherwise returns *NULL*.

VOpmWrite

VOpm Functions VO Routines

Writes a pixmap to an external file.

```
BOOLPARAM
VOpmWrite (
       OBJECT pixmap,
       V_PM_FORMAT_ENUM format,
       char *file_name)
```

VOpmWrite writes the *pixmap* to the specified external file, *file_name,* in the specified *format*. Valid *formats* are:

V_PM_PPM	*portable pixmap*
V_PM_TIFF	*Tag Interchange File Format*

Returns non-*NULL* if successful. Otherwise returns *NULL*.

VOpt (VOpoint)

VOpt Functions VO Routines

Manages point objects (*pt*). Point objects represent physical points in two-dimensional space and are usually used as control point subobjects for graphical objects. They can be drawn, but unlike other graphical objects, they have no attributes. Points are always drawn in the drawing foreground color and appear as crosses on the screen.

A point object can be either an absolute point or a relative point. The position of an absolute point object, which is most commonly used, is expressed directly in world coordinates in the range [-16383,16383]. A relative point object contains a point subobject, and its position is specified as an offset relative to this subpoint. Relative point object offsets are expressed either in world coordinates or in screen coordinates, which are device-dependent.

Vopt Functions

VOptBox	See VOobBox.
VOptClone	See VOobClone.
VOptCreate	Creates and returns a point object.
VOptDereference	See VOobDereference.
VOptFCreate	Creates a point object with *double* precision.
VOptGet	Gets point data in the point structure format.
VOptGetFloat	Gets point data in *FLOAT_POINT* format.
VOptGetParams	Gets the parameters that define a point.
VOptIntersect	See VOobIntersect.
VOptMove	Moves a point.
VOptMoveFloat	Moves a point by a floating point offset.
VOptRefCount	See VOobRefCount.
VOptReference	See VOobReference.
VOptStatistic	Returns statistics about points.
VOptTraverse	See VOobTraverse.
VOptValid	See VOobValid.
VOptXfBox	See VOobXfBox.
VOptXfGet	Gets transformed point in *GR* point format.
VOptXfGetFloat	Gets transformed point in *FLOAT_POINT* format.
VOptXformBox	See VOobXformBox.

A *VOpt* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOpt* routine to save the overhead of an additional routine call.

VOptCreate

VOpt Functions VO Routines

Creates and returns a point object.

```
OBJECT
VOptCreate (
       int format,
       int xcoord,
       int ycoord,
       OBJECT ref_pt)
```

VOptCreate creates and returns a point object. The point can be an absolute point or a relative point. An absolute point has the value (*xcoord, ycoord*) and a *NULL* value for the *ref_pt* argument. A relative point has the value (*xcoord* + *refx*, *ycoord* + *refy*) where *refx* and *refy* are the coordinates of the reference point, and *xcoord* and *ycoord* are the offset coordinates of the point with respect to the reference point. Relative points include the coordinates of the reference point object in the *ref_pt* argument. *format* specifies whether to express the relative point offset in world or screen coordinates with the value *WORLD_COORDINATES* or *SCREEN_COORDINATES* respectively. Absolute points ignore this flag since they are always specified in world coordinates. Points created in DV-Draw are absolute points.

VOptFCreate

VOpt Functions VO Routines

Creates a point object with *double* precision.

```
OBJECT
VOptFCreate (
       int format,
       double xcoord,
       double ycoord,
       OBJECT ref_pt)
```

VOptFCreate creates a point with floating point precision. For a description, see *VOptCreate* above. Note that this routine lets you represent fractional coordinates using *double* values for *xcoord* and *ycoord*. The coordinates must still be in the range [-16383,16383]. Returns the point object.

VOptGet

VOpt Functions VO Routines

Gets point data in the point structure format.

```
void
VOptGet (
       OBJECT point,
       DV_POINT *wpt,
       DV_POINT *spt_offset)
```

VOptGet gets the coordinates of the point object. The coordinates are returned in the form of a point structure and come in two parts: the world coordinates, *wpt*, and the offset in screen coordinates, *spt_offset*. An absolute point object is specified by its world coordinates in *wpt* with an *spt_offset* value of zero. A relative point object with an absolute point object as its reference and offsets in world coordinates is also specified by its world coordinates in *wpt* with an *spt_offset* value of zero. The *spt_offset* is *not* zero when a relative point object has an offset in screen coordinates or inherits an offset from its reference point, another relative point object. When *spt_offset* is non-zero, the actual coordinates of the point object are determined by converting the *wpt* point structure into screen coordinates, using the *TdpWorldToScreen* routine, and adding it to the *spt_offset* point structure. The result can then be converted back to world coordinates using *TdpScreenToWorld*. If the point object is a relative point, the returned coordinates always reflect the current value of its reference point.

VOptGetFloat

VOpt Functions VO Routines

Gets point data in *FLOAT_POINT* format.

```
void
VOptGetFloat (
       OBJECT point,
       FLOAT_POINT *wpt,
       FLOAT_POINT *spt_offset)
```

VOptGetFloat gets the coordinates of a point object using floating point precision. For a description, see *VOptGet* above. Note that this routine returns the coordinates in a *FLOAT_POINT* structure.

VOptGetParams

VOpt Functions VO Routines

Gets the parameters that define a point.

```
void
VOptGetParams (
       OBJECT point,
       int *is_float,
       int *is_world,
       double *xcoord,
       double *ycoord,
       OBJECT *ref_pt)
```

VOptGetParams gets the parameters that define a *point*. Gets the type of the point, its x and y coordinates, and its reference point. If the point is a *FLOAT_POINT*, sets *is_float* to *YES*. Otherwise, sets it to *NO*. If the point is in world coordinates, sets *is_world* to *YES*. Otherwise, sets it to *NO*. *xcoord* and *ycoord* are set to the x and y coordinates of the point. *ref_pt* is set to the reference point if there is one.

VOptMove

VOpt Functions VO Routines

Moves a point.

```
void
VOptMove (
       OBJECT point,
       int flag,
       int x,
       int y)
```

VOptMove changes the point object’s coordinates by an integer offset. *flag* indicates the types of points to be affected by the move. These values have the following meanings:

DV_ABSOLUTE	*Move absolute points to a new absolute position, (x,y).*
DV_RELATIVE	*Move absolute points by a relative amount, (x,y).*
ADJUST_OFFSET_WORLD	*Adjust the position of relative points to a new world coordinate offset.*
ADJUST_OFFSET_SCREEN	*Adjust the position of relative points to a new screen coordinate offset.*

Note that points created in DV-Draw are absolute points and should be moved using the *DV_ABSOLUTE* or *DV_RELATIVE* flags.

VOptMoveFloat

VOpt Functions VO Routines

Moves a point by a floating point offset.

```
void
VOptMoveFloat (
       OBJECT point,
       int flag,
       double deltax,
       double deltay)
```

VOptMoveFloat changes the point object’s coordinates by a floating point offset. For a description of the parameter *flag*, see *VOptMove* above. If the point was not created using *VOptFCreate*, the fractional part of the offset is ignored.

VOptStatistic

VOpt Functions VO Routines

Returns statistics about points.

```
LONG
VOptStatistic (
       int flag)
```

VOptStatistic returns statistics about point objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of point objects.

VOptXfGet

VOpt Functions VO Routines

Gets transformed point in *GR* point format.

```
void
VOptXfGet (
       OBJECT point,
       OBJECT xform,
       DV_POINT *pt)
```

VOptXfGet gets the coordinates of the point object, *point*, after applying the transformation, *xform*, and adding the screen coordinate offset, if any. The coordinates are returned in the point structure, *pt*.

VOptXfGetFloat

VOpt Functions VO Routines

Gets transformed point in *FLOAT_POINT* format.

```
void
VOptXfGetFloat (
       OBJECT point,
       OBJECT xform,
       FLOAT_POINT *pt)
```

VOptXfGetFloat gets transformed *point* in *FLOAT_POINT* format. This routine gives a more accurate number than *VOptXfGet*.

VOpy (VOpolygon)

VOpy Functions VO Routines

Manages polygon objects (*py*). A polygon object is defined by two or more point subobjects. Polygon attributes are foreground color, background color, line type, line width, fill status, and curve type.

The curve type attribute determines how the polygon is drawn. If this has a *NULL* value, the polygon is drawn with straight lines between the points. Three other curve types, *CLOSED_ENDS*, *OPEN_ENDS*, and *FLOATING_ENDS* specify the polygon to be drawn as a B-spline with closed, open, or floating ends respectively.

The polygon fill status can be *FILL*, *EDGE*, *EDGE_WITH_FILL*, *FILL_WITH_EDGE*, or *DV_TRANSPARENT*. When *EDGE* is used, the boundary is drawn using the line attributes. A polygon using *DV_TRANSPARENT* fill looks identical to one with *EDGE* only, but you can select it with the cursor anywhere in the interior of the shape. A transparent polygon does not visually obscure objects behind it, but they cannot be selected through it. When either *EDGE_WITH_FILL* or *FILL_WITH_EDGE* is used, the second feature listed in the fill status flag uses the background color attribute. The foreground color is used in all other cases. Filled polygons are implicitly closed, which means that the last point does not need to equal the first point.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	**VOpy**	VOsd	VOtx	VOxf
VOdbg	VOed						

Vopy Functions

VOpyAtGet	See VOobAtGet.
VOpyAtSet	See VOobAtSet.
VOpyBox	See VOobBox.
VOpyClone	See VOobClone.
VOpyCreate	Creates and returns a polygon object.
VOpyDereference	See VOobDereference.
VOpyIntersect	See VOobIntersect.
VOpyPtAdd	Adds a point to the polygon.
VOpyPtDelete	Deletes a point from the polygon.
VOpyPtGet	See VOobPtGet.
VOpyPtlistAdd	Adds a list of points to the polygon.
VOpyPtlistCreate	Creates a polygon object using a list of points.
VOpyPtSet	See VOobPtSet.
VOpyRefCount	See VOobRefCount.
VOpyReference	See VOobReference.
VOpyStatistic	Returns statistics about polygons.
VOpyTraverse	See VOobTraverse.
VOpyValid	See VOobValid.
VOpyXfBox	See VOobXfBox.
VOpyXformBox	See VOobXformBox.

A *VOpy* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOpy* routine to save the overhead of an additional routine call.

VOpyCreate

VOpy Functions VO Routines

Creates and returns a polygon object.

```
OBJECT
VOpyCreate (
       OBJECT pt1,
       OBJECT pt2,
       ATTRIBUTES *attributes)
```

VOpyCreate creates and returns a polygon object. *pt1* and *pt2* are the start and end points respectively. Valid *attributes* field flags are:

FOREGROUND_COLOR	*LINE_WIDTH*
BACKGROUND_COLOR	*LINE_TYPE*
FILL_STATUS	*CURVE_TYPE*

If *attributes* is *NULL*, default values are used. The default polygon is created using a straight line type. B-spline curve polygons can be created by setting *CURVE_TYPE* to *CLOSED_ENDS, OPEN_ENDS,* or *FLOATING_ENDS*, for closed end, open end, and floating end B-splines respectively. To add more points, use the *VOpyPtAdd* routine. To create a polygon from a list of points, see *VOpyPtlistCreate*.

VOpyPtAdd

VOpy Functions VO Routines

Adds a point to the polygon.

```
void
VOpyPtAdd (
       OBJECT polygon,
       int index,
       OBJECT point)
```

VOpyPtAdd adds a point object to a polygon after the *index*-th point. If *index* is zero, the point is added to the beginning. To add a point to the end of the polygon, call the routine as follows:

```
VOpyPtAdd (polygon, (int)VOpyPtGet (polygon,0), point);
```

If there is no *index*-th point, the routine displays an error message. For a description of *VOpyPtGet*, see the *VOob* chapter of this manual.

VOpyPtDelete

VOpy Functions VO Routines

Deletes a point from the polygon.

```
void
VOpyPtDelete (
       OBJECT polygon,
       int index)
```

VOpyPtDelete deletes a point object from the polygon. To delete a point from the end of the polygon, call the routine as follows:

```
VOpyPtDelete (polygon, (int)VOpyPtGet (polygon,0));
```

If there is no *index*-th point, the routine displays an error message. This routine does not allow a point count of less than two.

VOpyPtlistAdd

VOpy Functions VO Routines

Adds a list of points to the polygon.

```
void
VOpyPtlistAdd (
       OBJECT polygon,
       int index,
       OBJECT *point,
       int numpts)
```

VOpyPtlistAdd adds a list of points to a *polygon* after the *index*-th point. *numpts* is the number of points in the list. *VOpyPtlistAdd* is the same as *VOpyPtAdd* except that it allows adding more than one point to a polygon.

VOpyPtlistCreate

VOpy Functions VO Routines

Creates a polygon object using a list of points.

```
OBJECT
VOpyPtlistCreate (
       OBJECT *pt,
       int numpts,
       ATTRIBUTES *attributes)
```

VOpyPtlistCreate creates a polygon from a list of points, *pt*, with number of points in *numpts*. This is the same as *VOpyCreate* except that *VOpyPtlistCreate* lets you create of a polygon from a list of points. See *VOpyCreate* for list of valid *attribute* field flags. Returns the polygon object.

VOpyStatistic

VOpy Functions VO Routines

Returns statistics about polygons.

```
LONG
VOpyStatistic (
       int flag)
```

VOpyStatistic returns statistics about polygons, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of polygons.

VOre (VOrect)

VOre Functions VO Routines

Manages rectangle objects (*re*). A rectangle is defined by two point subobjects which represent diagonally opposite corners of the rectangle. Rectangle attributes are foreground color, background color, line type, line width, and fill status. The rectangle fill status can be *FILL*, *EDGE*, *EDGE_WITH_FILL*, *FILL_WITH_EDGE*, or *DV_TRANSPARENT*. When *EDGE* is used, the boundary is drawn using the line attributes. A rectangle using *DV_TRANSPARENT* fill looks identical to one with *EDGE* only, but you can select it with the cursor anywhere in the interior of the shape. A transparent rectangle does not visually obscure objects behind it, but they cannot be selected through it. When either *EDGE_WITH_FILL* or *FILL_WITH_EDGE* is used, the second feature listed in the fill status flag uses the background color attribute. The foreground color is used in all other cases.

Vore Functions

VOreAtGet	See VOobAtGet.
VOreAtSet	See VOobAtSet.
VOreBox	See VOobBox.
VOreClone	See VOobClone.
VOreCreate	Creates a rectangle object.
VOreDereference	See VOobDereference.
VOreIntersect	See VOobIntersect.
VOrePtGet	See VOobPtGet.
VOrePtSet	See VOobPtSet.
VOreRefCount	See VOobRefCount.
VOreReference	See VOobReference.
VOreStatistic	Returns statistics about rectangle objects.
VOreTraverse	See VOobTraverse.
VOreValid	See VOobValid.
VOreXfBox	See VOobXfBox.
VOreXformBox	See VOobXformBox.

A *VOre* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOre* routine to save the overhead of an additional routine call.

VOreCreate

VOre Functions VO Routines

Creates a rectangle object.

```
OBJECT
VOreCreate (
       OBJECT pt1,
       OBJECT pt2,
       ATTRIBUTES *attributes)
```

VOreCreate creates and returns a rectangle object. *pt1* and *pt2* are control points that define opposite corners of the rectangle. Valid *attributes* field flags are:

FOREGROUND_COLOR *FILL_STATUS*
BACKGROUND_COLOR *LINE_TYPE*
LINE_WIDTH

If *attributes* is *NULL*, default values are used.

VOreStatistic

VOre Functions VO Routines

Returns statistics about rectangle objects.

```
LONG
VOreStatistic (
       int flag)
```

VOreStatistic returns statistics about rectangle objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of rectangle objects.

VOru (VOrule)

VOru Functions VO Routines

Manages rule objects. A rule object connects a graphical object to a description of an action that depends on a specified event and condition. For the action to occur, the application must be written to interpret the components of the rule.

A rule has three components: an event, a condition, and an action. The event specifies what type of event triggers the rule; the condition specifies the conditions under which the event triggers the action. The file *dvrule.h* defines the event, condition, and action constants that you can use to define rules in an application. The *dvruletab.h* file contains tables to help interpret conditions and actions.

VOruCreate creates a default rule. Use *VOruSetInfo* and *VOruGetInfo* to modify and access rules. *VOruAddToOb* associates a rule object with a graphical object. *VOruDelFromOb* deletes a rule from an object. *VOruNumInOb* gets the number of rules in an object. *VOruGetFromOb* gets a particular rule.

It is recommended to use DV-Draw to create and attach rules to objects in a view. The rules are saved as part of the view.

```
#include "dvrule.h"
#include "dvruletab.h"
```

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	**VOru**	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

Voru Functions

VOruAddToOb	Adds a rule to the object after the *insert_index*-th rule.
VOruClone	See VOobClone.
VOruCreate	Creates a rule object with default values.
VOruDelFromOb	Deletes a rule from the object.
VOruDereference	See VOobDereference.
VOruGetDqFromOb	Returns the rule deque associated with the object.
VOruGetFromOb	Returns the *index*-th rule object of an object.
VOruGetInfo	Returns rule object's event, condition, and action information.
VOruNumInOb	Returns the number of rules in an object.
VOruRefCount	See VOobRefCount.
VOruReference	See VOobReference.
VOruSetInfo	Sets rule object's event, condition, and action information.
VOruStatistic	Returns statistics about rules.
VOruValid	See VOobValid.

A *VOru* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOru* routine to save the overhead of an additional routine call.

VOruAddToOb

VOru Functions VO Routines

Adds a rule to the object after the *insert_index*-th rule.

```
BOOLPARAM
VOruAddToOb (
       OBJECT object,
       OBJECT rule,
       int insert_index)
```

VOruAddToOb adds a rule object to the object, *object*, after the *insert_index*-th position in the list. The rule object can be a single rule or a deque of rules. If *insert_index* is zero, the rule is inserted at the beginning of the rule list. If the *insert_index* is *-1*, the rule is added to the end of the list. Rules are not added if *object*, *rule*, or *insert_index* is invalid. Returns *DV_FAILURE* if the rule cannot be added. Otherwise returns *DV_SUCCESS*.

VOruCreate

VOru Functions VO Routines

Creates a rule object with default values.

```
OBJECT
VOruCreate (void)
```

VOruCreate creates and returns a rule object. The default rule is “On *V_RE_PICK* If *V_RC_ALWAYS* Do *V_RA_NOTHING*.”

VOruDelFromOb

VOru Functions VO Routines

Deletes a rule from the object.

```
BOOLPARAM
VOruDelFromOb (
       OBJECT object,
       OBJECT rule)
```

VOruDelFromOb deletes a rule from the object. The rule object can be a single rule or a deque of rules. Returns *DV_FAILURE* if the rule cannot be added. Otherwise returns *DV_SUCCESS*.

VOruGetDqFromOb

VOru Functions VO Routines

Returns the rule deque associated with the object.

```
OBJECT
VOruGetDqFromOb (
       OBJECT object)
```

VOruGetDqFromOb returns the rule deque associated with the object. If the object has no rules, returns *NULL*.

VOruGetFromOb

VOru Functions VO Routines

Returns the *index*-th rule object of an object.

```
OBJECT
VOruGetFromOb (
       OBJECT object,
       int index)
```

VOruGetFromOb returns the *index*-th rule object associated with the object. Rule indices are 1-based. If the index is 0, returns the number of rules associated with the object.

VOruGetInfo

VOru Functions VO Routines

Returns rule object's event, condition, and action information.

```
void
VOruGetInfo (
        OBJECT rule,
            int flag, int *type_value,
            int flag, int *type_value,
            ...,
        V_END_OF_LIST)
```

VOruGetInfo gets information about *rule*. You can get information about some or all of the parameters of the rule. The information is specified using a zero-terminated list of flag-value sets. Each parameter set starts with a rule component flag that specifies the particular component of the rule to be queried, followed by variable number of values. The values require the address of a variable in which to return the information. The list must be terminate with *V_END_OF_LIST* or zero. Value sets are described below.

If the flag is *V_R_EVENT* the value set must contain a type value. Valid type values are:

V_RE_PICK	*V_RE_DONE*	*V_RE_EVENT_USED*
V_RE_CANCEL	*V_RE_DRAW*	*V_RE_UPDATE*

If the flag is *V_R_CONDITION* the value set must contain four values: a type and three arguments. Valid type values and their corresponding arguments are listed below. Dashes indicate that the information stored in the variable is unused.

Condition Type	**Arg1**	**Arg2**	**Arg3**
V_RC_ALWAYS	---	---	---
V_RC_PICK_BUTTON	---	---	mouse button
V_RC_PICK_ASCII	---	---	key presses
V_RC_DSV_VALUE	dsv	operator	value
V_RC_DSV_DSV	dsv	operator	dsv
V_RC_OBJ_VAR_VALUE	---	operator	value

If the flag is *V_R_ACTION* the value set must contain three values: a type and two arguments. Valid type values and their corresponding arguments are listed below. Dashes indicate that the information stored in the variable is unused.

Action Type	**Arg1**	**Arg2**
V_RA_NEXT	view name	---
V_RA_PREVIOUS	---	---
V_RA_OVERLAY_VIEW	view name	---
V_RA_DEL_OVERLAY_VIEW	view name	---
V_RA_OVERLAY_OBJ	obj name	from view name
V_RA_DEL_OBJECT	obj name	from view name
V_RA_POPUP_AT	obj name	from view name
V_RA_ERASE_ALL_POPUP_AT	---	---
V_RA_REDRAW	---	---
V_RA_QUIT	---	---
V_RA_NOTHING	---	---
V_RA_SYSTEM_CALL	call string	---
V_RA_ERASE_ALL_OVERLAYS	---	---
V_RA_START_DYNAMICS	---	---

V_RA_STOP_DYNAMICS	---	---
V_RA_INC_UPDATE_RATE	---	---
V_RA_DEC_UPDATE_RATE	---	---
V_RA_SET_DSV	dsv	value
V_RA_INC_DSV	dsv	value
V_RA_DEC_DSV	dsv	value

The following table shows how to interpret the values associated with the rule components. The argument values are based on the type values described above. All arguments are declared to be *RULE_ARG* and should be cast as shown below.

Rule Argument:	**Cast As:**	**Description:**
object or view name	(char *)	A character string indicating the object or view name.
condition operator	(int)	An operator chosen from the following set: *V_RC_EQUAL, V_RC_NOT_EQUAL, V_RC_LESS_THAN, V_RC_LESS_EQUAL_THAN, V_RC_GREATER_THAN, or V_RC_GREATER_EQUAL_THAN.*
mouse button	(int)	An integer specifying a mouse button: 1 = left; 2 = middle; 3 = right.
key press	(char *)	An ASCII code character string specifying a key.
data source variable	(DSVAR)	A data source variable.
variable value	(char *)	A character string. All values are saved as character strings so text variables and numerical variables can be stored in the same *RULE_ARG*. A numerical value must be converted from ASCII to its associated data source variable type.

The following code fragments illustrate how to set, get, and interpret a rule's condition. These examples use the right mouse button for the condition.

```
RULE_ARG arg1, arg2, arg3;
int type, button=3;
OBJECT rule;
/* Setting a rule's condition */
VOruSetInfo (rule, V_R_CONDITION,
    V_RC_PICK_BUTTON,
    (RULE_ARG)0, (RULE_ARG)0, (RULE_ARG)button,
    V_END_OF_LIST);
/* Getting a rule's condition */
VOruGetInfo (rule, V_R_CONDITION,
    &type, &arg1, &arg2, &arg3,
    V_END_OF_LIST);
/* Interpreting the rule's condition. Note the int cast on arg3. */
if (type == V_RC_PICK_BUTTON && button == (int)arg3)
    Do_Action;
```

VOruNumInOb

VOru Functions VO Routines

Returns the number of rules associated with an object.

```
*int
VOruNumInOb (
       OBJECT object)
```

VOruSetInfo

VOru Functions VO Routines

Sets rule object’s event, condition, and action information.

```
void
VOruSetInfo (
       OBJECT rule,
           int flag, int type_value, RULE_ARG arg_value,
           int flag, int type_value, RULE_ARG arg_value,
           ...,
       V_END_OF_LIST)
```

VOruSetInfo sets rule information. You can set information about some or all of the parameters of the rule. The information is specified using a zero-terminated list of flag-value sets. See *VOruGetInfo* for valid flag-value sets. If no flag-value set is passed, the parameters are set to default values. The default parameters are the *V_RE_PICK* event, the *V_RC_ALWAYS* condition, and the *V_RA_NOTHING* action.

VOruStatistic

VOru Functions VO Routines

Returns statistics about rules.

```
LONG
VOruStatistic (
       int flag)
```

VOruStatistic returns statistics about rules, depending on the value of *flag*. If *flag* is *OBJECT_COUNT*, returns the current number of rules. Valid flag values are defined in *VOstd.h.*

Examples

The following code fragment illustrates how to process a rule associated with an object. Assume *proto_info* is a structure containing application-specific information.

```
LOCAL void Handle_Rules (proto_info, obj, event)
   PROTO_INFO *proto_info;
   OBJECT obj;
   int event;
{
int i, num_rules, event_type, cond_type, action_type;
RULE_CONDITION cond;        /* defined in dvrule.h */
RULE_ACTION action;         /* defined in dvrule.h */

num_rules = VOruNumInOb (obj);

/* If the rule event matches the current event, process it */
for (i=1; i <= num_rules; i++)
   {
   VOruGetInfo (VOruGetFromOb (obj, i),
                  V_R_EVENT, &event_type,
                  V_R_CONDITION, &cond_type,
                         &cond.arg[0], &cond.arg[1], &cond.arg[2],
                  V_R_ACTION, &action_type,
                         &action.arg[0], &action.arg[1],
                  V_END_OF_LIST);

   cond.type = (char) cond_type;
   action.type = (char) action_type;

   if (event==event_type && Cond_Met (proto_info, obj, &cond))
      Do_Action (proto_info, &action);
   }
}
```

VOsc (VOscreen)

VOsc Functions VO Routines

Manages screen objects (*sc*). The screen object is the DV-Tools interface to the display device and contains low-level information such as the color look-up-table, last locator action, and device name. Only one screen object can be opened for each device or window in the system. Unlike other objects, the *VOsc* routines maintain a system global variable called the current screen, and most of the routines act on this current screen. In order to send graphics commands, you must first set the current screen with a call to *VOscSelect* or *TscSetCurrentScreen*.

The screen object is the highest object in the DataViews hierarchy of data structures. Screen objects contain drawports which contain views, which contain drawing objects, which contain graphical objects.

The screen object keeps track of the drawport ordering, meaning which drawport is in front of which, by keeping a visibility list of drawports. This list is updated when you create, change, or move drawports. Also, when a graphical object is drawn in a drawport, DV-Tools must clip the object so it is in the viewport and out of the obscuring viewports. This is done automatically when you use the *T* and *VO* routines, as opposed to the *GR* routines which do not perform any clipping.

The *VOsc* routines keep track of the current screen, which is the screen object to which screen operations are performed. Most *VOsc* routines operate on the current screen. The only functions that require a screen object argument are *VOscSelect*, which sets the current screen to the specified screen, and *VOscValid*, which determines if the screen object is valid.

VOscWinEventMask and *VOscWinEventPoll* use flags and fields from the *WINEVENT* structure, which contains information about events that occur in windowing systems, such as key strokes, mouse motion, and window resizing and exposure. A listing of the structure is located under *DataViews Public Types* in the *Include Files* chapter.

Note that some routines work even if no screens are open, although most routines return immediately if there is no current screen.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	**VOsc**	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

VOsc Functions

VOscBackcolor	Sets background color for the screen.
VOscClose	Closes a screen for display.
VOscClosePoll	Ends locator polling.
VOscCurrent	Returns the currently selected screen.
VOscDeviceName	Returns device name of the current screen.
VOscDraw	Redraws all the viewports, without erasing.
VOscForecolor	Sets foreground color for the screen.
VOscLocate	Synchronous locator read for the screen.
VOscLoSet	Sets initial locator position for the screen.
VOscOpen	Opens a screen for display.
VOscOpenClut	Opens screen for color table display.
VOscOpenClutSet	Opens screen for color table display and sets window attributes.
VOscOpenPoll	Starts locator polling.
VOscOpenSet	Opens screen and sets window attributes.
VOscPoll	Polls the locator device of the screen.
VOscRedraw	Erases and redraws all the viewports.
VOscReset	Resets the size of the current screen.
VOscSelect	Selects the screen as the current output device.
VOscSize	Returns size of the screen.
VOscUnlocate	Pushes the location onto the cursor event queue.
VOscValid	See VOobValid.
VOscWinEventMask	Sets the screen's window event mask.
VOscWinEventPoll	Gets the next window event from the queue of the current screen.

A *VOsc* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOsc* routine to save the overhead of an additional routine call.

VOscBackcolor

VOsc Functions VO Routines

Sets background color for the screen.

```
OBJECT
VOscBackcolor (
       OBJECT color_obj)
```

VOscBackcolor sets the background color for the current screen to *color_obj*. Returns the old color. If the background color is *NULL*, returns the current color.

VOscClose

VOsc Functions VO Routines

Closes the current screen for display.

```
void
VOscClose (void)
```

VOscClosePoll

VOsc Functions VO Routines

Closes locator cursor polling for the current screen.

```
void
VOscClosePoll (void)
```

VOscCurrent

VOsc Functions VO Routines

Returns the currently selected screen.

```
OBJECT
VOscCurrent (void)
```

VOscDeviceName

VOsc Functions VO Routines

Returns device name of the currently selected screen.

```
char *
VOscDeviceName (
       OBJECT screen)
```

VOscDraw

VOsc Functions VO Routines

Redraws all the viewports, without erasing.

```
void
VOscDraw (
       RECTANGLE *svp)
```

VOscDraw redraws, without erasing, all the viewports in *svp*. This routine should be used when erasing the background is unnecessary, such as when the screen has just been erased, or parts of drawings have been erased and only pieces need to be put back in. This is usually faster than erasing and completely redrawing the screen.

VOscForecolor

VOsc Functions VO Routines

Sets foreground color for the screen.

```
OBJECT
VOscForecolor (
       OBJECT color_obj)
```

VOscForecolor sets the foreground color for the current screen to *color_obj*. Returns the old color. If the foreground color is *NULL*, returns the current color.

VOscLocate

VOsc Functions VO Routines

Synchronous locator read for the screen.

```
OBJECT
VOscLocate (void)
```

VOscLocate is a synchronous locator read for the current screen. This routine waits for a keyboard press or a locator pick, then returns the location object for that pick.

VOscLoSet

VOsc Functions VO Routines

Sets initial locator position for the screen.

```
OBJECT
VOscLoSet (
       DV_POINT *p)
```

VOscLoSet puts the initial locator position for the current screen into the point *p*.

VOscOpen

VOsc Functions VO Routines

Opens a screen for display.

```
OBJECT
VOscOpen (
       char *device)
```

VOscOpen opens *device* for display, and returns the associated screen object.

VOscOpenClut

VOsc Functions VO Routines

Opens screen for color table display.

```
OBJECT
VOscOpenClut (
       char *device_name,
       char *clutfile)
```

VOscOpenClut opens a screen for display with the color lookup table contained in the file, *clutfile*. This file is a list of red, green, and blue intensities in the range [0,255], one set for each index. See also *TscOpen* and *TscOpenWindow*.

VOscOpenClutSet

VOsc Functions VO Routines

Opens screen for color table display and sets window attributes.

```
OBJECT
VOscOpenClutSet (
       char *dev_name,
       char *clutfile,
           ULONG flag, <type> value,
           ULONG flag, <type> value,
           ...,
       V_END_OF_LIST)
```

VOscOpenClutSet opens the device, *dev_name*, specifies the color lookup table, *clutfile*, sets device attributes, and returns a new screen object representing that device. The device attributes are set using a variable length argument list of *flag/value* pairs. The list must terminate with *V_END_OF_LIST* or *0*. See *TscOpenSet* for descriptions of the device attributes. The attribute flags, defined in the header file *dvGR.h*, are also used by *GRopen_set*, *GRset*, *VUopendev_set*, and *VOscOpenSet*. See *VOscOpenSet* below for an example of opening a screen using the attribute flags.

VOscOpenPoll

VOsc Functions VO Routines

Starts locator cursor polling for the current screen.

```
void
VOscOpenPoll (void)
```

VOscOpenSet

VOsc Functions VO Routines

Opens screen and sets window attributes.

```
OBJECT
VOscOpenSet (
       char *dev_name,
           ULONG flag, <type> value,
           ULONG flag, <type> value,
           ...,
       V_END_OF_LIST)
```

VOscOpenSet opens the device, *dev_name*, sets device attributes, and returns a new screen object representing that device.

The device's attributes are set using a variable length argument list of *flag/value* pairs. Each pair of parameters starts with an attribute flag which specifies the particular attribute of the device to be set. The second argument sets the value of the attribute. The list must terminate with *V_END_OF_LIST* or *0*. See *TscOpenSet* for the attribute flags and descriptions of the attributes.

For example, to open a screen as an X11 window 800 pixels high by 600 pixels wide with an upper left position of (100, 100) relative to the screen origin, you could call:

```
screen = VOscOpenSet ("X1", V_WINDOW_X, 100, V_WINDOW_Y, 100, V_WINDOW_WIDTH, 800,
                         V_WINDOW_HEIGHT, 600, V_END_OF_LIST);
```

Examples of attributes are window width and height, window name, and for externally created windows, the window id. The attributes are specified as integer constant flags. The attribute flags, defined in the header file *dvGR.h*, are also used by *TscOpenSet*, *GRopen_set*, *GRset*, *VUopendev_set*, and *VOscOpenClutSet*.

VOscPoll

VOsc Functions VO Routines

Polls the locator device of the screen.

```
OBJECT
VOscPoll (void)
```

VOscPoll polls the locator for the current screen and returns a locator object. The locator object gives the current position and key press, if any.

VOscRedraw

VOsc Functions VO Routines

Erases and redraws all the viewports.

```
void
VOscRedraw (
       RECTANGLE *svp)
```

VOscRedraw erases and redraws all the viewport objects that intersect the viewport, *svp*, specified in screen coordinates. If the viewport is *NULL*, the entire screen is redrawn.

VOscReset

VOsc Functions VO Routines

Resets the size of the current screen.

```
void
VOscReset (void)
```

VOscReset resets the size of the current screen and all of the viewport objects. To be used after resizing a window in a window system.

VOscSelect

VOsc Functions VO Routines

Selects the screen as the current output device.

```
OBJECT
VOscSelect (
       OBJECT screen)
```

VOscSelect selects *screen* as the current output device. This routine returns the previous current screen.

VOscSize

VOsc Functions VO Routines

Returns size of the screen.

```
DV_POINT *
VOscSize (void)
```

VOscSize returns a pointer to a point giving the pixel position of the upper right corner of the current screen. To convert the position coordinates to the actual screen size, add 1 to each coordinate value.

VOscUnlocate

VOsc Functions VO Routines

Pushes the location onto the cursor event queue.

```
void
VOscUnlocate (
       OBJECT location)
```

VOscUnlocate pushes the location onto the cursor event queue. This location is returned by a previous call to a simple polling routine such as *TloPoll*. This routine does not support location objects returned by window event polling.

VOscWinEventMask

VOsc Functions VO Routines

Sets the screen's window event mask.

```
OBJECT
VOscWinEventMask (
       ULONG mask,
       ULONG altmask)
```

VOscWinEventMask sets the current screen's event mask, *mask*, which specifies which DataViews window event types are returned by *VOscWinEventPoll*, *VOloType*, or *VOloWinEventPoll*. The mask is an unsigned long integer in which each bit represents a different type of window event. The mask is constructed by bitwise-ORing the *WINEVENT type* flags representing the events to be noted. The mask acts as a positive filter which passes only the desired events occurring in that window to the event queue. For example, the call:

```
VOscWinEventMask ((ULONG) V_KEYPRESS | V_MOTIONNOTIFY,
                  (ULONG) 0);
```

lets the polling routines report only key press and mouse motion events. The *WINEVENT type* flags are listed below. If no mask is set, the default mask passes the following events to the event queue: key press, key release, button press, button release, motion notify, window quit, enter notify, leave notify, iconify, expose, and resize. Note that you should include all event types required for the input objects in the window. For example, if you have a slider that updates on cursor motion and a button input object that responds to both button presses and releases, you should OR *V_MOTION_NOTIFY*, *V_BUTTONPRESS*, and *V_BUTTONRELEASE* in the event mask.

Certain event type flags require additional information to be specified in *altmask*. *altmask* is an unsigned long integer that is interpreted with a special flag in *mask*. For example, when the flag *V_XWINDOW_MASK* is ORed into *mask,* it tells *VOscWinEventMask* to look in *altmask* for an X11 event mask. This allows any X Window event to be returned. If the event does not fall into one of the standard DataViews event types, it is returned in the *WINEVENT type* field as *V_NON_STANDARD_EVENT*.

To interpret a system-dependent event, you can access the *eventdata* field of the *WINEVENT* structure, where the windowing system's event data structure is copied. For example, under X the *XEvent* structure is copied into the *eventdata* field. Refer to your windowing system manual for more information about how it handles events, including for flags for *altmask* and the system-specific event data structure.

Normally, *VOscWinEventMask* replaces the previous window event mask. However, if the *V_ADD_TO_MASK* flag is ORed into *mask*, the events are added to the existing mask. See also *GRwe_gmask* and *GRget*, which you can use to get the current mask and altmask respectively.

The following *WINEVENT type* flags can be used to construct the *mask* parameter:

V_KEYPRESS	*Any key press, including modifier keys (*<Shift>, <Control>, *etc.) and function keys.*
V_KEYRELEASE	*Any key release, including modifier keys (*<Shift>, <Control>, *etc.) and function keys.*
V_BUTTONPRESS	*Any mouse button press.*
V_BUTTONRELEASE	*Any mouse button release.*
V_MOTIONNOTIFY	*Any motion of the mouse, with or without the mouse buttons down.*
V_ENTERNOTIFY	*The mouse entering the window.*
V_LEAVENOTIFY	*The mouse leaving the window.*
V_WINDOW_ICONIFY	*User requests a window iconify.*
V_EXPOSE	*Some portion of the window is now exposed and needs to be redrawn.*
V_RESIZE	*The window size changes.*

V_WINDOW_QUIT	*User requests a window quit.*

The following modifiers can be ORed with the window event mask:

V_EVENTS_OFF	*Turns off all events, regardless of events that have been ORed into the mask.*
V_ADD_TO_MASK	*Indicates that the flags should be added to the current mask, not replace it. This applies only to* mask*, not* altmask.
V_XWINDOW_MASK	*Indicates* altmask *is an X11 event mask.*

Returns the current screen object when successful. Otherwise returns *NULL*.

VOscWinEventPoll

VOsc Functions VO Routines

Gets the next window event from the queue of the current screen.

```
OBJECT
VOscWinEventPoll (
       int mode)
```

VOscWinEventPoll returns a location object representing the next window event in the event queue. Only events from the current screen are returned. Only event types passed by the mask, either the default mask or one set by *VOscWinEventMask*, are returned. If no mask is set, the default mask passes the following events to the event queue: key press, key release, button press, button release, motion notify, window quit, enter notify, leave notify, iconify, expose, and resize. If the screen contains widgets, the event queue may contain non-DataViews events. These events are always passed onto the queue, regardless of the event mask.

mode specifies which type of polling mode to use. When the event queue is empty, if *mode* is *V_WAIT*, *VOscWinEventPoll* does not return until an event specified by *mask* or *altmask* is generated. If *mode* is *V_NO_WAIT*, *VOscWinEventPoll* does not wait until an event is generated, but returns *NULL* instead of the location object.

The difference between this routine and *VOscPoll* is that *VOscWinEventPoll* returns window events such as key releases, button releases, function keys, exposure, and resize. This information can be extracted from the location object using the *VOlocation* routines. Otherwise, location objects returned by *VOscWinEventPoll* can be used just like location objects returned from *TloPoll*. To get the next event from any window using the event queue, use *VOloWinEventPoll.*

VOsd (VOsubdrawing)

VOsd Functions VO Routines

Manages subdrawing objects (*sd*). The subdrawing object is the mechanism used to include high level objects in drawings. It lets a view refer to another view, either by directly including it, or by referencing the view filename. The latter approach lets you update subdrawings globally by changing the referenced view.

In a **referenced** subdrawing, the contents are not saved when the subdrawing is saved; only the filename is saved. When the subdrawing is read, the file containing the view is opened and read, and the subdrawing is updated with changes in the saved view. In an **included** subdrawing, the contents are saved with the subdrawing and the subdrawing is protected from changes to the referenced view. Using included subdrawings results in larger, self-contained files; using referenced subdrawings results in more compact files that reflect changes in the referenced views.

The dynamics of a view can be **enabled** or **disabled** when it is used as a subdrawing. These **internal** dynamics of the subdrawing should not be confused with dynamics applied to a subdrawing by attaching a dynamic control object. The internal dynamics of disabled subdrawings are not active. The internal dynamics of enabled subdrawings (**active subdrawings**) are active and can receive their data in two ways: from the data source variables in the referenced view (**source** data source variables) or from data source variables in the higher-level view (**destination** data source variables).

To receive data from data sources in the higher-level view, the source variables in the source view are **mapped** to destination variables in the higher-level view. When a source variable is mapped, all references to it are severed and rebound to the destination data source variable. The mapping is normally resolved when the high-level view is drawn, but can be resolved earlier. If you set the *DVSD_DEACT_POOL* configuration variable to *YES*, mappings are resolved at load time for all subdrawings. To resolve the mappings immediately for a particular subdrawing, call *VOsdPoolRemove*.

Note that you cannot clone a higher-level view after the mappings in it are resolved. To clone a view that contains mappings, you must clone it *before* the mappings are resolved. Also note that you must set the *DVSD_DEACT_POOL* configuration variable to *NO* to prevent the mappings from being resolved at load time.

You can change the mappings programmatically using *VOsdSetDsvMapping*. Note that source data source variables must be global to be mapped.

VOsd Functions

VOsdAtGet	See VOobAtGet.
VOsdAtSet	See VOobAtSet.
VOsdBox	See VOobBox.
VOsdClone	See VOobClone.
VOsdCreate	Creates and returns a subdrawing.
VOsdDereference	See VOobDereference.
VOsdFilename	Gets the filename of the subdrawing.
VOsdGetDsvMapping	Gets the mapping for a data source variable in a subdrawing.
VOsdGetDynamicFlag	Determines whether or not a subdrawing's dynamics are enabled.
VOsdGetSelectedObject	Gets the selected object in the subdrawing.
VOsdGetXform	Gets the transformation object of a subdrawing.
VOsdGetXformParams	Gets the transformation parameters.
VOsdHasDummyView	Returns the status of the view contained in the subdrawing.
VOsdIntersect	See VOobIntersect.
VOsdPoolFnmRemove	Removes a view filename from the pool.
VOsdPoolRemove	Removes a subdrawing from the pool.
VOsdPtGet	See VOobPtGet.
VOsdPtSet	See VOobPtSet.
VOsdRefCount	See VOobRefCount.
VOsdReference	See VOobReference.
VOsdRotate	Rotates the subdrawing.
VOsdScale	Scales the subdrawing.
VOsdSetDsvMapping	Sets the mapping for a data source variable in a subdrawing.
VOsdSetDynamicFlag	Controls whether or not a subdrawing's dynamics are enabled.
VOsdSetXformParams	Sets the transformation parameters.
VOsdStatistic	Returns statistics about subdrawings.
VOsdTraverse	See VOobTraverse.
VOsdValid	See VOobValid.
VOsdViGet	Gets the view referenced by a subdrawing.
VOsdViKeep	Determines whether to keep the view when saving the subdrawing.
VOsdViReplace	Replaces the view referenced by a subdrawing, returning the previous one.
VOsdViSet	Sets the view referenced by a subdrawing, destroying the previous one.
VOsdXfBox	See VOobXfBox.
VOsdXformBox	See VOobXformBox.
VOsdXScale	Scales the subdrawing in the x direction.
VOsdYScale	Scales the subdrawing in the y direction.

A *VOsd* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOsd* routine to save the overhead of an additional routine call.

VOsdCreate

VOsd Functions VO Routines

Creates and returns a subdrawing.

```
OBJECT
VOsdCreate(
       char *filename,
       VIEW view,
       OBJECT anchorpt,
       double scale,
       ATTRIBUTES *attributes)
```

VOsdCreate creates and returns a subdrawing. Either *filename* or *view* can be *NULL*. If both are *NULL*, the function returns *NULL* without doing anything. If a filename is specified, the subdrawing defaults to referenced. If a filename is specified but the file cannot be located, returns *NULL*. If the filename is *NULL*, the subdrawing defaults to included. If both are specified, it defaults to referenced. The anchor point, *anchorpt*, is the position in the drawing where the referenced view's origin is located. The origin, (0,0) in world coordinates, corresponds to the center of the view. The scale factor, *scale*, is used to convert from the referenced view's coordinate system to the drawing coordinate system. The valid field flag for *attributes* is *FOREGROUND_COLOR*. To support pre-9.0 code, a drawing object can be passed as *view*, but the internal dynamics of the subdrawing are always disabled.

VOsdFilename

VOsd Functions VO Routines

Gets the filename of the subdrawing.

```
char *
VOsdFilename (
       OBJECT subdrawing)
```

VOsdFilename returns the address of the filename string for the subdrawing. The filename string is an internal structure which should not be modified. Returns *NULL* if you created the subdrawing in DV-Tools and specified a *NULL* filename at that time.

VOsdGetDsvMapping

VOsd Functions VO Routines

Gets the mapping for a data source variable in a subdrawing.

```
int
VOsdGetDsvMapping (
       OBJECT subdrawing,
       int index,
       DSVAR *src_dsvar,
       DSVAR *dst_dsvar)
```

VOsdGetDsvMapping gets the mapping of a source variable in an active subdrawing to its destination variable. *index* is the one-based index of the source variable in the subdrawing's list of mapped data source variables. If *index* is *0*, returns the number of mapped variables. If *index* is greater than *1*, the return value is *0* and the mapping is returned in *src_dsvar* and *dst_dsvar*. If *src_dsvar* is not currently mapped, *NULL* is returned in *dst_dsvar*. This routine is only useful after calling *TdpDraw* because the mappings are not resolved until then.

VOsdGetDynamicFlag

VOsd Functions VO Routines

Determines whether or not a subdrawing's dynamics are enabled.

```
int
VOsdGetDynamicFlag (
       OBJECT subdrawing)
```

VOsdGetDynamicFlag returns a flag indicating whether or not the subdrawing's internal dynamics are enabled. Valid values for the returned flag are:

SD_DYN_NONE	*The subdrawing has no internal dynamics.*
SD_DYN_ENABLED	*The internal dynamics of the subdrawing are active.*
SD_DYN_DISABLED	*The internal dynamics of the subdrawing are inactive*

VOsdGetSelectedObject

VOsd Functions VO Routines

Gets the selected object in the subdrawing.

```
OBJECT
VOsdGetSelectedObject (
       OBJECT subdrawing,
       OBJECT location,
       OBJECT xform,
       int check_mode)
```

VOsdGetSelectedObject gets the object in the *subdrawing* selected by the location object, *location*. If the *subdrawing* is the direct child of the highest level drawing, *xform* is *NULL*. Otherwise, it is the transformation from the subdrawing to the highest level drawing, including all intermediate subdrawings. Use *VOsdGetXform* to get the transformation object for each level. Concatenate the transformations together to produce a single transformation from the subdrawing to the highest level drawing. The direction of the transformation must be from the subdrawing to the highest level drawing. If *check_mode* is *NAMED_SEARCH*, only checks named objects in the drawing. If *check_mode* is *FULL_SEARCH*, checks all objects. Returns the object if an object is selected. Otherwise, returns *NULL*.

VOsdGetXform

VOsd Functions VO Routines

Gets the transformation object of a subdrawing.

```
OBJECT
VOsdGetXform (
       OBJECT subdrawing)
```

VOsdGetXform gets the transformation object from a *subdrawing* to its parent object. The transformation object should not be altered. Returns the transformation object.

VOsdGetXformParams

VOsd Functions VO Routines

Gets the transformation parameters.

```
void
VOsdGetXformParams (
       OBJECT subdrawing,
       double *angle,
       double *xscale,
       double *yscale)
```

VOsdGetXformParams gets the transformation parameters *angle*, *xscale*, and *yscale* for the *subdrawing*.

VOsdHasDummyView

VOsd Functions VO Routines

Returns the status of the view contained in the subdrawing.

```
BOOLPARAM
VOsdHasDummyView (
       OBJECT subdrawing)
```

VOsdHasDummyView determines whether the external subdrawing file was available when it was created. *VOsdHasDummyView* returns TRUE if the external file was not available. (The user sees a view with a text object that gives the name of the missing file in place of the actual subdrawing.) Returns FALSE if the correct subdrawing is being displayed.

VOsdPoolFnmRemove

VOsd Functions VO Routines

Removes a view filename from the pool.

```
void
VOsdPoolFnmRemove (
       char *filename)
```

VOsdPoolFnmRemove removes a filename from the pool. The next time a subdrawing referring to the same filename is loaded or created, the view is loaded from the file and the filename is added to the pool again. This routine is useful when you have changed a view file and want subsequent subdrawings to reflect the changes.

VOsdPoolRemove

VOsd Functions VO Routines

Removes a subdrawing from the pool.

```
void
VOsdPoolRemove (
       OBJECT subdrawing)
```

VOsdPoolRemove removes a referenced subdrawing from the pool. This is useful when you plan to change the subdrawing’s view programmatically, but do not want the changes to affect other drawings that refer to the same view. When you remove the subdrawing from the pool, any changes are confined to the subdrawing and are not proliferated to other subdrawings that refer to the same view file. This routine also resolves the data source variable mappings in the subdrawing.

VOsdRotate

VOsd Functions VO Routines

Rotates the subdrawing.

```
double
VOsdRotate (
       OBJECT subdrawing,
       double angle;
```

VOsdRotate rotates the subdrawing by the angle, in degrees, and returns the new angle, which is the sum of the angle and all previous rotation angles. If the angle is zero, the routine doesn't change the angle setting for the subdrawing, but simply returns the current angle. A positive angle is counterclockwise.

VOsdScale

VOsd Functions VO Routines

Scales the subdrawing.

```
double
VOsdScale (
       OBJECT subdrawing,
       double scale)
```

VOsdScale scales the subdrawing to the scale and returns the new scale factor, which is the product of the scale factor and all previous scale factors. If the scale is 0, the routine returns the current scale factor without changing the old one.

VOsdSetDsvMapping

VOsd Functions VO Routines

Sets the mapping for a data source variable in a subdrawing.

```
int
VOsdSetDsvMapping (
       OBJECT subdrawing,
       DSVAR src_dsvar,
       DSVAR dst_dsvar)
```

VOsdSetDsvMapping sets the mapping of a source variable in an active subdrawing to its destination variable. The source variable, *src_dsvar*, must be a global data source variable in the view referenced by the subdrawing. If the destination variable, *dst_dsvar*, is *NULL*, the mapping is removed and the source variable subsequently supplies the data. Otherwise maps the source variable to the destination variable. Returns *DV_SUCCESS* if the mapping or unmapping was successful, otherwise returns *DV_FAILURE*. After a successful mapping, all variable descriptors and function data source arguments that previously obtained their data from *src_dsvar* now obtain their data from *dst_dsvar*. For the change to take effect, you must call *TdpDraw* after the remapping.

VOsdSetDynamicFlag

VOsd Functions VO Routines

Controls whether or not a subdrawing's dynamics are enabled.

```
void
VOsdSetDynamicFlag (
       OBJECT subdrawing,
       int flag)
```

VOsdSetDynamicFlag sets the flag controlling whether or not the dynamics within the subdrawing's internal dynamics are enabled. Valid values for *flag* are:

SD_DYN_ENABLED	*Makes the internal dynamics of the subdrawing active.*
SD_DYN_DISABLED	*Makes the internal dynamics of the subdrawing inactive.*
SD_DYN_RESET	*Resets the flag after a change to the internal dynamics.*

If the subdrawing is enabled after *TviOpenData* has been called, this routine must be followed by a call to *TviOpenData* on the referenced view. If the subdrawing is disabled after *TviOpenData* has been called, this routine must be followed by a call to *TviCloseData* on the referenced view.

If you modify the internal dynamics of a subdrawing, you must call *VOsdSetDynamicFlag* with *SD_DYN_RESET* to reset the subdrawing's dynamic state and update its internal deque of dynamic objects. If the subdrawing previously had no dynamics, the new state is *SD_DYN_DISABLED*. To enable the dynamics, you must call *VOsdSetDynamicFlag* a second time to set the dynamic state to *SD_DYN_ENABLED*. Note that you should not enable dynamics on a subdrawing within another subdrawing using this routine. You should do this only using DV-Draw.

VOsdSetXformParams

VOsd Functions VO Routines

Sets the transformation parameters.

```
void
VOsdSetXformParams (
       OBJECT subdrawing,
       double *angle,
       double *xscale,
       double *yscale)
```

VOsdSetXformParams sets the transformation parameters *angle*, *xscale*, and *yscale* for the *subdrawing*. The parameters must be passed by reference. If the address of the parameter is *NULL*, that parameter is unaffected.

VOsdStatistic

VOsd Functions VO Routines

Returns statistics about subdrawings.

```
LONG
VOsdStatistic (
	int flag)
```

VOsdStatistic returns statistics about subdrawings, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of subdrawings.

VOsdViGet

VOsd Functions VO Routines

Returns the view referenced by a subdrawing.

```
VIEW
VOsdViGet (
       OBJECT subdrawing)
```

VOsdViKeep

VOsd Functions VO Routines

Determines whether to keep the view when saving the subdrawing.

```
BOOLPARAM
VOsdViKeep (
       OBJECT subdrawing,
       int save_the_view)
```

VOsdViKeep sets the internal flag that determines how the subdrawing is saved. If *save_the_view* is *YES*, the view is saved with the subdrawing along with the name of the file containing the view. This is the included case. If the flag is *NO*, only the view filename is saved. This is the referenced case. If an invalid value of *save_the_view* is passed, the routine doesn't change the flag value; instead it returns the current value of the flag.

VOsdViReplace

VOsd Functions VO Routines

Replaces the view referenced by a subdrawing, returning the previous one.

```
VIEW
VOsdViReplace (
       OBJECT subdrawing,
       char *filename,
       VIEW view)
```

VOsdViReplace replaces the view referenced by the subdrawing. Either *filename*, *view*, or both must be valid. Returns the previous view.

VOsdViSet

VOsd Functions VO Routines

Sets the view referenced by a subdrawing, destroying the previous one.

```
void
VOsdViSet (
       OBJECT subdrawing,
       char *filename,
       VIEW view)
```

VOsdViSet sets the view referenced by the subdrawing. Either *filename*, *view*, or both must be valid. The previous view is destroyed.

VOsdXScale

VOsd Functions VO Routines

Scales the subdrawing in the x direction.

```
double
VOsdXScale (
       OBJECT subdrawing,
       double scale)
```

VOsdXScale scales the subdrawing's x coordinate and returns the new x scale factor. If the new scale factor is 0, the routine returns the current scale factor without change.

VOsdYScale

VOsd Functions VO Routines

Scales the subdrawing in the y direction.

```
double
VOsdYScale (
       OBJECT subdrawing,
       double scale)
```

VOsdYScale scales the subdrawing's y coordinate and returns the new y scale factor. If the new scale factor is 0, the routine returns the current scale factor without change.

VOsf (VOscalablefont)

VOsf Functions VO Routines

Manages scalable font objects (*sf*).

Scalable font objects scale with the drawing and are more flexible than vector text objects because you can use any native font on your system. This includes True Type fonts.

Scalable font attributes are underline, weight, point size, width, height, angle, slant, foreground color, and fontname. The scalable font object is attached to the drawing at an anchor point.

VOsf Functions

VOsfAtGet	See VOobAtGet.
VOsfAtSet	See VOobAtSet.
VOsfBox	See VOobBox.
VOsfClone	See VOobClone.
VOsfCreate	Creates and returns a scalable font object.
VOsfDereference	See VOobDereference.
VOsfGetString	Gets the string value of the scalable font object.
VOsfIntersect	See VOobIntersect.
VOsfPtGet	See VOobPtGet.
VOsfPtSet	See VOobPtSet.
VOsfRefCount	See VOobRefCount.
VOsfReference	See VOobReference.
VOsfSetString	Sets new string value for the scalable font object.
VOsfStatistic	Returns statistics about scalable font objects.
VOsfTraverse	See VOobTraverse.
VOsfValid	See VOobValid.
VOsfXfBox	See VOobXfBox.
VOsfXformBox	See VOobXformBox.

A *VOsf* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOsf* routine to save the overhead of an additional routine call.

VOsfCreate

VOsf Functions VO Routines

Creates and returns a scalable font object.

```
OBJECT
VOsfCreate (
       char *string,
       OBJECT anchor_pt,
       ATTRIBUTES *attributes)
```

VOsfCreate creates and returns a scalable font object. *string* is a *NULL*-terminated character string containing the text content of the object. The anchor point, *anchor_pt*, is the point object that defines where the text string appears on the screen. Valid *attributes* field flags are:

TEXT_UNDERLINE	*TEXT_WEIGHT*
TEXT_PTSIZE	*TEXT_WIDTH*
TEXT_SLANT	*TEXT_HEIGHT*
TEXT_ANGLE	*TEXT_FONTNAME*
FOREGROUND_COLOR	

If *attributes* is *NULL*, default values are used.

VOsfGetString

VOsf Functions VO Routines

Gets the string value of the scalable font object.

```
char *
VOsfGetString (
       OBJECT sftext)
```

VOsfGetString returns a pointer to the string associated with the scalable font object. This is a pointer to an internal data structure which should not be modified.

VOsfSetString

VOsf Functions VO Routines

Sets new string value for the scalable font object.

```
void
VOsfSetString (
       OBJECT sftext,
       char *newstring)
```

VOsfSetString sets a new string value, *newstring,* for the scalable font object. If the new string is shorter than the old string, it is simply copied into the old string's buffer. Otherwise, storage is reallocated to allow for the increased length.

VOsfStatistic

VOsf Functions VO Routines

Returns statistics about scalable font objects.

```
LONG
VOsfStatistic (
       int flag)
```

VOsfStatistic returns statistics about scalable font objects, depending on the value of *flag.* Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of scalable font objects.

VOsk (VOslotkey)

VOsk Functions VO Routines

Manages slots. A slot is a means of attaching information to objects. Slotkey objects associate a slot with the information describing what the slot contains. A slot can contain an integer, an array of integers, a float, an array of floats, an object, or a pointer to a *NULL*-terminated string.

You cannot create more than one slotkey with a given set of parameters. Slotkey creation is restricted by the absence of a create function. To define a slotkey, you must declare it using *VOskDeclare*. If it has already been declared, the routine returns the existing one. If the slotkey has not been declared, the routine creates and returns it. Slotkey objects are never destroyed. Reference, clone, and dereference functions are defined but do nothing. Utilities for operating on slots are provided in the *VOobSlotUtil* module described with the *VOob* routines.

The slotkey feature is intended for use by sophisticated DataViews users.

See Also

VOobSlotUtil

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	**VOsk**	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdb g	VOed						

VOsk Functions

VOskClone	Does nothing.
VOskDeclare	Declares a slotkey object.
VOskDereference	Does nothing.
VOskFind	Gets an existing slotkey by name.
VOskGetKeyName	Gets the name associated with the slotkey.
VOskGetType	Gets type information from the slotkey.
VOskRefCount	Does nothing.
VOskReference	Does nothing.
VOskStatistic	Returns statistics about slotkey objects.
VOskValid	See VOobValid.

A *VOsk* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOsk* routine to save the overhead of an additional routine call.

VOskDeclare

VOsk Functions VO Routines

Declares a slotkey object.

```
OBJECT
VOskDeclare (
       char *KeyName,
       int flag,
       LONG size)
```

VOskDeclare declares and returns a slotkey object that has the keyname, *KeyName*, and the specified *flag* value. *size* is an optional parameter that you use only to define a slotkey for an array type. DataViews reserves string names beginning with *V_*. The slotkey object differs from other objects in that there can only be one instance of any given keyname string and flag. Calling *VOskDeclare* with the keyname string and flag of an existing slotkey object is equivalent to calling *VOskFind*. The flag parameter determines what kind of data to associate with the slotkey object. Valid flags that can be used for defining slotkeys are:

VOSK_INT_TYPE	*VOSK_EXTERNAL_TYPE*
VOSK_INT_ARRAY_TYPE	*VOSK_STRING_TYPE*
VOSK_OBJECT_TYPE	*VOSK_FLOAT_TYPE*
VOSK_FLOAT_ARRAY_TYPE	

A slotkey declared with the flag *VOSK_EXTERNAL_TYPE* contains a pointer to external data types that must be managed by the application.

VOskFind

VOsk Functions VO Routines

Gets an existing slotkey by name.

```
OBJECT
VOskFind (
       char *KeyName)
```

VOskFind finds an existing slotkey with the specified name, *KeyName*. Returns the slotkey if it exists. Otherwise returns *NULL*.

VOskGetKeyName

VOsk Functions VO Routines

Gets the name associated with the slotkey.

```
char *
VOskGetKeyName (
       OBJECT slotkey)
```

VOskGetKeyName returns the slotkey's keyname. The keyname is a pointer to an internal buffer; do not modify the buffer directly.

VOskGetType

Gets type information from the slotkey.

```
void
VOskGetType (
       OBJECT slotkey,
       int *TypeFlag,
       LONG *size)
```

VOskGetType returns the slotkey's *TypeFlag*. Returns the *size* parameter when the slotkey is an array type. See *VOskDeclare* for a list of possible typeflags.

VOskStatistic

VOsk Functions VO Routines

Returns statistics about slotkey objects.

```
LONG
VOskStatistic (
        int Flag)
```

VOskStatistic returns statistics about slotkeys depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If flag is *OBJECT_COUNT*, returns the current number of existing slotkey objects.

Examples

The following code fragment declares a slotkey object that associates an integer slot with the keyname string *"INT"*:

```
OBJECT integer_sk;
integer_sk = VOskDeclare ("INT", VOSK_INT_TYPE);
```

The following example retrieves the slotkey object associated with the keyname string *"INT"*:

```
integer_sk = VOskFind ("INT");
```

VOtt (VOthreshold)

VOtt Functions VO Routines

Manages threshold table objects (*tt*). The threshold table object maps a numerical value range to a set of output values, either integers, floats, objects, or text strings. The table is a list of pairs in which each pair comprises a numerical threshold in the range of [0,32767] and its associated output value. The list is sorted by increasing order of the thresholds. All output values in a table must have the same type. If the output values are objects, however, you can use more than one kind of object. Threshold table objects are used by dynamic control objects to supply values for dynamic actions. See also the *VOdy* module.

When a threshold table is created, it has one output value and no thresholds. The output value is called an object, so a new threshold table has only one object and no thresholds. At this time, the table returns its one output value for all input data. In the following figure, the output value, or object, is labeled Ob_0.

After creating a threshold table, you can add object-threshold pairs. Each pair includes a threshold point, labeled T_1 in the figure below, and the output value above that point, labeled Ob_1.

A threshold represents a boundary between two output values. Incoming data that is greater than the threshold point maps to the output value associated with the threshold. Incoming data that is less than or equal to the threshold point maps to the output value of the previous threshold. Since the threshold table has an output value before it has any thresholds, it always has *n* thresholds and *n+1* objects, as illustrated below.

In this figure, the square bracket,], indicates that the output value maps to values "less than or equal to the next threshold point" and the parenthesis, (, indicates that the output value maps to values "greater than the associated threshold point."

Every time the data should be updated, such as after a call to *TdpDrawNext*, *VOdyUpdate*, or *VOttUpdate*, the table gets input data using a variable descriptor object which normalizes the data in the range [0,32767]. The threshold table compares the input datum to the thresholds in the table and generates an output datum of type *DATUM* (discussed below), which can be an integer, float, object, or text string. The output datum is called the "current output" of the table and is obtained by calling *VOttDataGet*. Before generating this output, the table saves the old "current output" as the "last output," which is obtained by calling *VOttLastGet*. If the table has been reset using *VOttReset*, the current and last output data are both set to the initial datum of the table.

Many of these routines use *DATUM* type data structures, which are described in the *#include* file, *VOstd.h*. See the examples section for an illustration of using the *DATUM* type data structure.

Updating the output of the threshold table can be handled at the higher level of the drawport or dynamic control object. Operations such as adding and deleting thresholds must be handled using routines in this module.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	**VOtt**	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdbg	VOed						

VOtt Functions

VOttAddThresh	*Adds* a threshold to the table.
VOttBox	Gets the union of the bounding boxes. Valid only for threshold tables of graphical objects. See VOobBox.
VOttClone	See VOobClone.
VOttCreate	Creates a threshold table of a specified type.
VOttDataGet	Gets the current object from the table.
VOttDatCreate	Creates a typed threshold table with datum.
VOttDelThresh	Deletes a threshold from the table.
VOttDereference	See VOobDereference.
VOttGetThresh	Gets a threshold from the table.
VOttHasThresh	Determines if the threshold table has a specific threshold.
VOttIntersect	Determines if the current datum intersects the viewport. Valid only for threshold tables of graphical objects. See VOobIntersect.
VOttLastGet	Gets the object before the current object.
VOttRefCount	See VOobRefCount.
VOttReference	See VOobReference.
VOttReset	Resets the threshold to initial state.
VOttScale	Scales thresholds into new range.
VOttSetDatum	Sets the datum for an existing threshold.
VOttSize	Gets the number of thresholds in the table.
VOttStatistic	Returns statistics about threshold table objects.
VOttTraverse	See VOobTraverse.
VOttTypeGet	Gets the type of the object returned by the threshold table.
VOttUpdate	Updates the object to show the current value.
VOttValid	See VOobValid.
VOttVd	Gets the variable descriptor object belonging to the table.
VOttXfBox	Gets the union of the bounding boxes in screen coordinates. Valid only for threshold tables of graphical objects. See VOobXfBox.
VOttXformBox	See VOobXformBox.

A *VOtt* routines that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOtt* routine to save the overhead of an additional routine call.

VOttAddThresh

VOtt Functions VO Routines

Adds a threshold to the table.

```
void
VOttAddThresh (
       OBJECT tt,
       int thresh,
       DATUM out)
```

VOttAddThresh adds a threshold-object pair (*thresh, out*) to the table, *tt*. The threshold value should be in the range [0, 32767]. If the threshold value is *V_UNDEFINED*, the initial object, which has no associated threshold, is to be replaced. If the threshold already exists, replaces its output datum with *out*. See the examples section for an illustration of passing a *DATUM*.

VOttCreate

VOtt Functions VO Routines

Creates a threshold table of a specified type.

```
OBJECT
VOttCreate (
       OBJECT vd,
       DATUM_TYPE type,
       <type> value)
```

VOttCreate creates and returns a threshold table given a variable descriptor object, *vd*, and a *typ*e-*value* pair. Valid *type-value* pairs are:

FLOAT_DATUM	*float*
INT_DATUM	*int*
OBJECT_DATUM (ob_type)	*OBJECT*
TEXT_DATUM	*DV_TEXT*

When *type* is *OBJECT_DATUM*, you must also supply the type of object, which is returned by *VOobType*. See also *VOttDatCreate*.

VOttDataGet

VOtt Functions VO Routines

Gets the current object from the table.

```
DATUM
VOttDataGet (
        OBJECT tt)
```

VOttDataGet returns the current object from the threshold table, *tt*, that corresponds to the current datum value. See the examples section for an illustration of how to get a value of a particular type from a *DATUM*.

VOttDatCreate

VOtt Functions VO Routines

Creates a typed threshold table with datum.

```
OBJECT
VOttDatCreate (
       OBJECT vd,
       DATUM_TYPE type,
       DATUM datum)
```

VOttDatCreate is the same as *VOttCreate* except that it passes in a *DATUM* instead of the actual value. See the examples section for an illustration of passing a *DATUM*.

VOttDelThresh

VOtt Functions VO Routines

Deletes a threshold from the table.

```
void
VOttDelThresh (
       OBJECT tt,
       int thresh)
```

VOttDelThresh deletes the threshold-object pair that has the threshold value, *thresh*, from the table, *tt*.

VOttGetThresh

VOtt Functions VO Routines

Gets a threshold from the table.

```
void VOttGetThresh (
       OBJECT tt,
       int index,
       int *thresh,
       DATUM *out)
```

VOttGetThresh gets the *index*-th threshold-object pair from the table, *tt*. If *index* is zero, the routine gets the original table entry, whose associated threshold value is returned as *V_UNDEFINED*.

VOttHasThresh

VOtt Functions VO Routines

Determines if the threshold table has a specific threshold.

```
int
VOttHasThresh (
       OBJECT tt,
       int thresh)
```

VOttHasThresh determines if the threshold table, *tt*, has the specified threshold, *thresh*. If the table has a threshold at that value, returns the 1-based index of the threshold. Otherwise returns *0*.

VOttLastGet

VOtt Functions VO Routines

Gets the object before the current object.

```
DATUM
VOttLastGet (
        OBJECT tt)
```

VOttLastGet returns the last output datum (the one that was the current output datum before the last call to *VOttUpdate*, *VOdyUpdate*, or *TdpDrawNext*) from the threshold table, *tt*. See the examples section for an illustration of how to get a value of a particular type from a *DATUM*.

VOttReset

VOtt Functions VO Routines

Resets the threshold to its initial state.

```
void
VOttReset (
       OBJECT tt)
```

VOttScale

VOtt Functions VO Routines

Scales thresholds into new range.

```
void
VOttScale (
       OBJECT tt,
       double scale_factor,
       double offset)
```

VOttScale scales thresholds into new range. Each threshold value is multiplied by *scale_factor* and added to *offset*. It is the programmer's responsibility to make sure these numbers do not result in threshold values outside the range [0,32767].

VOttSetDatum

VOtt Functions VO Routines

Sets the datum for an existing threshold.

```
void
VOttSetDatum (
       OBJECT tt,
       int thresh,
       DATUM out)
```

VOttSetDatum changes the output datum associated with a threshold, *thresh*. *out* is the new output datum. *thresh* must correspond to an existing threshold in the table; if not, no change occurs. To change the original threshold table entry, use *V_UNDEFINED* for *thresh*.

VOttSize

VOtt Functions VO Routines

Gets the number of thresholds in the table.

```
int
VOttSize (
       OBJECT tt)
```

VOttSize returns the number of thresholds in the table, *tt*. This does not include the original object in the table. Therefore, if *VOttAddThresh* has never been called, this routine returns zero.

VOttStatistic

VOtt Functions VO Routines

Returns statistics about threshold table objects.

```
LONG
VOttStatistic (
       int flag)
```

VOttStatistic returns statistics about threshold tables, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of threshold tables.

VOttTypeGet

VOtt Functions VO Routines

Gets the type of the object returned by the threshold table.

```
DATUM_TYPE
VOttTypeGet (
       OBJECT tt)
```

VOttTypeGet returns the type of the object returned by *tt*. To determine the type of the threshold, use the macros *IS_FLOAT_DATUM*, *IS_INT_DATUM*, *IS_OBJECT_DATUM* and *IS_TEXT_DATUM*, defined in *VOstd.h*. See *VOttCreate* for a list of valid threshold table types. If the threshold is type *OBJECT_DATUM*, the type also contains a sub-flag indicating the object type of the first object (*Datum0*) in the table (obtained using the *DATUM_O_TYPE* macro).

VOttUpdate

Updates the object to show the current value.

```
void
VOttUpdate (
       OBJECT tt)
```

VOttUpdate updates the threshold table, *tt*, to a new current output datum. This routine is called by these higher level functions that update drawings: *TdpDrawNext, TdpDrawNextObject, VOdyUpdate*.

VOttVd

VOtt Functions VO Routines

Returns the variable descriptor object associated with the table.

```
OBJECT
VOttVd (
       OBJECT tt)
```

Examples

Threshold tables can be built from various types, all of which are passed to the threshold table routines as *DATUM*s. A union, the *DATUM_DESC*, is used to convert *DATUM*s to the other types, and vice versa.

The following code fragment passes a *float* to *VOttAddThresh* as a *DATUM*.

```
DATUM_DESC dd;
float fnum;

dd.f = fnum;
VOttAddThresh (tt, threshold, dd.DATUM_alias);
```

The following code fragment gets a *DATUM* value from the threshold table, then converts it to a *float*.

```
dd.DATUM_alias = VOttDataGet (tt);
fnum = dd.f;
```

The following code fragment creates a threshold table of *double*s. You can also create a threshold table in DV-Draw.

```
OBJECT vd, threshtab;
float fnum1, fnum2, fnum3;
DATUM_DESC dd;

dsv = TdsvCreate();
TdsvEditAttributes (dsv, NULL, V_F_TYPE, 1, 1, NULL);
vd = VOvdCreate (dsv, 'n', (DATUM)defaultnumber);
vdp = VOvdGetVdp (vd);
VPvd_drange (vdp, 0.0, 1.0);     /*set vdp active range */
/* Create threshold table of doubles, with fnum1 as the first value. fnum1 is passed as a DATUM. */
dd.f = fnum1;
threshtab = VOttCreate (vd, FLOAT_DATUM, dd.DATUM_alias);
/* Add fnum2 and fnum3 to the threshold table, passing them as DATUMs. */
dd.f = fnum2;
VOttAddThresh (threshtab, 1*32767/3, dd.DATUM_alias);
dd.f = fnum3;
VOttAddThresh (threshtab, 2*32767/3, dd.DATUM_alias);
```

VOtx (VOtext)

VOtx Functions VO Routines

Manages text objects (*tx*). A text object is a screen coordinate-based object, which means it is a bitmap that is not affected by scaling or zooming into the drawing in which it is embedded. Text object attributes are foreground color, background color, text direction, text justification (position), and text size. The text object is attached to the drawing at an anchor point which can be in one of nine positions with respect to the string bitmap. These positions can be summarized as the cross-product of the sets:

```
{ AT_LEFT_EDGE, CENTERED, AT_RIGHT_EDGE } X
{ AT_TOP_EDGE, CENTERED, AT_BOTTOM_EDGE }
```

A point object can be created with screen coordinates relative to the anchor point, so that figures can be defined with respect to the string. For example, you can use these point objects to construct a box around the string which is always displayed around the string, regardless of the drawing's scale.

VOtx Functions

VOtxAtGet	See VOobAtGet.
VOtxAtSet	See VOobAtSet.
VOtxBox	See VOobBox.
VOtxClone	See VOobClone.
VOtxCreate	Creates and returns a text object.
VOtxDereference	See VOobDereference.
VOtxGetString	Gets the string value of the text object.
VOtxIntersect	See VOobIntersect.
VOtxPtGet	See VOobPtGet.
VOtxPtSet	See VOobPtSet.
VOtxRefCount	See VOobRefCount.
VOtxReference	See VOobReference.
VOtxSetString	Sets new string value for the text object.
VOtxStatistic	Returns statistics about text objects.
VOtxTraverse	See VOobTraverse.
VOtxValid	See VOobValid.
VOtxXfBox	See VOobXfBox.
VOtxXformBox	See VOobXformBox.

A *VOtt* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOtx* routine to save the overhead of an additional routine call.

VOtxCreate

VOtx Functions VO Routines

Creates and returns a text object.

```
OBJECT
VOtxCreate (
       char *string,
       OBJECT anchor_pt,
       ATTRIBUTES *attributes)
```

VOtxCreate creates and returns a text object. S*tring* is a *NULL*-terminated character string containing the text to be drawn when the object is drawn on the screen. The anchor point, *anchor_pt*, is the point object in the drawing where the text string is attached. Valid flag values for *attributes* are:

TEXT_DIRECTION *TEXT_POSITION*
TEXT_SIZE *FOREGROUND_COLOR*
BACKGROUND_COLOR

If *attributes* is *NULL*, default values are used.

VOtxGetString

VOtx Functions VO Routines

Gets the string value of the text object.

```
char *
VOtxGetString (
       OBJECT text)
```

VOtxGetString returns a pointer to the string associated with the text object. This pointer points to an internal data structure which should not be modified.

VOtxSetString

VOtx Functions VO Routines

Sets new string value for the text object.

```
void
VOtxSetString (
       OBJECT text,
       char *newstring)
```

VOtxSetString sets a new string value for the text object. If *newstring* is shorter than the old string, it is simply copied into the old string’s buffer. Otherwise, storage is re-allocated to allow for the increased length.

VOtxStatistic

VOtx Functions VO Routines

Returns statistics about text objects.

```
LONG
VOtxStatistic (
       int flag)
```

VOtxStatistic returns statistics about text objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of text objects.

VOu (VOutil)

VOu Functions   VO Routines

Utility routines for use with objects.

VOu Functions

VOuAtInit

VOu Functions VO Routines

Sets all attributes fields to *EMPTY_FIELD*.

```
void
VOuAtInit (
        ATTRIBUTES *attributes)
```

VOuAtInit sets all attribute fields to either *EMPTY_FIELD* or *EMPTY_FLOAT_FIELD*.

VOuAttr

VOu Functions VO Routines

Returns attributes structure from attribute-value pairs.

```
ATTRIBUTES *
VOuAttr (
       int attr1, <type> value1,
       int attr2, <type> value2,
       ...,
       V_END_OF_LIST)
```

VOuAttr returns a pointer to an internal attributes structure with fields that are set according to a variable length argument list of attribute-value pairs terminated by *V_END_OF_LIST*. Each attribute parameter is a constant flag representing the field of the attributes structure. The parameter following it contains the value of that field. Valid attribute flags are:

FOREGROUND_COLOR	*TEXT_FONT*
BACKGROUND_COLOR	*TEXT_FONTNAME*
LINE_WIDTH	*TEXT_SIZE*
LINE_TYPE	*TEXT_HEIGHT*
FILL_STATUS	*TEXT_WIDTH*
ARC_DIRECTION	*TEXT_DIRECTION*
CURVE_TYPE	*TEXT_POSITION*
TEXT_ANGLE	
TEXT_SLANT	
TEXT_CHARSPACE	
TEXT_LINESPACE	

VOuClearDgData

VOu Functions VO Routines

Clears the data buffers of data group objects.

```
void
VOuClearDgData (
       OBJECT object)
```

VOuClearDgData clears the data buffers associated with data group objects. *object* can be a data group object, a deque object, or a drawing object. If *object* is a deque or drawing object, this routine traverses *object* and clears the data buffers associated with all data group objects.

VOuDeleteDynamics

VOu Functions VO Routines

Deletes dynamic objects from the drawing.

```
void
VOuDeleteDynamics (
       OBJECT drawing)
```

VOuDeleteDynamics deletes all dynamic objects, such as data group objects, input objects, and dynamic control objects, from the drawing. Threshold table objects and variable descriptor objects are replaced by their static equivalents.

VOuDrListClear

VOu Functions VO Routines

Clears the list of drawings retrieved so far.

```
void
VOuDrListClear (void)
```

VOuDrRetrieve

VOu Functions VO Routines

Retrieves a drawing from a file.

```
OBJECT
VOuDrRetrieve (
       ADDRESS filename)
```

VOuDrRetrieve returns a drawing by reading a saved view from the file, *filename*, stripping the data sources and dynamics from it and returning the drawing object. This routine builds a list of the drawings that have been read in and saves them. If a drawing has already been retrieved, this routine simply returns the corresponding entry from the list. See also *VOuDeleteDynamics*.

VOuDyCoConvert

VOu Functions VO Routines

Converts an object with pre-8.0 color dynamics to post-8.0 dynamics.

```
void
VOuDyCoConvert (
       OBJECT color_object)
```

VOuDyCoConvert converts an object with pre-8.0 color dynamics to post-8.0 dynamics. *VOuDyCoConvert* creates a dynamic control object that uses the foreground color attribute for dynamics and attaches this dynamic control object to the *color_object*. See also *TviConvertDynamics*.

VOuDySdConvert

VOu Functions VO Routines

Converts an object with pre-8.0 subdrawing dynamics to post-8.0 dynamics.

```
void
VOuDySdConvert (
       OBJECT thresh_object,
       OBJECT *sdobject_ptr)
```

VOuDySdConvert converts an object with pre-8.0 subdrawing dynamics to post-8.0 dynamics. Given the threshold table object, *thresh_object*, and a pointer to a subdrawing object, *sdobject_ptr*, *VOuDySdConvert* creates a dynamic control object that emulates subdrawing dynamics. See also *TviConvertDynamics*.

VOuGetInList

VOu Functions VO Routines

Gets the list of objects in a viewport.

```
OBJECT
VOuGetInList (
       OBJECT candidates,
       OBJECT xform,
       RECTANGLE *vp)
```

VOuGetInList creates a list containing the objects in a drawing or a deque, *candidates*, that might intersect a given viewport, *vp*. The program applies a min-max test to the objects, comparing their *xform*-transformed bounding boxes to the viewport. Any object that might be in the viewport is added to the list. Therefore, the routine eliminates all objects that are definitely outside the viewport.

VOuGetMovePt

VOu Functions VO Routines

Gets the move point for an object.

```
void
VOuGetMovePt (
       OBJECT InObject,
       DV_POINT *pt)
```

VOuGetMovePt gets the move point for an object or a deque of objects, *InObject*. Sets the parameter, *pt*, to the world coordinates of the move point for *InObject*. The move point is the same as the move point seen in DV-Draw.

VOuHasColorDynamics

VOu Functions VO Routines

Determines if the object has dynamic color.

```
BOOLPARAM
VOuHasColorDynamics (
       OBJECT object)
```

VOuHasColorDynamics determines whether or not the object has pre-8.0 color dynamics. The routine determines this by traversing the object's subobjects looking for a dynamic color, which is a variable descriptor object of type *V_COLOR*. Returns *YES* or *NO*.

VOuIsDynamic

VOu Functions VO Routines

Determines if the object has dynamics.

```
BOOLPARAM
VOuIsDynamic (
       OBJECT object)
```

VOuIsDynamic determines whether or not the object has dynamics. The following objects are considered dynamic: input objects, data group objects, graphical objects with attached variable descriptor or dynamics control objects, and threshold table objects. Returns *YES* or *NO*.

VOuObGetNameSlot

VOu Functions VO Routines

Gets the name from the name slot of an object.

```
char *
VOuObGetNameSlot (
       OBJECT object)
```

VOuObGetNameSlot returns the name from the internal name slot of an object. Currently the only object that uses an internal name slot is the dynamic control object.

VOuObMatchNameSlots

VOu Functions VO Routines

Populates a deque with objects of a given type whose name slots match a given name.

```
int
VOuObMatchNameSlots (
       OBJECT start_obj,
       int obj_type,
       char *name,
       OBJECT deque)
```

VOuObMatchNameSlots populates a deque with objects of a given type whose internal name slot matches a given name. Currently the only object that uses an internal name slot is the dynamic control object. This routine starts checking at *start_obj* and copies into *deque* any objects and subobjects that match *obj_type* and *name*. If *name* is *NULL*, all objects of the given type are put into the deque. Once the deque is populated, use *VOdqGetEntry* or a traversal routine such as *TobForEachSubobject* to filter the objects. This routine provides a means of obtaining a named dynamic control object. See the example below. Returns the number of objects found.

VOuObMove

VOu Functions VO Routines

Moves an object.

```
void
VOuObMove (
       OBJECT object,
       int flag,
       int x,
       int y)
```

VOuObMove moves an object in world coordinates by a relative amount (*RELATIVE_MOVE*) or to an absolute position (*ABSOLUTE_MOVE*), depending on the flag value. When an object is moved to an absolute position, the object is centered on the absolute point.

VOuObSetNameSlot

VOu Functions VO Routines

Sets a name in the name slot of an object.

```
void
VOuObSetNameSlot (
       OBJECT object,
       char *name)
```

VOuObSetNameSlot sets the name in the name slot of an object to *name*. This routine provides a means of setting the internal name slot for a dynamic control object. Use this routine to change the name of the dynamic control object that was named in DV-Draw or to name a dynamic control object that you created using *VOdyCreate*.

VOuVpBound

VOu Functions VO Routines

Gets the boundary of transformed viewport.

```
void
VOuVpBound (
       RECTANGLE *vp,
       OBJECT xform,
       RECTANGLE *boundvp)
```

VOuVpBound gets the smallest viewport containing the viewport, *vp*, transformed by the transformation object, *xform*, which can include a rotation of the viewport.

VOuVpEmpty

VOu Functions VO Routines

Sets the viewport to indicate empty.

```
void
VOuVpEmpty (
       RECTANGLE *vp)
```

VOuVpEmpty sets the viewport, *vp*, to indicate empty. Sets the upper right of the viewport to the minimum coordinate values and the lower left coordinates to the maximum coordinate values. This lets *VOuVpUnion* merge viewports easily.

VOuVpObGet

VOu Functions VO Routines

Gets bounding viewport for object.

```
void
VOuVpObGet (
       OBJECT object,
       OBJECT xform,
       RECTANGLE *vp)
```

VOuVpObGet gets the bounding viewport, *vp*, for the object when it has been transformed by *xform*. This is calculated from the bounding box of the object.

VOuVpObscured

VOu Functions VO Routines

Determines if a viewport is partially obscured.

```
BOOLPARAM
VOuVpObscured (
       RECTANGLE *vp,
       RECTANGLE **obsvps)
```

VOuVpObscured determines whether or not any part of the viewport, *vp*, is obscured by any viewport in the specified *NULL*-terminated array of obscuring viewports, *obsvps*. Returns *YES* or *NO*.

VOuVpPtsGet

VOu Functions VO Routines

Gets the bounding viewport for the array of points.

```
void
VOuVpPtsGet (
       DV_POINT *pts,
       int numpts,
       RECTANGLE *vp)
```

VOuVpSort

VOu Functions VO Routines

Sorts the viewport's coordinates.

```
void
VOuVpSort (
       RECTANGLE *vp)
```

VOuVpSort sorts coordinates of the viewport, *vp*. This ensures that the lower left point (*ll*) is really lower and to the left of the upper right point (*ur*).

VOuVpUnion

VOu Functions VO Routines

Adjusts one viewport to contain the other.

```
void
VOuVpUnion (
       RECTANGLE *vp1,
       RECTANGLE *vp2)
```

VOuVpUnion adjusts the coordinates of the first viewport, *vp1*, to contain the second viewport, *vp2*.

VOuVpVisible

VOu Functions VO Routines

Determines if a viewport is visible.

```
BOOLPARAM
VOuVpVisible (
       RECTANGLE *testvp,
       RECTANGLE *invp,
       RECTANGLE **obsvps)
```

VOuVpVisible determines whether a portion of the viewport, *testvp*, is visible, where it is to be clipped into the viewport, *invp*, and where it is to be clipped outside the *NULL*-terminated viewport array, *obsvps*. Note that the input viewport, *testvp,* is modified. Returns *YES* or *NO*.

VOuXfDoesFlip

VOu Functions VO Routines

Determines if the transform flips the object.

```
BOOLPARAM
VOuXfDoesFlip (
       OBJECT xform)
```

VOuXfDoesFlip determines if *xform* changes the object from a right-hand coordinate system to a left-hand coordinate system. Returns *YES* if the object would be flipped by the transformation. Otherwise returns *NO.*

VOuXfDrFit

VOu Functions VO Routines

Creates a transformation for drawing in a viewport.

```
OBJECT
VOuXfDrFit (
       RECTANGLE *vp,
       BOOLPARAM all_visible)
```

VOuXfDrFit calculates the transformation that makes a drawing fit into a viewport. In general, there are two ways for the drawing to fit, since the aspect ratio of the drawing is 1:1 and the aspect ratio of viewport is usually not 1:1. The two cases are illustrated below. *YES* guarantees that the whole drawing is visible; *NO* guarantees that off-drawing space is not visible. *all_visible* should be set as desired.

VOuXfStretchCreate

VOu Functions VO Routines

Creates a transformation to map one rectangle to another.

```
OBJECT
VOuXfStretchCreate (
       RECTANGLE r1,
       RECTANGLE r2)
```

VOuXfStretchCreate calculates the transformation that maps one rectangle to another, stretching the x or y coordinate as necessary to make it fit. Note that this transformation does not preserve aspect ratio, so when the control points of certain objects get transformed, they may change their appearance with respect to other objects in the drawing. In particular, strange transformations will occur with arcs and circles. The transformation maps *r1* to *r2*.

Examples

The following code fragment shows how to obtain a named dynamic control object from a view. The dynamic control object's name is "*robot1_dynamics*" and the view's filename is "*robot.v*."

```
VIEW view;
OBJECT drawing, robot1_deque, robot1_dyn_object;

view = TviLoad ("robot.v");
drawing = TviGetDrawing (view);
robot1_deque = VOdqCreate (10);
VOuObMatchNameSlots (drawing, OT_DYNAMIC, "robot1_dynamics", robot1_deque);

/* Since "robot1_dynamics" is a unique name, only one object is in the deque */
robot1_dyn_object = VOdqGetEntry (robot1_deque, 1);
```

VOvd (VOvariabledescriptor)

VOvd Functions VO Routines

Manages variable descriptor objects (*vd*). Variable descriptor objects maintain lower-level data structures called variable descriptors (*vdp*). Variable descriptors describe variables that control the dynamic aspects of the display. See the *VP* and *VG* routines.

Variable descriptor objects should not share variable descriptors. If several connections to the same data are required, multiple variable descriptors should be created.

A variable descriptor object only controls one type of attribute. Variable descriptor objects return one of the following types of dynamic data:

Normalized datum	*Representing the current data value.*
Text	*Representing the current text value.*
Color object	*Maintained for compatibility with DataViews releases prior to version 8.0, but is considered obsolete. A variable descriptor object of type* COLOR *is used as a color attribute for graphical objects with pre-8.0 color dynamics.*

Variable descriptor objects supply normalized data to threshold table objects or dynamic control objects. See the *VOdy* and *VOtt* modules.

VOvd Functions

VOvdChanged	Determines if the value changed a noticeable amount.
VOvdClone	See VOobClone.
VOvdCreate	Creates and returns a variable descriptor object.
VOvdDereference	See VOobDereference.
VOvdDvGet	Gets the dynamic data value of a variable descriptor object.
VOvdGetVdp	Returns a pointer to the variable descriptor structure.
VOvdRefCount	See VOobRefCount.
VOvdReference	See VOobReference.
VOvdReset	Resets the variable descriptor object to an initial state.
VOvdStatistic	Returns statistics about variable descriptors.
VOvdSvGet	Gets the static value of a variable descriptor object.
VOvdSvPut	Sets the static value of a variable descriptor object.
VOvdSwitch	Changes the object's variable descriptor structure.
VOvdType	Returns variable type of the variable descriptor object.
VOvdValid	See VOobValid.

A *VOvd* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOvd* routine to save the overhead of an additional routine call.

VOvdChanged

VOvd Functions VO Routines

Determines if the value changed a noticeable amount.

```
BOOLPARAM
VOvdChanged (
       OBJECT vd)
```

VOvdChanged determines whether the value of the variable descriptor object, *vd*, has changed. Returns *YES* or *NO*.

VOvdCreate

VOvd Functions VO Routines

Creates and returns a variable descriptor object.

```
OBJECT
VOvdCreate (
       ADDRESS var,
       int type,
       DATUM statval)
```

VOvdCreate creates and returns a variable descriptor object. *var* specifies either an existing data source variable or an existing variable descriptor structure, *VARDESC*, to which the variable descriptor object is attached. If this parameter contains a data source variable, a variable descriptor structure is created, through which the data source variable is attached. If *var* is a variable descriptor, it must not already belong to another object. If the parameter *type* is defined to be *NUMBER*, a numeric variable descriptor object, *var*, is created. If the parameter *type* is defined to be *DV_TEXT*, a text variable descriptor object, *var*, is created. The *COLOR* type flag is obsolete but maintained for compatibility with previous releases. The default value, *statval*, is obsolete but maintained for compatibility with previous releases. It is used only for pre-8.0 dynamics when the data described by the variable descriptor object is unavailable or inappropriate. If *type* is *NUMBER*, *statval* should be an integer number within the normalized range [0, 32K]. If *type* is *DV_TEXT*, *statval* should be a text string. If *type* is *COLOR*, *statval* should be a color object.

VOvdDvGet

VOvd Functions VO Routines

Returns the current value of the dynamic data defined by a variable descriptor object.

```
DATUM
VOvdDvGet (
       OBJECT vd)
```

This routine only works on pre-8.0 dynamics.

VOvdGetVdp

VOvd Functions VO Routines

Returns a pointer to the variable descriptor structure.

```
ADDRESS
VOvdGetVdp (
       OBJECT vd)
```

VOvdGetVdp returns the address of the variable descriptor structure belonging to *vd*. See also the *VP* and *VG* routines.

VOvdReset

VOvd Functions VO Routines

Resets the variable descriptor object to an initial state.

```
void
VOvdReset (
       OBJECT vd)
```

VOvdStatistic

VOvd Functions VO Routines

Returns statistics about variable descriptors.

```
LONG
VOvdStatistic (
       int flag)
```

VOvdStatistic returns statistics about variable descriptor objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of variable descriptor objects.

VOvdSvGet

VOvd Functions VO Routines

Gets the default or static value of a variable descriptor object.

```
DATUM
VOvdSvGet (
       OBJECT vd)
```

This routine only works on pre-8.0 dynamics.

VOvdSvPut

VOvd Functions VO Routines

Sets the static value of a variable descriptor object to the value, *statval*.

```
void
VOvdSvPut (
       OBJECT vd,
       DATUM statval)
```

This routine only works on pre-8.0 dynamics. You cannot use this routine to specify an outgoing value when the incoming data has an undefined value. If the incoming data has an undefined value, the first value of the threshold table is used.

VOvdSwitch

VOvd Functions VO Routines

Changes the object's variable descriptor structure.

```
void
VOvdSwitch (
       OBJECT vd,
       VARDESC vdp)
```

VOvdSwitch replaces the variable descriptor used by *vd* with a new variable descriptor, *vdp*, and destroys the old one. The new variable descriptor must not already belong to another object.

VOvdType

VOvd Functions VO Routines

Returns variable type of the variable descriptor object.

```
int
VOvdType (
       OBJECT vd)
```

VOvdType returns the type of the variable descriptor object, *vd*. The type can be:

NUMBER *for a numerical variable descriptor object*

DV_TEXT *for a text variable descriptor object*

COLOR *for a color variable descriptor object. Obsolete, but maintained for compatibility with previous releases*

Examples

The following code fragment creates a variable descriptor object and sets its range:

```
OBJECT vd;
VARDESC vdp;
int defaultnumber;

defaultnumber = 0;
vd = VOvdCreate (dsv, NUMBER, (DATUM) defaultnumber)
vdp = VOvdGetVdp (vd);
VPvd_drange (vdp, 0.0, 1.0);
```

VOvt (VOvectortext)

VOvt Functions VO Routines

Manages vector text objects (*vt*). A vector text object is similar to an ordinary text object, except that it is drawn in world coordinate vectors, which are mapped to screen coordinates by the world-to-screen transform. Vector text objects can therefore be panned or zoomed without changing their relative size and position in the drawing. They can also be scaled in either dimension, rotated or slanted, and a variety of fonts are available, based on the Hershey fonts.

Vector text attributes are direction, position, width, height, angle, slant, character spacing, line spacing, foreground color, and fontname. The text object is attached to the drawing at an anchor point which can be at one of nine positions in the same manner as with *VOtx* text objects.

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	**VOvt**
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	VOxf
VOdb g	VOed						

VOvt Functions

VOvtAtGet	See VOobAtGet.
VOvtAtSet	See VOobAtSet.
VOvtBox	See VOobBox.
VOvtClone	See VOobClone.
VOvtCreate	Creates and returns a vector text object.
VOvtDereference	See VOobDereference.
VOvtFitRect	Finds dimensions of vector text to fit a rectangle.
VOvtGetBound	Gets the vector text boundary vectors.
VOvtGetString	Gets the string value of the vector text object.
VOvtIntersect	See VOobIntersect.
VOvtPtGet	See VOobPtGet.
VOvtPtSet	See VOobPtSet.
VOvtRefCount	See VOobRefCount.
VOvtReference	See VOobReference.
VOvtSetString	Sets new string value for the vector text object.
VOvtStatistic	Returns statistics about vector text objects.
VOvtTraverse	See VOobTraverse.
VOvtValid	See VOobValid.
VOvtXfBox	See VOobXfBox.
VOvtXformBox	See VOobXformBox.

A *VOvt* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOvt* routine to save the overhead of an additional routine call.

VOvtCreate

VOvt Functions VO Routines

Creates and returns a vector text object.

```
OBJECT
VOvtCreate (
       char *string,
       OBJECT anchor_pt,
       ATTRIBUTES *attributes)
```

VOvtCreate creates and returns a vector text object. *string* is a *NULL*-terminated character string containing the text content of the object. The anchor point, *anchor_pt*, is the point object that defines where the text string appears on the screen. Valid *attributes* field flags are:

TEXT_DIRECTION	*TEXT_POSITION*
TEXT_WIDTH	*TEXT_SLANT*
TEXT_HEIGHT	*TEXT_ANGLE*
TEXT_CHARSPACE	*TEXT_LINESPACE*
TEXT_FONTNAME	*FOREGROUND_COLOR*

If *attributes* is *NULL*, default values are used.

VOvtFitRect

VOvt Functions VO Routines

Finds dimensions of vector text to fit a rectangle.

```
void
VOvtFitRect (
       OBJECT vtext,
       RECTANGLE *wvp,
       float *width,
       float *height,
       DV_POINT *wpt_anchor)
```

VOvtFitRect gives the *height* and *width* attribute values and anchor point position, *wpt_anchor,* required to make the vector text object, *vtext*, fit exactly within the specified boundary viewport rectangle, *wvp*. *wpt_anchor* is specified in world coordinates. This routine does not change the vector text object.

VOvtGetBound

VOvt Functions VO Routines

Gets the vector text boundary vectors.

```
void
VOvtGetBound (
       OBJECT vtext,
       int *wx,
       int *wy,
       int *hx,
       int *hy)
```

VOvtGetBound returns four world coordinate values representing the boundary of the vector text object on the screen. This boundary can be a rectangle of any shape, size, and orientation and is defined by two vectors extending from the lower left corner of the text to the upper left corner (*hx,hy*), and from the lower left corner to the lower right corner (*wx,wy*). This yields a tighter boundary than the *VOobBox* routine which gives the minimum horizontal and vertical extents.

VOvtGetString

VOvt Functions VO Routines

Gets the string value of the vector text object.

```
char *
VOvtGetString (
       OBJECT vtext)
```

VOvtGetString returns a pointer to the string associated with the vector text object. This is a pointer to an internal data structure which should not be modified.

VOvtSetString

VOvt Functions VO Routines

Sets new string value for the vector text object.

```
void
VOvtSetString (
       OBJECT vtext,
       char *newstring)
```

VOvtSetString sets a new string value, *newstring,* for the vector text object. If the new string is shorter than the old string, it is simply copied into the old string's buffer. Otherwise, storage is reallocated to allow for the increased length.

VOvtStatistic

VOvt Functions VO Routines

Returns statistics about vector text objects.

```
LONG
VOvtStatistic (
       int flag)
```

VOvtStatistic returns statistics about vector text objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h.* If *flag* is *OBJECT_COUNT*, returns the current number of vector text objects.

VOxf (VOxform)

VOxf Functions VO Routines

Manages transform objects (*xf*). Transform objects map two-dimensional points from one coordinate system to another. Matrices post-multiply the points: [x y 1][mat].

VOob	VOdg	VOel	VOin	VOno	VOre	VOsf	VOu
VOar	VOdq	VOg	VOit	VOpm	VOru	VOsk	VOvd
VOci	VOdr	VOic	VOln	VOpt	VOsc	VOtt	VOvt
VOco	VOdy	VOim	VOlo	VOpy	VOsd	VOtx	**VOxf**
VOdbg	VOed						

VOxf Functions

VOxfCatCreate	Creates a concatenation of two transform objects.
VOxfDereference	See VOobDereference.
VOxfDpPoint	Transforms a point giving *double* coordinates.
VOxfInvCreate	Creates the inverse of a transform object.
VOxfMatCreate	Creates a general matrix transform object.
VOxfMatGet	Gets the 3x3 matrix for the *XFORM* object.
VOxfPoint	Transforms a point according to the transform object.
VOxfRefCount	See VOobRefCount.
VOxfReference	See VOobReference.
VOxfRotCreate	Creates a rotation matrix transform object.
VOxfScale	Gets the scale factor associated with the transform.
VOxfStatistic	Returns statistics about transforms.
VOxfStCreate	Creates a scale-translate transform object.
VOxfSxytCreate	Creates an x,y scale-translate transform object.
VOxfValid	See VOobValid.

A *VOxf* routine that refers to a *VOob* routine performs the same function and uses the same parameters as the *VOob* routine indicated. You can use the *VOxf* routine to save the overhead of an additional routine call.

VOxfCatCreate

VOxf Functions VO Routines

Creates a concatenation of two transform objects.

```
OBJECT
VOxfCatCreate (
       OBJECT xform,
       OBJECT xform2)
```

VOxfCatCreate creates and returns a transform that is the concatenation of the two specified transform objects.

VOxfDpPoint

VOxf Functions VO Routines

Transforms a point giving *double* coordinates.

```
void
VOxfDpPoint (
       OBJECT xform,
       double *x,
       double *y)
```

VOxfDpPoint transforms the point with coordinates (x,y) according to the transform object. Computes the point exactly, setting x and y to the double precision result.

VOxfInvCreate

VOxf Functions VO Routines

Creates the inverse of a transform object.

```
OBJECT
VOxfInvCreate (
       OBJECT xform)
```

VOxfInvCreate creates and returns the inverse of *xform*. The inverse of the scale-translate transformation is:

where s is the scale factor and (x,y) is the point offset.

VOxfMatCreate

VOxf Functions VO Routines

Creates a general matrix transform object.

```
OBJECT
VOxfMatCreate (
       float matrix[3][3])
```

VOxfMatCreate creates and returns a general matrix transform object. The matrix is a 3x3 homogeneous transformation matrix for two-dimensional coordinates. This type of transformation can represent translation, rotation, shear, and scaling. The general *xform* is arranged as follows:

VOxfMatGet

VOxf Functions VO Routines

Gets the 3x3 matrix for the *XFORM* object.

```
void
VOxfMatGet (
       OBJECT xform,
       float outmat[3][3])
```

VOxfMatGet gets the 3x3 matrix corresponding to *xform*. This matrix corresponds to a homogeneous transformation of a two-dimensional point. For an explanation of coordinate transformations, refer to any computer graphics textbook.

VOxfPoint

VOxf Functions VO Routines

Transforms a point according to the transform object.

```
int
VOxfPoint (
       OBJECT xform,
       DV_POINT *pt)
```

VOxfPoint transforms the point, *pt*, according to the transform object, *xform*. *VOxfPoint* transforms point data structures, not point objects. These point structures are the same as those used by the *GR* routines. Returns *DV_FAILURE* if the transformed point is out of range, that is, if it won’t fit in a *LONG*. Otherwise returns *DV_SUCCESS*.

VOxfRotCreate

VOxf Functions VO Routines

Creates a rotation matrix transform object.

```
OBJECT
VOxfRotCreate (
       double angle,
       LONG x,
       LONG y)
```

VOxfRotCreate creates and returns a rotation matrix transform object. The angle is in degrees. (x,y) is the center of rotation.

VOxfScale

VOxf Functions VO Routines

Returns the scale factor associated with the transform.

```
double
VOxfScale (
       OBJECT xform)
```

VOxfStatistic

VOxf Functions VO Routines

Returns statistics about transforms.

```
LONG
VOxfStatistic (
       int flag)
```

VOxfStatistic returns statistics about transform objects, depending on the value of *flag*. Valid flag values are defined in *VOstd.h*. If *flag* is *OBJECT_COUNT*, returns the current number of transform objects.

VOxfStCreate

VOxf Functions VO Routines

Creates a scale-translate transform object.

```
OBJECT
VOxfStCreate (
       double scale_factor,
       LONG x_offset,
       LONG y_offset)
```

VOxfStCreate creates and returns a scale-translate transform object. This generates a matrix of the following form:

where s is *scale_factor*, x is *x_offset*, and y is *y_offset*. A negative *scale_factor* makes the transform flip the object on which it operates.

VOxfSxytCreate

VOxf Functions VO Routines

Creates an x,y scale-translate transform object.

```
OBJECT
VOxfSxytCreate (
       double x_scale,
       double y_scale,
       LONG x_offset,
       LONG y_offset)
```

VOxfSxytCreate creates an (x,y) scale-translate transform object. This is a transform where the x and y scale factors are different. A negative *scale_factor* makes the transform flip the object on which it operates.

VUer Routines

Event handling routines.

VUer Modules

All modules in the VUer layer require the following include file:

```
#include "dvinteract.h"
#include "VUerfundecl.h"
```

VUerhandler	Routines that pass events to the event handler, then trigger appropriate service routines according to event requests.

VUerpost	Routines that post, activate, and deactivate event requests and service result requests with the event handler.

VUerpost

VUerpost Functions VUer Routines

Routines that post, activate, and deactivate event requests and service result requests with the event handler. They are called implicitly by the input objects through their interaction handlers, but they can also be called directly by the application programmer. Application programs using the *VUerPost* routines must include the header file *dvinteract.h*.

See Also

VUerHandleLocEvent for the order in which event requests and service result requests are serviced.

VUerpost Functions

VUerActivate	Activates an event request.
VUerActivateClient	Activates all event requests of a particular client.
VUerBoundaryEventDpPost	Same as VUerBoundaryEventPost, plus support for clipping.
VUerBoundaryEventPost	Posts a request for an event.
VUerCatchAllEventPost	Posts an event request for all events.
VUerClearAll	Clears all event requests of a particular client.
VUerClearAllForMonClient	Clears all service result requests posted on a monitored client.
VUerDeactivate	Deactivates an event request.
VUerDeactivateClient	Deactivates all event requests of a client.
VUerIsActive	Determines if an event request is active.
VUerObjectEdgeDpPost	Same as VUerObjectEdgePost, plus support for clipping.
VUerObjectEdgePost	Posts an object event request.
VUerRectEdgeDpPost	Same as VUerRectEdgePost, plus support for clipping.
VUerRectEdgePost	Posts a rectangle event request.
VUerServiceResultPost	Posts a service result request.
VUerWinEventPost	Posts a request for a window event.

VUerActivate

VUerpost Functions VUer Routines

Activates an event request.

```
void
VUerActivate (
       EVENT_REQUEST EventRequest)
```

VUerActivate activates an event request. When event requests are posted, they become active. Therefore, this routine is used to activate event requests after they have been deactivated by a call to VUerDeactivate.

VUerActivateClient

VUerpost Functions VUer Routines

Activates all event requests of a particular client.

```
void
VUerActivateClient (
       OBJECT Client)
```

VUerActivateClient activates all event requests associated with a particular client id, *Client*.

VUerBoundaryEventDpPost

VUerpost Functions VUer Routines

Same as VUerBoundaryEventPost, plus support for clipping.

```
EVENT_REQUEST
VUerBoundaryEventDpPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       int Label,
       ULONG ErInterpretation,
       ...,
```

Additional Parameters:

If *ErInterpretation* is *VUER_SE_EVENT:*

```
ULONG PickEventType,
ULONG *PickSyms,
DRAWPORT drawport)
```

If *ErInterpretation* is *VUER_BRE_EVENT*:

```
ULONG PickEventType,
ULONG *PickSyms,
RECTANGLE *BndingRect,
BOOLPARAM InOut,
DRAWPORT drawport,
RECTANGLE *cliprect)
```

If *ErInterpretation* is *VUER_DOE_EVENT*:

```
ULONG PickEventType,
ULONG *PickSyms,
OBJECT EdgeObj,
OBJECT XformObj,
BOOLPARAM InOut,
DRAWPORT drawport,
RECTANGLE *cliprect)
```

If *ErInterpretation* is *VUER_POS_EVENT*:

```
RECTANGLE *BndingRect,
BOOLPARAM InOut,
DRAWPORT drawport,
RECTANGLE *cliprect)
```

If *ErInterpretation* is *VUER_OPOS_EVENT*:

```
OBJECT EdgeObj,
OBJECT XformObj,
BOOLPARAM InOut,
DRAWPORT drawport,
RECTANGLE *cliprect)
```

```
int
fcn (
       OBJECT Client,
       EVENT_REQUEST Request,
       int Label,
       OBJECT Loc,
       ADDRESS Args)
```

VUerBoundaryEventDpPost posts an event request for five types of event interpretations. This routine has a variable length list of parameters, depending on the event request interpretation, which you pass explicitly in the *ErInterpretation* parameter. For descriptions of the parameters see *VUerBoundaryEventPost*. The additional parameters required for each of the *ErInterpretation* flags are:

- *drawport*: the drawport for which you are making the event request. This parameter can be used to distinguish between overlapping drawports.
- *cliprect*: the clipped rectangle for which you are making the event request. This parameter should be *NULL* when you want the event request applied to the entire object or region. This parameter should be specified when you want the event request applied only to the clipped part of the object or region. This parameter is not used when *ErInterpretation* is *VUER_SE_EVENT*, and is ignored when *InOut* is *V_OUTSIDE*.

VUerBoundaryEventPost

VUerpost Functions VUer Routines

Posts a request for an event.

```
EVENT_REQUEST
VUerBoundaryEventPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       int Label,
       ULONG ErInterpretation,
       ...,
```

Additional Parameters:

If *ErInterpretation* is *VUER_SE_EVENT*:

```
ULONG PickEventType,
ULONG *PickSyms)
```

If *ErInterpretation* is *VUER_BRE_EVENT*:

```
ULONG PickEventType,
ULONG *PickSyms,
RECTANGLE *BndingRect,
BOOLPARAM InOut)
```

If *ErInterpretation* is *VUER_DOE_EVENT*:

```
ULONG PickEventType,
ULONG *PickSyms,
OBJECT EdgeObj,
OBJECT XformObj,
BOOLPARAM InOut)
```

If *ErInterpretation* is *VUER_POS_EVENT*:

```
RECTANGLE *BndingRect,
BOOLPARAM InOut)
```

If *ErInterpretation* is *VUER_OPOS_EVENT*:

```
OBJECT EdgeObj,
OBJECT XformObj,
BOOLPARAM InOut)
```

```
int
fcn (
      OBJECT Client,
      EVENT_REQUEST Request,
      int Label,
      OBJECT Loc,
      ADDRESS Args)
```

VUerBoundaryEventPost posts an event request for five types of event interpretations. This routine should be used when drawport clipping is not an issue, such as when you have only one drawport on your screen. For situations with multiple drawports, use *VUerBoundaryEventDpPost*.

This routine has a variable length list of parameters depending on the event request interpretation, which you pass explicitly in the *ErInterpretation* parameter. The parameters that are common are:

Client: client id making the request.

fcn: pointer to the service routine called when the request is satisfied.

Args: argument structure passed to the service routine when it is called.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure, which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

Label: a label given by the programmer to identify this event request.

ErInterpretation: a flag indicating how the event request should be interpreted. The additional parameters in the variable length argument list, which depend on the event request interpretation, are listed below. The valid flags are:

VUER_SE_EVENT	A request for a key or button event anywhere on the screen, also called a simple edge event. Requires *PickEventType* and *PickSyms*.
VUER_BRE_EVENT	A request for a key or button event inside or outside a rectangle specified in screen coordinates, also called a boundary edge event. Requires *PickEventType*, *PickSyms*, *BndingRect*, and *InOut*.
VUER_DOE_EVENT	A request for a key or button event inside or outside a graphical object, also called a object edge event. Requires *PickEventType*, *PickSyms*, *EdgeObj*, *XformObj*, and *InOut*.
VUER_POS_EVENT	A request for a motion or position event inside or outside a rectangle specified in screen coordinates, also called a position event. Requires *BndingRect* and *InOut*.
VUER_OPOS_EVENT	A request for a motion or position event inside or outside a graphical object, also called an object position event. Requires *EdgeObj*, *XformObj*, and *InOut*.

The additional parameters in the variable length declaration list are:

PickEventType: the event type. The valid event type flags are *V_KEYPRESS*, *V_KEYRELEASE*, *V_BUTTONPRESS*, and *V_BUTTONRELEASE*. The event type of the location object is compared to this flag to determine if it matches the request. You can only enter one event type flag. To post for more than one event type, call this routine again with another event type and the same service routine.The definitions of these flags are located in *dvGR.h*.

PickSyms: an array of flags representing keyboard or mouse picks. The last item in the array must be zero. The key symbol or button of the location object is compared to this array to determine if it matches the request. Use *1*, *2*, or *3* for mouse button 1, 2, or 3. The definitions of the key symbol flags are located in *GRkeysymdef.h*.

BndingRect: a pointer to a rectangle specified in screen coordinates. The coordinates of the location object are compared to this rectangle to determine if the event occurred inside or outside this region.

InOut: a flag that specifies whether the event request should be interpreted as an inside event request or an outside event request with respect to the specified rectangle or graphical object. Valid flags are *V_INSIDE* and *V_OUTSIDE*.

EdgeObj: the graphical object. The coordinates of the location object is compared to this object to determine if the event occurred inside or outside the object. For objects with a fill status of *EDGE*, the location object is outside the object unless it directly intersects the edge of the object. The object must be visible for the request to be serviced.

XformObj: the transform object required for converting the graphical object's world coordinates to screen coordinates. You can get this parameter by calling *TdpGetXform* with the *DR_TO_SCREEN* flag.

The event request interpretation and the *InOut* flag determine the order in which *VUerHandleLocEvent* or *VUerHandler* service the event requests. Simple edge and inside event requests are posted to one list, and outside event requests are posted to a second list. The most recently posted matching simple edge or inside event request is the first and only event request serviced. If no requests from the first list are serviced, all matching outside event requests are serviced, starting with the most recently posted.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine's parameters, *Request* is the posting routine's return value, and *Loc* is the location object passed to the event handler that satisfied the event request. The service routine must return a user defined value or one of four service result flags, which are listed in the description of *VUerServiceResultPost*.

VUerCatchAllEventPost

VUerpost Functions VUer Routines

Posts an event request for all events.

```
EVENT_REQUEST
VUerCatchAllEventPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       int Label)

    int
    fcn (
           OBJECT Client,
           EVENT_REQUEST Request,
           int Label,
           OBJECT Loc,
           ADDRESS Args)
```

VUerCatchAllEventPost posts an event request to catch any event. This routine can be used to request any event, but is particularly useful for requesting events that do not fit into any of the categories covered by the posting routines VUerBoundaryEventPost, VUerObjectEdgePost, VUerRectEdgePost, or VUerWinEventPost. Since this posting routine does not contain parameters for sorting events into types, you must handle those tasks in the service routine.

Client: the client id making the request.

fcn: pointer to the service routine called when the request is satisfied, i. e. when any event is received that does not fulfill any of the other posted requests.You can specify how to interpret the events and what actions to take in this routine.

Args: the argument structure passed to the service routine.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure, which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

Label: a label given by the programmer to identify this event request.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine's parameters, *Request* is the posting routine's return value, and *Loc* is the location object passed to the event handler that satisfied the event request. The service routine must return a user defined value or one of four service result flags, which are listed in the description of *VUerServiceResultPost*.

VUerClearAll

VUerpost Functions VUer Routines

Clears all event requests of a particular client.

```
void
VUerClearAll (
       OBJECT Client)
```

VUerClearAll removes all events requests with the specified client id, *Client*, from the event handler.

VUerClearAllForMonClient

VUerpost Functions VUer Routines

Clears all service result requests posted on a monitored client.

```
void
VUerClearAllForMonClient (
       OBJECT MonitoredClient)
```

VUerClearAllForMonClient clears all service result requests that specify a particular monitored client, regardless of which client posted the request. This routine is similar to VUerClearAll, but acts based on the monitored client instead of the client.

VUerDeactivate

VUerpost Functions VUer Routines

Deactivates an event request.

```
void
VUerDeactivate (
       EVENT_REQUEST EventRequest)
```

VUerDeactivate deactivates an event request. This lets an input object deactivate specific event requests for a specific time period. Event requests can be reactivated using VUerActivate.

VUerDeactivateClient

VUerpost Functions VUer Routines

Deactivates all event requests of a client.

```
void
VUerDeactivateClient (
       OBJECT Client)
```

VUerDeactivateClient deactivates all event requests associated with the client id, *Client*. The event handler then ignores the event requests until VUerActivateClient iscalled.

VUerIsActive

VUerpost Functions VUer Routines

Determines if an event request is active.

```
BOOLPARAM
VUerIsActive (
       EVENT_REQUEST erp)
```

VUerIsActive determines if an event request is active. Event requests are changed by their posting, VUerActivate or VUerDeactivate. Returns *YES* or *NO*.

VUerObjectEdgeDpPost

VUerpost Functions VUer Routines

Same as VUerObjectEdgePost, plus support for clipping.

```
EVENT_REQUEST
VUerObjectEdgeDpPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       OBJECT EdgeObject,
       OBJECT XformObject,
       BOOLPARAM InOut,
       char *KeyStr,
       int Label,
       DRAWPORT drawport,
       RECTANGLE *cliprect)

    int
    fcn (
           OBJECT Client,
           EVENT_REQUEST Request,
           int Label,
           OBJECT Loc,
           ADDRESS Args)
```

VUerObjectEdgeDpPost posts an event request for object position and object edge events. It requires the same arguments as *VUerObjectEdgePost* plus the following arguments:

drawport: the drawport for which you are making the event request. This parameter can be used to distinguish between overlapping drawports.

cliprect: the clipped rectangle for which you are making the event request. This parameter should be *NULL* when you want the event request applied to the entire object or region. This parameter should be specified when you want the event request applied only to the clipped part of the object or region. This parameter is ignored when *InOut* is *V_OUTSIDE*.

VUerObjectEdgePost

VUerpost Functions VUer Routines

Posts an object event request.

```
EVENT_REQUEST
VUerObjectEdgePost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       OBJECT EdgeObject,
       OBJECT XformObject,
       BOOLPARAM InOut,
       char *KeyStr,
       int Label)

    int
    fcn (
          OBJECT Client,
          EVENT_REQUEST Request,
          int Label,
          OBJECT Loc,
          ADDRESS Args)
```

VUerObjectEdgePost posts an event request for object position and object edge events. This routine should be used when drawport clipping is not an issue, such as when you have only one drawport on your screen. For situations with multiple drawports, use *VUerObjectEdgeDpPost*. It requires the following arguments:

Client: the client id making the request.

fcn: pointer to the service routine called when the request is satisfied.

Args: argument structure passed to the service routine when it is called.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure, which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

EdgeObject: the graphical object. The coordinates of the location object is compared to this object to determine if the event occurred inside or outside the object. For objects with a fill status of *EDGE*, the location object is outside the object unless it directly intersects the edge of the object. The object must be visible for the request to be serviced.

XformObject: the transform object required for converting the graphical object's world coordinates to screen coordinates. You can get this parameter by calling TdpGetXform withthe *DR_TO_SCREEN* flag.

InOut: a flag that specifies whether the event request should be interpreted as an inside event request or an outside event request. *V_INSIDE* indicates an inside event request, satisfied when the locator is inside the object; *V_OUTSIDE* indicates an outside event request, satisfied when the locator is outside the object.

KeyStr: a *NULL*-terminated string containing the keys that can be pressed to satisfy the event request. A zero-length string means that any key is valid. When this parameter is *NULL*, the event request is interpreted implicitly as an object position event request, *VUER_OPOS_EVENT*. When the parameter is not *NULL*, the request is interpreted as a object edge event request, *VUER_DOE_EVENT*.

Label: a label given by the programmer to identify this event request.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine's parameters, *Request* is the posting routine's return value, and *Loc* is the location object passed to the event handler that satisfied the event request. The service routine must return a user defined value or one of four service result flags, which are listed in the description of VUerServiceResultPost.

VUerRectEdgeDpPost

VUerpost Functions VUer Routines

Same as VUerRectEdgePost, plus support for clipping.

```
EVENT_REQUEST
VUerRectEdgeDpPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       RECTANGLE *BndingRect,
       BOOLPARAM InOut,
       char *KeyStr,
       int Label,
       DRAWPORT drawport,
       RECTANGLE *cliprect)

   int
   fcn (
          OBJECT Client,
          EVENT_REQUEST Request,
          int Label,
          OBJECT Loc,
          ADDRESS Args)
```

VUerRectEdgeDpPost posts an event request for simple edge, boundary edge, or position events. It requires the arguments required by VUerRectEdgePost, plus the following arguments:

- *drawport*: the drawport for which you are making the event request. This parameter can be used to distinguish between overlapping drawports.
- *cliprect*: the clipped rectangle for which you are making the event request. This parameter should be *NULL* when you want the event request applied to the entire object or region. This parameter should be specified when you want the event request applied only to the clipped part of the object or region. This parameter is ignored when *InOut* is *V_OUTSIDE*.

VUerRectEdgePost

VUerpost Functions VUer Routines

Posts a rectangle event request.

```
EVENT_REQUEST
VUerRectEdgePost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       RECTANGLE *BndingRect,
       BOOLPARAM InOut,
       char *KeyStr,
       int Label)

    int
    fcn (
           OBJECT Client,
           EVENT_REQUEST Request,
           int Label,
           OBJECT Loc,
           ADDRESS Args)
```

VUerRectEdgePost posts an event request for simple edge, boundary edge, or position events. This routine should be used when drawport clipping is not an issue, such as when you have only one drawport on your screen. For situations with multiple drawports, use VUerRectEdgeDpPost. It requires the following arguments:

Client: the client id making the request.

fcn: pointer to the service routine called when the request is satisfied.

Args: the argument structure passed to the service routine.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure, which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

BndingRect: a pointer to a rectangle specified in screen coordinates. The coordinates of the location object are compared to this rectangle to determine if the event occurred inside or outside this region. When this parameter is *NULL*, the request is interpreted implicitly as a simple edge event request, *VUER_SE_EVENT*.

InOut: a flag that specifies whether the event request should be interpreted an inside event request or an outside event request. *V_INSIDE* indicates as an inside event request, satisfied when the locator is inside the region; *V_OUTSIDE* indicates an outside event request, satisfied when the locator is outside the region.

KeyStr: a *NULL*-terminated string containing the keys that can be pressed to satisfy the event request. A zero-length string means that any key is valid. When this parameter is *NULL*, the request is interpreted implicitly as a position event request, *VUER_POS_EVENT*. When this parameter and *BndingRect* are both non-*NULL*, the request is interpreted implicitly as a boundary edge event request, *VUER_BRE_EVENT*.

Label: a label given by the programmer to identify this event request.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine’s parameters, *Request* is the posting routine’s return value, and *Loc* is the location object passed to the event handler that satisfied the event request. The service routine must return a user defined value or one of four service result flags, which are listed in the description of VUerServiceResultPost.

VUerServiceResultPost

VUerpost Functions VUer Routines

Posts a service result request.

```
EVENT_REQUEST
VUerServiceResultPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int ArgSize,
       OBJECT MonitoredClient,
       int ResultMask,
       int Label)

    int
    fcn (
          OBJECT Client,
          EVENT_REQUEST Request,
          int Label,
          OBJECT Loc,
          ADDRESS Args)
```

VUerServiceResultPost posts a service result request with the event handler. It requires the following arguments:

Client: client id posting the service result request.

fcn: pointer to service result routine to call when the service result request is satisfied.

Args: the argument structure to pass on to the service result routine when it is called.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

MonitoredClient: input object being monitored or client id of initial service routine necessary to satisfy the service result request.

ResultMask: a mask that specifies which types of service result flags satisfy the service result request. The following flags can be bitwise OR'ed together to make the mask.

INPUT_UNUSED	indicates that no event request was satisfied.
INPUT_DONE	indicates that input sequence was completed.
INPUT_ACCEPT	indicates that the input was used by an input handler.
INPUT_CANCEL	indicates that the input activity was canceled.

Label: a label defined by the programmer to identify this service result request.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine's parameters, *Request* is the service result posting routine's return value, and *Loc* is the location object passed to the event handler that satisfied the original event request. If you want to pass the input object being monitored to the service routine, you should pass it as the label or as a member of the argument block. The service routine must return a user defined value or one of the four service result flags listed above.

VUerWinEventPost

VUerpost Functions VUer Routines

Posts a request for a window event.

```
EVENT_REQUEST
VUerWinEventPost (
       OBJECT Client,
       VUERFCNFUNPTR fcn,
       ADDRESS Args,
       int Label,
       ULONG WinEventType)

    int
    fcn (
           OBJECT Client,
           EVENT_REQUEST Request,
           int Label,
           OBJECT Loc,
           ADDRESS Args)
```

VUerWinEventPost posts an event request for window events on the current screen. The device number of the current screen is posted internally with the request. In a multiple-screen application, the event handler compares the device number in location object's *WINEVENT* structure against the device number in each request to determine which event request the location object satisfies. This routine requires the following arguments:

Client: the client id making the request.

fcn: pointer to the service routine called when the request is satisfied.

Args: the argument structure passed to the service routine.

ArgSize: the size in bytes of the *Args* structure. If *Args* is non-*NULL* and *ArgSize* is non-zero, the event handler makes a copy of the *Args* structure, which it frees when the event request is cleared. If *Args* is non-*NULL* and *ArgSize* is zero, the event handler keeps the pointer to the structure without making a copy. In this case, the structure is not freed when the event request is cleared.

Label: a label given by the programmer to identify this event request.

WinEventType: a flag indicating how the event request should be interpreted. You can only enter one event request flag. To post for more than one event type, call this routine again with another event type and the same service routine. The valid flags are:

VUER_RESIZE_EVENT	A request for a resize event.
VUER_WINQUIT_EVENT	A request for a window quit event.
VUER_ICONIFY_EVENT	A request for an iconify event.
VUER_EXPOSE_EVENT	A request for an expose event.
VUER_WIN_ENTER_EVENT	A request for a window enter event.
VUER_WIN_LEAVE_EVENT	A request for a window leave event.

The service routine is user-defined and should have the following form:

```
fcn (Client, Request, Label, Loc, Args);
```

where *Client*, *Label*, and *Args* are passed from the posting routine's parameters, *Request* is the posting routine's return value, and *Loc* is the location object passed to the event handler that satisfied the event request. The service routine must return a user defined value or one of four service result flags, which are listed in the description of VUerServiceResultPost.

VUerhandler

VUerhandler Functions VUer Routines

These routines pass events to the event handler, then trigger appropriate service routines according to event requests. Event requests are posted with the event handler by input objects through their interaction handlers, but they can also be posted directly by the application programmer. Applications using these routines must include the header file *dvinteract.h.*

See *VUerPost*, the next module, and the *Interaction Handler* chapter of the manual for more information about event requests.

See Also

VUerpost Routines, Interaction Handlers, *VOin* Routines, *VOit* Routines

VUerhandler Functions

VUerGetKeys

VUerhandler Functions VUer Routines

Gets keys corresponding to a given action type.

```
char *
VUerGetKeys (
       int ActionType)
```

VUerGetKeys returns the keys associated with a given action type specified in *ActionType*. See VUerPutKeys below for possible values of *ActionType*.

VUerHandleLocEvent

VUerhandler Functions VUer Routines

Handles a single event.

```
int
VUerHandleLocEvent (
       OBJECT LocObject)
```

VUerHandleLocEvent services a single event by determining if it satisfies any posted event requests, calls the associated service routine if it does, and triggers any service result routines. For more information about the way the event handler services events, see the *Event Handling* chapter in the *DV-Tools User's Guide*. Returns a result flag from the last service routine called:

INPUT_UNUSED	indicates that no event request was satisfied.
INPUT_DONE	indicates that input sequence was completed.
INPUT_ACCEPT	indicates that the input was used by an input handler.
INPUT_CANCEL	indicates that the input activity was canceled.

VUerHandleLocEvent services applicable event and service result requests in the following order:

1. Services only the most recently posted window event request. Does not service any other requests.
2. Services only the most recently posted inside or simple edge event request and then services result requests. Does not service outside event requests.
3. Services all the posted outside event requests, starting with the most recently posted.
4. Services all the posted service result requests, starting with the most recently posted.

VUerHandler

VUerhandler Functions VUer Routines

Starts an event service loop.

```
void
VUerHandler (
       int TermFlag,
       VUERFCNFUNPTR TermFcn,
       ADDRESS Args,
       OBJECT *Loc,
       int *TermCond)

    int
    TermFcn(
          OBJECT Client,
          EVENT_REQUEST Request,
          int Label,
          OBJECT Loc,
          ADDRESS Args)
```

VUerHandler enters a continuous event service loop, calling *TloPoll*, using the *LOC_POLL* polling method, to gather events, and VUerHandleLocEvent to handle them. It returns control to the caller depending on termination flags or the result of a programmer-defined function. It also returns control when it collects an event which does not satisfy any event requests. The routine arguments have the following functions:

TermFlag: a flag mask specifying handler states making the handler return control to the caller. The constants for the flags below are predefined in *dvinteract.h.*

Flag	**Comment**
ER_STOP_ON_ANY_EDGE	Any key press or release.
ER_STOP_ON_LEAD_EDGE	Reserved for future enhancements.
ER_STOP_ON_ANY_USE	*Result != INPUT_UNUSED*
ER_STOP_ON_UNUSED	*Result == INPUT_UNUSED*
ER_STOP_ON_DONE	*Result == INPUT_DONE*
ER_STOP_ON_ACCEPT	*Result == INPUT_ACCEPT*
ER_STOP_ON_CANCEL	*Result == INPUT_CANCEL*
ER_STOP_ON_USED	*Result == INPUT_USED*

TermFcn: an optional user-defined function called after each input event to determine if control should be returned. An example would be a user-written time-out function. This function is called with the argument *Args*; for example, *(*TermFcn)(Args)*. If the function returns *NULL*, then the handler continues to process events. If it returns non-*NULL*, then the handler returns control to the caller.

Loc: returns the location object if a request is unserviced and *NULL* if it is serviced.

TermCond: returns non-*NULL* if the routine terminates due to a *TermFlag* condition. A bit is set indicating which *TermFlag* condition caused termination.

If both *Loc* and *TermCond* are *NULL*, then termination is due to a *TermFcn* condition.

VUerPutKeys

VUerhandler Functions VUer Routines

Associates keys with a particular action type.

```
void
VUerPutKeys (
       int ActionType,
       char *Keys)
```

VUerPutKeys associates a string of keys, *Keys*, with a specified user action type, *ActionType*. Possible values for *ActionType* are:

DONE_KEYS	*CANCEL_KEYS*
SELECT_KEYS	*RESTORE_KEYS*
CLEAR_KEYS	*TOGGLE_POLLING_KEYS*

These action types are used by the interaction technique objects. The flag constants are predefined in *dvinteract.h*. *CLEAR_KEYS i*s implemented only for text entry interactions, and *TOGGLE_POLLING_KEYS* is currently implemented only for slider2D interactions.

The key string for *VUerPutKeys* and *VUerGetKeys* is a *NULL* terminated character string. Each character in the string indicates one of the defined keys. The character values \001, \002, and \003 correspond to the left, middle, and right mouse buttons respectively. To specify that no keys are defined, use *NULL* in place of the string. An empty string with a *NULL* termination binds *all* keys to the action.

Interaction Handlers (VN)

The behavior and appearance of input objects are controlled respectively by the interaction handlers and templates. Interaction handlers are sets of internal routines that determine the general method by which an input object interacts with the user. The interaction handler is attached to the input object's input technique object, and must be externally referenced using a *GLOBALREF* declaration. Interaction handlers work in conjunction with input objects and input technique objects, which are covered in the VOin and VOit modules.

Interaction Handlers

Templates	Drawing Objects composed of three rectangle object ares.
Key Bindings and Action Types	A list of action type flags
Echo Functions	Customize input object behavior at critical points when drawn
Modifying Active Input Objects	Methods for modifying active input objects

Name	Description
VNbutton	Implements a button interaction.
VNchecklist	Implements a checklist interaction.
VNcombiner	Allows multiple input objects to be embedded and controlled within a single input object.
VNmenu	Implements a menu-based interaction.
VNmultiplexor	Implements a multiplexor-based interaction.
VNpalette	Implements a color palette-based interaction.
VNslider	Implements a valuator-based interaction.
VNslider2D	Implements a 2-dimensional valuator-based interaction.
VNtext	Implements a single line text entry interaction.
VNtextedit	Implements a multi-line text editing interaction.
VNtoggle	Implements a toggle-based interaction.

The following interaction handlers implement interactions using Motif or OPEN LOOK widgets. They are described separately in the *DataViews and the View Widget in the X Environment Manual*:

VNwcheck	Implements a widget-based checklist interaction.
VNwmenu	Implements a widget-based menu interaction.
VNwradio	Implements a widget-based radio button list interaction.
VNwslider	Implements a widget-based valuator interaction.
VNwtext	Implements a widget-based text entry interaction.
VNwtoggle	Implements a widget-based text toggle interaction.

Templates

VN Description **Layout.area** **Key Bindings** **Echo Functions** **Modifying Active**

Templates are drawing objects composed of three rectangle object areas: the **Layout** area, the **Objects** area, and the **Flags** area. Within the template, strict naming conventions must be followed for the areas and the objects within those areas.

The **Layout** area contains the physical layout of the interaction. If the layout area is empty, it means that the interaction handler does not echo, leaving the echoing to the caller. Also, default actions are used to control the input sequence. The layout area must be named *Layout.area*.

The **Objects** area contains items that are displayed sequentially in buttons, scrolling checklists, scrolling menus, and toggles. The objects area must be named *Objects.area*.

The **Flags** area contains optional flags that affect the appearance and behavior of the input devices. The objects that define the individual flag must be bounded by an object named *Flags.area*. The objects' names are constrained by the naming conventions, but the content and type of the object itself varies depending on the flag and the desired effect. Most of the objects in the *Flags.area* are named text objects, so if another object type is not specified, it is assumed to be a text object. The names of the flags must match exactly; they are case sensitive, and must not contain leading or embedded blanks. The text string must contain a colon (:) followed by the flag value; unlike the names, the strings are case insensitive. Any text preceding the colon is ignored. See the description of each interaction handler for the flags specific to that interaction handler.

Layout.area

VN Description **Templates** **Key Bindings** **Echo Functions** **Modifying Active**
Restore.area **Done.area** **Cancel.area** **Flags.area**

Layout.area defines the physical layout and is mapped onto the boundary of the input object. *Layout.area* is stretched to fit, so its aspect ratio can be changed. The size, shape, and position of the components can be changed by editing the template. Typical objects within the *Layout.area* are echo areas and pickable areas, which are usually named with the suffix *.area*.

Objects that are not named using the naming convention are drawn as they appear in the template, letting you customize the template with graphical ornamentation. Named text objects frequently serve as labels for other objects in *Layout.area*. Hardware text objects that are named according to the **.text* convention are scaled down and cropped, if necessary, to fit the available space in the input object. All vector text objects are scaled automatically and are never cropped.

The following objects, contained in *Layout.area*, are common to all interaction handlers except *VNbutton*. See the description of each interaction handler for the objects specific to that interaction handler.

Restore.area lets the user restore the input variable attached to the input object to its original value by selecting this area. This area is optional. If it is not present, restoration of the interaction is signalled by pressing a "Restore" key. If no "Restore" key is defined and no *Restore.area* exists, the user cannot restore the interaction.

Restore.text contains the label for *Restore.area*. In the templates supplied with DV-Tools, this string is set to "Restore." This string can be changed in the template. For example, the "Restore" area can be labeled "Refresh."

Restore.button is a button input object that lets the user restore the input variable. If used with *Restore.area*, the button is scaled to fit *Restore.area*. To add a label, edit the label for the button input object; *Restore.text* is mutually exclusive and cannot be used with this object.

Done.area lets the user signal that the interaction is complete by selecting this area. This area is optional. If it is not present, completion of the interaction is signalled by pressing a "Done" key. If no "Done" key is defined and no *Done.area* exists, the user cannot complete the interaction.

Done.text contains the label for the *Done.area*. In the templates supplied with DV-Tools, this string is set to "Done." This string can be changed. For example, the "Done" area can be labeled "Finish" or "Exit."

Done.button is a button input object that lets the user signal that the interaction is complete. If used with *Done.area*, the button is scaled to fit *done.area*. To add a label, edit the label for the button input object; *Done.text* is mutually exclusive and cannot be used with this object.

Cancel.area lets the user abort the current interaction by selecting this area. This area is optional. If it is not present, the interaction must be aborted by pressing a "Cancel" key. If no "Cancel" key is defined and no *Cancel.area* exists, the user cannot abort the interaction.

Cancel.text contains the label for *Cancel.area*. In the templates supplied with DV-Tools, this string is set to "Cancel." This string can be changed. For example, the "Cancel" area can be labeled "Abort" or "Stop Input."

Cancel.button is a button input object that lets the user abort the current interaction. If used with *Cancel.area*, the button is scaled to fit *Cancel.area*. To add a label, edit the label for the button input object; *Cancel.text* is mutually exclusive and cannot be used with this object.

Flags.area contains objects used to customize the interaction, such as flags controlling polling and echoing options. The following flags are common to all interaction handlers.

PostType.flag is an optional flag that controls the test determining whether a pick actually intersected a pickable object. Valid text strings are:

PostType:RECT indicates that the bounding rectangle of a pickable object is used for the intersection test.

PostType:OBJECT indicates that the pickable object itself is used for the intersection test. This flag value permits greater precision in interpreting where picks are located, but reduces interaction speed. Since picking objects with the *EDGE* fill status attribute can be difficult, pickable objects should be filled or transparent.

VNtype.flag is a required flag that identifies the template type to the interaction handler. If the *VNtype.flag* matches the interaction handler type, the template is accepted. If the flag does not match, an error message is generated. If no type is specified, operation continues. VNtype flags are required for templates used by input objects edited in DV-Draw. Valid text strings are:

Flag Text String	**Interaction Type**
VNtype:VNbutton	button
VNtype:VNchecklist	object and text checklists
VNtype:VNcombiner	combiner of embedded input objects
VNtype:VNmenu	object and text menus
VNtype:VNmultiplexor	menu of embedded input objects
VNtype:VNpalette	color palette
VNtype:VNslider	sliders and scrollbars
VNtype:VNslider2D	two-dimensional slider
VNtype:VNtext	text entry
VNtype:VNtextedit	two-dimensional text editing
VNtype:VNtoggle	object and text toggles

Key Bindings and Action Types

VN Description **Templates** **Echo Functions** **Modifying Active**

VUerGetKeys, VUerPutKeys, VOitGetKeys, and VOitPutKeys support the definition and querying of current global and local key bindings associated with the following *ActionTypes* used with input objects. For the results from key actions, see the*Interpretation of Action Types* section for each input object.

Action	**ActionType Flag**
Done	DONE_KEYS
Cancel	CANCEL_KEYS
Select	SELECT_KEYS
Restore	RESTORE_KEYS
Clear	CLEAR_KEYS
Toggle Poll	TOGGLE_POLLING_KEYS

VOitKeyOrigin supports definition and querying of the origin of the key bindings as either global or local. The VUer routines handle the global key bindings, and the VOit routines handle the local key bindings. For more information on key bindings and origins, see the VUer and VO modules.

Echo Functions

VN Description **Templates** **Key Bindings** **Modifying Active**

An echo function lets the user customize the behavior of an input object at the critical points when it is being drawn, erased, updated, or when it is taking input. The echo function is attached to the input technique object using VOitPutEchoFunction, and is invoked whenever the input object is initially drawn, takes input, is updated, is redrawn, or is erased.

The echo function can be called with seven parameters. For parameter descriptions, see VOitPutEchoFunction. Echo functions can be written with any selection of these arguments, depending on what is useful in the application. A synopsis of the echo function unique to each interaction handler is included at the end of each interaction's description.

Modifying Active Input Objects

VN Description **Templates** **Key Bindings** **Echo Functions**

Input objects can be modified after they are drawn using one of several methods.

- Modify the template drawing attached to the input technique and redraw.
- Use VOitPutTemplate to associate a new template drawing with the input technique, and follow with a redraw.
- Change the text strings in a menu or multiplexor using VOitListStart followed by TdpDrawNext or TdpDrawNextObject.
- Change a variety of template objects by accessing internal structures of the input object. For more details on the internal structures, see VOinGetInternal.

Interaction Handlers: VNbutton

VN Description

VNbutton	VNmenu	VNslider	VNtextedit
VNchecklist	VNmultiplexor	VNslider2D	VNtoggle
VNcombiner	VNpalette	VNtext	

VNbutton

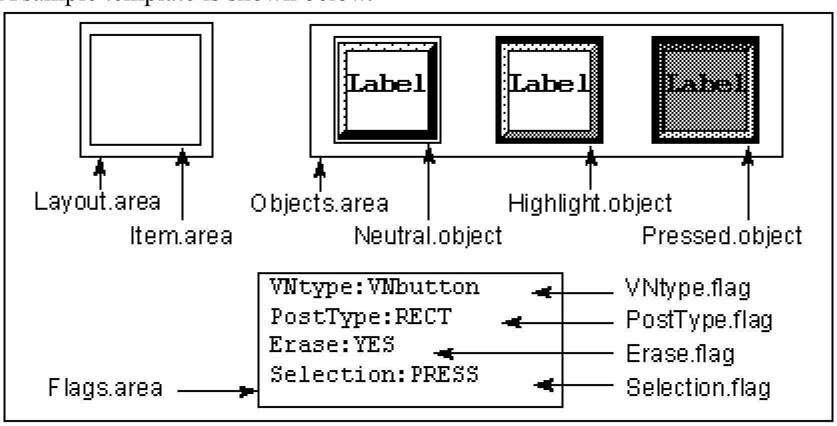
The Button interaction handler presents a single selectable item and echoes the state of selection in one of two ways: while the button is being selected (a push button) or until the button is selected again to deselect the item (a toggle button). Both kinds of buttons can also echo an highlight state when the cursor is within the button boundary. The appearance of the button in these different states is controlled by objects, usually subdrawings, in the template. Button behavior can be customized by editing the objects as well as by editing the flags. Button input objects require both button presses and releases to update properly. The text string for the button is set using VOitPutList. The associated variable, defined by VOinPutVarList, is set to 32K when the button is echoing its selection. *SELECT_KEYS* are the only key bindings used to interact with buttons. Requires a layout template.

Synopsis

```
GLOBALREF INHANDLER VNbutton;
```

Template

A sample template is shown below.



Sample Template (for a three-state push button)

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Item.area - object defining the area where the button states are displayed. If *Item.area* is not defined, the button states are mapped to the boundary of the input object, and the input object outline may not be visible.

Active.area - object defining the pickable area of the button, usually matching a particular part of the subdrawings representing the button states. If *Active.area* is not defined, *Item.area* defines the pickable area. If neither is defined, the entire button is pickable.

Objects.area:

Label.object defines the text attributes for the label. *Label.object* can be a text, vector text, or a subdrawing object that refers recursively to a text or vector text object named *Label.object*. When *Label.object* is hardware text, the label on the button is scaled to fit within the defined area.

Off_neutral.object - object representing the button when it is off and the cursor is not in the button.

Off_highlight.object - object representing the button when it is off and the cursor is in the button.

Off_pressed.object - object representing the button when it is off and a select key is being pressed in the button.

On_neutral.object - object representing the button when it is on and the cursor is not in the button.

On_highlight.object - object representing the button when it is on and the cursor is in the button.

On_pressed.object - object representing the button when it is on and the select key is being pressed.

Neutral.object - used for push buttons. Object representing the button when the cursor is not in the button. Equivalent to *Off_neutral.object*.

Highlight.object - used for push buttons. Object representing the button when the cursor is in the button. Equivalent to *Off_highlight.object*.

Pressed.object - used for push buttons. Object representing the button when a select key is being pressed in the button. Equivalent to *Off_pressed.object*.
Using these objects, you can create buttons with the following states:

A two-state push button that uses *Neutral.object* and *Pressed.object*.

A two-state push button, called a poll push button, that uses *Neutral.object* and *Highlight.object*. Selection occurs when the cursor enters the button. This kind of button is not generally recommended.

A three-state push button that uses all three objects.

A two-state toggle button that uses *Off_neutral.object* and *On_neutral.object.*

A two-state toggle button, called a poll toggle button, that uses *Off_neutral.object* and *On_neutral.object.* Selection and deselection occur when the cursor enters the button. This kind of button is not generally recommended.

A four-state toggle button that uses *Off_neutral.object*, *Off_pressed.object*, *On_neutral.object*, and *On_pressed.object.*

A four-state toggle button with highlighting that uses *Off_neutral.object*, *Off_highlight.object*, *On_neutral.object*, and *On_highlight.object.*

A six state toggle button with highlighting that uses *Off_neutral.object*, *Off_highlight.object*, *Off_pressed.object*, *On_neutral.object*, *On_highlight.object*, and *On_pressed.object*.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNbutton*.

Erase.flag - controls whether the object representing the previous button state is erased before drawing the new state. The default is *YES*. Valid text strings are:

Erase:YES - erases the previous state before drawing the new state.

Erase:NO - draws the new state without erasing the previous state, which can reduce flashing. To work effectively, the objects that define the button states should overlap exactly, so when the object for the new state is drawn, it completely covers the object for the previous state.

Selection.flag - controls whether the *INPUT_DONE* service result is generated on the key press or the key release. The default is *PRESS*. Valid text strings are:

Selection:PRESS - *INPUT_DONE* service result is generated on the key press.

Selection:RELEASE - *INPUT_DONE* service result is generated on the key release. This option works only when the *DVUSE_KEYRELEASE_IN_BUTTON* configuration variable is set to *yes* and is only applicable to these kinds of buttons: two-state push button, three-state push button, four-state toggle button without highlighting, and six-state toggle button.

Echo Function

The echo function for the button interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNbutton

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In active area	INPUT_DONE	Draw echo, update vdp
Motion (buttons with highlighting)	In active area	INPUT_ACCEPT	Update highlight
Motion (poll buttons)	In active area	INPUT_DONE	Draw echo, update vdp

Summary of Template Areas, Objects, and Flags for VNbutton

Required areas for both push and toggle buttons:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Objects.area	rectangle	boundary of objects area
Flags.area	rectangle	boundary of flags area

Required objects for push buttons (in the objects area):

Name	Object Type	Function
Neutral.object	graphic	unselected state for the button
Pressed.object	graphic	selected state for the button (except for poll button)

Required objects for toggle buttons (in the objects area):

Name	Object Type	Function
Off_neutral.object	graphic	neutral unselected state for the button
On_neutral.object	graphic	neutral selected state for the button (except for poll button)

Required flags for both push and toggle buttons (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNbutton	match to input object

Optional objects for both push and toggle buttons (in the layout area):

Name	Object Type	Function

Item.area	graphic	display area for objects representing states
Active.area	graphic	pickable area within the button

Optional objects for push buttons (in the objects area):

Name	**Object Type**	**Function**
Highlight.object	graphic	highlighted state for the button

Optional objects for toggle buttons (in the objects area):

Name	**Object Type**	**Function**
Off_highlight.object	graphic	highlighted, unselected state for the button
Off_pressed.object	graphic	pressed, unselected state for the button
On_highlight.object	graphic	highlighted, selected state for the button
On_pressed.object	graphic	pressed, selected state for the button

Optional flags for both push and toggle buttons (in the flags area):

Name	**Type**	**Content**	**Function**
Erase.flag	text	Erase:YES	erase previous state before drawing new state
		Erase:NO	draw new state over previous state
Selection.flag	text	Selection:PRESS	selection occurs with the key press
		Selection:RELEASE	selection occurs with the key release
PostType.flag	text	PostType:RECT	pick in bounding box
		PostType:OBJECT	pick on object only

Interaction Handlers: VNchecklist

VN Description

VNbutton VNmenu VNslider VNtextedit
VNchecklist VNmultiplexor VNslider2D VNtoggle
VNcombiner VNpalette VNtext

VNchecklist

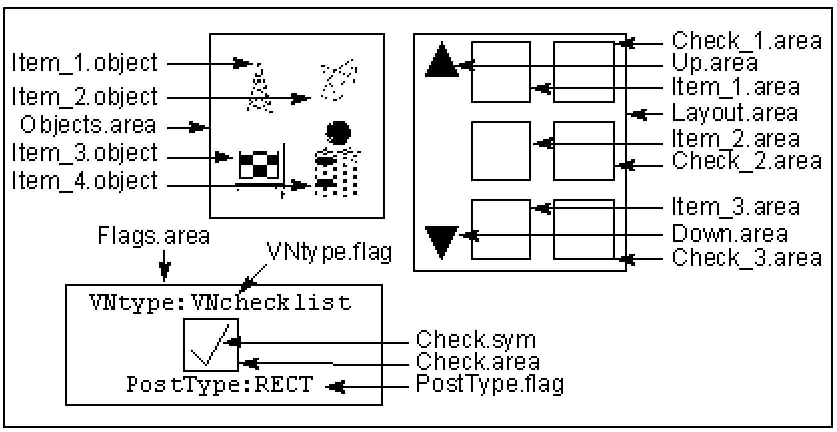
The Checklist interaction handler allows selection and deselection from a list of items by using a “Select” key. Selection is typically echoed by the appearance of a check of the programmer’s design beside the selected item, but button items use only their own echoing. Each item corresponds to a variable descriptor that has been associated with the interaction handler using VOinPutVarList. By default, (de)selecting an item sets the corresponding variable to (0.0)1.0. The select values for the items can be changed using VOitPutListValues. Text strings for the checklist interaction are set using VOitPutList. Requires a layout template.

Synopsis

```
GLOBALREF INHANDLER VNchecklist;
```

Template

A sample template is shown below.



Sample Template (for a scrolling object checklist)

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Item_%d.area - selectable areas that correspond to the list of values and item choices (text, object, or button). Note that *%d* is replaced by the number assigned to that item. For example, item areas are named *Item_1.area*, *Item_2.area*, etc. Numbers are assigned sequentially beginning at one. The item choices, either *Item_%d.text*, *Item_%d.object*, or *Item_%d.button*, appear within these selectable areas. The item choices are all *Item_%d.text*, all *Item_%d.object*, or all *Item_%d.button*, but not a mixture. The items can be placed in the objects area or the layout area. If placed in the layout area, the items maintain their position with respect to the item areas. If placed in the objects area, the items are centered when displayed in the item areas.

- *Item_%d.text* - specifies the attributes of the associated displayed label. VOitPutList can be used to set the text strings programmatically, in which case the strings in the template are ignored. If insufficient text items are supplied by VOitPutList, the excess template items are ignored. For scrolling checklists, text items may be placed in the objects area and the text attributes scroll with the labels.
- *Item_%d.object* - object in an object checklist. Objects can be either a single object or a subdrawing and must fit within the item areas. Ignores VOitPutList. For scrolling checklists, object items must be placed in the objects area. Object checklists can support labels when the items (*Item_%d.object*) are subdrawings which use *Label.area* and *Label.object* in the subdrawing views.

Item_%d.button - button item in a checklist. Buttons are scaled to fit the item areas. If the buttons support labels, VOitPutList can be used to set the text strings programmatically, in which case the button labels in the template are ignored. For scrolling checklists, button items may be placed in the objects area and the button appearance scrolls with the labels.

Check_%d.area - objects defining the areas where selection is echoed. Note that *%d* is replaced by the number assigned to that item, where the first check area is *Check_1.area*, the second is *Check_2.area*, etc. Check areas are not drawn when buttons are used as the items. Buttons provide their own echoing, so check areas are redundant and should not be used.
Scroll.object - a slider or scrollbar input object that controls the scrolling of the items being displayed. The template for this input object should not include up or down areas or buttons; they should be in the menu template.

Scroll.area - area that defines where *Scroll.object* will be drawn.

Up.area - when selected, scrolls the items being displayed up.

Up.text - text string containing the label for the *Up.area*.

Up.button - button input object for scrolling the items being displayed up. If used with *Up.area*, the button is scaled to fit *Up.area*. To add a label, edit the label for the button input object; *Up.text* is mutually exclusive and cannot be used with this object.

Down.area - when selected, scrolls the items being displayed down.

Down.text - text string containing the label for the *Down.area*.

Down.button - button input object for scrolling the items being displayed down. If used with *Down.area*, the button is scaled to fit *Down.area*. To add a label, edit the label for the button input object; *Down.text* is mutually exclusive and cannot be used with this object.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNchecklist*.

Check.sym - object or subdrawing that is used as a check mark indicating that a particular item is selected. This object should be fully enclosed in *Check.area*. Not used with button items.

Check.area - area that is used to map *Check.sym* into each *Check_%d.area* when the corresponding item is selected. Not used with button items.

CheckArea.flag - controls whether each *Check_%d.area* is drawn, but is not used with button items. The default is *DRAWN*. Valid text strings are:

CheckArea:DRAWN - each *Check_%d.area* is drawn.

CheckArea:UNDRAWN - no *Check_%d.area* is drawn.

Increment.flag - controls the number of items scrolled at a time. The default is *1*.

Additional Information

If a checklist item has been selected and there is no *Check.area* or no *Check.sym*, *Check_%d.area* is drawn in the foreground color. If there is no *Check_%d.area*, the bounding box of the item is drawn in the foreground color. In each case, when the checklist item is deselected, the area is redrawn in the background color. This does not apply to button items, which handle their own echoing.

Echo Function

The echo function for the Checklist interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *ValList,
      ADDRESS *VdpList,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNchecklist

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In item areas	INPUT_ACCEPT	Draw echo; update vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore original vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore original vdp
SELECT_KEYS	In slider or scrollbar	INPUT_ACCEPT	Scroll text block
SELECT_KEYS	In scroll areas	INPUT_ACCEPT	Scroll items
DONE_KEYS	In input object	INPUT_DONE	Update vdp
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore original vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore original vdp

Summary of Template Areas, Objects, and Flags for VNchecklist

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Optional areas:

Name	Object Type	Function
Objects.area	rectangle	boundary of objects area

Required objects (in the layout area or objects area):

Name	Object Type	Function
Item_%d.area	graphic	display areas for items (in layout area only)
Item_%d.text or	text	checklist items (text, objects, and buttons
Item_%d.object or	graphic	cannot be mixed). For button items, toggle
Item_%d.button	button input object	buttons are recommended.

Required flags (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNchecklist	match to input object

Optional objects (in the layout area):

Name	Object Type	Function
Check_%d.area	graphic	display areas for check symbols
Up.area	graphic	pickable area to scroll up through items
Up.text or	text	label for up area
Up.button	button input object	push button to scroll up through items
Down.area	graphic	pickable area to scroll down through items
Down.text or	text	label for down area
Down.button	button input object	push button to scroll down through items
Scroll.object	slider or scrollbar	input object to control scrolling
Scroll.area	rectangle	display area for slider or scrollbar
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional objects (in the flags area):

Name	Object Type	Function
Check.sym	graphic or text	check symbol graphic
Check.area	graphic	mapped into layout check areas

Optional flags (in the flags area):

Name	Type	Content	Function
CheckArea.flag	text	CheckArea:DRAWN	draw the check areas
		CheckArea:UNDRAWN	don’t draw the check areas
PostType.flag	text	PostType:RECT	pick in bounding box
		PostType:OBJECT	pick on object only
Increment.flag	text	Increment:n	the number of items to scroll by

Interaction Handlers: VNcombiner

VN Description

VNbutton VNmenu VNslider VNtextedit
VNchecklist VNmultiplexor VNslider2D VNtoggle
VNcombiner VNpalette VNtext

Introduction
Synopsis
Template
Additional Information
Echo Function
Interpretation of Action Types
Summary of Template, Areas, Objects, and Flags

VNcombiner

The Combiner interaction handler allows multiple input objects to be embedded and controlled within a single input object, allowing the construction of complex composite interaction objects. The embedded input objects, which must be fully defined before being used in the combiner, are controlled as a unit. Each embedded input object is paired with an input variable, which can be defined individually using VOinPutVarList before embedding, or as a group by calling VOinPutVarList for the combiner input object. Combiner input objects cannot contain an embedded combiner or multiplexor, but the embedded input objects can use different interaction handlers. Requires a layout template.

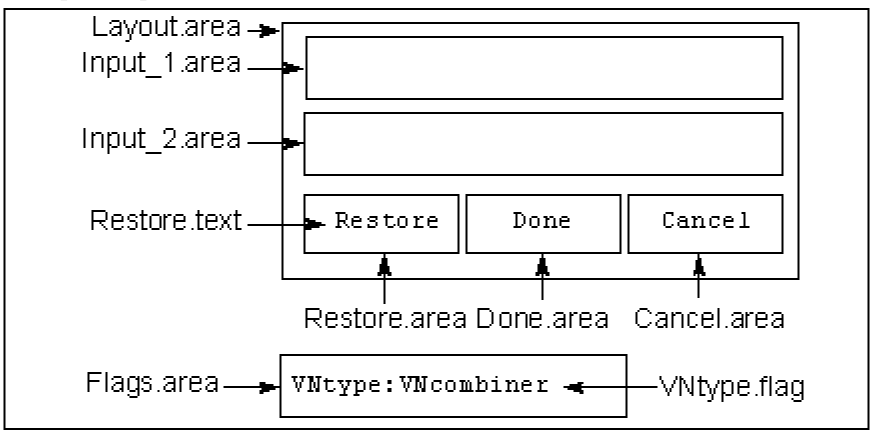
Selecting Restore, Done, and Cancel areas affects the composite input object regardless of the state of embedded input objects.

Synopsis

```
GLOBALREF INHANDLER VNcombiner;
```

Template

A sample template is shown below.



Sample Template

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Input_%d.area - areas within which the embedded input objects function. The previously defined input objects are mapped into these areas, so aspect ratio should be considered. Note that *%d* is replaced by the number assigned to that item, where the first item is *Input_1.area*, the second *Input_2.area*, etc. Numbers are assigned sequentially beginning at one.

Input_%d.text - text string used to document the combined form. It is not displayed when the interaction is run.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNcombiner*.

Additional Information

Since the combiner is treated as a unit, service result posting is done as a unit. To post a service result request for an embedded input object, use VOinGetInternal to access the embedded input objects.

Echo Function

The echo function for the combiner interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
       OBJECT Input,
       int Origin,
       int State,
       double *ValList,
       ADDRESS *VdpList,
       RECTANGLE *EchoVP,
       ADDRESS args)
```

Interpretation of Action Types for VNcombiner

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore embedded objects
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore embedded objects
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore embedded objects
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore embedded objects

Summary of Template Areas, Objects, and Flags for VNcombiner

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required objects (in layout area):

Name	Object Type	Function
Input_%d.area	rectangle	embedded input object area
Input_%d.text	text	label for input object area

Required flags (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNcombiner	match to input object

Optional objects (in the layout area):

Name	Object Type	Function
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in the flags area):

Name	Type	Content	Function
PostType.flag	text	PostType:RECT	pick in bounding box
		PostType:OBJECT	pick on pickable area only

Interaction Handlers: VNmenu

VN Description

VNbutton | **VNmenu** | VNslider | VNtextedit
VNchecklist | VNmultiplexor | VNslider2D | VNtoggle
VNcombiner | VNpalette | VNtext

Introduction
Synopsis
Template
Additional Information
Echo Function
Interpretation of Action Types
Summary of Template, Areas, Objects, and Flags

VNmenu

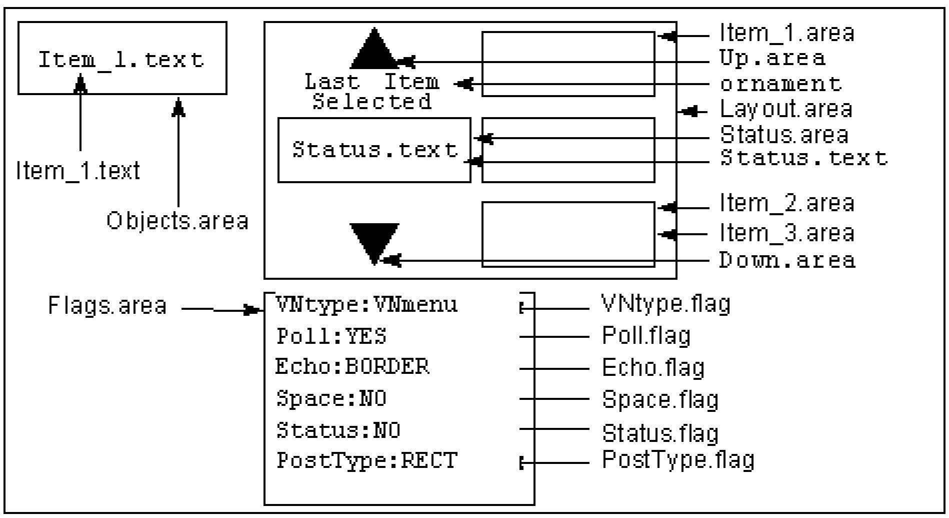
The Menu interaction handler gets an item selection from the user and echoes the selection within the specified area. Text menu items are echoed by toggling the fill of the item area or the thickness of the bounding box. Object menus are echoed only by drawing the item area contained in the template. Button items are echoed using the echoing inherent in the button. The associated variable, defined by VOinPutVarList, is set to whatever value corresponds to the menu entry that is currently echoed, defined by VOitPutListValues. If VOitPutListValues is not called, the variable is set to the index in the item's name. Text strings for the menu interaction are set using VOitPutList. A template is optional for text menu interactions and required for object menu interactions.

Synopsis

```
GLOBALREF INHANDLER VNmenu;
```

Template

A sample template is shown below.



Sample Template (for a scrolling text menu)

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Item_%d.area - selectable areas that correspond to the list of values and item choices (label, object, or button). Note that *%d* is replaced by the number assigned to that item. For example, item areas are named *Item_1.area*, *Item_2.area*, etc. Numbers are assigned sequentially beginning at one. The item choices, either *Item_%d.text*, *Item_%d.object*, or *Item_%d.button*, appear within these selectable areas. The item choices are all *Item_%d.text*, all *Item_%d.object*, or all *Item_%d.button*, but not a mixture. The items can be placed in the objects area or the layout area. If placed in the layout area, the items maintain their position with respect to the item areas. If placed in the objects area, the items are centered when displayed in the item areas.

Item_%d.text - specifies the attributes of the associated displayed label. *VOitPutList* can be used to set the text strings programmatically, in which case the strings in the template are ignored. If insufficient text items are supplied by *VOitPutList*, the excess template items are ignored. For scrolling menus, text items may be placed in the objects area and the text attributes scroll with the labels.

Item_%d.object - object in an object menu. Objects can be either a single object or a subdrawing and must fit within the item areas. Ignores *VOitPutList*. For scrolling menus, object items may be placed in the objects area. Object menus can support labels when the items (*Item_%d.object*) are subdrawings which use *Label.area* and *Label.object* in the subdrawing views.

Item_%d.button - button item in a menu. Buttons are scaled to fit the item areas. If the buttons support labels, *VOitPutList* can be used to set the text strings programmatically, in which case the button labels in the template are ignored. For scrolling menus, button items may be placed in the objects area.

Status.area - area for displaying the last selected item. It is most useful when the menu has scrolling, since the last selected item may be scrolled from view.

Status.text - text or vector text object that specifies the attributes for displaying the label of the last selected item. Required for displaying the last selected item when using button items; highly recommended when the text items are placed in the layout area instead of the objects area; not useful for object menus.

Scroll.object - a slider or scrollbar input object that controls the scrolling of the items being displayed.

Scroll.area - area that defines where *Scroll.object* will be drawn.

Up.area - when selected, scrolls the items being displayed up.

Up.text - text string containing the label for the *Up.area*.

Up.button - button input object for scrolling the items being displayed up. If used with *Up.area*, the button is scaled to fit *Up.area*. To add a label, edit the label for the button input object; *Up.text* is mutually exclusive and cannot be used with this object.

Down.area - when selected, scrolls the items being displayed down.

Down.text - text string containing the label for the *Down.area*.

Down.button - button input object for scrolling the items being displayed down. If used with *Down.area*, the button is scaled to fit *Down.area*. To add a label, edit the label for the button input object; *Down.text* is mutually exclusive and cannot be used with this object.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNmenu*.

Echo.flag - defines the type of menu echoing for text or object items. Button items use their own echoing. The default is *BORDER*. Valid text strings are:

Echo:BORDER - toggles the line thickness attribute of *Item_%d.area* between thick and thin. If *Item_%d.area* is drawn with a thick line it is highlighted with a thin line and vice versa. In object menus, the objects are drawn without borders; the border is drawn only to highlight the chosen object.

Echo:FILL - toggles the fill of *Item_%d.area* between filled and unfilled. The area is drawn highlighted until another item is pointed to. The highlight fill color is the fill color of the bounding box of the menu item. This applies only to text menus.

Echo:NONE - menu items are never highlighted. This option is particularly useful when using an echo function to draw your own echoes or using immediate action menus where the menu is erased after a selection is made.

Poll.flag - controls whether the menu pays attention to non-pick cursor position within menu items. Button items use their own polling. The default is *YES*. Valid text strings are:

Poll:YES - menu updates whenever the cursor is positioned within a selectable area, regardless of whether or not a pick occurs.

Poll:NO - menu updates only when a “Done” or “Select” key is pressed. You must assign both *DONE_KEYS* and *SELECT_KEYS* bindings.

Space.flag - determines whether highlighting of the menu item is deactivated when the cursor is not on the *Item_ %d.area* or the bounding box of the menu item object. The default is *NO*. This flag is only effective when the *Poll.flag* is *YES*. Valid text strings are:

Space:NO - last menu item remains highlighted when the cursor is not in an item area.

Space:YES - whenever the cursor is in screen space other than a menu item no item is highlighted. The *YES* option requires more overhead and does not provide current status.

Status.flag - determines whether the value of the menu's control variable is used to highlight a menu choice when the menu is initially drawn or when the value is reset below the lowest value associated with an item. The default is *NO*. Valid text strings are:

Status:NO - no item is highlighted when the menu is initially drawn and the variable value is initially set to less than the minimum value associated with the menu. Whenever the variable value is reset to a value below the minimum value associated with the menu, no item is highlighted.

Status:YES - current value of the variable is used as an item index. The variable value is mapped to the nearest value of an item.

Increment.flag - controls the number of items scrolled at a time. The default is *1*.

Additional Information

When no template is used, the default menu is restricted to text items and is internally generated by the menu interaction handler. The size of the menu is determined by the number and length of the text strings set using VOitPutList. The upper left corner of the menu is drawn as close to the cursor location as possible, but is constrained to fit into the input object's drawing area. If the menu is too large to fit into the input object's drawing area, it is cropped to fit. The text is drawn using hardware text, size 2. If insufficient text items are supplied, the excess template items are ignored.

Echo Function

The echo function for the menu interaction handler is set up by a call to *VOitPutEchoFunction*. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNmenu

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In item areas	INPUT_DONE	Update highlight and vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore original vdp

SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore original vdp
SELECT_KEYS	In slider or scrollbar	INPUT_ACCEPT	Scroll text block
SELECT_KEYS	In scroll areas	INPUT_ACCEPT	Scroll items
DONE_KEYS	In input object	INPUT_DONE	Update vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore original vdp
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore original vdp
Motion (`POLL:YES`)	In item areas	INPUT_ACCEPT	Update highlight and vdp
Motion (`SPACE:YES`)	In input object	INPUT_ACCEPT	No highlight if outside item area
Motion (`SPACE:NO`)	In input object	INPUT_ACCEPT	Last highlight remains in menu

Summary of Template Areas, Objects, and Flags for VNmenu

Required areas:

Name	**Object Type**	**Function**
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Optional areas:

Name	**Object Type**	**Function**
Objects.area	rectangle	boundary of objects area

Required objects (in the layout area or objects area):

Name	**Object Type**	**Function**
Item_%d.area	graphic	display areas for items (in layout area only)
Item_%d.text or	text	menu items (text, objects, and buttons
Item_%d.object or	graphic	cannot be mixed). For button items, toggle
Item_%d.button	button input object	buttons are recommended.

Required flags (in the flags area):

Name	**Type**	**Content**	**Function**
VNtype.flag	text	VNtype:VNmenu	match to input object

Optional objects (in the layout area):

Name	**Object Type**	**Function**
Up.area	graphic	pickable area to scroll up through items
Up.text or	text	label for up area
Up.button	button input object	push button to scroll up through items
Down.area	graphic	pickable area to scroll down through items
Down.text or	text	label for down area
Down.button	button input object	push button to scroll down through items
Scroll.object	slider or scrollbar	input object to control scrolling
Scroll.area	rectangle	display area for slider or scrollbar
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel
Status.area	graphic	display area for last selected item

Status.text	text	label for last selected text item

Optional flags (in the flags area):

Name	**Type**	**Content**	**Function**
Echo.flag	text	Echo:BORDER	border of pickable area echoes selection
		Echo:FILL	echo toggles pickable area between fill and edge
		Echo:NONE	no echoing
Poll.flag	text	Poll:YES	polls cursor position only for selection
		Poll:NO	polls picks only for selection
Space.flag	text	Space:NO	item stays highlighted until another is selected
		Space:YES	item highlighted only when cursor in item areas
Status.flag	text	Status:NO	no item highlighted when menu is initialized or current value of variable is less any item value
		Status:YES	item with value nearest current value of variable is highlighted
PostType.flag	text	PostType:RECT	pick in bounding box of area
		PostType:OBJECT	pick on Item_%d.object only: object menus only
Increment.flag	text	Increment:n	the number of items to scroll by

Interaction Handlers: VNmultiplexor

VN Description

VNbutton	VNmenu	VNslider	VNtextedit
VNchecklist	**VNmultiplexor**	VNslider2D	VNtoggle
VNcombiner	VNpalette	VNtext	

Introduction
Synopsis
Template
Echo Function
Interpretation of Action Types
Summary of Template, Areas, Objects, and Flags

VNmultiplexor

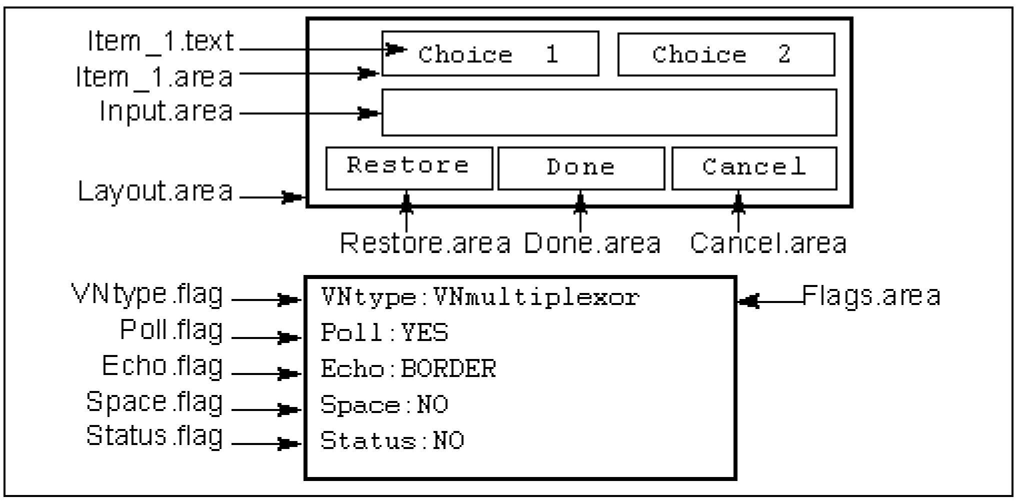
The Multiplexor interaction handler is a menu in which each selection activates a different input object in the shared input area. Text items are echoed by toggling the fill of the item area or the thickness of the bounding box. Object items are echoed only by drawing the item area contained in the template. Button items are echoed using the echoing inherent in the button. The selections are labeled using the names of the variable descriptors, defined by *VOinPutVarList*, associated with the embedded input objects, defined by *VOitPutList*. The variables associated with each input object can be assigned individually by using *VOinPutVarList*, or as a group by calling *VOinPutVarList* for the multiplexor input object. A multiplexor input object cannot contain an embedded combiner or multiplexor, but the embedded input objects can use different interaction handlers. Binding the *SELECT_KEYS* and *DONE_KEYS* to the same list of keys is recommended. Requires a layout template.

Synopsis

```
GLOBALREF INHANDLER VNmultiplexor;
```

Template

A sample template is shown below.



Sample Template

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Item_%d.area - selectable areas that correspond to the input objects embedded in the multiplexor. Note that *%d* is replaced by the number assigned to that item. For example, item areas are named *Item_1.area*, *Item_2.area*, etc. Numbers are assigned sequentially beginning at one. The item choices, either *Item_%d.text*, *Item_%d.object*, or *Item_%d.button*, appear within these selectable areas. The item choices are all *Item_%d.text*, all *Item_%d.object*, or all *Item_%d.button*, but not a mixture. The items can be placed in the objects area or the layout area. If placed in the layout area, the items maintain their position with respect to the item areas. If placed in the objects area, the items are centered when displayed in the item areas.

Item_%d.text - specifies the attributes of the associated displayed label. The names of the variable descriptors associated with the embedded input objects serve as the labels. *VOitPutList* can be used to set the text strings programmatically, in which case the strings in the template are ignored. If insufficient text items are supplied by *VOitPutList*, the excess template items are ignored. For scrolling multiplexors, text items may be placed in the objects area and the text attributes scroll with the labels.

Item_%d.object - object identifying a choice. An object can be either a single object or a subdrawing and must fit within the item areas. For scrolling multiplexors, object items may be placed in the objects area. Object checklists can support labels when the items (*Item_%d.object*) are subdrawings which use *Label.area* and *Label.object* in the subdrawing views.

Item_%d.button - button identifying a choice. Buttons are scaled to fit the item areas. If the buttons support labels, *VOitPutList* can be used to set the text strings programmatically, in which case the button labels in the template are ignored. For scrolling multiplexors, button items may be placed in the objects area.

Input.area - area shared by the embedded input objects associated with the selectable areas. As each area is selected, the corresponding input object is activated within the shared input area. The templates of the embedded input objects should be the same or have similar aspect ratios. Otherwise, the embedded input objects appear distorted.

Input_area.text - text string used to document the shared input area. It is not displayed when the interaction is run.

Scroll.object - a slider or scrollbar input object that controls the scrolling of the items being displayed.

Scroll.area - area that defines where *Scroll.object* will be drawn.

Up.area - when selected, scrolls the items being displayed up.

Up.text - text string containing the label for the *Up.area*.

Up.button - button input object for scrolling the items being displayed up. If used with *Up.area*, the button is scaled to fit *Up.area*. To add a label, edit the label for the button input object; *Up.text* is mutually exclusive and cannot be used with this object.

Down.area - when selected, scrolls the items being displayed down.

Down.text - text string containing the label for the *Down.area*.

Down.button - button input object for scrolling the items being displayed down. If used with *Down.area*, the button is scaled to fit *Down.area*. To add a label, edit the label for the button input object; *Down.text* is mutually exclusive and cannot be used with this object.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNmultiplexor*.

Poll.flag - controls whether the multiplexor pays attention to non-pick cursor position within text or object items. Button items use their own polling. The default is *YES*. Valid text strings are:

Poll:YES - updates whenever the cursor is positioned within a selectable area regardless of whether or not a pick occurs.

Poll:NO - no updating occurs unless a “Select” or “Done” pick occurs.

Echo.flag - determines the type of item echoing used for a multiplexor with text or object items. Button items use their own echoing. The default is *BORDER*. Valid text strings are:

Echo:BORDER - echoes the currently selected item by toggling the line thickness attribute of the *Item_%d.area* between thick and thin. If the *Item_%d.area* is drawn with a thick line, it is highlighted with a thin line and vice versa. In multiplexors using *Item_%d.object*, the objects are drawn without borders and the border is drawn to highlight the object.

Echo:FILL - echoes the currently selected item by toggling the fill of the item area. The highlight fill color is the fill color of the item’s bounding box. This applies only to multiplexors using *Item_%d.text*.

Echo:NONE - items are never highlighted. This option is particularly useful when using an echo function to draw your own echoes or using immediate action multiplexors where the multiplexor is immediately erased after a selection is made.

Space.flag - determines whether highlighting of the menu item is deactivated when the cursor is not on the *Item_%d.area* or the bounding box of the menu item object. The default is *YES*. This flag is only effective when the *Poll.flag* is *YES*. Valid text strings are:

Space:NO - last item remains highlighted when the cursor is not in an item area.

Space:YES - whenever the cursor is in screen space other than an item no item is highlighted. The *YES* option requires more overhead and does not provide current status.

Status.flag - determines whether the initial value of the multiplexor's control variable is used to highlight a choice when the multiplexor is initially drawn. The default is *NO*. Valid text strings are:

Status:NO - no item is highlighted when the multiplexor is initially drawn and the variable value is initially set to less than the minimum value associated with the multiplexor. Whenever the variable value is reset to a value below the minimum value associated with the multiplexor, no item is highlighted.

Status:YES - current value of the variable is used as an item index. The variable value is mapped to the nearest value of an item.

Increment.flag - controls the number of items scrolled at a time. The default is *1*.

Echo Function

The echo function for the multiplexor interaction handler is set up by a call to *VOitPutEchoFunction*. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *ValList,
      ADDRESS *VdpList,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNmultiplexor

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	**Locator Position**	**Service Result**	**Services**
SELECT_KEYS	In item areas	INPUT_ACCEPT	Update vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore to original vdp and embedded obj vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore to original vdp and embedded obj vdp
SELECT_KEYS	In slider or scrollbar	INPUT_ACCEPT	Scroll text block
SELECT_KEYS	In scroll areas	INPUT_ACCEPT	Scroll items
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore to original vdp and

			embedded obj vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore to original vdp and embedded obj vdp
Motion (`POLL:YES`)	In item areas	INPUT_ACCEPT	Update highlight and vdp
Motion (`SPACE:YES`)	In input object and not in item area	INPUT_ACCEPT	No highlight in menu
Motion (`SPACE:NO`)	In input object and not in item area	INPUT_ACCEPT	Last highlight remains in menu

Summary of Template Areas, Objects, and Flags for VNmultiplexor

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Optional areas:

Name	Object Type	Function
Objects.area	rectangle	boundary of objects area

Required objects (in the layout area or objects area):

Name	Object Type	Function
Item_%d.area	graphic	display areas for items (in layout area only)
Item_%d.text or	text	menu items (text, objects, and buttons
Item_%d.object or	graphic	cannot be mixed). For button items, toggle
Item_%d.button	button input object	buttons are recommended.
Input.area	graphic	shared input object area (in layout area only)

Required flags (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNmenu	match to input object

Optional objects (in the layout area):

Name	Object Type	Function
Input_area.text	text	label for shared input object area
Up.area	graphic	pickable area to scroll up through items
Up.text or	text	label for up area
Up.button	button input object	push button to scroll up through items
Down.area	graphic	pickable area to scroll down through items
Down.text or	text	label for down area
Down.button	button input object	push button to scroll down through items
Scroll.object	slider or scrollbar	input object to control scrolling
Scroll.area	rectangle	display area for slider or scrollbar
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in the flags area):

Name	Type	Content	Function
Echo.flag	text	Echo:BORDER	border of pickable area echoes selection
		Echo:FILL	echo toggles pickable area between fill and edge
		Echo:NONE	no echoing
Poll.flag	text	Poll:YES	polls cursor position only for selection
		Poll:NO	polls picks only for selection
Space.flag	text	Space:NO	item stays highlighted until another is selected
		Space:YES	item highlighted only when cursor in item areas
Status.flag	text	Status:NO	no item highlighted when menu is initialized or current value of variable is less any item value
		Status:YES	item with value nearest current value of variable is highlighted
PostType.flag	text	PostType:RECT	pick in bounding box of area
		PostType:OBJECT	pick on Item_%d.object only: object menus only
Increment.flag	text	Increment:n	the number of items to scroll by

Interaction Handlers: VNpalette

VN Description

VNbutton VNmenu VNslider VNtextedit
VNchecklist VNmultiplexor VNslider2D VNtoggle
VNcombiner **VNpalette** VNtext

VNpalette

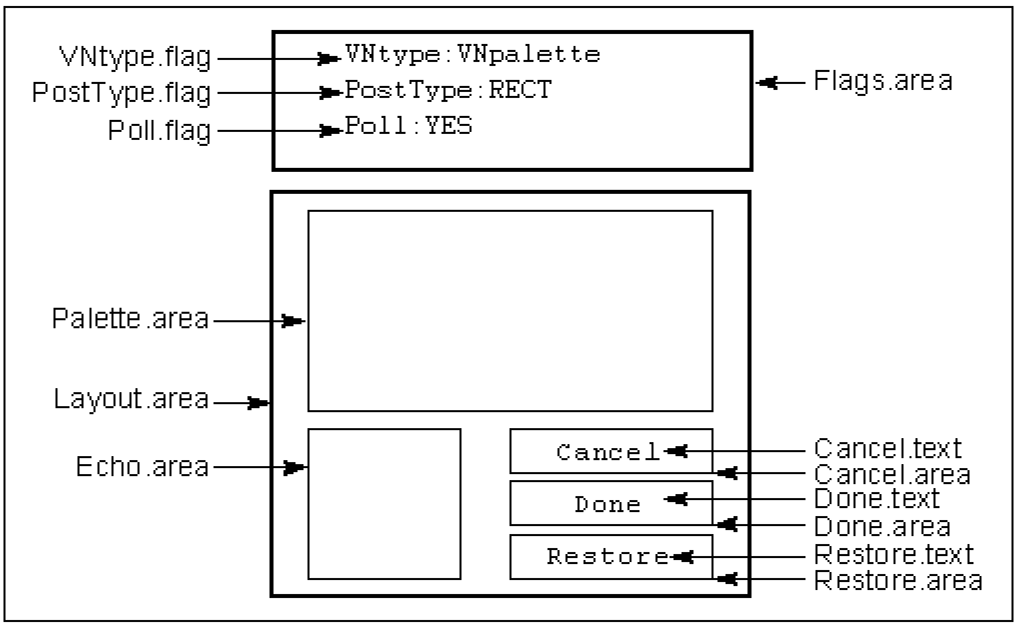
The Palette interaction handler gets a color selection from the user and echoes it in *Echo.area*. The associated variable, defined by VOinPutVarList, is set to the index of the selected color. A template is optional. When no template is used, the palette fills the entire input object, no echoing is done, and the variable updates when a “Select” key is pressed.

Synopsis

```
GLOBALREF INHANDLER VNpalette;
```

Template

A sample template is shown below.



Sample Template

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Palette.area - sensitive area in the input template in which the color selection takes place. The *Palette.area* is used to display a color palette from which a single color can be chosen. If no *Palette.area* is specified, the palette fills the entire layout area.

Palette.text - labels the palette area for use in DV-Draw. It is not displayed when the interaction is run. This label is optional.

Echo.area - area in which the currently selected palette color is echoed. The echo area can be any DataViews object.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNpalette*.

Poll.flag - controls whether the palette acknowledges non-pick cursor position within palette items. The default is

YES when a template is used. Valid text strings are:

Poll:YES - updates whenever the cursor is positioned within a palette item.

Poll:NO - updates only when a "Done" or "Select" key is pressed. You must assign both *DONE_KEYS* and *SELECT_KEYS* bindings.

Echo Function

The echo function for the palette interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNpalette

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	**Locator Position**	**Service Result**	**Services**
SELECT_KEYS (`POLL:YES`)	In `Palette.area`	INPUT_DONE	Update vdp
None (`POLL:YES`)	In `Palette.area`	INPUT_ACCEPT	Update vdp
SELECT_KEYS (`POLL:NO`)	In `Palette.area`	INPUT_DONE	Update vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore to original vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore to original vdp
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore to original vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore to original vdp

Summary of Template Areas, Objects, and Flags for VNpalette

Required areas:

Name	**Object Type**	**Function**
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required flags (in the flags area):

Name	**Type**	**Content**	**Function**
VNtype.flag	text	VNtype:VNpalette	match to input object

Optional objects (in the layout area):

Name	Object Type	Function
Palette.area	rectangle	boundary of palette area
Palette.text	text	label for palette area (in template only)
Echo.area	graphic	boundary of echo area
Echo.text	text	label for echo area (in template only)
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in the flags area):

Name	Type	Content	Function
Poll.flag	text	Poll:YES	polls cursor position only for selection
		Poll:NO	polls picks only for selection
PostType.flag	text	PostType:RECT	pick in bounding box
		PostType:OBJECT	pick on pickable area only

Interaction Handlers: VNslider

VN Description

VNbutton	VNmenu	**VNslider**	VNtextedit
VNchecklist	VNmultiplexor	VNslider2D	VNtoggle
VNcombiner	VNpalette	VNtext	

VNslider

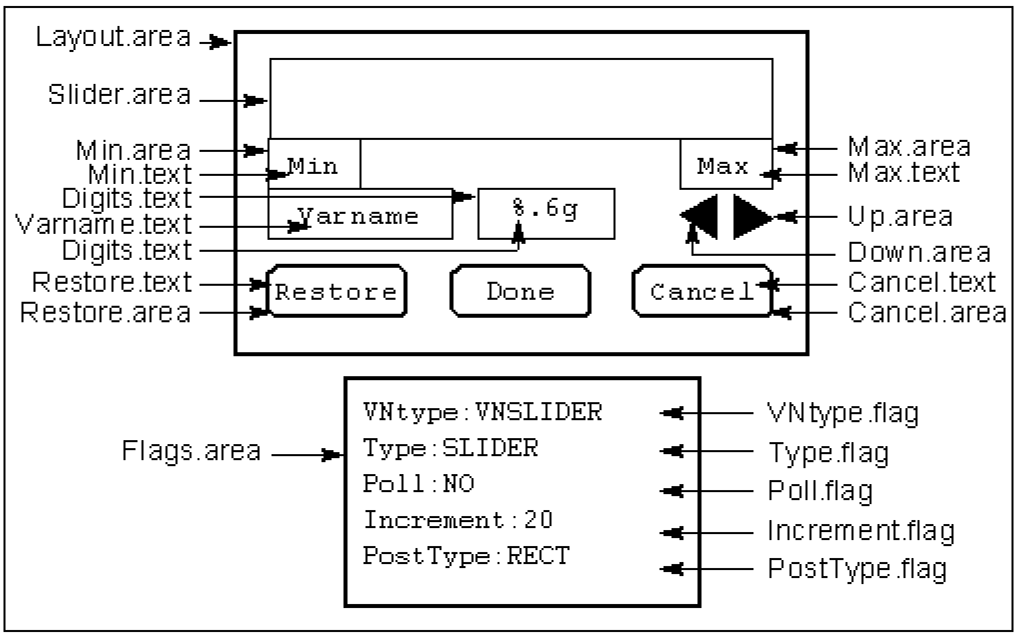
The Slider interaction handler acts as a sliding valuator to get a value from the user. The current value echoes as the position of a slider or a scrollbar along its track. The associated variable, defined by VOinPutVarList, is set to the value, which is within the range set for the variable by *VPvd_irange* or *VPvd_drange*. A template is optional for sliders but required for scrollbars.

Synopsis

```
GLOBALREF INHANDLER VNslider;
```

Template

A sample template is shown below.



Sample Template (for a slider)

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Slider.area - sensitive area in the Input Template in which the slider action takes place. The *Slider.area* is filled along the major axis in the input object's foreground color. The portion of the slider between the current value and the maximum value is filled with the input object's background color. This area is required if the slider is to echo the current value.

Slider.text - labels the slider area for use in DV-Draw. It is not displayed when the interaction is run. This label is optional.

Min.area - an optional area for displaying the minimum value per *Min.text* below. The area is not drawn in the input object.

Min.text - controls the position and appearance of the minimum value of the slider. This string is optional. It is replaced by the actual minimum value associated with the variable descriptor attached to the input object.

Max.area - an optional area for displaying the maximum value per *Max.text* below. The area is not drawn in the input object.

Max.text - controls the position and appearance of the maximum value of the slider. This string is optional. It is replaced by the actual maximum value associated with the variable descriptor attached to the input object.

Varname.area - an optional area for displaying the variable name per *Varname.text* below. The area is not drawn in the input object.

Varname.text - controls the position and appearance of the input variable name. This string is optional. The name is the name field of the variable descriptor, which is set using *VPvdvarname*.

Digits.area - displays the digital value of the input variable. This area is optional, but must appear if *Digits.text* exists.

Digits.text - controls the position and appearance of the digital display of the input variable. *Digits.text* must be a valid C format string; for example, *%6.3f*. This string is optional but must appear if *Digits.area* exists.

Up.area - when selected, increments the current value of the input variable by a percentage of the range of the variable descriptor controlling the input variable. See *Increment.flag*.

Up.text - text string containing the label for the *Up.area*.

Up.button - button input object for incrementing the current value. If used with *Up.area*, the button is scaled to fit *Up.area*. To add a label, edit the label for the button input object; *Up.text* is mutually exclusive and cannot be used with this object.

Down.area - when selected, decrements the current value of the input variable by a percentage of the range of the variable descriptor controlling the input variable. See *Increment.flag*.

Down.text - text string containing the label for the *Down.area*.

Down.button - button input object for decrementing the current value. If used with *Down.area*, the button is scaled to fit *Down.area*. To add a label, edit the label for the button input object; *Down.text* is mutually exclusive and cannot be used with this object.

Flags.area:

VNtype.flag - the correct text string for both sliders and scrollbars is *VNtype:VNslider*.

Poll.flag - controls whether the slider or scrollbar pays attention to non-pick cursor position within *Slider.area*. The default is *YES*. Valid text strings are:

Poll:YES - updates whenever the cursor is positioned within the slider regardless of whether or not a pick is detected.

Poll:NO - updates only when a "Select" key is pressed.

Increment.flag - controls the percentage of the variable range by which the slider position changes when the *Up.area* and *Down.area* objects are picked. The contents of the text string after the colon (:) are interpreted as a float percentage of the variable range.

Direction.flag - determines the direction of slider movement. If no flag is specified, the default is movement along the longer dimension of the slider. Valid text strings are:

Direction:Horizontal - slider moves right and left.

Direction:Vertical - slider moves up and down.

Type.flag - selects a *SCROLLBAR* or *SLIDER* representation when drawing the slider. The default is *SLIDER*. Valid text strings are:

Type:SLIDER - draws valuator using slider representation.

Type:SCROLLBAR - pays attention to *Anchor.flag* and *PageSize.flag*.

Anchor.flag - determines where the cursor is anchored to the scrollbar page. Valid text strings are:

Anchor:Middle - places the scrollbar page centered around the last cursor position used as an update.

Anchor:Start - depends on the orientation. In a horizontal scrollbar, the page is to the right of the current position. In a vertical scrollbar, the page is above the current position.

Anchor:End - depends on the orientation. In a horizontal scrollbar, the page is to the left of the current position. In a vertical scrollbar, the page is below the current position.

PageSize.flag - controls the percentage of the variable range used as the scrollbar page size. The text string after the colon (:) is interpreted as a float percentage of the variable range. If no *PageSize.flag* is specified, a scrolling line appears in place of the scrollbar.

Echo Function

The echo function for the slider interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNslider

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS (`POLL:YES`)	In `Slider.area`	INPUT_DONE	Update slider and vdp
SELECT_KEYS (`POLL:NO`)	In `Slider.area`	INPUT_DONE	Update slider and vdp
SELECT_KEYS	In Done.area	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore original vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore original vdp
SELECT_KEYS	In increment areas	INPUT_ACCEPT	Update slider and vdp
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore original vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore original vdp
Motion (`POLL:YES`)	In `Slider.area`	INPUT_ACCEPT	Update slider and vdp

Summary of Template Areas, Objects, and Flags for VNslider

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required object (in layout area):

Name	Object Type	Function
Slider.area	rectangle	pickable area and track for movement

Required flags (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNslider	match to input object

Additional required flags for scrollbars (in the flags area):

Name	Type	Content	Function
Type.flag	text	Type:SCROLLBAR	scrollbar input object
		Type:SLIDER	slide input object
Anchor.flag	text	Anchor:Middle	scrollbar centered on current value
		Anchor:Start	scrollbar to the right or above current value
		Anchor:End	scrollbar to the left or below current value

Optional objects (in the layout area):

Name	Object Type	Function
Slider.text	text	labels Slider.area (in template only)
Varname.area	graphic	area for variable label
Varname.text	text	controls style and position of the variable name
Min.area	graphic	area for minimum label
Min.text	text	controls style and position of the minimum value setting
Max.area	graphic	area for maximum label
Max.text	text	controls style and position of the maximum value setting
Digits.area	graphic	controls position of current value reading
Digits.text	text	controls style of current value reading
Up.area	graphic	pickable area to increment the value up
Up.text or	text	label for up area
Up.button	button input object	push button to increment the value up
Down.area	graphic	pickable area to increment the value down
Down.text or	text	label for down area
Down.button	button input object	push button to increment the value down
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in the flags area):

Name	Type	Content	Function
Poll.flag	text	Poll:YES	polls cursor position only for selection in slider.area
		Poll:NO	polls picks only for selection in slider.area
Increment.flag	text	Increment:%	sets change increment as a percent of range

Direction.flag	text	Direction:Horizontal	slider moves horizontally
		Direction:Vertical	slider moves vertically
PostType.flag	text	PostType:RECT	pick in bounding box of area
		PostType:OBJECT	pick on area only
PageSize.flag	text	PageSize:%	scrollbar size as percentage of slider dimension (for scrollbars only)

Interaction Handlers: VNslider2D

VN Description

VNbutton	VNmenu	VNslider	VNtextedit
VNchecklist	VNmultiplexor	**VNslider2D**	VNtoggle
VNcombiner	VNpalette	VNtext	

Introduction
Synopsis
Template
Echo Function
Interpretation of Action Types
Summary of Template, Areas, Objects, and Flags

VNslider2D

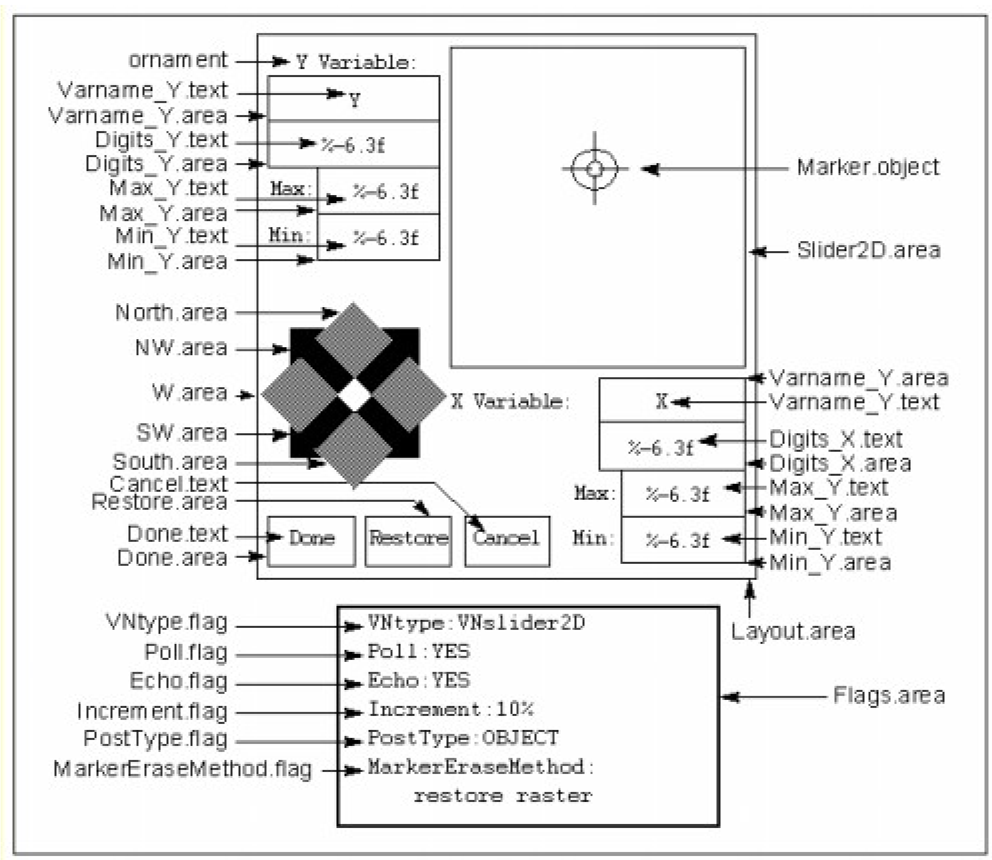
The Slider2D interaction handler acts as a two-dimensional valuator to get values from the user. It echoes the current value as the position of a marker within the rectangular slider plane. The associated variables, defined by VOinPutVarList, are set to the x and y values, which are within the range set for the variables by *VPvd_irange* or *VPvd_drange*. A template is optional.

Synopsis

```
GLOBALREF INHANDLER VNslider2D;
```

Template

A sample template is shown below.



The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Slider2D.area - sensitive area in which the slider action takes place. This area is required if the slider is to echo the current values.

Slider2D.text - labels the slider area for use in DV-Draw. It is not displayed when the interaction is run. This label is optional.

Max_X.area, *Min_X.area*, *Max_Y.area*, *Min_Y.area* - display the areas for the slider's maximum and minimum dimension labels. These strings are optional, but must appear if the corresponding *Max_X.text*, *Min_X.text*, *Max_Y.text*, or *Min_Y.text* exists.

Max_X.text, *Min_X.text*, *Max_Y.text*, *Min_Y.text* - control the position and appearance of the maximum and minimum values of the slider. These strings are optional, but must appear if the corresponding *Max_X.area*, *Min_X.area*, *Max_Y.area*, or *Min_Y.area* exists. They are replaced at run-time by the maximum and minimum values associated with the variable descriptors attached to the input object.

Varname_X.area, *Varname_Y.area* - display the areas for the dimension labels. These strings are optional, but must appear if the corresponding *Varname_X.text* or *Varname_Y.text* exists.

Varname_X.text, *Varname_Y.text* - control the position and appearance of the input variable names. These strings are optional, but must appear if the corresponding *Varname_X.area* or *Varname_Y.area* exists. They are replaced at run-time by the name fields of the variable descriptors, set using *VPvdvarname*.

Digits_X.area, *Digits_Y.area* - display the digital value of the input variable. These areas are optional, but they must appear if the corresponding *Digits_X.text* or *Digits_Y.text* exists.

Digits_X.text, *Digits_Y.text* - control the position and appearance of the digital displays of the input variable. *Digits.text* must be a valid C format string; for example, *%-6.3f*. These strings are optional, but must appear if the corresponding *Digits_X.area* or *Digits_Y.area* exists.

North.area, *NE.area*, *NW.area*, *South.area*, *SE.area*, *SW.area*, *East.area*, *West.area* - when selected, increments or decrements the current values of the corresponding input variables by a percentage, set by *Increment.flag*, of the range of the variable descriptor controlling the input variable. The movements are: North = up, South = down, East = right, West = left and NE, NW, SE, SW correspond to the diagonal directions.

North.text, *NE.text*, *NW.text*, *South.text*, *SE.text*, *SW.text*, *East.text*, *West.text* - text strings containing the labels for the corresponding increment areas.

North.button, *NE.button*, *NW.button*, *South.button*, *SE.button*, *SW.button*, *East.button*, *West.button* - button input objects for incrementing or decrementing the current values. If used with the corresponding areas, the buttons are scaled to fit the areas. To add a label to a button, edit the label for the button input object; the corresponding text labels (**.text*) are mutually exclusive and cannot be used with the button input objects.

Marker.object is a custom marker that can be a primitive object or a subdrawing. If it is a subdrawing, the anchor for positioning is the center of the view. Centering and scaling should be made in the view before loading as a subdrawing. If the marker is a primitive object, the move point serves as the anchor. If an additional echo marker is specified using *EchoMethod.flag*, that marker appears superimposed on *Marker.object*.

Flags.area:

VNtype.flag - the correct text string is *VNtype:VNslider2D*.

Poll.flag - controls whether the slider pays attention to non-pick cursor positions within *Slider2D.area*. The default value is *YES*. Valid text strings are:

Poll:YES - updates whenever the cursor is positioned within the slider regardless of whether a pick is detected.

Poll:NO - updates only when a “Select” key is pressed.

The *ActionType* flag, *TOGGLE_POLLING_KEYS*, supports toggling of *Poll.flag* during interaction. When *Poll:Yes* or no *Poll.flag* is set, this action key lets the user use the cursor to move the marker in *Slider2D.area*, change the polling using a “Toggle Poll” action key, and move out of *Slider2D.area* without affecting the marker position or current x and y values. Toggle polling is currently valid only for the Slider2D. At least one key must be defined as a “Select” key for toggling to be effective. See the *Key Bindings and Action Types* at the beginning of this chapter for more information.

Increment.flag - controls the percentage of the variable range by which the slider position changes when the *North.area*, *NE.area*, *NW.area*, *South.area*, *SE.area*, *SW.area*, *East.area*, *West.area* objects are picked. The contents of the text string after the colon (:) are interpreted as a float percentage of the variable range. The default is 5%.

IncrementX.flag and *IncrementY.flag* are used to control axis increments separately.

Echo.flag - specifies whether a marker echoes the current values. The default is *YES*. Valid text strings are:

Echo:YES - a marker echoes the current values in the *Slider2D.area*.
Echo:NO - no marker echoes the current values in the *Slider2D.area*.

EchoMethod.flag - specifies the geometric form of the echo marker. The default is *plus:unfilled circle*. The valid text strings are:

EchoMethod:dot
EchoMethod:plus
EchoMethod:filled circle
EchoMethod:unfilled circle
EchoMethod:filled rect
EchoMethod:unfilled rect

The markers can be combined. For example:

EchoMethod:dot:unfilled circle

specifies an unfilled circle with a dot in its center.

Fixed.flag - determines where the anchor point for the current values is positioned on the marker's bounding box. This flag is only effective with the markers specified using the *EchoMethod.flag*. Whenever a *Marker.object* is specified, *Fixed.flag* is ignored and markers are centered. The default anchor point is the center. Valid text strings are:

Text String:	**Position on bounding box:**
corner:ul	upper left corner
corner:ur	upper right corner
corner:ll	lower left corner
corner:lr	lower right corner
edge:top	center point of the top edge
edge:bottom	center point of the bottom edge
edge:left	center point of the left edge
edge:right	center point of the right edge

IconSize.flag - specifies the size, in screen coordinates, of the marker specified by *EchoMethod.flag*. The default is 20.

MarkerEraseMethod.flag - if present, specifies how erasing is performed. The default is *restore raster* if the workstation supports it, *erase* otherwise. Valid text strings are:

MarkerEraseMethod:restore raster - restore the background using the saved raster.
MarkerEraseMethod:erase - erase the marker, but do not restore the background.

Echo Function

The echo function for the slider2D interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
```

```
VARDESC Vdp,
RECTANGLE *EchoVP,
ADDRESS args)
```

Interpretation of Action Types for VNslider2D

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS (`POLL:YES`)	In `Slider2D.area`	INPUT_DONE	Update slider and vdp
SELECT_KEYS (`POLL:NO`)	In `Slider2D.area`	INPUT_DONE	Update slider and vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore original vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore original vdp
SELECT_KEYS	In increment areas	INPUT_ACCEPT	Update slider and vdp
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore original vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore original vdp
Motion (`POLL:YES`)	In `Slider2D.area`	INPUT_ACCEPT	Update slider and vdp
TOGGLE_POLLING_KEYS	In `Slider2D.area`	INPUT_ACCEPT	Toggle polling NO/YES

Summary of Template Areas, Objects, and Flags for VNslider2D

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required object (in layout area):

Name	Object Type	Function
Slider2D.area	rectangle	plane for marker positioning, pickable area

Required flags (in the flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNslider2D	match to input object

Optional objects (in the layout area):

Name	Object Type	Function
Slider2D.text	text	labels Slider2D.area
Marker.object	graphic	custom marker for Slider2D.area
Varname_X.area	graphic	area for X dimension label
Varname_X.text	text	controls style and position of the X dimension name
Varname_Y.area	graphic	area for Y dimension label
Varname_Y.text	text	controls style and position of the Y dimension name
Min_X.area	graphic	area for minimum X-dimension label
Min_X.text	text	controls style and position of the minimum X value setting
Max_X.area	graphic	area for maximum X-dimension label
Max_X.text	text	controls style and position of the maximum X value setting

Min_Y.area	graphic	area for minimum Y-dimension label
Min_Y.text	text	controls style and position of the minimum Y value setting
Max_Y.area	graphic	area for maximum Y-dimension label
Max_Y.text	text	controls style and position of the maximum Y value setting
Digits_X.area	graphic	controls position of current X value reading
Digits_X.text	text	controls style of current X value reading
Digits_Y.area	graphic	controls position of current Y value reading
Digits_Y.text	text	controls style of current Y value reading

Additional optional objects (in the layout area):

Name	Object Type	Function
North.area	graphic	pickable area for incrementing value
North.text or	text	label for North area
North.button	button input object	push button for incrementing value
NE.area	graphic	pickable area for incrementing value
NE.text or	text	label for NE area
NE.button	button input object	push button for incrementing value
East.area	graphic	pickable area for incrementing value
East.text or	text	label for East area
East.button	button input object	push button for incrementing value
SE.area	graphic	pickable area for incrementing value
SE.text or	text	label for SE area
SE.button	button input object	push button for incrementing value
South.area	graphic	pickable area for incrementing value
South.text or	text	label for South area
South.button	button input object	push button for incrementing value
SW.area	graphic	pickable area for incrementing value
SW.text or	text	label for SW area
SW.button	button input object	push button for incrementing value
West.area	graphic	pickable area for incrementing value
West.text or	text	label for West area
West.button	button input object	push button for incrementing value
NW.area	graphic	pickable area for incrementing value
NW.text or	text	label for NW area
NW.button	button input object	push button for incrementing value
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in the flags area):

Name	Type	Content	Function
Echo.flag	text	Echo:YES	marker echoes current values
		Echo:NO	no marker echoing
Poll.flag	text	Poll:YES	polls cursor position only for selection in Slider2D.area
		Poll:NO	polls picks only for selection in Slider2D.area
Increment.flag		Increment:%	sets change increment as a percent of X and Y ranges
IncrementX.flag		IncrementX:%	sets change increment as a percent of X range
IncrementY.flag		IncrementY:%	sets change increment as a percent of Y range

IconSize.flag	IconSize:pixels	sets marker's size in pixels
Fixed.flag	corner:ul	sets the anchor point on the marker's bounding box for positioning according to the current values. For edges, the anchor is the center point of the chosen edge.
	corner:ur	
	corner:ll	
	corner:lr	
	edge:top	
	edge:bottom	
	edge:left	
	edge:right	

Additional optional flags (in the flags area):

Name	**Type**	**Content**	**Function**
PostType.flag	text	PostType:RECT	pick in area's bounding box
		PostType:OBJECT	pick on area only
EchoMethod.flag	text	EchoMethod:dot	marker type
		EchoMethod:plus	marker type
		EchoMethod:filled circle	marker type
		EchoMethod:unfilled circle	marker type
		EchoMethod:filled rect	marker type
		EchoMethod:unfilled rect	marker type
MarkerEraseMethod.flag	text	MarkerEraseMethod:restore raster	restores saved raster image
		MarkerEraseMethod:erase	draws rectangle in background color

Interaction Handlers: VNtext

VN Description

VNbutton	VNmenu	VNslider	VNtextedit
VNchecklist	VNmultiplexor	VNslider2D	VNtoggle
VNcombiner	VNpalette	**VNtext**	

Introduction
Synopsis
Template
Additional Information
Echo Function
Interpretation of Action Types
Summary of Template, Areas, Objects, and Flags

VNtext

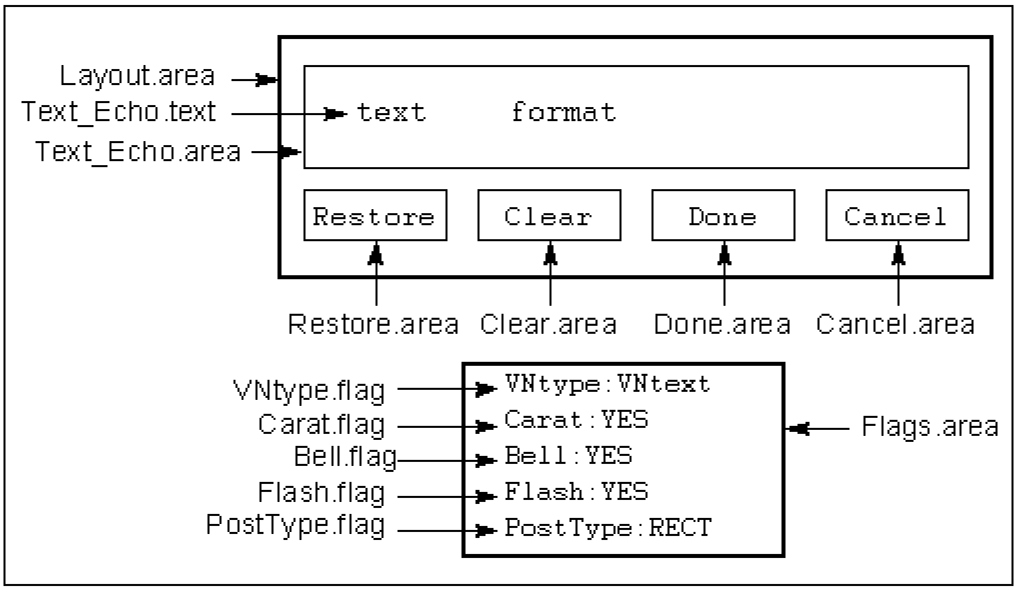
The Text interaction handler gets a line of text from the user and echoes the string using hardware text. If the string is too long to fit in the viewport, it scrolls to the left as necessary. The text interaction handler only lets the user enter characters up to the maximum length allowed by the variable descriptor set by *VOinPutVarList* and *VPvddim*. Only single line text input is supported. A template is optional.

Synopsis

```
GLOBALREF INHANDLER VNtext;
```

Template

A sample template is shown below.



Sample Template

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Text_Echo.area - area within which the text input is echoed as it is entered. The text entry is centered along the vertical dimension and starts at the left edge of the area. If the input string is too long to fit, it scrolls to the left.

Text_Echo.text - hardware text object that is used to define the attributes for echoing text. It is not displayed when the interaction is run. The color and size attributes are used in the echoed string.

Prompt.area - optional area that contains a prompt message. This area appears in the input object as drawn in *Layout.area*.

Prompt.text - text string containing the prompt message. The text appears in the input object as drawn in *Layout.area*.

Clear.area - optional area that lets the user erase the current input text string.

Clear.text - text string containing the label for the *Clear.area*. The string can be anything, but in the template supplied with DV-Tools, this string is set to “Clear.”

Flags Area:

VNtype.flag - the correct text string is *VNtype:VNtext*.

Carat.flag - marks the current cursor position. The default is *Carat:REVERSE*. Valid text strings are:

Carat:YES or *Carat:REVERSE* - current position displayed in reverse video.

Carat:NONE - no cursor is displayed. In-line positioning and on-line editing are disabled. Allows deleting from the end of the line.

Carat:BAR - a vertical bar is displayed to the left of the current position.

Carat:BOX - an open box is displayed around the current position.

Carat:UNDERSCORE - a horizontal bar is displayed beneath the current position.

Bell.flag - determines whether the bell sounds when there is too much text for the interaction handler to accept. The default is *YES*. Valid text strings are:

Bell:YES - sets the bell to ring.

Bell:NO - sets the bell to not ring.

Flash.flag - controls the flashing of the text area background. The default is *YES*. Valid text strings are:

Flash:YES - the text area is flashed in the text background color when an error occurs during text entry.

Flash:NO - the text area is not flashed when errors occur during text entry.

Direction.flag - controls the default direction of text entry. The default is *L_TO_R*. Valid text strings are:

Direction:L_TO_R - text entry is from left to right.

Direction:R_TO_L - text entry is from right to left.

Additional Information

Characters entered anywhere within the screen area of the input object are checked for use as both text and as control keys. The action keys, set using VUerPutKeys or VOitPutKeys, should be control characters so that they do not conflict with the keys interpreted as text. The text interaction handler also uses the following line editing characters:

Text Editing Commands

Operation	Character
Position cursor in text string	*select* in text
Go forward one character	^*L* or right arrow key
Go back one character	^*N* or left arrow key
Go forward to next word	^P
Go back to previous word	^O
Go to beginning of line	^*F* or up arrow key
Go to end of line	^*G* or down arrow key
Delete previous character	*Delete* or *Backspace*
Delete current character	^V
Delete to next white space	^E
Delete to previous white space	^W
Delete string	*Clear* Keys or ^*U*
Reverse text entry direction	^\ (Ctrl-Backslash)
Restores string to original	*Restore* Keys
Cancel	*Cancel* Keys
Done	*Esc*, *Return*, *Line Feed*, or *Done* Keys

Echo Function

The echo function for the text interaction handler is set up by a call to VOitPutEchoFunction. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      char **Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNtext

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Echo & restore original text
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Echo & restore original text
SELECT_KEYS	In `Clear.area`	INPUT_ACCEPT	Clear text and vdp
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Echo & restore original text
CANCEL_KEYS	In input object	INPUT_CANCEL	Echo & restore original text
CLEAR_KEYS	In input object	INPUT_ACCEPT	Clear text and vdp
ESC,RET,NEWLN	In `Text_Echo.area`	INPUT_DONE	Echo and update vdp
^U	In `Text_Echo.area`	INPUT_ACCEPT	Echo and update vdp
Other Keys	In `Text_Echo.area`	INPUT_ACCEPT	Echo and update vdp

Summary of Template Areas, Objects, and Flags for VNtext

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required object (in layout area):

Name	Object Type	Function
Text_Echo.area	rectangle	boundary of the text entry box

Required flags (in flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNtext	match to input object

Optional objects (in layout area):

Name	Object Type	Function
Text_Echo.text	text (hardware only)	defines the size of the text
Prompt.area	graphic	boundary of prompt area
Prompt.text	text	label for prompt area
Clear.area	graphic	boundary of clear area

Clear.text or	text	label for clear area
Clear.button	button input object	push button to signal clear
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in flags area):

Name	Type	Content	Function
Carat.flag	text	Carat:YES or	reverse video rectangle marks current position
		Carat:REVERSE	
		Carat:BAR	vertical bar is left of current position
		Carat:NONE	no echo of current position
		Carat:BOX	unfilled box marks current position
		Carat:UNDERSCORE	underscore marks current position
Bell.flag	text	Bell:YES	bell rings when text entered exceeds limit or entry error is made
		Bell:NO	no bell sounds
Flash.flag	text	Flash:YES	text area flashes when text entered exceeds limit or entry error is made
		Flash:NO	no flashing occurs
Direction.flag	text	Direction:L_TO_R	text entry is from left to right
		Direction:R_TO_L	text entry is from right to left
PostType.flag	text	PostType:RECT	pick in bounding box of area
		PostType:OBJECT	pick on area only

Interaction Handlers: VNtextedit

VN Description

VNbutton VNmenu VNslider **VNtextedit**
VNchecklist VNmultiplexor VNslider2D VNtoggle
VNcombiner VNpalette VNtext

VNtextedit

The Text Editor lets the user enter and edit a block of text and echoes the block using hardware text. If the block is too large to fit in the text echo area, it scrolls up and to the left and the cursor can scroll outside of the text echo area. The text editor only lets the user enter characters up to the maximum length specified by the variable descriptor set by VOinPutVarList and *VPvddim*. A template is optional. If you do not use a template, the input object uses an editing window defined by internal defaults, like the one used for editing commands and view comments.

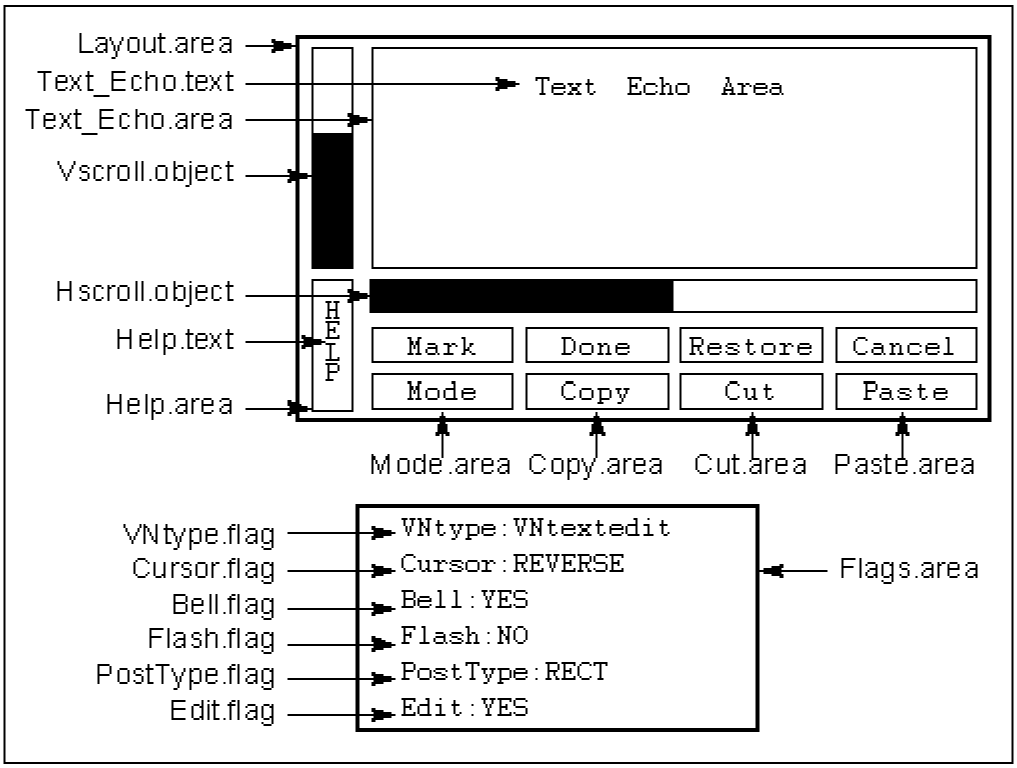
The example program *text_editor.c* shows how to load a form from a file into a text editor and save the edited text to a file. If the text loaded into the editor contains tabs, the tabs are displayed as single spaces.

Synopsis

```
GLOBALREF INHANDLER VNtextedit;
```

Template

A sample template is shown below.



Sample Template

The following components are unique to this interaction handler. Components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Text_Echo.area - area within which the text input is echoed as it is entered. The cursor is initially located in the upper left corner. If the input text is too large to fit, it scrolls up and to the left. The text echo area always displays at least one row or one column of text even if the characters are drawn beyond the boundary of the text echo area. The text echo area can display a maximum of 256 characters on each line, although the user can enter more than 256 characters. *Text_Echo.area* should be specified if any other objects are specified in *Layout.area*.

Text_Echo.text - hardware text object that is used to define the attributes for echoing text. It is not displayed when the interaction is run. The color and size attributes are used in the echoed string.

HScroll.object, *VScroll.object* - optional slider or scrollbar input objects that control the horizontal and vertical scrolling of the text being displayed. Using *Poll:NO* as the polling flag in the slider or scrollbar templates and setting a select key for the text editor are recommended. The scrollbars or sliders respond to the same action keys as the text editor.

HScroll.area, *VScroll.area*- optional areas that define where *HScroll.object* and *VScroll.object* will be drawn.

Prompt.area - optional area that contains a prompt message. This area appears in the input object as drawn in *Layout.area*.

Prompt.text - text string containing the prompt message. The text appears in the input object as drawn in *Layout.area*.

Clear.area - optional area that lets the user erase the current input text string.

Clear.text - text string containing the label for the *Clear.area*. The string can be anything, but in the template supplied with DV-Tools, this string is set to "Clear."

Clear.button - button input object for clearing. If used with *Clear.area*, the button is scaled to fit *Clear.area*. To add a label, edit the label for the button input object; *Clear.text* is mutually exclusive and cannot be used with this object.

Help.area - optional area that lets the user alternately display and erase a list of the control keys and their corresponding editing actions. The list appears in the *Text_Echo.area* in place of the text and can be scrolled if it is too large to fit.

Help.text - text string containing the label for the *Help.area*. The string can be anything, but in the template supplied with DV-Tools, this string is set to "Help."

Help.button - button input object for displaying help. If used with *Help.area*, the button is scaled to fit *Help.area*. To add a label, edit the label for the button input object; *Help.text* is mutually exclusive and cannot be used with this object.

Mark.area - optional area that lets the user enter a highlighting mode. After entering the highlight mode, press the left mouse button to indicate the start position for the highlight. Press the left mouse button again the indicate the end position for the highlight. The highlighted text can be cut, copied, or pasted.

Mark.text - text string containing the label for the *Mark.area*. The string can be anything, but in the template supplied with DV-Tools, this string is set to "Mark."

Mark.button - button input object for entering the highlight mode. If used with *Mark.area*, the button is scaled to fit *Mark.area*. To add a label, edit the label for the button input object; *Mark.text* is mutually exclusive and cannot be used with this object.

Copy.area, *Cut.area*, *Paste.area*, *Mode.area*- optional areas that let the user move highlighted blocks of text in the following ways: copying text into a paste buffer, cutting text from the display and placing it in the paste buffer, or pasting the text from the paste buffer into the display. You cannot paste text from one input object into another input object. Selecting *Mode.area* switches between the three highlighting modes: area, rectangle, and lines.

Copy.text, *Cut.text*, *Paste.text*, *Mode.text* - text strings containing the labels for the *Copy.area*, *Cut.area*, *Paste.area*, and *Mode.area*. The strings can be anything.

Copy.button, *Cut.button*, *Paste.button*, *Mode.button* - button input objects for handling highlighted blocks of text. If used with the corresponding areas, the buttons are scaled to fit the areas. To add a label to a button, edit the label for the button input object; the corresponding text labels (**.text*) are mutually exclusive and cannot be used with the button input objects.

Left.area, *Right.area*, *Up.area*, *Down.area* - optional areas that let the user scroll the text block left, right, up, and down.

Left.text, *Right.text*, *Up.text*, *Down.text* - text strings containing the labels for the scroll areas. The strings can be anything.

Left.button, *Right.button*, *Up.button*, *Down.button* - button input objects for scrolling the text. If used with the corresponding areas, the buttons are scaled to fit the areas. To add a label to a button, edit the label for the button input object; the corresponding text labels (**.text*) are mutually exclusive and cannot be used with the button input objects.

Flags Area:

VNtype.flag - the correct text string is *VNtype:VNtextedit*.

Cursor.flag - determines the style of the cursor marking the current position. The default is *Cursor:UNDERSCORE*. Valid text strings are:

Cursor:REVERSE - current position is displayed in reverse video.

Cursor:COLOR - a colored rectangle is displayed at the current position. The foreground and background colors of the flag determine the foreground and background of the character at the cursor position. These colors should be different from the colors of *Text_Echo.text*.

Cursor:UNDERSCORE - a horizontal bar is displayed beneath the current position.

Bell.flag - determines whether or not the bell sounds when there is too much text for the interaction handler to accept. The default is *YES*. Valid text strings are:

Bell:YES - sets the bell to ring.

Bell:NO - sets the bell to not ring.

Flash.flag - determines whether or not the text area background flashes when there is too much text for the interaction handler to accept. The default is *YES*. Valid text strings are:

Flash:YES - the text area is flashed in the text background color.

Flash:NO - the text area is not flashed.

Edit.flag - determines whether or not text editing is enabled. The default is *YES*. Valid text strings are:

Edit:YES - text editing is enabled.

Edit:NO - text editing is not enabled. This is useful for displaying text that the user should not edit.

Direction.flag - controls the default direction of text editing. The default is *L_TO_R*. Valid text strings are:

Direction:L_TO_R - text entry is from left to right, the text is left-justified, and carriage returns move the cursor to the left end of the next line.

Direction:R_TO_L - text entry is from right to left, the text is right-justified, and carriage returns move the cursor to the right end of the next line.

Additional Information

Characters entered anywhere within the screen area of the input object are checked for use as both text and as control keys. The action keys, set using VUerPutKeys or VOitPutKeys, should be control characters so they do not conflict with the keys interpreted as text. Note that you can reassign the control keys listed below as action keys, but they no longer function as editing commands. The following table shows the control characters and editing commands:

Text Editing Commands

Operation	Character
Select position of cursor or highlight	select in text
Toggle Help display on and off	^Q
Go to the left	^X ^L

Go to the right	^X ^R
Go forward one character	^L or right arrow key
Go back one character	^N or left arrow key
Go forward to next word	^P
Go back to previous word	^O
Go to beginning of line	^F
Go to end of line	^G
Go up one line	^X u* or up arrow key
Go down one line	^X d* or down arrow key
Go up one page	^X ^U
Go down one page	^X ^D
Delete previous character	Delete or Backspace
Delete current character	^V
Delete to end of word	^E
Delete to beginning of word	^W
Delete current line	^U
Delete to end of line	^K
Delete all contents of editor	Clear Keys
Add new line	Return or LineFeed
Toggle insert/overwrite mode	^I
Enter highlight mode	^X h*, <select>
Cut highlighted region and put in paste buffer	^X t*
Copy highlighted region and put in paste buffer	^X c*
Paste highlighted region or paste buffer to cursor position	^X p*
Toggle highlight mode between area, rectangle, and lines	^X m*
Reverse text editing direction	^\ (Ctrl-Backslash)
Abort without saving changes	^X ^S or Cancel Keys
Restore original text	^R or Restore Keys
Done	Esc or Done Keys

* case-sensitive

Note that <*Restore*> keys toggle between the unchanged text and the most recently changed version of the text.

Echo Function

The echo function for the text interaction handler is set up by a call to *VOitPutEchoFunction*. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      char **Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNtextedit

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In `Text_Echo.area`	INPUT_ACCEPT	Change cursor or highlight
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Echo & restore original text
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Echo & restore original text
SELECT_KEYS	In `Clear.area`	INPUT_ACCEPT	Clear text and vdp
SELECT_KEYS	In `Help.area`	INPUT_ACCEPT	Display or erase help
SELECT_KEYS	In `Mark.area`	INPUT_ACCEPT	Enter highlight mode
SELECT_KEYS	In `Mode.area`	INPUT_ACCEPT	Switch highlight mode
SELECT_KEYS	In Cut, Copy, etc. areas	INPUT_ACCEPT	Echo and update vdp
SELECT_KEYS	In sliders or scrollbars	INPUT_ACCEPT	Scroll text block
SELECT_KEYS	In Up, Down, etc. areas	INPUT_ACCEPT	Scroll text block
RESTORE_KEYS	In input object	INPUT_ACCEPT	Echo & restore original text
CANCEL_KEYS	In input object	INPUT_CANCEL	Echo & restore original text
CLEAR_KEYS	In input object	INPUT_ACCEPT	Clear text and vdp
DONE_KEYS	In input object	INPUT_DONE	None
ESC	In `Text_Echo.area`	INPUT_DONE	None
^X ^S	In `Text_Echo.area`	INPUT_CANCEL	Echo & restore original text
^X ^H	In `Text_Echo.area`	INPUT_ACCEPT	Enter highlight mode
^X ^M	In `Text_Echo.area`	INPUT_ACCEPT	Switch highlight mode
^Q	In `Text_Echo.area`	INPUT_ACCEPT	Display or erase help
Motion (`POLL:YES`)	In sliders or scrollbars	INPUT_ACCEPT	Scroll text block
Other Keys	In `Text_Echo.area`	INPUT_ACCEPT	Scroll or echo and update vdp

Summary of Template Areas, Objects, and Flags for VNtextedit

Required areas:

Name	Object Type	Function
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area

Required object (in layout area):

Name	Object Type	Function
Text_Echo.area	rectangle	boundary of the text entry box

Required flags (in flags area):

Name	Type	Content	Function
VNtype.flag	text	VNtype:VNtextedit	match to interaction

Optional objects (in layout area):

Name	Object Type	Function
Text_Echo.text	text (hardware only)	defines the attributes of the text (appears only in the template)
Hscroll.area	rectangle	boundary for horizontal scrolling slider or scrollbar
Hscroll.object	input object	slider or scrollbar for scrolling the text
Vscroll.area	rectangle	boundary for vertical scrolling slider or scrollbar
Vscroll.object	input object	slider or scrollbar for scrolling the text
Up.area	graphic	boundary of up scrolling area
Up.text or	text	label for up scrolling area

Up.button	button input object	push button to scroll up through text
Down.area	graphic	boundary of down scrolling area
Down.text or	text	label for down scrolling area
Down.button	button input object	push button to scroll down through text
Left.area	graphic	boundary of left scrolling area
Left.text or	text	label for left scrolling area
Left.button	button input object	push button to scroll left through text
Right.area	graphic	boundary of right scrolling area
Right.text or	text	label for right scrolling area
Right.button	button input object	push button to scroll right through text
Prompt.area	graphic	boundary of prompt area
Prompt.text	text	label for prompt area
Help.area	graphic	boundary of help area
Help.text or	text	label for help area
Help.button	button input object	push button for help action
Clear.area	graphic	boundary of clear area
Clear.text or	text	label for clear area
Clear.button	button input object	push button to signal clear
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Additional optional objects (in layout area):

Name	Object Type	Function
Mark.area	graphic	boundary of mark area
Mark.text or	text	label for mark area
Mark.button	button input object	push button for mark action
Mode.area	graphic	boundary of mode area
Mode.text or	text	label for mode area
Mode.button	button input object	push button for mode action
Cut.area	graphic	boundary of cut area
Cut.text or	text	label for cut area
Cut.button	button input object	push button for cut action
Copy.area	graphic	boundary of copy area
Copy.text or	text	label for copy area
Copy.button	button input object	push button for copy action
Paste.area	graphic	boundary of paste area
Paste.text or	text	label for paste area
Paste.button	button input object	push button for paste action

Optional flags (in flags area):

Name	Type	Content	Function
Cursor.flag	text	Cursor:REVERSE	reverse video rectangle marks current position
		Cursor:COLOR	colored rectangle marks current position
		Cursor:UNDERSCORE	underscore marks current position
Bell.flag	text	Bell:YES	bell rings when text entered exceeds limit or entry error is made

		Bell:NO	no bell sounds
Flash.flag	text	Flash:YES	text area flashes when text entered exceeds limit or entry error is made
		Flash:NO	no flashing occurs
Edit.flag	text	Edit:YES	text editing is enabled
		Edit:NO	text editing is not enabled
Direction.flag	text	Direction:L_TO_R	text editing is from left to right
		Direction:R_TO_L	text editing is from right to left
PostType.flag	text	PostType:RECT	pick in bounding box of area
		PostType:OBJECT	pick on area only

Interaction Handlers: VNtoggle

VN Description

VNbutton VNmenu VNslider VNtextedit
VNchecklist VNmultiplexor VNslider2D **VNtoggle**
VNcombiner VNpalette VNtext

VNtoggle

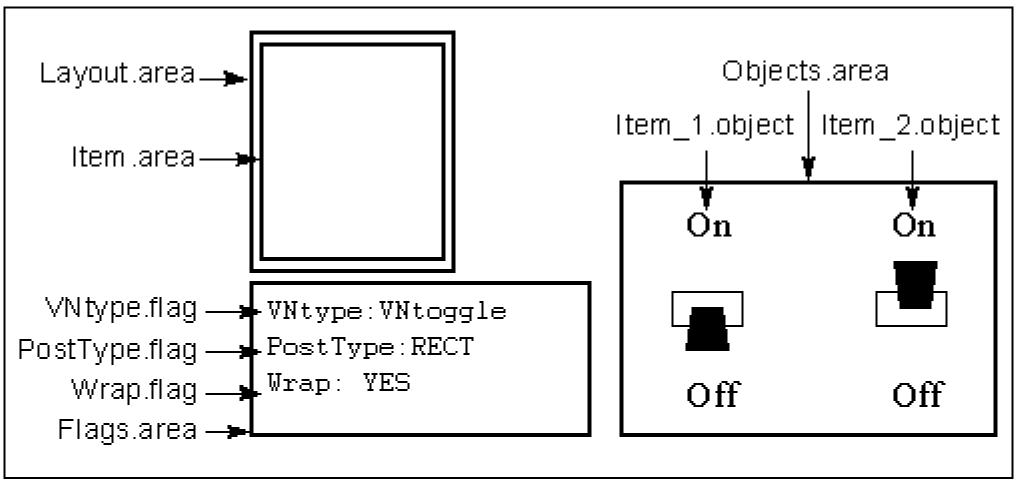
The Toggle interaction handler gets an item selection from the user and echoes the selection within the specified viewport. The associated variable, set by *VOinPutVarList*, is set to the value that corresponds to the toggle entry currently displayed. Values are defined by *VOitPutListValues*. If there is no corresponding value, the variable is set to the index of the current toggle item. Toggle items can be text strings set using *VOitPutList*, button input objects with labels set using *VOitPutList*, or objects. A template is optional for text toggle interactions and required for object toggle interactions.

Synopsis

```
GLOBALREF INHANDLER VNtoggle;
```

Template

A sample template is shown below.



Sample Template (for an object toggle)

The following components are unique to this interaction handler. The components common to all interaction handlers are described in the chapter introduction.

Layout.area:

Item.area - area in which the toggle items are displayed. Selecting this area toggles the displayed item to the next item in the sequence. The item choices are all *Item_%d.text*, all *Item_%d.object*, or all *Item_%d.button*, but not a mixture.

Item_%d.text - text object used to define the attributes used to display the text toggle items. The text string is not displayed during the interaction. The background color of the text is the erase color for the toggle items in an object toggle. *VOitPutList* can be used to set the text strings programmatically. Usually only one text item is used, but multiple text items may be placed in the objects area and then the labels are centered in the item area and the text attributes toggle with the labels.

Item_%d.object - object item. Objects can be either a single object or a subdrawing and must be placed in the objects area. Ignores *VOitPutList*. The items must fit within the item area, are centered in the item area when displayed, and are erased using the background color of *Item.text*. Object toggles can support labels when the items (*Item_%d.object*) are subdrawings which use *Label.area* and *Label.object* in the subdrawing views.

Item_%d.button - button item. Buttons are scaled to fit the item area. If the buttons support labels, *VOitPutList* can be used to set the text strings programmatically. Usually only one button item is used, but multiple button items may be placed in the objects area and then the buttons are centered in the item area and the button appearance toggles with the labels.

Next.area - when selected, toggles to the item with the next highest number.

Next.text - text string containing the label for the *Next.area*.

Next.button - button input object for toggling to the next item. If used with *Next.area*, the button is scaled to fit *Next.area*. To add a label, edit the label for the button input object; *Next.text* is mutually exclusive and cannot be used with this object.

Previous.area - when selected, toggles to the item with the next lowest number.

Previous.text - text string containing the label for the *Previous.area*.

Previous.button - button input object for toggling to the previous item. If used with *Previous.area*, the button is scaled to fit *Previous.area*. To add a label, edit the label for the button input object; *Previous.text* is mutually exclusive and cannot be used with this object.

Flags.area:

Wrap.flag - controls how the toggle behaves when you attempt to pass the beginning or end of a list of items. The interaction handler wraps around to the first item in the list, or starts back down the list; decrementing the list until it reaches the beginning, and starts incrementing again. The default value is *YES*. Valid text strings are:

Wrap:YES - wraps around. After the toggle displays the last item, the first item follows in a cyclical sequence.

Wrap:NO - goes back and forth along the list. After the toggle displays the last item, the second to last item follows.

Echo Function

The echo function for the toggle interaction handler is set up by a call to *VOitPutEchoFunction*. It has the following unique call structure:

```
void
echo_fcn (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args)
```

Interpretation of Action Types for VNtoggle

The following table of action types specifies how certain key presses are to be interpreted based on the interaction handler and the context of the action. Valid action types are:

- DONE_KEYS
- CANCEL_KEYS
- SELECT_KEYS
- RESTORE_KEYS
- CLEAR_KEYS
- TOGGLE_POLLING_KEYS

Action Type	Locator Position	Service Result	Services
SELECT_KEYS	In item area	INPUT_ACCEPT	Echo and update vdp
SELECT_KEYS	In `Done.area`	INPUT_DONE	None
SELECT_KEYS	In `Restore.area`	INPUT_ACCEPT	Restore original vdp
SELECT_KEYS	In `Cancel.area`	INPUT_CANCEL	Restore original vdp
SELECT_KEYS	In increment areas	INPUT_ACCEPT	Echo and update vdp
DONE_KEYS	In input object	INPUT_DONE	None
RESTORE_KEYS	In input object	INPUT_ACCEPT	Restore original vdp
CANCEL_KEYS	In input object	INPUT_CANCEL	Restore original vdp

Summary of Template Areas, Objects, and Flags for VNtoggle

Required areas:

Name	**Object Type**	**Function**
Layout.area	graphic	boundary of layout area
Flags.area	rectangle	boundary of flags area
Objects.area	rectangle	boundary of objects area (required for object toggles; optional for text toggles)

Required objects (in the layout area or objects area):

Name	**Object Type**	**Function**
Item.area	graphic	display area for items (in layout area only)
Item_%d.text or	text	toggle items (text, objects, and buttons cannot be mixed). For button items, push buttons are recommended. Object items must be in objects area
Item_%d.object or	graphic	
Item_%d.button	button input object	

Required flags (in the flags area):

Name	**Type**	**Content**	**Function**
VNtype.flag	text	VNtype:VNtoggle	match to input object

Optional objects (in layout area):

Name	**Object Type**	**Function**
Next.area	graphic	pickable area to toggle to next numbered item
Next.text or	text	label for next area
Next.button	button input object	push button to toggle to next numbered item
Previous.area	graphic	pickable area to toggle to previous numbered item
Previous.text or	text	label for previous area
Previous.button	button input object	push button to toggle to previous numbered item
Done.area	graphic	boundary of done area
Done.text or	text	label for done area
Done.button	button input object	push button to signal done
Restore.area	graphic	boundary of restore area
Restore.text or	text	label for restore area
Restore.button	button input object	push button to signal restore
Cancel.area	graphic	boundary of cancel area
Cancel.text or	text	label for cancel area
Cancel.button	button input object	push button to signal cancel

Optional flags (in flags area):

Name	**Type**	**Content**	**Function**
Wrap.flag	text	Wrap:YES	sequence wraps around, first item follows last
		Wrap:NO	sequence ascends, then descends
PostType.flag	text	PostType:RECT	pick in bounding box
		PostType:OBJECT	pick on pickable area only

VD - Display Formatters

Display Formatters (VD)

Introduction

Data structures that display the graphic encoding of data on the screen. To use one of these data structures, you must first declare it using *GLOBALREF*, then use *VPdgdf* to attach the display formatter to the data group you want to display.

In this chapter, the term variable is used to mean variable descriptor.

All variables within a data group must have the same dimension. Variables within a single graph should also have the same range, unless the graph description explicitly states that the display formatter can handle more than one range. For additional information, see *VPvddim*.
Elements of matrix variables are displayed from the lower left of the matrix to the upper right. Vectors are displayed from left to right. For example, if the shape of a matrix variable descriptor is 3 columns x 2 rows, then the variable elements are displayed in the following order:

1,2	2,2	3,2
1,1	2,1	3,1

Display Formatters

Name	**Description**
VDbars	
VDbar	Vertical bar chart.
VDbarhoriz	Horizontal bar graph.
VDbarpacked	Bar graph, no spaces between bars.
VDbarsolid	Bar chart, each bar a single color.
VDcenter	Centered bar chart.
VDpig	Piggyback bar chart.
VDpigdist	Statistical distributions using piggyback bars.
VDblocks	
VDcprects	Packed rectangles with changing color.
VDrects	Rectangles with changing color.
VDbullseye	
VDbullseye	Cartesian graph of (x,y) points.
VDclock	
VDanclock	Simulated analog clock.
VDcolorbar	
VDcolorbar	Horizontal legend showing the color threshold table of the variable.
VDcombos	
VDbarline	Vertical bar and line graph combination.
VDbarpackedline	Packed bar and line graph combination.
VDbarplstacked	Stack of packed bar-line graphs.
VDhilobar	Vertical bar and high-low-close graph combination.
VDhiloline	Line and high-low-close graph combination.
VDptsline	Points and line graph combination.
VDcontours	
VDcontour	Contour plot of a matrix variable.
VDfcontour	Filled contour plot of a matrix variable.

VDcontrollers

VDcontroller	Combination of bar and point graphs.
VDhorizcontroller	Combination of horizontal bar and point graphs.

VDdials

VDdial	180-degree dial.
VDdial360	360-degree dial.
VDhistdial	Dials with dots for previous values.

VDdigit

VDdigits	Digital display.

VDdrawings

VDdrawing	Runs a view created with DV-Draw.
VDmovedrawing	Rotates, scales, and moves a drawing.

VDface

VDface	Face with changing features.

VDfader

VDfader	Fader display.

VDfan

VDfan	Nested fans.

VDhighlowopen-close

VDhighlow	High-low-open-close display.

VDhorizon

VDhorizon	Artificial horizon graph.

VDindicator

VDindicator	Marker display of current variable value.

VDknob

VDknob	Knob with a 270 degree range.

VDlegend

VDlegend	Legend for each variable.

VDlines

VDline	Line graph.
VDlinedist	Statistical distributions using filled lines.
VDlinefill	Line graph filled below lines.
VDlinefstacked	Stack of filled line graphs.
VDlinestacked	Stack of line graphs.
VDstep	Horizontal value lines connected by vertical lines.

VDmeter

VDmeter	Logarithmic meter.

VDpie

VDpie	Pie chart.

VDpoint

VDpoints	Points graph.
VDprimitives	
VDbox	Box with changing color and shape.
VDcircle	Circle with changing color and size.
VDtriangle	Triangles with changing color and shape.
VDradials	
VDne_radial	Polar coordinate graph, no erasing.
VDradial	Polar graph, erasing after 360 degrees.
VDscatters	
VDimpulse	Scatter plot with vertical lines.
VDimpulseto0	Scatter plot with vertical lines symmetrically about zero.
VDscatter	Scatter plot.
VDsize	
VDsize	Rectangles with changing length and width.
VDspectros	
VDspectro	Colored bar for each sample of the variable.
VDspectrointp	Interpolated colored bar for each sample of the variable.
VDspectrointpstkd	Stack of interpolated spectro graphs.
VDspectrostacked	Stack of spectro graphs.
VDstrips	
VDstrip	Line graph that scrolls with time.
VDstripras	Strip chart that scrolls using raster images.
VDstripstacked	Stack of strip charts.
VDvstrip	Strip chart that scrolls up.
VDvstrip_r	Strip chart that scrolls up using raster images.
VDwaterfall	Strip chart that scrolls down.
VDwaterfall_r	Strip chart that scrolls down using raster images.
VDsurface	
VD3dsurface	Three-dimensional surface graph.
VDtext	
VDmessage	One or more text graphs.
VDtext	Text in the center of the viewport.
VDtime	
VDrtline	Line graph with time-stamped values.
VDrtstep	Stacked step graph with time-stamped values.
VDvectors	
VDflowfield	Scatter plot of vectors.
VDvector	Array of vectors (x, y, and color).
VDwebs	
VDmultiyweb	Scatter plot with lines connecting each point to adjacent points and multiple vertical value axes.
VDweb	Serial (x,y) points connected by lines.

VDbars

Bar charts.

Synopses

```
GLOBALREF DISPFORM VDbar;
GLOBALREF DISPFORM VDbarhoriz;
GLOBALREF DISPFORM VDbarpacked;
GLOBALREF DISPFORM VDbarsolid;
GLOBALREF DISPFORM VDcenter;
GLOBALREF DISPFORM VDpig;
GLOBALREF DISPFORM VDpigdist;
```

Descriptions

VDbars	*DV-Draw Graph Type:* see routines below	
Var Shape: scalar, vector, matrix	Min Variables: 1	*Max Variables:* 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y) *(first variable only)*		

Bar graphs display data values using one bar for each data element. Additional variables are displayed using additional bars, lines, or whole bar graphs. The dimension of the bar is proportional to the variable value.

The bar color is determined by the color or color threshold table associated with the variable.

Bar graphs wrap around to the beginning of the data viewport, scroll left, or scroll up, depending on the value of *VPdgscroll_amount*. The default is to wrap around to the beginning.

VDbar draws a vertical bar graph. The corresponding DV-Draw graph type is *Bar Chart*.

VDbarhoriz draws a horizontal bar graph. The corresponding DV-Draw graph type is *Horizontal Bar Chart*.

VDbarpacked draws a vertical bar graph without spaces between the bars. The corresponding DV-Draw graph type is *Packed Bar Chart*.

VDbarsolid draws a vertical bar graph where each bar is filled with a single color. The bar color is determined by the color or color threshold table associated with the variable. If there is no color threshold table, *VDbarsolid* is identical to *VDbar*. The corresponding DV-Draw graph type is *Solid Bar Chart*.

VDcenter draws a centered vertical bar graph with the columns mirrored around the base line and centered vertically in the data viewport. The corresponding DV-Draw graph type is *Centered Bar Chart*.

VDpig draws a stacked bar graph in which the variable values are stacked vertically with the first variable on the bottom. The corresponding DV-Draw graph type is *Piggyback Bar Chart*.

The range of the graph equals the sum of the variable ranges. The variables should be either all logarithmic or all linear.

VDpigdist draws a stacked bar graph in which the variable values are stacked on top of each other with the first variable on the bottom. The corresponding DV-Draw graph type is *Piggyback Bar Distribution*.

All variables must have the same range. The range of the graph equals the range of the attached variables. The sum of the variable values for each sample should not exceed the maximum range value. The range of the value scale equals the sum of the ranges of the variables attached. For example, if a graph had three variables with a range of 0 to 1, the range of the graph is 0 to 10. A given sample of the three variables might have values of 2, 3, and 5 or 2, 3, and 4 but not 2, 3, and 6.

See also *VDpig*.

VDblocks

VDrects, *VDcprects* - rectangular color patch graphs.

Synopses

```
GLOBALREF DISPFORM VDcprects;
GLOBALREF DISPFORM VDrects;
```

Descriptions

Both formatters display an array of rectangles. The color of each block is determined by the color or color threshold table associated with the variable. The data value is displayed in the center of each block.

VDcprects *DV-Draw Graph Type:* Packed Block

Variable Shape: scalar, vector, matrix Min Variables: 1 Max Variables: 5

History: No Min Samples: 1 Max Samples: 1

Axis Types: Value Tick Label (digital value display), Horizontal (columns), Vertical (rows), Time Tick Label (iteration number)

VDcprects displays each box without separating outlines.

VDrects *DV-Draw Graph Type:* Block

Variable Shape: scalar, vector, matrix Min Variables: 1 Max Variables: 5

History: No Min Samples: 1 Max Samples: 1

Axis Types: Value Tick Label (digital value display), Horizontal (columns), Vertical (rows), Time Tick Label (iteration number)

VDrects displays each box outlined in the background color.

VDbullseye

Displays (x,y) points on a Cartesian graph.

Synopses

```
GLOBALREF DISPFORM VDbullseye;
```

Descriptions

VDbullseye	*DV-Draw Graph Type:* Bullseye	
Variable Shape: scalar, vector[2]	Min Variables: see below	Max Variables: 25
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Axis Grid (rectilinear target lines) or Value Axis Ticks (radial target lines)		

VDbullseye accepts either scalar or vector[2] variables. If scalar, two variables are required for each graph point to provide the x and y values respectively. If vector, the variable can only have two elements, for the x and y values respectively.

The range of the variables must be symmetrical around zero. All variables must have the same range.
To control the axes and target lines, use the following *VPdgcontext* flags:

To display the axes, set the *V_FV_GRID* flag to *YES*. Use *VPdggrid_attr* to control the color and line type of the grid. The grid consists only of the x and y axes.

To display radial target lines, set the *V_FV_TICS* flag to *YES*.

To display rectilinear target lines, set the *V_FT_GRID* flag to *YES*.

To change the number of target lines, set *VPdgtime_start_incr* to the desired number of target lines.

When using vector variables, the first *n* variables provide target line values where *n* is the number of target lines specified. If the target lines are radial, the first element of the vector is the radius of the circle and the second element is ignored. If the target lines are rectilinear, the first element of the vector provides the x position and the second element provides the y position. The remaining variables are plotted as x,y coordinates.

When using scalar variables, the first *n* variables provide target line values for radial target lines where *n* is the number of target lines specified. Each target line value provides the radius of a circle. If the target lines are rectilinear, 2*n* variables are required for the target line values. In each pair of variables, the first variable provides the x position and the second provides the y position. The remaining variables are plotted as x,y coordinates, so there must be an even number of graph variables. The graph attributes are determined by the second variable (the y variable) in each pair.

The graph variables are plotted alternately as a solid vector and a clock hand, starting with a solid vector, to prevent vector pairs from overlapping. The hour hand can be hollow or filled. To produce a hollow hour hand, set the *V_FV_TICS* to *YES* and the *V_FV_LABEL_TICS* flags to *NO*. To produce a filled hour hand, set both the *V_FV_TICS* and *V_FV_LABEL_TICS* flags to *YES*.

The color of the solid vectors and clock hands is determined by the color associated with the variable if you are using a vector variable, or by the color associated with the second variable if you are using scalar variables. If the determining variable has a color threshold table, the color is determined by that variable value and the corresponding color in that threshold table.

The scale of the graph corresponds to the range of the variables. This scale is applied to both the x and y axes. This display formatter does not currently allow separate scale control of the x and y axes.

This display formatter supports color threshold tables when using radial target lines. The number of entries in the color threshold table should be one more than the number of target lines. This provides a color range between each pair of target lines and beyond the innermost and outermost target lines. The actual numerical values of the thresholds are ignored; thresholds are set to equal the range bar values. When the variable value crosses a range bar value, the color of the vector or clock hand changes. If there are not enough thresholds, no color dynamics are used. If there are extra thresholds, the graph starts with the lowest threshold and uses only as many thresholds as it needs.

The number of history slots displayed is determined by the number specified by *VPdgslots*. To display history, the value must be greater than 1 *and* the variable must have any marker except the null marker. If either of these conditions is not true, no history is displayed. When using history, each new vector and clock hand leaves a history marker in the color of the vector or clock hand.

VDclock

Draws a simulated analog clock.

Synopses

```
GLOBALREF DISPFORM VDanclock;
```

Descriptions

VDanclock	*DV-Draw Graph Type:* Clock	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 2
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Horizontal (columns), Vertical (rows), Value Ticks (ticks around clock)		

VDanclock maps the data range onto the circumference of the clock face starting with the minimum value at the top and proceeding in a clockwise direction. For example, if the range is [0,1], the first variable is 0.25, and the second variable is 0.5, the hour hand points to three o'clock and the minute hand points to six o'clock.

The first variable displays the hour hand, the second variable displays the minute hand.

Variables need not have the same range.

The first variable determines the placement of the tick marks.

VDcolorbar

Displays the color threshold table of the variable as a horizontal legend.

Synopses

```
GLOBALREF DISPFORM VDcolorbar;
```

Descriptions

VDcolorbar	*DV-Draw Graph Type:* Color Bar	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Value (x)		

The color bar appears at the top edge of the graph area. The height of the color bar is proportional to the width of the graph area, not to its height. If the graph area is not as high as the color bar, the complete color bar and axis still display correctly.

This display formatter works best with color thresholds.

The axis displays the value range of the color threshold table. The axis and variable name cannot be turned off.

VDcombos

Combination graph formatters: bar-line, hilo-bar, hilo-line, point-line.

Synopses

```
GLOBALREF DISPFORM VDbarline;
GLOBALREF DISPFORM VDbarpackedline;
GLOBALREF DISPFORM VDbarplstacked;
GLOBALREF DISPFORM VDhilobar;
GLOBALREF DISPFORM VDhiloline;
GLOBALREF DISPFORM VDptsline;
```

Descriptions

These display formatters display line graphs combined with bars, horizontal lines, or points.
The colors of the lines, bars, and points are determined by the color or color threshold table associated with the variables.
When multiple variables are used with different ranges, a second value axis is displayed on the right.

VDbarline	*DV-Draw Graph Type:* Bar Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y) (two if range of second variable is different from first)		

VDbarline displays the first variable as a bar chart and all subsequent variables as lines. Only the left value axis is displayed if all variables have the same range. If the second variable (the first to be displayed as a line) has a different range from the first, a second value axis is displayed on the right. If only one variable is used, this display formatter displays an overlapping bar and line graph using one variable.

VDbarpackedline	*DV-Draw Graph Type:* Packed Bar-Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y) (two if range of second variable is different from first)		

VDbarpackedline displays the first variable as a bar chart and all subsequent variables as lines. There are no gaps between the bars.

If only one variable is used, this display formatter displays an overlapping bar and line graph using one variable.

The legend of this display formatter lists only the first variable, displayed as a bar. The remaining variables do not appear in the legend. To list all of the variables in the legend, turn the legend off in the Packed Bar-Line graph and use the Legend display formatter to display the variables.

VDbarplstacked	*DV-Draw Graph Type:* Stacked Packed Bar-Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 32
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y) (two if range of second variable is different from first)		

VDbarplstacked displays each variable pair as a Packed Bar-Line Graph, stacking each graph above the previous one. The first variable of each pair is displayed as a bar; the second as a line. There is no space between the bars.

The value axis of each graph is displayed on the left side of the graph. The value axis is determined by the first variable of the pair. If the range of the second variable (the one displayed as a line) is different from the first, a second value axis is displayed on the right side of that graph.

If the number of variables is odd, the last graph displays an overlapping bar and line graph using one variable.

This display formatter displays a single title and a single legend for the stack of graphs. The legend lists all of the variables in the stack of graphs.

VDhilobar	*DV-Draw Graph Type:* High Low Bar

Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 13
History: Yes	Min Samples: 1	Max Samples: unlimited

Axis Types: Time (x) vs Value (y) (two if range of fourth variable is different from first)

VDhilobar displays the first three variables as a high-low-close graph and all subsequent variables as bar charts. If all variables have the same range, only the left value axis is displayed. If the fourth variable (the first one that generates a bar) has a different range from the first, a second axis is displayed on the right. When fewer than four variables are used, *VDhilobar* uses the last variable for the bar and the remaining variables for the high-low graph.

VDhiloline	*DV-Draw Graph Type:* High Low Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 13
History: Yes	Min Samples: 2	Max Samples: unlimited

Axis Types: Time (x) vs Value (y) (two if range of fourth variable is different from first)

VDhiloline displays the first three variables as high-low-close graph and all subsequent variables as lines. If the fourth variable (the first one that generates a bar) has a different range from the first, a second axis is displayed on the right. When fewer than four variables are used, *VDhilobar* uses the last variable for the line and the remaining variables for the high low graph.

VDptsline	*DV-Draw Graph Type:* Point-Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited

Axis Types: Time (x) vs Value (y) (two if range of last variable is different from first)

VDptsline displays all variables as points except for the last variable, which displays as an independent line graph. If the last variable (the one that generates a line) has a different range from the first, a second axis is displayed on the right. If only one variable is used, both the line and the points use the same variable and the line is superimposed on the points.

VDcontours

Contour plot of a matrix variable. Matrix element values are located at the midpoints of the display grid, with intermediate values mapped between one value and another. Contour lines are drawn through all points where the values correspond to the threshold values in the color threshold table.

These display formatters work best with a color threshold table.

Synopses

```
GLOBALREF DISPFORM VDcontour;
GLOBALREF DISPFORM VDfcontour;
```

Descriptions

VDcontour, VDfcontour	*DV-Draw Graph Type:* Contour, Filled Contour	
Variable Shape: matrix only	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Horizontal (columns), Vertical (rows), Time Tick Label (iteration number)		

VDcontour displays a contour plot. If there is no color threshold table, the graph calculates two or more equidistant contours, depending on the size of the data area.

VDfcontour displays a filled contour plot. The areas between contour lines are filled with the corresponding color threshold color.

VDcontrollers

Draws a combination of bar graphs and point graphs.

These display formatters can be used with or without range bars. If range bars are used, the first two variables supply the values for the range bars and subsequent variables provide the values for the graphs. Therefore, three variables are required when using range bars. If range bars are not used, all variables provide values for the graphs and only one variable is required. Range bars can be used a s a visual cue that the data is inside or outside a critical range. The data values used for range bars should be constants.

Each variable can have a separate color threshold table. If range bars are used, the first two color thresholds of every color threshold table are set equal to the range bar values and subsequent threshold values are ignored. If the range bars move, so do the color threshold values.

Each bar or symbol displays in a single solid color. If the variable value crosses a threshold value, the whole bar or symbol is redrawn in the new color.

Synopses

```
GLOBALREF DISPFORM VDcontroller;
GLOBALREF DISPFORM VDhorizcontroller;
```

Descriptions

VDcontroller	*DV-Draw Graph Type:* Controller		
Variable Shape: scalar, vector, matrix		Min Variables: 1	Max Variables: 12
History: No		Min Samples: 1	Max Samples: 1
Axis Types: Value (y), Time Ticks (range bars using first two variables)			

VDcontroller displays each variable as either a vertical bar or as a point, according to the variable's graph marker type. If the marker is null, the data value is represented by a vertical bar. If the marker is a symbol, the variable is represented by a marker with its center point at a vertical position proportional to the variable value.

VDhorizcontroller	*DV-Draw Graph Type:* Horizontal Controller		
Variable Shape: scalar, vector, matrix		Min Variables: 1	Max Variables: 12
History: No		Min Samples: 1	Max Samples: 1
Axis Types: Value (y), Time Ticks (range bars using first two variables)			

VDhorizcontroller displays each variable as either a horizontal bar or as a point, according to the variable's graph marker type. If the marker is null, the data value is represented by a horizontal bar. If the marker is a symbol, the variable is represented by a marker with its center point at a horizontal position proportional to the variable value.

VDdials

Dial display formatters is which the data values are represented by needles or hands pointing to the corresponding values. The color of the needle is determined by the color or color threshold table associated with the variable descriptor.

Synopses

```
GLOBALREF DISPFORM VDdial;
GLOBALREF DISPFORM VDdial360;
GLOBALREF DISPFORM VDhistdial;
```

Descriptions

VDdial, VDhistdial	*DV-Draw Graph Type:* Dial or Dial with History	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 5
History: No (Dial), Yes (Dial w Hist)	Min Samples: 1	Max Samples: 1
Axis Types: Horizontal (columns), Vertical (rows), Value Ticks (dial ticks), Value Tick Labels (digital value display), Time (iteration number)		

VDdial draws a dial encompassing 180 degrees, with the lowest value at the left and the highest value at the right. The needle points to the corresponding value.

VDdial360	*DV-Draw Graph Type:* Dial 360	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Horizontal (columns), Vertical (rows), Value Ticks (dial ticks), Value Tick Labels (digital value display), Time (iteration number)		

VDdial360 draws a dial encompassing 360 degrees. The variable is represented by two hands pointing to the corresponding value: the small hand codes the most significant digit, the large hand codes the second most significant digit. For example, if the data range is [0,999], a value of 550 is displayed by a large hand at 6 o'clock and a small hand halfway between 6 and 7 o'clock.

The data range maps to the circumference with zero at the top. The range should be from 0 to a power of 10.

VDdigit

Digital display formatter.

Synopses

```
GLOBALREF DISPFORM VDdigits;
```

Descriptions

VDdigits	*DV-Draw Graph Type:* Digits Graph	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 5
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Label (iteration number), Horizontal (columns), Vertical (rows)		

It is not necessary for all variables to have the same range.

VDdigits displays an array of numbers that displays the actual data in the variable. The digits are displayed in the largest text size that fits into the display area. Adding text dynamics to text objects can produce the similar results. You can justify the digits display by calling *VPdgdfargs* with the "Justify" argument, as shown in the example. The available options are *"Left," "Right,"* and *"Center."* The default is *"Center."* These arguments are case insensitive.

VDdigits uses the data variable range to determine the number of significant digits displayed using the following criteria:

three (sometimes, four) digits
the number of significant digits in the variable's minimum value
the number of significant digits in the variable's maximum value

For example:

If the range is:	It must allow at least:	If the range is:	It must allow at least:
[0,1]	4 digits	[0,1001]	4 digits
[0,10]	4 digits	[0,10001]	5 digits
[0,100]	3 digits	[0,100001]	6 digits

If the digits graph shares a variable with an input object, the range of the digits graph must match the range of the input variable.
The "C Format" option in the Edit Graph Menu lets you specify the C format for displaying your data. The conversion character must be preceded by a % sign. The conversion character conforms to the Ansi C standard for format conversion, except for *g*, *G*. Valid conversion characters and the type of data they indicate are:

s character string
c single character
f float, double, decimal notation
e, E float or double converted to scientific notation
g, G converts to e, E or f, depending on whether the graph allows for the number of decimals specified
d, i integer converted to decimal
o unsigned octal
u unsigned decimal
x, X unsigned hexadecimal
p address

You can only have one conversion character per format string.

When you specify a *g*, *G* format in the form *x.y*, *y* specifies the number of decimal places, not the total width of the field.

Other characters in your string appear as you enter them. These include *\n* for a newline, *\t* for a tab, and \

octal_digits for special characters.

This display formatter can display data that is outside the variable range.

Diagnostics

This formatter does not display more than six significant digits, so data precision is reduced if the range limits have more than six significant digits. Room is allowed to display six digits.

Example

This code fragment defines a format to be used by the digits formatter, and displays the digits left justified.

```
DATAGROUP dgp;
NAME_VALUE_PAIR arg[2];
arg[0].name = "Value Format";
arg[0].value = "%5.2f";        /* C format for digits */
arg[1].name = "Justify";
arg[1].value = "Left";
VPdgdfargs (dgp, &arg, 2);
```

VDdrawings

VDdrawing runs a view created using DV-Draw. *VDmovedrawing* rotates, scales, and moves a drawing. These display formatters are obsolete, but are provided for compatibility for applications that were developed using earlier releases. The needs addressed by these display formatters can now be handled by object dynamics and active subdrawings.

Synopses

```
GLOBALREF DISPFORM VDdrawing;
GLOBALREF DISPFORM VDmovedrawing;
```

Descriptions

VDdrawing — *DV-Draw Graph Type:* Dynamic Drawing

Variable Shape: scalar, matrix — Min Variables: 1 — Max Variables: unlimited

History: No — Min Samples: 1 — Max Samples: 1

Axis Types: None

VDdrawing binds a view's data source variables to the graph's variables in the order in which they appear. The view is then run in the graph's viewport. The data group title must be the filename of a view created using DV-Draw. Note that this display formatter is obsolete. You can achieve many of the same results by enabling the dynamics within a subdrawing.

If the view variables have default attributes such as color, line type, and symbol type, these attributes are replaced by the data group variable attributes. Non-default attributes are only replaced by data group variable attributes if the latter have non-default values. Defaults attributes are: single color, solid line, null symbol; non-default attributes are: color threshold table, patterned lines, non-null symbols.

The shapes of the variables must match the shapes of the variables in the view.

Unmatched variables are set to constants.

VDmovedrawing — *DV-Draw Graph Type:* Moving Drawing

Variable Shape: scalar — Min Variables: 1 — Max Variables: 4

History: No — Min Samples: 1 — Max Samples: 1

Axis Types: None

VDmovedrawing displays a subdrawing. The first variable determines the rotation angle between -180 and +180 degrees; the second variable determines the scale; the third variable determines the x position; and the fourth variable determines the y position. Note that this display formatter is obsolete. You can achieve many of the same results by adding motion dynamics to the subdrawing.

The data group title must be the filename of a drawing created using DV-Draw. The display formatter reads in the static part of the view, and positions it according to the variables, as described below:

Angle is determined by the first variable. The value variable is mapped from -180 degrees (measured clockwise from the zero-degree line) to +180 degrees. Thus, a value for the variable that is in the middle of its range is equivalent to an unrotated drawing.

Scale is determined by the second variable. The value is not normalized to its range before it is used so range is irrelevant for this variable. The drawing is scaled by the value of this variable. For example, if the variable value is 1.0, the drawing appears the same size as it was originally drawn in DV-Draw. If the scale is 2.0, the drawing size is doubled. A good way to set the size of a drawing is to attach the second variable to a constant and adjust the value of the constant until the drawing appears the correct size.

X and **Y** coordinates of the drawing's center point are determined by the third and fourth variables. The range of these variables maps to the entire range of the graph's viewport. This means the drawing can extend outside of the viewport. For the x coordinate, the minimum value is at the left edge of the viewport and the maximum value is at the right edge of the viewport. For the y coordinate, the minimum is at to the bottom edge of the viewport and the maximum is at to the top edge. To place the drawing in different portions of the viewport, you can adjust the ranges of the third and fourth variables. To make the drawing move in the correct area, you may need to adjust the scale factor in conjunction with the third and fourth variable ranges.

VDface

Face display formatter.

Synopses

```
GLOBALREF DISPFORM VDface;
```

Descriptions

VDface		*DV-Draw Graph Type:* Face Graph	
Variable Shape: scalar, vector, matrix		Min Variables: 1	Max Variables: 5
History: No		Min Samples: 1	Max Samples: 1
Axis Types:	Time Tick Label (iteration number), Value Tick Label (digital value display) Horizontal (columns), Vertical (rows)		

VDface displays stylized faces with eyes, eyebrows, and mouth. The greater the value, the more the corners of the mouth point up, the larger the eyes become, and the more the eyebrows rise. The lower the value, the more the corners of the mouth point down, the smaller the eyes become, and the more the eyebrows tilt down.

The color of the features is determined by the color or color threshold table associated with the variable.

Diagnostics

While this display formatter is similar to a Chernoff face in which multiple variables can be displayed using one variable per feature, it currently supports only one variable, using the entire face to reflect the variable value. Although five variables can be used, multiple variables display on top of each other, making the values difficult to distinguish.

VDfader

Fader display formatter.

Synopses

```
GLOBALREF DISPFORM VDfader;
```

Descriptions

VDfader	*DV-Draw Graph Type:* Fader	
Variable Shape: scalar	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: None		

VDfader displays the variable value in a format that resembles a stereo equalizer control. The position of the horizontal bar is proportional to the variable value.

The fader bar color is determined by the color or color threshold table of the variable.

VDfan

Fan display formatter.

Synopses

```
GLOBALREF DISPFORM VDfan;
```

Descriptions

VDfan	*DV-Draw Graph Type:* Fan Graph	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 2
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Labels (iteration number), Value Tick Labels (digital value display), Horizontal (columns), Vertical (rows)		

VDfan displays nested fans that open in a clockwise direction. A fan is a filled arc resembling a pie slice. The greater the data values, the larger the fan. The lowest value is an empty circle. The highest value shows a full circle in the colors of the variable. Intermediate values create shapes like pie pieces. Multiple variables display as fans superimposed on each other with decreasing radii.

The shape of the variable determines the number of fans displayed. Multiple variables display as fans superimposed on each other with decreasing radii.

The color of the fan is determined by the color or color threshold table associated with the variable.

VDhighlowopenclose

High-low-open-close display formatter.

Synopses

```
GLOBALREF DISPFORM VDhighlow;
```

Descriptions

VDhighlow	*DV-Draw Graph Type:* High Low	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 4
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDhighlow draws a high-low-open-close graph such as those used to display stock market data. the first two variables are displayed as a vertical line between the highest and lowest data values. If a third variable is used, it is displayed as a horizontal line marking the “close” value.

If only two variables are used, they determine the high and low values of the vertical line, and the second variable determines the value of the horizontal line. If only one variable is used, only a horizontal line appears.

If a fourth variable is used, it is displayed as a horizontal line marking the “open” values. In this case, the vertical line is located in the center of the time slot, with the two horizontal lines on either side.

The color of the vertical bar is determined by the color or color threshold table associated with the first variable.

The value axis is labeled using the range of the first variable only.

VDhorizon

Artificial horizon display formatter.

Synopses

```
GLOBALREF DISPFORM VDhorizon;
```

Descriptions

VDhorizon	*DV-Draw Graph Type:* Artificial Horizon	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 4
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Roll, Pitch		

VDhorizon draws a horizon line, runway, the representation of airplane wings, and a track circle within a 360-degree dial-shaped graph. Four variables can be used. Their values determine the roll, pitch, roll error, and pitch error respectively. Roll error and pitch error are optional.

The first variable value determines the **roll** angle. The roll value is represented by rotation of the horizon, sky, runway, pitch axis, and a red arrowhead indicator. A positive roll value rotates these objects counter-clockwise; a negative value rotates them clockwise.

If the range of the roll variable values is smaller than -180 to 180, the tick labels are limited correspondingly. If the range is greater than -180 to 180, values wrap around the dial. For example, a value of 240 appears as -120. Values outside the range are clipped to the range limits.

The second variable determines the **pitch** angle. The pitch value is represented by the position of the horizon with respect to the pitch axis. Positive values move the horizon down the scale; negative values move it up.

The maximum pitch value is mapped to the bottom of the scale (100% sky and 0% ground); the minimum pitch value is mapped to the top of the scale (0% sky and 100% ground) with zero at the mid-point. Tick marks are drawn along the pitch axis.
The following *VPdgcontext* flags control the roll and pitch axis ticks and tick labels. To display, set the flag value to *YES*; to turn the ticks or tick labels off, set the flag value to *NO*.

Roll axis ticks:	*V_FROLL_TICS*
Roll axis tick labels:	*V_FROLL_LABEL_TICS*
Pitch axis ticks:	*V_FPITCH_TICS*
Pitch axis tick labels:	*V_FPITCH_LABEL_TICS*

The third variable value determines the **roll error**. The roll error value is represented by a short line perpendicular to the dial's horizontal axis. The roll error variable uses the range of the roll variable. The range is mapped to the horizontal axis, with zero in the center. Positive values move the line proportionally to the left; negative values move it to the right.

The fourth variable value determines the **pitch error**. The pitch error value is represented by a short line perpendicular to the dial's vertical axis. The pitch error variable uses the range of the pitch variable. The range is mapped to the vertical axis, with zero in the center. Positive values move the line down proportionally, and negative values move it up.

If a variable range is not symmetrical around zero, this display formatter interprets it as if it were, using the larger absolute value for both the positive and negative limits. For example, a variable range of -45 to 90 is interpreted as -90 to 90.

VDindicator

Indicator displaying current variable value as a marker.

Synopses

```
GLOBALREF DISPFORM VDindicator;
```

Descriptions

VDindicator	*DV-Draw Graph Type:* Indicator	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (y), Value (x)		

The horizontal position of the marker is proportional to the variable value. The vertical position of each marker represents a new time sample, not a spatial (x,y) value. Multiple variables overlap in the same slot space. If the graph cannot display all the samples at the same time, the markers wrap around from top to bottom.

VDknob

Knob display formatter.

Synopses

```
GLOBALREF DISPFORM VDknob;
```

Descriptions

VDknob	*DV-Draw Graph Type:* Knob	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Value Tick Labels (digital value display) Horizontal (columns), Vertical (rows)		

VDknob draws a knob with a 270 degree travel. The lowest value is in the lower left, the highest in the upper right.

The knob color is determined by the color or color threshold table associated with the variable. On a monochrome display, the knob color does not change if a color threshold table is associated with the variable.

The object foreground color determines the color of the background panel. If the value axis ticks are “on” to show value markings around the rim of the knob, the object foreground color must contrast well with black to make the markings visible.

See Also

VDmeter

VDlegend

Legend display formatter.

Synopses

```
GLOBALREF DISPFORM VDlegend;
```

Descriptions

VDlegend	*DV-Draw Graph Type:* Legend Graph	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 20
History: No	Min Samples: 1	Max Samples: 1
Axis Types: None		

VDlegend draws a legend listing the name and color threshold table of each variable attached to the graph. The legend is static; it is drawn once and is not updated while running. The legend appears as a centered column in the viewport. *VDlegend* scales the legend to fit into the viewport. There is no context except the outline.

See Also

VDtext

VDlines

Line display formatters.

Synopses

```
GLOBALREF DISPFORM VDline;
GLOBALREF DISPFORM VDlinedist;
GLOBALREF DISPFORM VDlinefill;
GLOBALREF DISPFORM VDlinefstacked;
GLOBALREF DISPFORM VDlinestacked;
GLOBALREF DISPFORM VDstep;
```

Descriptions

The line formatters draw a line graph for each variable, starting at the left edge of the graph. Each time these display formatters are invoked, they put the next data value into the next available slot. When the data area fills up, the display wraps around to the beginning of the data viewport or scrolls left, depending on the value set by *VPdgscroll_amount*. If the scroll amount is greater than zero, the graph scrolls to the left. The default is to wrap around to the beginning.

The value axis displays the range of the first variable.

The time and value grids are supported.

VDline	*DV-Draw Graph Type:* Line Graph	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDline displays a simple line graph

The line color is determined by the color or color threshold table associated with the variable.

Different line types can be assigned to different variables to make it easier to distinguish between them.

VDlinedist	*DV-Draw Graph Type:* Filled Line Distribution	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDlinedist displays a filled line graph in which the range of the graph equals the range of the attached variables.

All variables must have the same range. The range of the graph equals the range of the attached variables. The sum of the variable values for each sample should not exceed the maximum range value. For example, if a graph had three variables with a range of 0 to 10, the range of the graph is 0 to 10. A given sample of the three variables might have values of 2, 3, and 5 or 2, 3, and 4 but not 2, 3, and 6, since the sum is greater than the range of the graph.

If the minimum range value is not zero, the value axis ticks are only accurate for reading the total of the variables, at the top line of the filled line graph.

See also *VDlinefill*.

VDlinefill	*DV-Draw Graph Type:* Filled Line	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDlinefill displays a line graph for each variable and fills below the line with the variable color. The line graphs are stacked vertically, adding each variable value to the sum of the values beneath it. The first variable is on the bottom, the last variable on the top. The height of the line graph is proportional to the sum of the values of all the variables.

The color of the area below each line is determined by the color associated with the variable. If the variable has a color threshold table, the area is divided into sections of different colors to match the threshold table.

Variables do not need to have the same range. If the minimum range value is not zero, the value axis ticks are only accurate for reading the total of the variables, at the top line of the filled line graph.

VDlinefstacked	*DV-Draw Graph Type:* Stacked Filled Line Graph		
Var Shape: scalar, vector, matrix		Min Variables: 1	Max Variables: 10
History: Yes		Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)			

VDlinefstacked displays each variable as a Filled Line Graph, stacking each graph above the previous one.

The value axis of each graph is displayed on alternate sides of the graphs, starting at the left side of the bottom graph.

This display formatter displays a single title and a single legend for the stack of graphs. The legend lists all of the variables in the stack of graphs.

VDlinestacked	*DV-Draw Graph Type:* Stacked Line Graph	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 16
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDlinestacked: displays each variable as a Line Graph, stacking each graph above the previous one.

The value axis of each graph is displayed on alternate sides of the graphs, starting at the left side of the bottom graph.

This display formatter displays a single title and a single legend for the stack of graphs. The legend lists all of the variables in the stack of graphs.

Different line types can be assigned to the variables to make it easier to distinguish between them.

VDstep	*DV-Draw Graph Type:* Step Graph	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDstep displays each variable element as a horizontal line connected to the adjacent values by vertical lines. Different line types can be assigned to different variables to make it easier to distinguish between them.

Each horizontal line is plotted together with the following vertical line. Since the vertical line cannot be plotted until the next value is known, values are plotted with a delay of one time slot.

Diagnostics

Buffering the data so the buffered dimension equals the number of slots updates the display most efficiently. For additional information, see *VPvddim*. For example:

```
VPvddim (vdp, 10, 1, 1);
VPdgslots (dgp, 10);
```

See Also

VDlinefill, *VDstrip*

VDmeter

Meter display formatter.

Synopses

```
GLOBALREF DISPFORM VDmeter;
```

Descriptions

VDmeter	*DV-Draw Graph Type:* Meter	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 1
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Horizontal (columns), Vertical (rows), Value Ticks (ticks around meter), Value Axis Labels (labels on ticks)		

VDmeter draws a simulated meter with the lowest value at the left and the highest value at the right. The variable is represented by a needle pointing to the corresponding value. The meter is similar to the dial, but uses a logarithmic scale mapped to a 120 degree arc.

When the number of samples is greater than one, a dot appears at the tip of the meter needle. As the value changes, the graph leaves the dot of each value as a history of the values. The slot count specifies the number of dots displayed.

The needle color is determined by the color or color threshold table associated with the variable.

See Also

VDdial, *VDhistdial*, *VDknob*

VDpie

Pie chart display formatter.

Synopses

```
GLOBALREF DISPFORM VDpie;
```

Descriptions

VDpie		*DV-Draw Graph Type:* Pie Chart
Variable Shape: scalar	Min Variables: 1	Max Variables: 10
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Labels (iteration number), Value Tick Labels (displays digital value inside slice)		

VDpie is a standard pie chart that plots the ratios of several different variables. This display only makes sense if more than one variable is associated with the data group. If value labeling is turned on, each pie slice is labeled with the percentage of the total value corresponding to that variable's value. The routine totals values for all the variables, and displays a pie slice of a size proportional to the ratio:

variable_value : total_value

Each pie slice color is determined by the color or color threshold table associated with the variable.

Labels use the current foreground color of the formatter.

VDpoint

Point graph display formatter.

Synopses

```
GLOBALREF DISPFORM VDpoints;
```

Descriptions

VDpoints	*DV-Draw Graph Type:* Points Chart	
Variable Shape: scalar	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDpoints displays a points graph with wrap-around. The graph starts at the left boundary of the first slot and stops at the right boundary of the last slot, so there *n* points are plotted before wrap-around, where *n* is the number of slots. The height of the point is proportional to the value of the variable being plotted. If the variable has a marker associated with it, that marker is used to display the data.

Each time the display formatter is invoked it puts the next data value into the next available slot. When the data area fills up, the graph wraps around to the beginning of the data viewport or scrolls left, depending on the value set by *VPdgscroll_amount*. If the scroll amount is greater than zero, the graph scrolls to the left. The default is to wrap around to the beginning.

The value axis is labeled using the range of the first variable only.

The time and value grids are supported.

Each marker color is determined by the color or color threshold table associated with the variable.

VDprimitives

Display formatters that use an array of primitive shapes, changing their size and color to reflect data values.

Synopses

```
GLOBALREF DISPFORM VDbox;
GLOBALREF DISPFORM VDcircle;
GLOBALREF DISPFORM VDtriangle;
```

Descriptions

These display formatters provide no context except for the outline.

The shape of the variable determines the number of primitives displayed. If the variable is scalar, the formatter draws the largest shape possible in the specified viewport.

The first variable determines the color. The color of each shape is determined by the color or color threshold table associated with the variable.

Remaining variables determine the dimensions of the primitive. The maximum value of a dimension variable produces the largest size possible. If a variable is missing, the maximum value is used in its place.

VDbox
Variable Shape: scalar, vector, matrix
History: No
Axis Types: None

DV-Draw Graph Type: Box
Min Variables: 1 Max Variables: 3
Min Samples: 1 Max Samples: 1

VDbox draws a rectangle using up to three variables. The first variable determines the color of the rectangle; the second variable determines the width; and the third variable determines the height.

VDcircle
Variable Shape: scalar, vector, matrix
History: No
Axis Types: None

DV-Draw Graph Type: Circle
Min Variables: 1 Max Variables: 2
Min Samples: 1 Max Samples: 1

VDcircle draws a circle using up to two variables. The first variable determines the color and the second determines the radius of the circle.

VDtriangle
Variable Shape: scalar, vector, matrix
History: No
Axis Types: None

DV-Draw Graph Type: Triangle
Min Variables: 1 Max Variables: 3
Min Samples: 1 Max Samples: 1

VDtriangle draws a primitive triangle using up to three variables. The first variable determines the color; the second determines the width of the triangle at its base; and the third determines the height of the triangle.

VDradials

Radial strip chart display formatters.

Synopses

```
GLOBALREF DISPFORM VDne_radial;
GLOBALREF DISPFORM VDradial;
```

Descriptions

Radial formatters plot a line graph in polar coordinates. The graph starts at the 3 o'clock position and moves counter-clockwise.

The variable value is mapped to the distance from the center shape to the outside of the circle, with the maximum value at the outside.

The number of time slots is mapped to the circumference of the circle and formatter plots that number of points per revolution, connecting the points with linear arcs. A linear arc is a linear function in polar coordinates, which is a function of the form:

```
radius = constant * angle + constant2
```

The line color is determined by the color or color threshold table associated with the variable.

The value axis displays the range of the first variable.

VDne_radial	*DV-Draw Graph Type:* Radial Graph, no erase	
Variable Shape: scalar	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Tick Labels (iteration number), Value (y)		

VDne_radial (ne = no erase) does not erase previous values as it wraps around, plotting new values together with old values. This plots faster than *VDradial*. It is useful for cyclic data.

If the display formatter is redrawn by *TdpRedraw*, *TscRedraw*, or any other method of redrawing, only the most recent number of time slots specified by *VGdgslots* are redrawn. Previous values are not preserved.

VDradial	*DV-Draw Graph Type:* Radial Graph	
Variable Shape: scalar	Min Variables: 1	Max Variables: 10
History: No	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Tick Labels (iteration number), Value (y)		

VDradial erases old data in each time slot when it wraps around to that time slot again.

VDscatters

Scatter plot display formatters

Synopses

```
GLOBALREF DISPFORM VDimpulse;
GLOBALREF DISPFORM VDimpulseto0;
GLOBALREF DISPFORM VDscatter;
```

Descriptions

For each pair of variables, these formatters plot a marker whose x coordinate is the value of the first variable and whose y coordinate is the value of the second variable. These formatters use an even number of variables; unpaired variables are ignored. If either value in a variable pair is out of range, the marker falls outside the data viewport and is not drawn. In the impulse graphs, if a point is above the range, the marker is not drawn, but the vertical line is drawn from the horizontal axis to the top of the data viewport. If a point is below the given range, the marker is not drawn. In the impulse graph, no line is drawn; in the impulse to zero graph, the line is drawn from the horizontal axis to the bottom of the data viewport.

The legends, markers, and vertical lines (if used in the impulse graphs) use the color associated with the second variable of each pair. If the variable has a color threshold table, the color is determined by the variable value and the corresponding color in the threshold table. Vertical lines are divided into sections of different colors according to the variable value and the corresponding color in the threshold table.

This display formatter displays the x and y value axes. The time value appears as a numerical value centered below the value axis.

Only the value grid is supported.

VDimpulse	*DV-Draw Graph Type:* Impulse Graph	
Variable Shape: scalar	Min Variables: 2	Max Variables: 20
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Tick Labels (iteration number), Value (x=first variable range, y=second variable range)		

VDimpulse plots a scatter plot with vertical lines from each point to the horizontal axis.

VDimpulseto0	*DV-Draw Graph Type:* Impulse to Zero	
Variable Shape: scalar	Min Variables: 2	Max Variables: 20
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Tick Labels (iteration number), Value (x=first variable range, y=second variable range)		

VDimpulseto0 plots a scatter plot with a vertical line from each marker to the zero line.

VDscatter	*DV-Draw Graph Type:* Scatter Plot	
Variable Shape: scalar	Min Variables: 2	Max Variables: 20
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time Tick Labels (iteration number), Value (x=first variable range, y=second variable range)		

VDscatter plots a scatter plot.

Diagnostics

To update the display most efficiently, set the size of the variable descriptor buffer equal to the number of slots. For example:

```
VPvddim (vdp, 10, 1, 1);
VPdgslots (dgp, 10);
```

See Also

VDweb, VPvddim

VDsize

Size display formatter.

Synopses

```
GLOBALREF DISPFORM VDsize;
```

Descriptions

VDsize	*DV-Draw Graph Type:* Size Graph		
Variable Shape: scalar, vector, matrix		Min Variables: 1	Max Variables: 3
History: No		Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Labels (iteration number), Value Tick Labels (digital value display), Horizontal (columns), Vertical (rows)			

VDsize displays up to three variables as sets of geometric shapes whose sizes change as the variable values change. The first variable appears as an unfilled rectangle, the second as an unfilled diamond superimposed on the rectangle, and the third as an unfilled star superimposed on the diamond and rectangle. The default shapes can be replaced by associating markers with the variables. If there is only one variable, the geometrical shape is a filled rectangle.

If the value tick labels are on, the data values are displayed digitally directly above the shape sets.

The color of each shape is determined by the color or color threshold table associated with the variable.

VDspectros

Displays a colored bar for each sample of a vector variable. The bar is divided vertically into the number of elements of the variable and each region of the bar is colored to reflect the value of the element according to the color threshold table. If the variable is scalar, each bar is a single solid color. You must use one of the stacked display formatters if you are displaying more than one variable. The interpolated display formatters display gradual transitions between the color regions.

Synopses

```
GLOBALREF DISPFORM VDspectro;
GLOBALREF DISPFORM VDspectrointp;
GLOBALREF DISPFORM VDspectrointpstkd;
GLOBALREF DISPFORM VDspectrostacked;
```

Descriptions

These display formatters work best with a vector variable that has a color threshold table. Matrix data is not meaningful with this display formatter.

The legend is a color bar that shows the colors corresponding to the threshold values. The variable values are mapped uniformly to the axis of the color bar, not only to the threshold values.

The data for the first sample appears in the leftmost slot of each graph. When the data area fills up, the graph wraps around to the beginning of the data viewport or scrolls left, depending on the value set by *VPdgscroll_amount*. If the scroll amount is greater than zero, the graph scrolls to the left. The default is to wrap around to the beginning.

The vertical axis displays the numbers of the elements in a sample. For example, the vertical axis of a vector variable with a length of 8 has values from 1 to 8. The tick marks appear at the center of each element's height. The value of each element is indicated by the color of the rectangle, not by its vertical position. This axis is called the value axis in DV-Draw for compatibility with previous releases, but is actually the first spatial axis. To set this axis label using DV-Tools, you must call *VPdgaxlabel* using the *V_FIRST_AXIS* flag instead of calling *VPvdvallabel*.

The maximum length of a vector variable is 250.

VDspectro	*DV-Draw Graph Type:* Spectro Graph	
Variable Shape: scalar, vector	Min Variables: 1	Max Variables: 1
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x), Vertical (y=number of elements in sample)		

VDspectro displays a color bar for a single vector variable. The legend color bar shows the color threshold table and range of the variable.

VDspectrointp	*DV-Draw Graph Type:* Smoothed Spectro	
Variable Shape: scalar, vector	Min Variables: 1	Max Variables: 1
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x), Vertical (y=number of elements in sample)		

VDspectrointp displays a color bar with interpolated color transitions between the elements in the vector variable and between the samples. The color transitions are drawn between neighboring values using the color thresholds.

The legend color bar shows the color threshold table and range of the variable.

This display formatter uses raster operations to interpolate the data, and therefore should not be obscured. It clips correctly only when the scroll amount is 0. If the display device does not support rasterops, the display formatter behaves like *VDspectro* and there is no interpolation.

VDspectrointpstkd	*DV-Draw Graph Type:* Stacked Smoothed Spectro	
Variable Shape: scalar, vector	Min Variables: 1	Max Variables: 16
History: Yes	Min Samples: 1	Max Samples: unlimited

Axis Types: Time (x), Vertical (y=number of elements in sample)

VDspectrointpstkd displays each variable as an Interpolated Spectro Graph, stacking each graph above the previous one. Each graph displays an interpolated colored bar for each data sample of a vector variable. The color transitions are drawn between neighboring values using the color thresholds.

The vertical axis of each graph is displayed on alternate sides of the graphs, starting at the left side of the bottom graph.

This display formatter displays a single title and legend for the stack of graphs.

The legend color bar shows the color threshold table and range of the last variable. Since there is only one legend color bar for all variables, the variable of each graph should have the same color threshold table.

This display formatter uses raster operations to interpolate the data, and therefore should not be obscured. It clips correctly only when the scroll amount is 0. If the display device does not support rasterops, the display formatter behaves like *VDspectrostacked* and there is no interpolation.

VDspectrostacked	*DV-Draw Graph Type:* Stacked Spectro	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 16
History: Yes	Min Samples: 1	Max Samples: unlimited

Axis Types: Time (x), Vertical (y=number of elements in sample)

VDspectrostacked displays each variable as a Spectro Graph, stacking each graph above the previous one. Each graph displays a colored bar for each data sample of a vector variable.

The vertical axis of each graph is displayed on alternate sides of the graphs, starting at the left side of the bottom graph.

This display formatter displays a single title and legend for the stack of graphs.

The legend color bar shows the color threshold table and range of the last variable. Since there is only one legend color bar for all variables, the variable of each graph should have the same color threshold table.

VDstrips

Strip chart display formatters.

Synopses

```
GLOBALREF DISPFORM VDstrip;
GLOBALREF DISPFORM VDstripras;
GLOBALREF DISPFORM VDstripstacked;
GLOBALREF DISPFORM VDvstrip;
GLOBALREF DISPFORM VDvstrip_r;
GLOBALREF DISPFORM VDwaterfall;
GLOBALREF DISPFORM VDwaterfall_r;
```

Descriptions

The value axis displays the range of the first variable.

The time and value grids are supported.

The number of samples specifies the number of history values displayed.

The line color is determined by the color or color threshold table associated with the variable.

Strip charts are slower than line graphs because they redraw the entire plot and time axis for each sample. To increase the speed, turn the time axis label off or use a raster version of the strip chart.

The raster versions take and display raster images to scroll the data, and therefore should not be obscured or partially clipped. Using raster images makes the raster versions more efficient than non-raster strip charts, which redraw all the data before plotting each sample. If the display device does not support rasterops, the raster versions behave like the non-raster versions and there is no improvement in efficiency.

Plotting strip charts overloads the plotter. Before you send a strip chart to the plotter, convert it to a line chart.

VDstrip	*DV-Draw Graph Type:* Strip Chart	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDstrip displays a scrolling line graph. *VDstrip* always puts the most recent value at the right end of the display area, moving the older data points to the left.

A more efficient strip chart can be created by using a line graph with a scroll amount of 1 or more. For additional information, see *VDline*.

VDstripras	*DV-Draw Graph Type:* Raster Strip Chart	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 3	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDstripras displays a scrolling line graph, taking a raster image of current data and shifting the image before plotting each new sample. *VDstripras* always puts the most recent value at the right end of the display area, moving the older data points to the left.

VDstripstacked	*DV-Draw Graph Type:* Stacked Strip Chart	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 16
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDstripstacked displays each variable as a Strip Chart, stacking each graph above the previous one. Each graph plots a line graph that begins at the right edge of the graph and scrolls toward the left of the graph. The most recent value appears at the right edge of the graph and the history shifts continually to the left.

The value axis of each graph is displayed on alternate sides of the graphs, starting at the left side of the bottom graph.

This display formatter displays a single title and a single legend for the stack of graphs. The legend lists all of the variables in the stack of graphs.

Different line types can be assigned to different graphs to make it easier to distinguish between them.

VDvstrip	*DV-Draw Graph Type:* Vertical Strip Chart	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDvstrip displays a line graph that scrolls up. *VDvstrip* always puts the most recent value at the bottom of the display area, moving the older data points to the top. The time axis is displayed on the left side of the graph and the value axis is displayed at the bottom.

VDvstrip_r	*DV-Draw Graph Type:* Vertical Raster Strip Chart	
Var Shape: scalar, vector, matrix	Min Variables: 2	Max Variables: 10
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: Time (x) vs Value (y)		

VDvstrip_r displays a line graph that scrolls up, taking a raster image of current data and shifting the image before plotting each new sample. *VDvstrip_r* always puts the most recent value at the bottom of the display area, moving the older data points to the top. The time axis is displayed on the left side of the graph and the value axis is displayed at the bottom.

VDwaterfall	*DV-Draw Graph Type:* Waterfall	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (y) vs Value (x)		

VDwaterfall displays a line graph that scrolls down. *VDwaterfall* always puts the most recent value at the top of the display area, moving the older data points to the bottom. The time axis is displayed on the left side of the graph and the value axis is displayed at the bottom.

VDwaterfall_r	*DV-Draw Graph Type:* Raster Waterfall	
Var Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 10
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (y) vs Value (x)		

VDwaterfall_r displays a line graph that scrolls down, taking a raster image of current data and shifting the image before plotting each new sample. *VDwaterfall_r* always puts the most recent value at the top of the display area, moving the older data points to the bottom. The time axis is displayed on the left side of the graph and the value axis is displayed at the bottom.

VDsurface

Three-dimensional surface graph.

Synopses

```
GLOBALREF DISPFORM VD3dsurface;
```

Descriptions

VD3dsurface	*DV-Draw Graph Type:* Surface Graph	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 1
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Value (y), Time Tick Label (iteration number)		

VD3dsurface displays a three-dimensional surface with the hidden lines removed.

The grid represents the data array positions; the position of each surface point above the grid corresponds to the element's location in the data array. The height of a point on the surface is proportional to the data value. The origin is in the lower right corner.

This display formatter works best with matrix data. A scalar variable plots as a plane.

The color of the surface lines is determined by the color or color threshold table associated with the variable.

VDtexts

Displays the contents of one or more text variables, adding each successive iteration of strings below the previous strings.

Synopses

```
GLOBALREF DISPFORM VDmessage;
GLOBALREF DISPFORM VDtext;
```

Descriptions

VDmessage	*DV-Draw Graph Type:* Message Graph	
Variable Shape: text, scalar	Min Variables: 1	Max Variables: 18
History: Yes	Min Samples: 1	Max Samples: unlimited
Axis Types: None		

VDmessage can only display text variables. To display numerical data, the data must be in text format. Non-text variables can be used to control aspects of the message display.

Multiple text variables display side by side. The first iteration of all text variables appear on the first line, the second iteration on the second line, etc. To separate entries on the same line, space must be included between items in the text files.

The number of samples specifies the number of text values in the current sampling.

The first scalar variable specifies which iteration of text values to display at the top of the graph from among the current sampling. If the scalar value is more than 1, each text value appears at the top of the graph then scrolls off the top until the specified iteration is displayed. That iteration remains at the top of the graph and the remaining values in the current sampling appear below it.

The first scalar variable can be used to scroll backward in the list, especially if you use an input object to control the iteration number by connecting it to the first scalar variable. The range of the input object should be equivalent to the number of samples specified.

If the graph is not large enough to display all the samples, it only displays enough samples to fill the graph. To make the graph scroll upward to display the latest iterations, use a scalar value of -1. In this case, the scalar value does not control which text value appears at the top of the graph, but only makes the text values scroll up with every iteration after the specified number of samples is displayed.

The second scalar variable controls the text size. The text size variable should be a constant. If it is not a constant, the graph uses only the first value to determine text size. The text size value must be in the range of 1 to 4, with 1 representing the smallest text size and 4 representing the largest.

This display formatter can use a maximum number of 16 text variables and 2 numerical variables.

VDtext	*DV-Draw Graph Type:* Text	
Variable Shape: text, scalar	Min Variables: 1	Max Variables: 2
History: No	Min Samples: 1	Max Samples: 1
Axis Types: None		

VDtext displays text in the center of the viewport.

To display dynamic text, the first variable must be a text variable. A second variable of any type can be added to determine the text color. If you only use a text variable, the text appears in an arbitrary color.

If the first variable is not a text variable, only the graph title appears, centered in the viewport.

If the first variable is a text variable, the title is justified in the upper left corner of the viewport, and the text from the text variable is vertically centered along the left edge.

The graph title uses the graph foreground color.

VDtime

GraphsVDtime with time-stamped values.

Synopses

```
GLOBALREF DISPFORM VDrtline;
GLOBALREF DISPFORM VDrtstep;
```

Descriptions

VDrtline, VDrtstep	*DV-Draw Graph Type:* see routines listed below	
Variable Shape: scalar, vector	Min Variables: 3	Max Variables: 102
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time (x) *(first two variables)* vs Value (y)		

These display formatters draw either a line or stacked step graph using a time axis that displays a day counter and time stamp.

The first variable displays as a day counter; the second variable displays as a time stamp representing the time elapsed since the beginning of the day in tenths of milliseconds. The first two variables must be in binary *ULONG* format. Subsequent variables can be in any DataViews data format and are plotted as lines. Up to ten variables can be displayed as data.

Typically the data used in this graph has already been collected; the time stamp data represents the times when data was taken rather than the current system time.

The first two variables must both have values that only increase *or* only decrease. Values for the time stamp (the second variable) need not represent regular intervals; the time axis is labeled in regular intervals regardless.

The real-time graphs display data differently from other graphs that display time series data such as bar charts and line graphs. Instead of displaying one data value per slot, the real-time graphs plot the data at the proper place along the time axis based on the value of the time stamp (the second variable). Since each sample of a data variable is paired with the corresponding time stamp, the horizontal gap between data values can vary. Multiple data points can even be plotted at the same point in time if the same time stamp value occurs more than once.
Because of this different approach to plotting data, some features of the real-time graphs are controlled differently from those of other graph types:

- The time span displayed along the time axis is controlled by a variable range, not by the slot count.
- The scroll amount is controlled by the slot count and the scroll amount.
- The format for the time axis tick labels is controlled by a variable range.
- The number of data points redrawn after an expose event is controlled by the slot count.

Time span. The range of the second variable controls the span of time displayed along the time axis. The basic unit is a tenth of a millisecond. For example, a range of [0,100] displays 100 tenths of milliseconds in 10 intervals of 10 milliseconds each. A range of [0,50] displays 50 tenths of milliseconds in 5 intervals of 10 milliseconds each.

Scroll amount. The graph scrolls only when it must make room for new time stamp data. The slot count and scroll amount determine the amount scrolled. For example, if the slot count is 20 and scroll amount is 4, the graph scrolls 20% of the time axis. To eliminate scrolling, make the scroll amount greater than the slot count. In this case, all old data is erased at once and the new data is drawn starting at the left.

You can think of the slot count as an estimate of the number of data points that will be displayed in the time span. Then the scroll amount specifies the estimated number of data points to scroll by.

Time axis tick labels. The range of the second variable also controls the format for the time axis tick labels. For example, a range of [0,100] displays time axis labels in the format SS.TTT.T (*seconds.milliseconds.tenths of milliseconds*). A range of [0,10000] displays time axis labels in the format MM:SS.TTT (*minutes:seconds.milliseconds*). A range of [0,1000000] displays time axis labels in the format HH:MM:SS (*hours:minutes:seconds*).

Data points plotted on an expose. The slot count controls the number of data points that can be redisplayed on an expose event. However, some data may be lost on the redisplay if the graph was displaying more data points than estimated in the slot count.

Display direction. The real-time graphs let you reverse the direction of the data display. Note that the time stamps must correspond to the graph direction, so if you change the direction of the graph, the time stamps must reverse direction at the same time. Time stamps must be increasing whenever the graph is going forward, and must be decreasing whenever the graph is going backward.
To reverse the direction, send the *VDTIME_CHANGE_DIRECTION* flag to the graph using *VPdgdfmessage*. For example:

```
#define VDTIME_CHANGE_DIRECTION  1
VPdgdfmessage (dgp, VDTIME_CHANGE_DIRECTION, NULL);
```

You can send this message before the first call to *TdpDraw*.

To change the direction back again, repeat the call to *VPdgdfmessage*.

Changing direction resets the graph and all history is lost.

Units per second. The real-time graphs let you change the number of units per second to match your data resolution.
To change the units per second, send the *VDTIME_UNITS_PER_SECOND* flag to the graph using *VPdgdfmessage*. For example:

```
#define VDTIME_UNITS_PER_SECOND  2
VPdgdfmessage (dgp, VDTIME_UNITS_PER_SECOND, (ULONG) value);
```

where *value* represents the new number of units per second. Values that work best include 10, 100, 10,000. The default is 10,000. When you change the number of units per second, you do not have to change the range of the second variable, which controls the span of time displayed along the time axis; this is handled internally by the graph.

You can send this message before the first call to *TdpDraw*.

VDrtline draws a line graph with a time axis that displays a day counter and time stamp. The corresponding DV-Draw graph type is *Real Time Line Graph*.

VDrtstep displays each variable element as a step graph, stacking each graph above the previous one. The time axis displays a day counter and time stamp. The corresponding DV-Draw graph type is *Real Time Step Graph*.

VDvectors

Vector plot display formatters.

Synopses

```
GLOBALREF DISPFORM VDflowfield;
GLOBALREF DISPFORM VDvector;
```

Descriptions

Any variables not specified are set to zero.

For the angles to be meaningful, the variable ranges should be symmetrical around zero.

The color of each vector is determined by the color or color threshold table associated with the last variable.

VDflowfield	*DV-Draw Graph Type:* Flowfield	
Variable Shape: scalar, vector, matrix	Min Variables: 3	Max Variables: 5
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Labels (iteration number), Value (x=first variable range, y=second variable range)		

VDflowfield displays up to five variables as points, each with a vector attached. A minimum of three variables is required to supply the x and y coordinates of the points and the length of the vectors. The first variable provides the x coordinate of each point; the second provides the y value. The third variable provides the x component of the vector, and the forth variable, if used, provides the y component of the vector. Each vector is drawn with its corresponding plotted point as its origin. The fifth variable, if used, provides the z component of the vector. The z component, if used, is displayed as color changes using the color threshold table of the fifth variable.

The third, fourth, and fifth variables should all have the same range.

For more information about the component display formatters, see *VDscatter* and *VDvector*.

VDvector	*DV-Draw Graph Type:* Vector Graph	
Variable Shape: scalar, vector, matrix	Min Variables: 1	Max Variables: 3
History: No	Min Samples: 1	Max Samples: 1
Axis Types: Time Tick Label (iteration number), Horizontal (columns), Vertical (rows)		

VDvector plots a three-dimensional vector field. The origin of each vector is constant. For each vector, the first variable provides the x component, the second variable provides the y component, and the third variable provides the z component. The z component is represented by the color of the line.

See Also

VDscatter

VDwebs

Scatter plot display formatter with points connected.

Synopses

```
GLOBALREF DISPFORM VDweb;
GLOBALREF DISPFORM VDmultiyweb;
```

Descriptions

The value axis displays the range of the second variable.

Only the value grid is supported.

The legend and marker use the color associated with the second variable of each pair. If the variable has a color threshold table, the color is determined by the variable value and the corresponding color in the threshold table.

Different markers can be assigned to different variables to make it easier to distinguish between them.

VDmultiyweb	*DV-Draw Graph Type:* Multiple-Y Web	
Variable Shape: scalar, vector, matrix	Min Variables: 2	Max Variables: 20
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time Tick Label (iteration number), Value (x=first variable range, y=second variable range)		

VDmultiyweb draws a Scatter Plot with lines connecting each point to the adjacent points and multiple vertical value axes. For each pair of variables, the graph plots a marker whose x coordinate is the value of the first variable and whose y coordinate is the value of the second variable. This graph uses an even number of variables; unpaired variables are ignored.

The y axis is displayed for each variable pair. The values are determined by the second variable of each pair. The axis color matches the color of the variable. The axis is displayed for every variable pair even if the variable range is not unique.

Each y axis is labeled with the name of the second variable in the pair. If the second variable in any pair has been given a null name using *VPvdvarname*, and a vertical axis label has been assigned using *VPdgaxlabel*, the vertical axis label is used to label the whole set of vertical axes. Normally this vertical axis label is ignored.

Different line types can be assigned to different variables to make it easier to distinguish between them.

This display formatter displays the x and y value axes. The time value appears as a numerical value centered below the value axis.

VDweb	*DV-Draw Graph Type:* Web Chart	
Variable Shape: scalar	Min Variables: 2	Max Variables: 20
History: Yes	Min Samples: 2	Max Samples: unlimited
Axis Types: Time Tick Label (iteration number), Value (x=first variable range, y=second variable range)		

VDweb displays a Scatter Plot with lines connecting each point to the adjacent points. For each pair of variables, the graph plots a marker whose x coordinate is the value of the first variable and whose y coordinate is the value of the second variable. This graph uses an even number of variables; unpaired variables are ignored.

This display formatter displays the x and y value axes. The time value appears as a numerical value centered below the value axis.

Diagnostics

To update the display most efficiently, set the size of the variable descriptor buffer equal to the number of slots. For example:

```
VPvddim (vdp, 10, 1, 1);
VPdgslots (dgp, 10);
```

See Also

VDscatter, *VPvddim*, DataViews Technical Note #4, *Using Vector and Flowfield Formatters*.

VG Routines

Vg Routines

Routines that get information from data group and variable descriptor data structures.

VG Modules

```
#include "std.h"
#include "dvstd.h"
#include "VGfundecl.h"
```

VGdg	Gets basic information from a data group.
VGdgcolor	Gets the color information from a data group.
VGdgcontext	Gets the context information from a data group.
VGdgdf	Gets information related to the display formatter from a data group.
VGdgdfargs	Gets the display formatter arguments to a data group.
VGdgvd	Gets the address or number of variable descriptors from a data group.
VGdgviewport	Gets the viewport of a data group in virtual, screen, or normalized device coordinates.
VGvd	Gets basic information from a variable descriptor.
VGvdaccess	Gets the access information from a variable descriptor.
VGvdcontext	Manages the context for variable descriptors.
VGvdctt	Utilities for specifying the variable color.
VGvdrange	The variable value range utilities.
VGvdvarvalue	Routines to set variables associated with variable descriptors.

VGdg

VGdg Functions VG Routines

Gets basic information from a data group.

VGdg	VGdgdfargs	VGvd	VGvdctt
VGdgcolor	VGdgvd	VGvdaccess	VGvdrange
VGdgcontext	VGdgviewport	VGvdcontext	VGvdvarvalue
VGdgdf			

VGdg Functions

VGdgdevice	Gets the device index of a data group.
VGdgget	Gets the data group's address or the number of data groups.

VGdgdevice

VGdg Functions VGdg Routines

Gets the device index of a data group.

```
int
VGdgdevice (
       DATAGROUP dgp)
```

VGdgdevice returns the **Error! Reference source not found.**device index for the data group pointed to by *dgp*. The device index specifies which device the data group is to be displayed on. The user can specify a device for a data group by calling VPdgdevice. Valid device indices can be obtained by calling VUopendevice. *VUopendevice* must be given the name of the desired device.

VGdgget

VGdg Functions VGdg Routines

Gets the data group's address or the number of data groups.

```
DATAGROUP
VGdgget (
       int index)
```

VGdgget accepts an **Error! Reference source not found.**index and returns a pointer to the data group referenced by that index. The first data group has an index of 1. Returns the current number of data groups if *index* is zero. Returns *NULL* if *index* refers to a non-existent data group.

VGdgcolor

VGdgcolor Functions   VG Routines

Gets the color information from a data group.**Error! Reference source not found.**

See Also

VPdgcolor

Example

The following code fragment prints the current foreground color.

```
COLOR_SPEC color;
DATAGROUP dgp;

VGdgfrcolor (dgp, &color);
if (color.rgb_rep.rgb_rep_flag >= 0)
   printf ("The foreground color index is %d\n", color.color_index);
else
   {
      printf ("The foreground color is ");
      printf ("red = %d; green = %d; blue = %d\n", color.rgb_rep.red,
             color.rgb_rep.green, color.rgb_rep.blue);
   }
```

VGdg	VGdgdfargs	VGvd	VGvdctt
VGdgcolor	VGdgvd	VGvdaccess	VGvdrange
VGdgcontext	VGdgviewport	VGvdcontext	VGvdvarvalue
VGdgdf			

VGdgcolor Functions

VGdgbkcolor	Gets the background color of a data group.
VGdgfrcolor	Gets the foreground color of a data group.

VGdgbkcolor

VGdgcolor Functions VGer Routines

Gets the background color of a data group.

```
void
VGdgbkcolor (
       DATAGROUP dgp,
       COLOR_SPEC *color)
```

VGdgbkcolor gets the background color associated with the data group. The viewport is set to this color when it is erased.

VGdgfrcolor

VGdgcolor Functions VGRoutines

Gets the foreground color of a data group.

```
void
VGdgfrcolor (
       DATAGROUP dgp,
       COLOR_SPEC *color)
```

VGdgfrcolor gets the foreground color associated with the data group. This is the color of the static context of the data group display, such as the title or viewport outline.

For both of these routines, *dgp* must point to a previously created data group, and *color* should point to a *COLOR_SPEC* data structure in which the routine stores the desired color information. The color is either in RGB form or in device-dependent color index form. The *COLOR_SPEC* data structure includes a flag indicating the form in which the data is stored. See *COLOR_SPEC typedef* in the *Include Files* chapter.

VGdgcontext

VGdgcontext Functions   VG Routines

Gets the context information from a data group.**Error! Reference source not found.**

See Also

VGvdcontext, VPdgcontext

Example

The following code fragment gives a data group a time axis label, and then retrieves the label.

```
DATAGROUP *dgp;
char *label;

/* dgp points to a previously created data group. */
VPdgaxlabel (dgp, V_TIME_AXIS, "MONTHS");
label = VGdgaxlabel (dgp, V_TIME_AXIS);

/* label now points to the copy of MONTHS in the DATAGROUP data structure. */
```

The following code fragment determines whether any axis tick marking has been turned on, and whether context drawing has been turned on.

```
/* Is axis tick marking on? */
if (VGdgcontext (dgp, V_FT_TICS | V_FV_TICS | V_FD1_TICS | V_FD2_TICS))
   printf ("Axis tick marking enabled.\n");

/* Is context drawing enabled? */
if (VGdgcontext (dgp, V_FCONTEXT))
   printf ("Context drawing enabled.\n");
```

The following code fragment gets the current grid attributes.

```
COLOR_SPEC color;
int ltype, lwidth;
DATAGROUP dgp;

VGdggrid_attr (dgp, &color, &ltype, &lwidth);
printf ("The current grid line type is %d\n", ltype);
```

The following code fragment determines whether a graph scrolls or wraps around.

```
DATAGROUP dgp;
int amount;

amount = VGdgscroll_amount (dgp);
if (amount == 0)
   printf ("The graph wraps around.\n");
else
   printf ("The graph scrolls.\n");
```

The following code fragment displays the number of slots assigned to a previously created data group.

```
DATAGROUP dgp;
int num_slots;

num_slots = VGdgslots (dgp);
printf ("The number of slots in the data group is %d\n", num_slots);
```

The following code fragment gets the time increment between adjacent time slices.

```
DATAGROUP dgp;
float increment;

VGdgtime_start_incr (dgp, NULL, &increment);
printf ("The time between time slices is %5.2f.\n", increment);
```

The following code fragment prints the title associated with a previously created data group, pointed to by *dgp*.

```
printf ("The data group title is: %s\n", VGdgtitle (dgp));
```

VGdgcontext Functions

VGdgaxlabel	Gets the time or space axis label.
VGdgcontext	Gets the context control mask of a data group.
VGdggrid_attr	Gets the grid attributes for a graph.
VGdgscroll_amount	Gets the graph scroll amount.
VGdgslots	Gets the number of data group slots.
VGdgticlabfcn	Gets the tick labeling function of a data group.
VGdgtime_start_incr	Gets the time axis start and increment.
VGdgtitle	Gets the title of the data group.

VGdgaxlabel

VGdgcontext Functions VG Routines

Gets the time or space axis label.

```
char *
VGdgaxlabel (
       DATAGROUP dgp,
       int axis_type)
```

VGdgaxlabel returns the **Error! Reference source not found.**axis label of the time axis or either of the two spatial axes that are used if the variable being displayed is a matrix. Returns a pointer to a *NULL*-terminated character string that is the label associated with the specified axis. The choice of axis is indicated by the character *axis_type*: Valid flags are:

V_TIME_AXIS	For the time axis.
V_FIRST_AXIS	For the first spatial axis, which runs horizontally to indicate the columns.
V_SECOND_AXIS	For the second spatial axis, which runs vertically to indicate the rows.

The pointer that is returned points into part of the *DATAGROUP* data structure. If you change the string that is pointed to, you can affect the data group. To change the string, first make a copy, then assign the new label to the data group using VPdgaxlabel.

Returns *NULL* if *dgp* is invalid.

To get the value axis label, use VGvdvallabel.

VGdgcontext

VGdgcontext Functions VG Routines

Gets the context control mask of a data group.

```
LONG
VGdgcontext (
       DATAGROUP dgp,
       LONG mask)
```

VGdgcontext returns a *LONG* containing the status of a data group's context control flags. These flags control how much information the display formatter puts in the display context. *dgp* is a pointer to the data group. *mask* should contain a "1" bit in the position corresponding to each control flag to be checked, and a "0" bit in all other positions. See the description of VPdgcontext for a discussion of the context control flags, their meanings, and pre-defined constant names.

VGdggrid_attr

Gets the grid attributes for a graph.

```
void
VGdggrid_attr (
       DATAGROUP dgp,
       COLOR_SPEC *color,
       int *linetype,
       int *linewidth)
```

VGdggrid_attr gets the g**Error! Reference source not found.Error! Reference source not found.**rid color, line type, and line width for the display formatter from the time axis. If the attributes are not defined, the color is set to the data group foreground color, line type is set to zero, and line width is set to one.

VGdgscroll_amount

VGdgcontext Functions VG Routines

Gets the graph scroll amount.

```
int
VGdgscroll_amount (
       DATAGROUP dgp)
```

VGdgscroll_amount gets the **Error! Reference source not found.Error! Reference source not found.**amount to be scrolled when graphs with history fill all their slots. This does not apply to all graphs.

VGdgslots

VGdgcontext Functions VG Routines

Gets the number of data group slots.

```
int
VGdgslots (
       DATAGROUP dgp)
```

VGdgslots returns an *int* count of the number of **Error! Reference source not found.Error! Reference source not found.**slots or time slices to fit into one display of the data associated with the data group. Generally, a display formatter erases previous data values when displaying the next set of time slices. If the data being displayed are scalars, the number of slots is the number of data points that are displayed. If the data being displayed are matrices or vectors, the display formatter only displays one time slice at a time, regardless of the number of slots specified.

VGdgticlabfcn

VGdgcontext Functions VG Routines

Gets the tick labeling function of a data group.

```
DV_TICLABELFUNPTR
VGdgticlabfcn (
       DATAGROUP dgp,
       int axis_type)

   void
   ticlabelfunc (
          ADDRESS argpcopy,
          double *value,
          ADDRESS output,
          TIC_DATA *tdp)
```

VGdgticlabfcn returns the tick labeling function for a data group axis. The axes are indicated by:

V_TIME_AXIS	For the time axis.
V_FIRST_AXIS	For the first spatial axis, which runs horizontally to indicate the columns.
V_SECOND_AXIS	For the second spatial axis, which runs vertically to indicate the rows.

To get the value axis labeling function, use *VGvdticlabfcn*.

VGdgtime_start_incr

VGdgcontext Functions VG Routines

Gets the time axis start and increment.

```
void
VGdgtime_start_incr (
       DATAGROUP dgp,
       float *start,
       float *increment)
```

VGdgtime_start_incr gets the**Error! Reference source not found.Error! Reference source not found.** time axis start and increment values, used to label the time axis. The arguments are pointers to floats. If the pointer is *NULL*, that argument is not to be set.

VGdgtitle

VGdgcontext Functions VG Routines

Gets the title of the data group.

```
char *
VGdgtitle (
        DATAGROUP dgp)
```

VGdgtitle returns a pointer to the **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**title associated with the data group, *dgp*. The title is a *NULL*-terminated string. Returns *NULL* if *dgp* is invalid.

The pointer that is returned points into part of the *DATAGROUP* data structure. If you change the string that is pointed to, you can affect the data group. To change the string, first make a copy, then assign the new title to the data group using VPdgtitle.

VGdgdf

VGdgdf Functions VG Routines

Gets information related to the display formatter from a data group.

See Also

VPdgdf, VPdgdraw

VGdg	VGdgdfargs	VGvd	VGvdctt
VGdgcolor	VGdgvd	VGvdaccess	VGvdrange
VGdgcontext	VGdgviewport	VGvdcontext	VGvdvarvalue

VGdgdf

VGdgdf Functions

VGdgdf	Gets the display formatter associated with a data group.
Vgdgdfbuffer	Gets the data buffer associated with a data group.
Vgdgdfbuffernum	Gets the number of data elements to be stored in the buffer.
Vgdgdfdata	Gets the pointer to a formatter data area.
Vgdgdfstatus	Gets the drawing status of the display formatter.

VGdgdf

VGdgdf Functions VG Routines

Gets the display formatter associated with a data group.

```
DISPFORM
VGdgdf (
       DATAGROUP dgp)
```

VGdgdf returns the display formatter associated with the data group, *dgp*. Returns *NULL* if there is no display formatter attached.

VGdgdfbuffer

VGdgdf Functions VG Routines

Gets the data buffer associated with a data group.

```
ADDRESS
VGdgdfbuffer (
       DATAGROUP dgp)
```

VGdgdfbuffer returns the address of the data buffer associated with the data group, *dgp*.

VGdgdfbuffernum

VGdgdf Functions VG Routines

Gets the number of data elements to be stored in the buffer.

```
int
VGdgdfbuffernum (
       DATAGROUP dgp)
```

VGdgdfbuffernum returns the number of data elements to be stored in the buffer associated with a data group, *dgp*.

VGdgdfdata

VGdgdf Functions VG Routines

Gets the pointer to a formatter data area.

```
ADDRESS
VGdgdfdata (
       DATAGROUP dgp)
```

VGdgdfdata returns the pointer to a **Error! Reference source not found.Error! Reference source not found.**data area attached to the data group, *dgp*. When the display formatter is called to set up a graph for drawing, it creates the data area and attaches it to the data group. The data area contains information about the graph setup that is required across calls to the display formatter. The data area is attached to the data group by VPdgdfdata, which saves a pointer to the data area in the data group. *VGdgdfdata* is primarily called from display formatters. Returns *NULL* if no data area has been assigned. *dgp* must contain a valid data group since this routine does not determine whether or not the data group is valid. This routine is intended for use by experienced DataViews users who are creating new display formatters. See the *DataViews Graph Development Guide*.

VGdgdfstatus

VGdgdf Functions VG Routines

Gets the drawing status of the display formatter.

```
LONG
VGdgdfstatus (
       DATAGROUP dgp,
       LONG mask)
```

VGdgdfstatus returns the status of the display formatter associated with the data group, *dgp*. *mask* is a bit mask of flags that are OR'ed together where each flag requests different status information. Returns the mask of request flags AND'ed to the current status. Valid flags are:

V_DGDF_CANT_DRAW	Did the setup fail?
V_DGDF_SETUP_DONE	Was the display formatter set up?
V_DGDF_CONTEXT_DRAWN	Was the context drawn?
V_DGDF_ALL	Return the result of all three request flags.

VGdgdfargs

VGdgdfargs Functions VG Routines

Gets the display formatter arguments to a data group.

See Also

VPdgdfargs. See the *Display Formatters (VD)* chapter for formatters that accept paired name-value arguments.

Example

The following code fragment prints the display formatter arguments for a data group:

```
NAME_VALUE_PAIR *dfarg;
int i, dfargsize;

VGdgdfargs (dgp, &dfarg, &dfargsize);
if (dfargsize == 0)
   printf ("There are no arguments\n");
else
   {
      printf ("There are %d argument pairs:\n");
      for (i = 0; i < dfargsize; i++)
         printf (" Name: %s; Value: %s\n", dfarg[i].name, dfarg[i].value);
   }
```

To get the value string associated with a given argument name:

```
DATAGROUP dgp;
char *value;

value = VGdgdfarg_value (dgp, "Argument Name");
```

VGdg VGdgdfargs VGvd VGvdctt
VGdgcolor VGdgvd VGvdaccess VGvdrange
VGdgcontext VGdgviewport VGvdcontext VGvdvarvalue
VGdgdf

VGdgdfargs Functions

VGdgdfarg_value	Gets the value associated with a given argument.
VGdgdfargs	Gets the display formatter arguments.

VGdgdfarg_value

VGdgdf Functions VG Routines

Gets the value associated with a given argument.

```
char *
VGdgdfarg_value (
       DATAGROUP dgp,
       char *name)
```

VGdgdfarg_value returns a pointer to the value string associated with the argument name string, *name*. Returns *NULL* if there is no argument with that name. Note that the pointer is to an internal string which must not be modified. This routine is case-insensitive.

VGdgdfargs

VGdgdf Functions VG Routines

Gets the display formatter arguments.

```
void
VGdgdfargs (
       DATAGROUP dgp,
       NAME_VALUE_PAIR **dfargarray,
       int *dfargsize)
```

VGdgdfargs gets display formatter arguments, *dfargs*, from the specified data group, *dgp*. *dfargarray* is set to the address of an array of *dfargsize* name-value pairs that communicate display formatter-specific information to the display formatter associated with the data group.

A *NAME_VALUE_PAIR* structure contains two pointers: the first points to a name string, which tells the display formatter which value is being specified; the second points to a value string, which the display formatter interprets. The structure pointed to by *dfargarray* is an internal data structure and should not be modified. If changes are required, first make a copy, then use VPdgdfargsto set the new value.

VGdgvd

VGdgvd Functions VG Routines

Gets the address or number of variable descriptors from a data group.

See Also

VGvd, VPdgvd, VPvd

VGdg | VGdgdfargs | VGvd | VGvdctt
VGdgcolor | **VGdgvd** | VGvdaccess | VGvdrange
VGdgcontext | VGdgviewport | VGvdcontext | VGvdvarvalue
VGdgdf

VGdgvd Functions

VGdgvd Gets the address or number of variable descriptors from a data group.

VGdgvd

VGdgvd Functions VG Routines

Gets the address or number of variable descriptors from a data group.

```
VARDESC
VGdgvd (
       DATAGROUP dgp,
       int index)
```

VGdgvd accepts an index and a pointer to a data group and returns a pointer to the variable descriptor in the data group, *dgp*, referenced by *index*. Returns the number of variable descriptors if *index* is 0. If the data group pointer is *NULL*, the routine uses the list of “hanging” variable descriptors, which are variable descriptors that have not yet been associated with a data group.

Returns a pointer to a variable descriptor or an *int*, depending on whether *index* is greater than zero or equal to zero. Returns *NULL* if *index* refers to a non-existent variable descriptor.

VGdgvd returns two different types of data: *int*s and pointers to a *VARDESC*. You must cast the result to the proper type.

VGdgviewport

VGdgviewport Functions VG Routines

Gets the **Error! Reference source not found.**viewport of a data group in virtual, screen, or normalized device **Error! Reference source not found.**coordinates.

See Also

VPdgviewport, VPdg

VGdg VGdgdfargs VGvd VGvdctt
VGdgcolor VGdgvd VGvdaccess VGvdrange
VGdgcontext **VGdgviewport** VGvdcontext VGvdvarvalue
VGdgdf

VGdgviewport Functions

VGdgNDCvp	Gets the viewport of a data group in normalized device coordinates.
VGdgscreenvp	Gets the viewport of a data group in screen coordinates.
VGdgvp	Gets the viewport of a data group in virtual coordinates.

VGdgNDCvp

VGdgviewport Functions VG Routines

Gets the viewport of a data group in normalized device coordinates.

```
void
VGdgNDCvp (
       DATAGROUP dgp,
       FLOAT_POINT *ll,
       FLOAT_POINT *ur)
```

VGdgNDCvp gets the normalized device coordinates of the data group, *dgp*, and returns them in *ll* and *ur*. Normalized device coordinates are *float*s where (0.0, 0.0) corresponds to the lower left of the screen and (1.0, 1.0) corresponds to the upper right of the screen. For example, if the viewport was zoomed to twice the width and height of the screen, the viewport's normalized device coordinates would be *ll* = (0.0, 0.0) and *ur* = (2.0, 2.0).

VGdgscreenvp

VGdgviewport Functions VG Routines

Gets the viewport of a data group in screen coordinates.

```
void
VGdgscreenvp (
       DATAGROUP dgp,
       RECTANGLE *scvp)
```

VGdgscreenvp gets the screen viewport, *scvp*, associated with the data group, *dgp*. Fills the *RECTANGLE* structure pointed to by *scvp* with the viewport screen coordinates. In screen coordinates, (0, 0) corresponds to the lower left corner of the screen and the upper right corner depends on the size of the screen.

VGdgvp

VGdgviewport Functions   VG Routines

Gets the viewport of a data group in virtual coordinates.

```
void
VGdgvp (
       DATAGROUP dgp,
       RECTANGLE *viewport)
```

VGdgvp gets the virtual viewport, *viewport*, associated with the data group, *dgp*. Fills the *RECTANGLE* structure pointed to by *viewport* with the viewport virtual coordinates. The coordinate values are in the range [0, 32767], where (0, 0) corresponds to the lower left corner of the screen and (32767, 32767) corresponds to the upper right corner.

VGvd

VGvd Functions VG Routines

Gets the basic information from a variable descriptor.

See Also

VPvd

Example

The following code fragment gets the variable's dimension from the variable descriptor, *vdp*, and prints out a message describing the shape of the variable.

```
VARDESC vdp;
int d1, d2, d3;

VGvddim (vdp, &d3, &d2, &d1);
printf ("The variable is a ");
if (d3 > 1)
   printf ("time-buffered ");
if (d1 == 1)
   if (d2 == 1)
      printf ("scalar\n");
   else
      printf ("column vector\n");
else if (d2 == 1)
   printf ("row vector\n");
else
   printf ("matrix\n");
```

VGdg	VGdgdfargs	**VGvd**	VGvdctt
VGdgcolor	VGdgvd	VGvdaccess	VGvdrange
VGdgcontext	VGdgviewport	VGvdcontext	VGvdvarvalue
VGdgdf			

VGvd Functions

VGvddim	Gets the dimensions of a variable descriptor.
VGvdrefcount	Gets the variable descriptor's reference count.
VGvdtype	Gets the type of the variable descriptor.

VGvddim

VGvd Functions VG Routines

Gets the dimensions of a variable descriptor.

```
void
VGvddim (
       VARDESC vdp,
       int *dim3,
       int *dim2,
       int *dim1)
```

VGvddim gets the **Error! Reference source not found.**dimensions of the variable associated with the variable descriptor *vdp*. By default, the dimension values are all one (*1*). If the data is stored in row-major order, *dim1* is the dimension that varies the fastest and *dim3* is the dimension that varies the slowest. DataViews treats *dim1* (rows) and *dim2* (columns) as the two spatial dimensions and *dim3* as the time dimension.

VGvdrefcount

VGvd Functions VG Routines

Gets the variable descriptor's reference count.

```
int
VGvdrefcount (
       VARDESC vdp)
```

VGvdrefcount returns the reference count of a variable descriptor. The reference count starts at zero when the variable descriptor is created.

VGvdtype

VGvd Functions VG Routines

Gets the type of the variable descriptor.

```
int
VGvdtype (
        VARDESC vdp)
```

VGvdtype returns the type of the **Error! Reference source not found.**variable described by the variable descriptor, *vdp*. The type is defined by *VPvdtype* or by VPvdcreate when the variable descriptor is created. Valid data types, defined in *dvstd.h*, are:

Flag	**Data Type**	**Size in bits**
V_C_TYPE	char	8
V_UC_TYPE	unsigned char, UBYTE	8
V_S_TYPE	short	16
V_US_TYPE	unsigned short	16
V_L_TYPE	int, LONG	32
V_UL_TYPE	unsigned int, ULONG	32
V_F_TYPE	float	32 (or 64 for some systems)
V_D_TYPE	double	64 (or 128 for some systems)
V_T_TYPE	NULL-terminated string	no set size

VGvdaccess

VGvdaccess Functions VG Routines

Gets the access information from a variable descriptor.

See Also

VPvdaccess

VGvdaccess Functions

VGvd_accmode

VGvdaccess Functions VG Routines

Gets the access mode flag.

```
int
VGvd_accmode (
       VARDESC vdp)
```

VGvd_accmode returns the **Error! Reference source not found.Error! Reference source not found.**access mode flag, which determines how the base address in the variable descriptor, *vdp*, is interpreted. If the access mode is direct, the base address, returned by *VGvdbase*, is the actual base address of the variable. If the variable is accessed indirectly, the base address is the address of a pointer to the base address of the variable. The access mode is indicated by the following flags:

V_DIR_ACCESS	direct access
V_INDIR_ACCESS	Indirect access
V_DS_BOUND	Indirect access through a DataViews data source variable

Normally, DataViews assumes that the base address saved with the variable descriptor points directly to the data to be displayed. However, you can call VPvd_*accmode*to change the interpretation of the address to an indirect mode. The base address is assigned when the variable descriptor is created using VPvdcreate, and it can be re-assigned using VPvdbase. Using TvdPutBuffer to rebind a variable descriptor automatically changes the access mode to *V_DIR_ACCESS*.

VGvdaccess

VGvdaccess Functions VG Routines

Gets the variable descriptor's data access function.

```
void
VGvdaccess (
       VARDESC vdp,
       ADDRESS *fcnp,
       ADDRESS *argp)
```

VGvdaccess gets the information about the access function of the variable descriptor. *fcnp* contains a pointer to the access function. *argp* contains a pointer to the argument block of the access function. A copy of the argument block is saved in the data group. Returns the pointer to the copy, which must not be modified. This routine is intended for use by sophisticated users who are creating new display formatters.

VGvdbase

VGvdaccess Functions VG Routines

Gets the variable's base address.

```
ADDRESS
VGvdbase (
       VARDESC vdp)
```

VGvdbase returns the **Error! Reference source not found.**base address of a variable in a variable descriptor. The base address is defined when the variable descriptor is created by calling VPvdcreate.

VGvddirect_access

VGvdaccess Functions VG Routines

Gets the variable’s data access mode flag.

```
BOOLPARAM
VGvddirect_access (
       VARDESC vdp)
```

VGvddirect_access returns *YES* if the variable is addressed directly. Returns *NO* if the variable is addressed indirectly.

VGvdcontext

VGvdcontext Functions VG Routines

Gets the context information from a variable descriptor.

VGvdcontext Functions

VGvdlog	Gets the log scale flag.
VGvdltype	Gets the type of a line.
VGvdlwidth	Gets the width of a line.
VGvdsymbol	Gets the symbol for the variable descriptor.
VGvdticlabfcn	Gets the tick labeling function.
VGvdvallabel	Gets the variable’s value axis label.
VGvdvarname	Gets a pointer to the variable name.

VGvdlog

VGvdcontext Functions   VG Routines

Gets the log scale flag.

```
int
VGvdlog (
       VARDESC vdp)
```

VGvdlog returns *YES* or *NO* indicating whether the log of the **Error! Reference source not found.Error! Reference source not found.**variable is displayed. *YES* indicates that the variable is displayed on a log scale. *NO* indicates that it is displayed on a linear scale.

VGvdltype

VGvdcontext Functions VG Routines

Gets the type of a line.

```
void
VGvdltype (
       VARDESC vdp,
       int *type)
```

VGvdltype gets the **Error! Reference source not found.**line type index associated with the variable descriptor, *vdp*. The line type is placed in the *int* variable pointed to by *type*. Typically the number of the index is between 0 and 15; however, not all devices support 16 line types. Line types are currently supported by the following formatters: *VD - Lines, VD - Strips, VD - Controllers, VD - Combos* except *VDhilobar, VDbullseye, VDimpulse,* and *VDimpulseto0*.

VGvdlwidth

VGvdcontext Functions VG Routines

Gets the width of a line.

```
void
VGvdlwidth (
       VARDESC vdp,
       int *width)
```

VGvdlwidth gets the pixel width of the line associated with the variable descriptor, *vdp*. The line width is placed in the *int* variable pointed to by *width*. Line types are currently supported by the following formatters: *VD - Lines, VD - Strips, VD - Controllers, VD - Combos* except *VDhilobar, VDbullseye, VDimpulse,* and *VDimpulseto0*.

VGvdsymbol

VGvdcontext Functions VG Routines

Gets the symbol for the variable descriptor.

```
int
VGvdsymbol (
       VARDESC vdp)
```

VGvdsymbol returns the symbol in the attributes section for the variable descriptor, *vdp*. The symbol can have one of the following values:

V_NULL_SYMBOL	= ’ ’	- Default
V_ASTERISK	= ’*’	- Asterisk
V_DOT	= ’.’	- Dot
V_PLUS	= ’+’	- Plus
V_CROSS	= ’x’	- X
V_DIAMOND	= ’d’	- Diamond
V_FILLED_DIAMOND	= ’D’	- Filled Diamond
V_CIRCLE	= ’o’	- Circle
V_FILLED_CIRCLE	= ’O’	- Filled Circle
V_BOX	= ’r’	- Box
V_FILLED_BOX	= ’R’	- Filled Box
V_TRIANGLE	= ’t’	- Triangle (apex up)
V_FILLED_TRIANGLE	= ’T’	- Filled Triangle (apex up)
V_INVERTED_TRIANGLE	= ’v’	- Triangle (apex down)
V_FILLED_INVERTED_TRIANGLE	= ’V’	- Filled Triangle (apex down)
V_TRIANGLE_RIGHT	= ’)’	- Triangle (apex right)
V_FILLED_TRIANGLE_RIGHT	= ’>’	- Filled Triangle (apex right)
V_TRIANGLE_LEFT	= ’(’	- Triangle (apex left)
V_FILLED_TRIANGLE_LEFT	= ’<’	- Filled Triangle (apex left)
V_VERTICAL_LINE	= ’|’	- Vertical Line
V_HORIZONTAL_LINE	= ’-’	- Horizontal Line

VGvdticlabfcn

VGvdcontext Functions VG Routines

Gets the tick labeling function.

```
void
VGvdticlabfcn (
       VARDESC vdp,
       DV_TICLABELFUNPTR ticlabelfunc,
       ADDRESS *argp)

    void
    ticlabelfunc (
           ADDRESS argpcopy,
           double *value,
           ADDRESS output,
           TIC_DATA *tdp)
```

VGvdticlabfcn gets the **Error! Reference source not found.Error! Reference source not found.**tick label function, *ticlabelfunc*, from a variable descriptor, *vdp*, and a pointer to the internally-stored argument block, *argp*, for that function.

Since *argp* is set to an internal data structure which should only be modified with care.

VGvdvallabel

VGvdcontext Functions VG Routines

Gets the variable's value axis label.

```
char *
VGvdvallabel (
        VARDESC vdp)
```

VGvdvallabel returns the **Error! Reference source not found.Error! Reference source not found.**value axis label from the variable descriptor, *vdp*. This label was previously defined by calling VPvdvallabel. Returns *NULL* if no value axis label is defined.

Returns a pointer that points into part of the variable descriptor data structure. If you change the string to which it points, it affects the display. To change the string, first make a copy, then assign the new version to the variable descriptor using *VPvdvallabel*.

VGvdvarname

VGvdcontext Functions VG Routines

Gets a pointer to the variable name.

```
char *
VGvdvarname (
       VARDESC vdp)
```

VGvdvarname returns a pointer to the **Error! Reference source not found.**variable name string in the variable descriptor pointed to by *vdp*. This name was previously defined by calling VPvdvarname. Returns *NULL* if no variable name is defined.

Returns a pointer that points into part of the variable descriptor data structure. If you change the string to which it points, you could destroy the integrity of the structure. If a change is required, first make a copy, then assign the new version to the variable descriptor using VPvdvarname.

VGvdctt

VGvdctt Functions VG Routines

Gets the color information from a variable descriptor.**Error! Reference source not found.**

See Also

VPvdcolor

Example

The following code fragment gets the address of the color threshold table and the number of colors in the table from the variable descriptor, *vdp*. It then prints the contents of the table, with attention to the format of the color specification.

```
VARDESC vdp;
int num_colors;
COLOR_THRESHOLD *ctp;

VGvdctt (vdp, &num_colors, &ctp);
for (i=0; i < num_colors; i++)
    {
       printf ("entry #%d: ", i);
       if (ctp[i].threshcolor.rgb_rep.rgb_rep_flag < 0)
          printf ("red = %d; green = %d; blue = %d\n",
             ctp[i].threshcolor.rgb_rep.red,
             ctp[i].threshcolor.rgb_rep.green,
             ctp[i].threshcolor.rgb_rep.blue);
       else /* ctp[i].threshcolor.rgb_rep.rgb_rep_flag >= 0 */
          printf ("color table index = %d\n", ctp[i].threshcolor.color_index);
    }
```

VGvdctt Functions

VGvdctt

VGvdctt Functions VG Routines

Gets the color information from a variable descriptor.

```
void
VGvdctt (
       VARDESC vdp,
       int *num_colors,
       COLOR_THRESHOLD **ctp)
```

VGvdctt gets the number of colors, *num_colors*, and the address of the color threshold table, *ctp*, stored in the variable descriptor, *vdp*. If the color associated with the variable descriptor is not in color threshold table format, *VGvdctt* first converts it to that format. Therefore, the number of colors is always greater than zero. Returns a pointer that points into part of the *DATAGROUP* data structure. If you change the color threshold table that is pointed to, it affects the display. To change the table, first make a copy, then assign the updated version to the data group using *VPvdctt*.

VGvdrange

VGvdrange Functions VG Routines

Gets the range information from a variable descriptor.**Error! Reference source not found.Error! Reference source not found.**

These routines get the minimum, *low*, and maximum, *high*, values of the variable associated with the variable descriptor, *vdp*.

See Also

VPvdrange

Example

The following code fragment gets the range of values in both formats.

```
VARDESC vdp;
LONG low, high;
double dlow, dhigh;

VGvd_irange (vdp, &low, &high);
VGvd_drange (vdp, &dlow, &dhigh);
/* dlow may or may not equal (double) low, depending on how the */
/* range was set. The same is true for high and dhigh. */
```

VGvdrange Functions

VGvd_drange	Gets the range in double precision float format.
VGvd_irange	Gets the range in long integer format.

VGvd_drange

VGvdrange Functions VG Routines

Gets the range in double precision float format.

```
void
VGvd_drange (
       VARDESC vdp,
       double *low,
       double *high)
```

VGvd_drange converts the values to double precision floating point format before returning them.

VGvd_irange

VGvdrange Functions VG Routines

Gets the range in long integer format.

```
void
VGvd_irange (
       VARDESC vdp,
       LONG *low,
       LONG *high)
```

VGvd_irange converts the values to long integer format before returning them.

VGvdvarvalue

VGvdvarvalue Functions VG Routines

Gets values from a variable descriptor. Normalized data is transformed into the range [0,32767] where 0 corresponds to the minimum data value and 32767 corresponds to the maximum data value. The indices into the variable descriptor's array are zero-based; therefore, to get the first element in the array, make the following call:**Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**

```
VGvdValue (vdp, 0, 0, 0);
```

See Also

VPvdvarvalue

VGvdvarvalue Functions

VGvarLastValue

VGvdvarvalue Functions VG Routines

Uses the access function to get the last value in unnormalized form.

```
double
VGvarLastValue (
       VGDOUBLEACCESSFUNPTR accessfun,
       ADDRESS args)

    double *
    accessfun(
          ADDRESS args,
          int i,
          int j,
          int k)
```

VGvarLastValue returns the unnormalized data value for the specified access function and the position in the data array that was last read.

VGvarNextValue

VGvdvarvalue Functions VG Routines

Uses the access function to get the next normalized data value.

```
int
VGvarNextValue (
       VGLONGACCESSFUNPTR accessfun,
       ADDRESS args)

    LONG
    accessfun (
          ADDRESS args,
          int i,
          int j,
          int k)
```

VGvarNextValue returns the normalized data value for the specified access function and the next position in the data array.

VGvarValue

VGvdvarvalue Functions VG Routines

Uses the access function to get the specified normalized data value.

```
int
VGvarValue (
       VGLONGACCESSFUNPTR accessfun,
       ADDRESS args,
       int time,
       int row,
       int column)

    LONG
    accessfun (
          ADDRESS args,
          int i,
          int j,
          int k)
```

VGvarValue returns the normalized data value for the specified access function and offset, where offset is defined by *time*, *row*, and *column*, which are indices into the variable's array.

VGvdDValue

VGvdvarvalue Functions VG Routines

Uses the variable descriptor to get the specified value as a double.

```
double
VGvdDValue (
       VARDESC vdp,
       int time,
       int row,
       int column)
```

VGvdDValue returns the current value at the position indicated by the indices as a *double*.

VGvdLastValue

VGvdvarvalue Functions VG Routines

Uses the variable descriptor to get the last value in unnormalized form.

```
double
VGvdLastValue (
       VARDESC vdp)
```

VGvdLastValue returns the unnormalized data value for the specified variable descriptor.

VGvdNextValue

VGvdvarvalue Functions VG Routines

Uses the variable descriptor to get the next normalized data value.

```
int
VGvdNextValue (
       VARDESC vdp)
```

VGvdNextValue returns the normalized data value for the specified variable descriptor and the next position in the data array.

VGvdValue

VGvdvarvalue Functions   VG Routines

Uses the variable descriptor to get the specified normalized data value.

```
int
VGvdValue (
       VARDESC vdp,
       int time,
       int row,
       int column)
```

VGvdValue returns the normalized data value for the specified variable descriptor. *time*, *row*, and *column* are indices into the variable's array.

VP Routines

Vp Routines

Routines that put (create, modify, and delete) information into the *DATAGROUP* and *VARDESC* data structures.

The *VP* routines set up the data structures that DataViews uses to manage the display of the data in a program. First, they define the attributes of the data to be displayed by setting up a variable descriptor (*vd*) for each variable, using the routines starting with *VPvd*. Then, the variable descriptors are collected into data groups (*dg*) which contain additional attributes for specifying how the group of variables is displayed. Any changes made to a data group or its variable descriptors after the data group has been set up are not reflected until the data group is reset using *VPdgdfreset*.

VP Modules

All modules in the *VP* layer require the following headers:

```
#include "std.h"
#include "dvstd.h"
#include "VPfundecl.h"
```

Any special headers required by a particular *VP* module are listed in the synopsis section for that module.

VPdg	Manages the basic aspects of data groups.
VPdgcolor	Data group color utilities.
VPdgcontext	Controls the context of a data group.
VPdgdf	Manages communication between the data group and the display formatter.
VPdgdfargs	Data group display formatter argument utilities.
VPdgdraw	Draws and updates the context and data for data groups.
VPdgvd	Variable descriptor utilities.
VPdgviewport	Sets the viewport of a data group using virtual, screen, or normalized device coordinates.
VPvd	Manipulates the basic aspects of the variable descriptors.
VPvdaccess	Access routines for variable descriptors.
VPvdcolor	Utilities for specifying the variable color.
VPvdcontext	Manages the context for variable descriptors.
VPvdrange	The variable value range utilities.
VPvdvarvalue	Routines to set variables associated with variable descriptors.

VPdg

Vpdg Functions VP Routines

Manages the basic aspects of data groups.

See Also

VPdgdraw, VPdgviewport, VGdg

Examples

The following code fragment illustrates the standard way of creating a data group.

```
DATAGROUP dgp;

dgp = VPdgcreate();
```

The following code fragment opens a graphic device for output and assigns the predefined data group, *dgp*, to that device.

```
int devnum;
```

The following code fragment makes a copy of a data group so that you can see the same data displayed in two different ways in two different places.

```
DATAGROUP dgp, dgpnew;
GLOBALREF DISPFORM VDbar, VDline;
RECTANGLE vp1 = {0, 0, 32767, 16383};
RECTANGLE vp2 = {0, 16384, 32767, 32767};
RECTANGLE *clipvp1, *clipvp2, **obsvps;

clipvp1 = &vp1; clipvp2 = &vp2;
```

```
obsvps = NULL;
```

/* Convert the clipped viewports from virtual to screen coordinates. */

```
GRvcs_to_scs (&clipvp1.ll, &clipvp1.ll);
GRvcs_to_scs (&clipvp1.ur, &clipvp1.ur);
GRvcs_to_scs (&clipvp2.ll, &clipvp2.ll);
GRvcs_to_scs (&clipvp2.ur, &clipvp2.ur);
```

/* Display the original data group as a bar graph in the lower half of the screen. */

```
VPdgvp (dgp, &vp1);
VPdgdf (dgp, VDbar);
VPdgdraw (dgp, clipvp1, obsvps);
```

/* Display the original data group as a line graph in the upper half of the screen. */

```
dgpnew = VPdgclone (dgp);
VPdgvp (dgpnew, &vp2);
VPdgdf (dgpnew, VDline);
VPdgdraw (dgp, clipvp2, obsvps);
```

VPdg	VPdgdfargs	VPvd	VPvdcontext
VPdgcolor	VPdgdraw	VPvdaccess	VPvdrange
VPdgcontext	VPdgvd	VPvdcolor	VPvdvarvalue
VPdgdf	VPdgviewport		

VPdg Functions

VPdgclone	Makes a copy of a data group.
VPdgcreate	Creates and initializes a data group.
VPdgdelete	Deletes a data group and frees the associated memory.
VPdgdevice	Assigns a graphical output device to a data group.

VPdgclone

VPgd Functions VP Routines

Makes a copy of a data group.

```
DATAGROUP
VPdgclone (
       DATAGROUP dgp)
```

VPdgclone creates a copy of a data group and returns a pointer to that copy. When the data group is copied, it is set up as if the display formatter had just been connected and the data group had not been displayed yet, even if the original data group has already been displayed once. This means that the context is redrawn the next time the copy is displayed. Returns a pointer to the copy of the data group. After a clone is made, changes to the original are not reflected in the copy.

VPdgcreate

VPgd Functions VP Routines

Creates and initializes a data group.

```
DATAGROUP
VPdgcreate (void)
```

VPdgcreate creates a new data group, sets its parameters to their default settings, and returns a pointer to the new data group. This routine also adds the new data group to the list of data groups maintained by the system.

VPdgdelete

VPgd Functions VP Routines

Deletes a data group and frees the associated memory.

```
void
VPdgdelete (
        DATAGROUP dgp)
```

VPdgdelete removes a data group from the list maintained by DataViews and frees its allocated memory.

VPdgdevice

VPgd Functions VP Routines

Assigns a graphical output device to a data group.

```
void
VPdgdevice (
       DATAGROUP dgp,
       int device_index)
```

VPdgdevice assigns a device index to the data group, indicating the device on which it is to be displayed. *device_index* must contain the number returned by *VUopendevice*. This is distinct from the physical device number obtained when calling *GRopen*.

VPdgcolor

Vpdgcolor Functions VP Routines

Data group color utilities.

See Also

VPvdcolor, VGdgcolor

VPdgcolor Functions

VPdgbkclrndx	Assigns the background color using the lookup table index.
VPdgbkcolor	Assigns the background color using the *COLOR_SPEC* format.
VPdgbkrgb	Assigns the background color using the RGB format.
VPdgfrclrndx	Assigns the foreground color using the lookup table index.
VPdgfrcolor	Assigns the foreground color using the *COLOR_SPEC* format.
VPdgfrrgb	Assigns the foreground color using the RGB format.

VPdgbkclrndx, VPdgbkcolor, VPdgbkrgb

Vpdgcolor Functions VP Routines

Assign a background color to a data group.

```
void
VPdgbkclrndx (
       DATAGROUP dgp,
       int clrndx)

void
VPdgbkcolor (
       DATAGROUP dgp,
       COLOR_SPEC *color)

void
VPdgbkrgb (
       DATAGROUP dgp,
       int r,
       int g,
       int b)
```

VPdgbkclrndx, *VPdgbkcolor*, and *VPdgbkrgb* assign a background color to a data group. The background color is the color used to erase the viewport of the data group. *VPdgbkcolor* expects the color in *COLOR_SPEC* format; *VPdgbkclrndx* expects the color as a device-dependent color lookup table index; *VPdgbkrgb* expects the color in RGB format. The default background color is black.

VPdgfrclrndx, VPdgfrcolor, VPdgfrrgb

Assign a foreground color to a data group.

```
void
VPdgfrclrndx (
       DATAGROUP dgp,
       int clrndx)

void
VPdgfrcolor (
       DATAGROUP dgp,
       COLOR_SPEC *color)

void
VPdgfrrgb (
       DATAGROUP dgp,
       int r,
       int g,
       int b)
```

VPdgfrclrndx, *VPdgfrcolor*, and *VPdgfrrgb* assign a foreground color to a data group. This color is used to draw the context (e.g., the title) of the data display when you are using routines from the *VPdgdraw* module. *VPdgfrcolor* expects the color in *COLOR_SPEC* format; *VPdgfrclrndx* expects the color as a device-dependent color lookup table index; *VPdgfrrgb* expects the color in RGB format. The default foreground color is a middle-level gray.

RGB format specifies a color using three numbers in the range [0,255], where each number corresponds to the intensity of one of the additive primary colors: red, green, and blue.

Note that these routines affect the static part (the context) of the data display; the *VPvd* routines define the colors for the data variables (the dynamic part).

Diagnostics

The foreground color routine affects all of the data group’s context when it is displayed. There is no way to selectively set the colors of the parts of the context.

VPdgcontext

Vpdgcontext Functions VP Routines

Controls the context of a data group.

See Also

VPdgdraw, VPvdcontext, VGdgcontext

Examples

The following code fragment sets the time axis label to “MONTHS.”

```
VPdgaxlabel (dgp, V_TIME_AXIS, "MONTHS");
```

The following code fragment defines a function to label the time axis with month names.

```
/* Tell the data group to use the routine month_name() to label the time axis. */
VPdgticlabfcn (dgp, 't', (DV_TICLABELFUNPTR) month_name, NULL, 0);
VPdgcontext (dgp, V_FT_LABEL_TICS | V_FV_LABEL_TICS | V_FT_TICS | V_FV_TICS, YES);
    .   .   .
}

static void month_name (argp, value, output, tdp)
    int *argp;
    double *value;
    union{
        char string[4];
        LABEL_SIZE size;
        } *output;
    ADDRESS tdp;
    {
        static char *months[12] = {
            "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
            "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"};
        if (value == NULL) /* Describe the largest possible tick label */
            {
                output->size.StringLength = 3;
                output->size.NumLines = 1;
                output->size.LongestLine = 3;
            }
        else        /* Return a copy of the appropriate string */
            strcpy (output->string,
                        months[((int)(*value - 1)) %12]);
    }
```

The following code fragment makes the time axis start at 0 and to be incremented 0.5 units per time slice.

```
DATAGROUP dgp;
float start, incr;

start = 0.0;
incr = 0.5;

VPdgtime_start_incr (dgp, &start, &incr);
```

The following code fragment turns on all axis tick marking and turns off labeling on tick marks with values.

```
VPdgcontext (dgp, V_FT_TICS | V_FV_TICS | V_FD1_TICS | V_FD2_TICS, YES);
```

```
VPdgcontext (dgp, V_FT_LABEL_TICS | V_FV_LABEL_TICS | V_FD1_LABEL_TICS
             | V_FD2_LABEL_TICS, NO);
```

The following code fragment lets you make a formatter scroll left one slot at a time.

```
DATAGROUP dgp;
VPdgscroll_amount (dgp, 1);
```

The following code fragment makes the formatter clear the data area whenever it fills up.

```
DATAGROUP dgp;
VPdgscroll_amount (dgp, VGdgslots (dgp));
```

The following code fragment sets the title of a data group.

```
VPdgtitle (dgp, "TITLE");
```

The following code fragment sets the grid attributes to red, double-width, patterned lines.

```
COLOR_SPEC color;
int ltype, lwidth;

color.rgb_rep.rgb_rep_flag = -1;
color.rgb_rep.red = 255;
color.rgb_rep.green = 0;
color.rgb_rep.blue = 0;

ltype = 3;
lwidth = 2;

VPdggrid_attr (dgp, &color, &ltype, &lwidth);
```

[VPdg](#) [VPdgdfargs](#) [VPvd](#) [VPvdcontext](#)
[VPdgcolor](#) [VPdgdraw](#) [VPvdaccess](#) [VPvdrange](#)
VPdgcontext [VPdgvd](#) [VPvdcolor](#) [VPvdvarvalue](#)
[VPdgdf](#) [VPdgviewport](#)

***VPdgcontext* Functions**

VPdgaxlabel	Assigns labels to the axes of a data group.
VPdgcontext	Sets the data group context control flags.
VPdggrid_attr	Assigns the grid attributes for a data group.
VPdgscroll_amount	Sets the scrolling for a data group.
VPdgslots	Sets the number of time slices.
VPdgticlabfcn	Assigns a tick labeling function to a data group axis.
VPdgtime_start_incr	Sets the start and increment values of the time axis.
VPdgtitle	Sets the title of a data group.

VPdgaxlabel

Vpdgcontext Functions VP Routines

Assigns labels to the axes of a data group.

```
void
VPdgaxlabel (
       DATAGROUP dgp,
       int axis_type,
       char *label)
```

VPdgaxlabel defines a label for the axis of any display of the data group. The axes that can be labeled by this routine are the time axis and the two spatial axes. The spatial axes are available on certain display formatters when the variables in the data group are vector or matrix. See the *Display Formatters* chapter to determine which display formatters support spatial axes. *axis_type* specifies the axis to be labeled. Valid flags are:

V_TIME_AXIS	For the time axis.
V_FIRST_AXIS	For the first spatial axis, which runs horizontally to indicate the columns.
V_SECOND_AXIS	For the second spatial axis, which runs vertically to indicate the rows.

The value axis is specified using *VPvdvallabel*.

VPdgcontext

Vpdgcontext Functions VP Routines

Sets the data group context control flags.

```
void
VPdgcontext (
       DATAGROUP dgp,
       LONG flag_mask,
       BOOLPARAM on_off_flag)
```

VPdgcontext sets and clears the context control flags that tell the display formatter how much information to put in the display context. The display context is the static part of the display that helps the viewer interpret the graphical encoding of the data. The static context includes such features as the title of the display, the legend, and the axis tick marks.

dgp is a pointer to the data group. *flag_mask* indicates which flags are to be changed and *on_off_flag* tells whether all these flags are to be turned on (*YES* or *1*) or off (*NO* or *0*).

dvstd.h contains pre-defined constants that can be used as *flag_mask* values. They can be ORed together, so an arbitrary number of flags can be set with each call.

Context Control Flags:

V_F_ALL: If *YES*, the display formatter displays all the context options described below. If *NO*, the display formatter draws the display with none of the context options.

V_FPRE_ERASE: If *YES*, the display formatter clears the viewport before drawing in it. If *NO*, the display formatter overlays whatever is already in the viewport. This does not guarantee that repeated calls to the display formatter result in clean overlays since the display formatter must do some erasing in order to update the display. The default setting is *YES*.

V_FCONTEXT: If *YES*, the display formatter displays the context according to the settings of the remaining context control flags. If *NO*, the display formatter ignores the other flags except for *V_FVPBOX* and *V_FPRE_ERASE*. If this flag is *NO*, the only thing displayed is the graphical encoding of the data and the viewport box (if the *V_FVPBOX* flag is set). The default setting is *YES*.

V_FLEGEND: If *YES*, the display formatter displays a legend that associates the variable name with the color or color threshold table specified for the variable. This lets you identify the variable in the display according to its color. The default setting is *YES*.

V_FVPBOX: If *YES*, the display formatter draws a box around the viewport. This sometimes helps the display look cleaner. The default setting is *YES*.

Axis Flags:

The display formatter axes (time, value, spatial axis 1st dimension, spatial axis 2nd dimension, roll, pitch) have the following valid flags:

V_FT_TICS,
V_FV_TICS,
V_FD1_TICS,

If *YES*, the display formatter puts tick marks on the axis. If *NO*, the display formatter ignores the settings for the minimum number of tick marks and the tick labels.

V_FD2_TICS, V_FROLL_TICS, V_FPITCH_TICS	The default setting is *YES*.
V_FT_MINTICS, V_FV_MINTICS, V_FD1_MINTICS, V_FD2_MINTICS	If *YES*, and if the tick flag is also *YES*, the display formatter labels the ticks with appropriate values. Note that these tick labels are in addition to the axis labels that may have been specified by a call to *VPdgaxlabel*. The default setting is *YES*.
V_FT_LABEL_TICS, V_FV_LABEL_TICS, V_FD1_LABEL_TICS, V_FD2_LABEL_TICS, V_FROLL_LABEL_TICS, V_FPITCH_LABEL_TICS	If *YES*, and if the tick flag is also *YES*, the display formatter displays the minimum number of tick marks, which is two, with one at each end of the axis. The default setting is *NO*. The roll and pitch axes do not use this flag.

Grid Flags:

Users can define grids for some graphs where the grid attributes are defined using *VPdggrid_attr* as explained in the *Display Formatters (VD)* chapter. Valid flags are:

V_FV_GRID:	If *YES*, grid lines are drawn for each major tick on the value axis. The default setting is *NO*.
V_FT_GRID:	If *YES*, grid lines are drawn for each major tick on the time axis. The default setting is *NO*.

VPdggrid_attr

Vpdgcontext Functions VP Routines

Assigns the grid attributes for a data group.

```
void
VPdggrid_attr (
       DATAGROUP dgp,
       COLOR_SPEC *color,
       int *linetype,
       int *linewidth)
```

VPdggrid_attr assigns grid color and line type for the data group. Line width is not currently supported, so the *linewidth* parameter is ignored by the display formatter, although you can set and get the *linewidth* value. The grid is associated with the time and value axes after it is turned on by appropriate calls to *VPdgcontext*.

VPdgscroll_amount

Vpdgcontext Functions VP Routines

Sets the scrolling for a data group.

```
void
VPdgscroll_amount (
       DATAGROUP dgp,
       int amount)
```

VPdgscroll_amount specifies the amount a graph with history scrolls when it fills up. Sets the number of slots that are made available each time all the slots are filled.

Reasonable values for *amount* are:

- *0* Sets the formatter to wrap around.
- *SlotCount* The number of slots or time slices. The formatter erases the entire data area when it fills up. See *VGdgslots*.
- *n* Where 0 < *n* < *SlotCount*. The formatter moves the data left *n* slots, redrawing *SlotCount* - *n* data items and leaving *n* slots empty.

The default value is *0*.

VPdgslots

Vpdgcontext Functions VP Routines

Sets the number of time slices.

```
void
VPdgslots (
       DATAGROUP dgp,
       int count)
```

VPdgslots sets the number of time slices that are to fit into the display. For example, if the slot count is set to 8, it means that the display formatter makes room for eight time slices. For a bar graph, this means making room for eight sets of bars. The graph could then be updated by calling *VPdgdraw* or *VPdgtakedata* and *VPdgdrdata* eight times before it fills up. At the ninth call, it performs some remedial action such as wrapping around or scrolling the bars to the left. This continually displays the eight most recent values of the data on the display.

Note that some display formatters can only display the most recent data value. These display formatters usually give a warning message if the slot count is greater than one.

The default for *count* is *1* for display formatters that can only display one time slice, and *10* for display formatters that can display more than one time slice.

VPdgticlabfcn

Vpdgcontext Functions VP Routines

Assigns a tick labeling function to a data group axis.

```
void
VPdgticlabfcn (
       DATAGROUP dgp,
       int axis_type,
       DV_TICLABELFUNPTR ticlabelfunc,
       char *argp,
       int argsize)

    void
    ticlabelfunc (
          ADDRESS argpcopy,
          double *value,
          ADDRESS output,
          TIC_DATA *tdp)
```

VPdgticlabfcn assigns a tick labeling function to an axis. The tick labeling function is called once by the display formatter when it sets up the context. It is called twice when the display formatter draws or updates labels; first to determine whether or not the labels fit, and second to draw the labels. If the context is set to label the tick marks of an axis, the display formatter calls this labeling function with the value associated with the tick. The function then gets a label to be attached to that tick mark.

Valid arguments are:

dgp: Pointer to the data group.
axis_type: Character flag indicating which axis receives the axis labeling function. Valid flag values are: *V_TIME_AXIS* for the time axis, *V_FIRST_AXIS* for the first dimension spatial axis indicating the columns of a matrix variable, and *V_SECOND_AXIS* for the second dimension spatial axis indicating the rows of a matrix variable.
ticlabelfunc: Function that calculates the tick label. This function is described below.
argp: Pointer to a user-defined argument block to be passed to the tick labeling function.
argsize: the number of bytes in the user-defined argument block. This is included so a copy can be made of the argument block in the data group. A pointer to this copy is passed to the tick labeler, ensuring that the argument block contains all information required for *VPdgticlabfcn*.

The function *ticlabelfunc* must be defined with four arguments:

```
void
ticlabelfunc (
       ADDRESS argpcopy,
       double *value,
       union
           {
           char string[10];
           LABEL_SIZE size;
           } *output,
       TIC_DATA *tdp)
```

argpcopy: Constant or pointer to a user-defined data structure which can contain several values that determine how the ticks are labeled. A copy of this data structure is stored in the data group and is passed unmodifed to the *ticlabelfunc* function. The size of the copy can be determined using *argsize*.

value: Pointer to a double precision floating point number that gives the value associated with the tick mark. By default *value* is initially *1* and increments by *1* each time the labeling function is called. To change its initial value or increment amount call *VPdgtime_start_incr*. When *value* is *NULL*, the caller wants the tick labeler function to return the size of the longest string it can possibly generate.

output: Pointer to a data structure designated to receive the information provided by this routine. If *value* is non-*NULL*, then *output* is a pointer to a string array which receives the tick label generated for *value*. If *value* is *NULL*, *output* points to a *LABEL_SIZE* structure which receives the size of the text string. The graph asks for this size information to determine how much space to allocate for tick labels. *LABEL_SIZE* is defined in *dvstd.h* and has three fields:

StringLength: The number of characters in the longest label. If the tick labels have multiple lines, this includes all the characters on all the lines, including the newline characters.

NumLines: The maximum number of lines in a tick label. For a label with no embedded newline characters, this should be set to 1.

LongestLine: The number of characters in the longest line of the tick label. For a single line label, this number is the same as *StringLength*.

tdp: Pointer to a DataViews data structure of type *TIC_DATA* which describes how the ticks are to be laid out on the axis. This structure is documented in the *#include* file *dvtypes.h*.

The *TIC_DATA* data structure is currently intended only for internal use, since the *dvtypes.h* file is not included with the subroutine package. When defining the tick labeling function, declare *tdp* as type *ADDRESS* and ignore it in the function. See the example below. Note that this is intended for use by sophisticated DataViews users.

VUdgticlabtab is a utility routine that uses *VPdgticlabfcn* to define a table of strings used as axis tick mark labels.

VPdgtime_start_incr

Vpdgcontext Functions VP Routines

Sets the start and increment values of the time axis.

```
void
VPdgtime_start_incr (
       DATAGROUP dgp,
       float *start,
       float *increment)
```

VPdgtime_start_incr sets the time axis start and increment values. The arguments are pointers to floats. If a pointer is *NULL*, the argument is unchanged. This gives the first slot a value of *start*, and the n-th slot a value of (n - 1) * *increment* + *start*.

VPdgtitle

Vpdgcontext Functions VP Routines

Sets the title of a data group.

```
void
VPdgtitle (
        DATAGROUP dgp,
        char *title)
```

VPdgtitle assigns a character string to be used by the data group as its title. When the data group is displayed, this title appears at the top of the display.

VPdgdf

Vpdgdf Functions VP Routines

Manages communication between the data group and the display formatter.

See Also

VPdgdraw, VGdgdf, Display Formatters (*VD*)

Examples

The following code fragment displays the data group information as a bar graph.

```
RECTANGLE *clipvp, **obsvps;
obsvps = NULL;

/* Bar graph display formatter */
GLOBALREF DISPFORM VDbar;

/* Attach the bar graph to the data group */
VPdgdf (dgp, VDbar);

/* Display the bar graph */
VGdgscreenvp (dgp, &clipvp);
VPdgdraw (dgp, clipvp, obsvps);
```

The following code fragment waits for a cursor position and determines which slot in a graph was picked.

```
DATAGROUP dgp;
DV_POINT SlotSize, CursorPosition;
RECTANGLE DataArea;
RECTANGLE *clipvp, **obsvps;
obsvps = NULL;

VGdgscreenvp (dgp, &clipvp);
VPdgdraw (dgp, clipvp, obsvps);

/* Poll, then get the cursor position in screen coordinates. */
. . .

if (DV_SUCCESS == VPdgdfquery (dgp,
                               V_Q_DATAVP, NULL, &DataArea)
      && DV_SUCCESS == VPdgdfquery (dgp,
                               V_Q_SLOTSIZE, NULL, &SlotSize))

   if (   CursorPosition.x < DataArea.ll.x ||
      CursorPosition.x > DataArea.ur.x ||
      CursorPosition.y < DataArea.ll.y ||
      CursorPosition.y > DataArea.ur.y)
      printf ("Cursor outside data area.\n");

   else
      printf ("Cursor in slot # %d.\n",
         (CursorPosition.x - DataArea.ll.x) / SlotSize.x);

else
   printf ("Couldn’t get info from the formatter.\n");
```

The following code fragment draws a graph, makes some changes in the data group, then calls *VPdgdfreset* to effect the changes.

```
x = 0.6;                           /* Initialize variable being displayed */
VPdgdraw (dgp, clipvp, obsvps);    /* Display the data group */
x = 4 * x * (1 - x);               /* Update displayed variable */
VPdgtitle (dgp, "NEW TITLE");      /* Change the data group title */
VPdgdraw (dgp, clipvp, obsvps);    /* Update the display (still with old title) */
x = 4 * x * (1 - x);               /* Update displayed variable */
VPdgdfreset (dgp);                 /* Display it with new title */
VPdgdraw (dgp, clipvp, obsvps);
x = 4 * x * (1 - x);   /* Update displayed variable */
```

[VPdg](#) [VPdgdfargs](#) [VPvd](#) [VPvdcontext](#)
[VPdgcolor](#) [VPdgdraw](#) [VPvdaccess](#) [VPvdrange](#)
[VPdgcontext](#) [VPdgvd](#) [VPvdcolor](#) [VPvdvarvalue](#)
VPdgdf [VPdgviewport](#)

VPdgdf Functions

VPdgdf	Assigns a display formatter to a data group.
VPdgdfbuffer	Assigns a data buffer to a data group.
VPdgdfbuffernum	Sets the number of data elements to be stored in the buffer.
VPdgdfdata	Assigns a display formatter data area pointer into a data group.
VPdgdfmessage	Sends a message or information to the display formatter.
VPdgdfquery	Queries a display formatter for information.
VPdgdfreset	Resets the display formatter associated with a data group.

VPdgdf

Vpdgdf Functions VP Routines

Assigns a display formatter to a data group.

```
void
VPdgdf (
       DATAGROUP dgp,
       DISPFORM df)
```

VPdgdf associates a display formatter, *df*, with the specified data group, *dgp*. The display formatter is specified by a global pointer which must be declared accordingly in order to compile correctly.

VPdgdfbuffer

Vpdgdf Functions VP Routines

Assigns a data buffer to a data group.

```
void
VPdgdfbuffer (
       DATAGROUP dgp,
       ADDRESS buffer)
```

VPdgdfbuffer assigns a data buffer, *buffer*, to the data group, *dgp*. This routine is normally called by a display formatter to attach the data buffer to the data group. The data buffer holds both displayed and undisplayed data so data can be redisplayed after the data group is resized or exposed. This routine is intended for use by sophisticated DataViews users who are creating new display formatters. See the *DataViews Graph Development Guide*.

VPdgdfbuffernum

Vpdgdf Functions VP Routines

Sets the number of data elements to be stored in the data buffer.

```
void
VPdgdfbuffernum (
       DATAGROUP dgp,
       int num)
```

VPdgdfbuffernum sets the number of data elements, *num*, to be stored in the data buffer attached to the data group, *dgp*. This lets you specify a maximum number of data elements in situations where you don't want to limit the buffer to the number of slots. The default value of *num* is equal to the number of slots, which is the minimum required to support redrawing the graph. If *num* is less than the number of data slots in the data group, an error message is displayed.

VPdgdfdata

Vpdgdf Functions VP Routines

Assigns a display formatter data area pointer into a data group.

```
void
VPdgdfdata (
       DATAGROUP dgp,
       ADDRESS data_area)
```

VPdgdfdata associates a pointer to a data area with the data group. This routine is normally called by a display formatter which uses it to attach an allocated data area to the data group. The data area is defined by the display formatter to save information relevant to the data group across display calls. This routine is intended for use by sophisticated DataViews users who are creating new display formatters. See the *DataViews Graph Development Guide*.

The display formatter can't save the information internally because it may be servicing several data groups at once. Therefore, in order to isolate the display formatter from information specific to a data group, the data area is attached to the data group itself. This information may include the current values being displayed, the display slot sizes, and copies of important data in the data group.

CAUTION: The caller is responsible for managing the data that is pointed to. For example, if there is already a data pointer in the data group and you want to attach a new one, you must deallocate the space pointed to by the old pointer.

VPdgdfmessage

Vpdgdf Functions VP Routines

Sends a message or information to the display formatter.

```
BOOLPARAM
VPdgdfmessage (
        DATAGROUP dgp,
        int flag,
        ADDRESS indatum)
```

VPdgdfmessage sends the display formatter a message or information. Can be used to change contextual information about the graph. *flag* indicates the type of information to be received. *indatum* is the address of a structure containing the information. This routine is intended for use by sophisticated DataViews users who are creating new display formatters. To use this routine, you must define the *recv_message* entry point to process the flags and structures you send using the *flag* and *indatum* parameters. See *VDtime* and the *DataViews Graph Development Guide*.

VPdgdfquery

Vpdgdf Functions VP Routines

Queries a display formatter for information.

```
BOOLPARAM
VPdgdfquery (
       DATAGROUP dgp,
       int flag,
       ADDRESS indatum,
       ADDRESS outdatum)
```

VPdgdfquery retrieves information from a display formatter that has been invoked for a data group. It is used after drawing the data group with *VPdgdrdata, VPdgdraw*, or *VPdgdisplay*. *flag* indicates what type of information is to be returned. *indatum* is the address of a structure containing additional information related to the query. This structure is *NULL* for some queries. *outdatum* is the address of a structure designated to hold the data that answers the query. The routine returns *DV_SUCCESS* if the query is answered; otherwise returns *DV_FAILURE*.

The query flags fall into two categories: general and feedback. The general query flags get information relating to the data group as a whole. The feedback query flags get information relating to a particular point in the data group. The general query flags are:

V_Q_DATAVP	Gets the area of the screen where the formatter is encoding the data. Set *indatum* to *NULL* and make *outdatum* a pointer to a *RECTANGLE*.
V_Q_DOES_CLIPPING	Determines whether the formatter clips to obscuring viewports. Returns *YES* in outdatum if the formatter clips; otherwise returns *NO*. Set *indatum* to *NULL* and make *outdatum* a pointer to a *DV_BOOL*.
V_Q_LEGSIZE	Applies only to *VDlegend*. Gets the size of the legend. Set *indatum* to *NULL* and make *outdatum* a pointer to a *DV_POINT*.
V_Q_SLOTSIZE	Gets the size of a single “slot” in a graph. A slot records one time slice of data for scalar variables or one value in vector or matrix variables. Set *indatum* to *NULL* and make outdatum a pointer to a *DV_POINT*.
V_Q_DATA_SLOTSIZE	Applies only to the spectro display formatters. Gets the size of a single element where one value in the vector of data is drawn. Set *indatum* to *NULL* and make *outdatum* a pointer to a *DV_POINT*.
V_Q_VDTITLE_TEXTVP	Applies only to *VDtext* and *VDmessage*. Gets the screen coordinates of the smallest bounding box encompassing the text. Set *indatum* to *NULL* and make *outdatum* a pointer to a *RECTANGLE*.
V_Q_VDTITLE_CHARSIZE	Applies only to *VDtext* and *VDmessage*. Gets the character size used to display the text in the range [1,4]. Set *indatum* to *NULL* and make *outdatum* a pointer to an *int*.

The feedback query flags let you get information displayed at a particular point in the data group. If the point comes from a user pick, this feature lets you feedback information about the data displayed at the pick. Currently, only certain display formatters support the feedback query flags. These are listed after the flags.

The feedback queries use pointers to *DV_POINT* or *V_Q_PICK_VDP* structures as *indatum*. Points must be in screen coordinates. To get a point in screen coordinates from a location object, call *VOloScpGet*. The *V_Q_PICK_VDP* structure contains a *DV_POINT*, a *vdp*, and the index of the *vdp* in the data group’s list of *vdp*s. To get a list of *vdp*s at a point in the data group, use the *V_Q_VDPS_AT_LOCATION* query flag. You can use *outdatum* from this query to get information for *indatum* for additional queries.

The feedback query flags are:

V_Q_VDPS_AT_LOCATION	Gets an array of structures containing the *vdp*s whose data is drawn at or near the point. Set *indatum* to a pointer to a *DV_POINT* and make *outdatum* a pointer to a *V_Q_VDP_LIST*. See the table later in this description for the pickable graphics and the accuracy required for picking. If no *vdp*s display data at or near the point, the routine returns *DV_FAILURE*.
V_Q_SLOT_AT_LOCATION	Gets the 1-based index of the slot at the point. If the point is not in the data viewport, the query returns *-1.0* for *outdatum*. Set *indatum* to a pointer to a *DV_POINT* and make *outdatum* a pointer to an *int*.
V_Q_DATA_SAMPLE	Gets the iteration number of the data closest to the point. If the point is not in the data viewport, the query returns *-1.0* for *outdatum*. Set *indatum* to a pointer to a *DV_POINT* and make *outdatum* a pointer to a *double*.
V_Q_SAMPLE_AT_LOCATION	Gets the interpolated iteration number at the point. If the point is not in the data viewport, the query returns -*1.0* for *outdatum*. Set *indatum* to a pointer to a *DV_POINT* and make *outdatum* a pointer to a *double*.
V_Q_DATA_VALUE	Gets the value of the datum displayed at the point. Set *indatum* to a pointer to a *V_Q_PICK_VDP* and make *outdatum* a pointer to a *double*.
V_Q_VALUE_AT_LOCATION	Gets the interpolated value of the point with respect to the *vdp*'s range. Set *indatum* to a pointer to a *V_Q_PICK_VDP* and make *outdatum* a pointer to a *double*.
V_Q_FLOOR_VALUE	Applies only to *VDpig* and *VDlinefill*. Gets the visual base value at the point. The floor value lets you take into account the data values stacked beneath the datum at the point. Set *indatum* to a pointer to a *V_Q_PICK_VDP* and make *outdatum* a pointer to a *double*.
V_Q_SECTOR_AT_LOCATION	Applies only to *VDradial* and *VDne_radial*. Gets the 1-based sector at the point. A sector is similar to a slot, but starts and ends at a sample. A slot starts and ends midway between samples. Set *indatum* to a pointer to a *V_Q_PICK_VDP* and make *outdatum* a pointer to an *int*.

The following display formatters support the feedback queries. They must be displaying scalar data.

Bars	VDbar, VDbarhoriz, VDbarpacked, VDbarsolid, VDcenter, VDpig
Combos	*VDbarline, VDbarpackedline, VDhilobar, VDhiloline*, *VDptsline*
Strips	*VDstrip*, *VDstripras*, *VDvstrip*, *VDvstrip_r*, *VDwaterfall*, *VDwaterfall_r*
Misc.	*VDhighlow*, *VDline*, *VDlinefill*, *VDne_radial*, *VDpoints*, *VDradial*, *VDstep*

The following table lists the graphics that are pickable for each display formatter and the accuracy required for picking. The accuracy is expressed in pixels. An accuracy of 0 indicates that the object requires an exact pick within the width of the bar or marker. An accuracy of 5×5 indicates that the object must be within a 5×5 pixel rectangle centered on the pick location.

Display Formatter	Pickable Graphics	Accuracy
VDbar VDbarhoriz VDbarpacked VDbarsolid VDcenter VDpig	bar	0
VDbarline, VDbarpackedline	bar line	0 5×5
VDhighlow	either endpoint of a vertical line (high, low)	5×5
	horizontal line (open, close)	5×5
VDhilobar	bar	0
	either endpoint of a vertical line (high, low)	5×5
	horizontal line (close)	5×5
VDhiloline	line	5×5
	either endpoint of a vertical line (high, low)	5×5
	horizontal line (close)	5×5
VDline	line	5×5
VDlinefill	area	0
VDpoints	marker	0
VDptsline	marker	0
	line	5×5
VDne_radial VDradial	line	5×5
VDstep	step (horizontal line only)	5×5
VDstrip VDstripras VDvstrip VDvstrip_r VDwaterfall VDwaterfall_r	line	5×5

VPdgdfreset

Vpdgdf Functions VP Routines

Resets the display formatter associated with a data group.

```
void
VPdgdfreset (
       DATAGROUP dgp)
```

VPdgdfreset resets the display formatter associated with the data group by deleting any temporary storage associated with the display formatter. The next time the data group is displayed, it starts from the beginning, redrawing the context before displaying any data. This routine is an alternative to *VPdgcleanup* for use with *VPdgdraw*. The next time *VPdgdraw* is called, it resets the data group and draws the context before drawing data. When used with pre-9.0 display formatters, it resets the entry point of pre-9.0 display formatters to *initial_draw*.

VPdgdfargs

Vpdgdfargs Functions VP Routines

Data group display formatter argument utilities.

See Also

VPdgdraw, VGdgdfargs. For formatters that accept paired name-value arguments, see the *Display Formatters (VD)* chapter.

Example

The following code fragment passes special arguments to a hypothetical display formatter.

```
NAME_VALUE_PAIR arg[2];

arg[0].name = "Argument 0";
arg[0].value = "10";
arg[1].name = "Argument 1";
arg[1].value = "20";

VPdgdfargs (dgp, arg, 2);
```

VPdgdfargs Functions

VPdgdfaddarg	Adds or replaces a specific name-value pair.
VPdgdfargs	Adds the display formatter arguments.
VPdgdfdelarg	Deletes a specific name-value pair.

VPdgdfaddarg

Vpdgdfargs Functions VP Routines

Adds or replaces a specific name-value pair to a data group.

```
void
VPdgdfaddarg (
       DATAGROUP dgp,
       char *name,
       char *value)
```

VPdgdfargs

Vpdgdfargs Functions VP Routines

Adds the display formatter arguments.

```
void
VPdgdfargs (
        DATAGROUP dgp,
        NAME_VALUE_PAIR *dfargarray,
        int dfargsize)
```

VPdgdfargs adds an array of display formatter arguments, *dfargarray*, to the data group, *dgp*. The array contains *dfargsize* name-value pairs that communicate display formatter-specific information to the display formatter associated with the data group.

A *NAME_VALUE_PAIR* structure contains two pointers: the first points to a name string which indicates which value is being specified; the second points to a corresponding value string.

VPdgdfdelarg

Vpdgdfargs Functions VP Routines

Deletes a specific name-value pair, *name,* from a data group.

```
void
VPdgdfdelarg (
       DATAGROUP dgp,
       char *name)
```

VPdgdraw

Vpdgdraw Functions VP Routines

Draws and updates the context and data for data groups. Five routines constitute the basic calls for displaying data groups. *VPdgsetup* sets up the required internal structures. *VPdgdrcontext* draws the context. *VPdgtakedata* and *VPdgdrdata* take and display data, and are usually called in the update loop of the application. *VPdgcleanup* deallocates the internal structures. The data group should be reset to reflect changes made using any *VPdg* or *VPvd* function with the exception of the functions in the *VPvdvarvalue* module. Use *VPdgdfreset* to reset the data group.

VPdgdraw combines the setup, context drawing, initial data retrieval, and initial data display into a single call. It can also be used in place of *VPdgtakedata* and *VPdgdrdata* in the update loop.

To draw and update data groups using pre-9.0 display formatters, use *VPdgdisplay* in conjunction with *VPdgdfcontext_only*. See also *VPdgdfentry*.

See Also

VPdgdf, VGdgdf

Examples

The following code fragment sets up a data group, draw its context, and draws the first iteration of data.

```
flag = VPdgsetup (dgp);

/* If the display formatter can be drawn, display the context and the first data iteration.
if (flag == DV_SUCCESS)
    {
        VPdgdrcontext (dgp, clipvp, obsvps, V_BF_UNDISP);
        VPdgtakedata (dgp);
        VPdgdrdata (dgp, clipvp, obsvps, V_BF_UNDISP);
    }
    else
        printf ("The graph cannot be set up properly.");
```

The following code fragment is functionally equivalent to the previous fragment, but uses *VPdgdraw*:

```
if (! (DV_SUCCESS == VPdgdraw (dgp, clipvp, obsvps)))
    printf ("The display formatter cannot be drawn.");
```

The following code fragment sets up a data group, retrieves two new iterations of data, draws the context, and draws the latest *n* iterations of data, where *n* equals the number of slots in the data group:

```
VPdgslots (dgp, 2);

flag = VPdgsetup (dgp);

VPdgtakedata (dgp);
VPdgtakedata (dgp);

/* If the display formatter can be drawn, display the context and the latest n iterations of data. */
if (flag == DV_SUCCESS)
    {
        VPdgdrcontext (dgp, clipvp, obsvps, V_BF_LATEST_N);
        VPdgdrdata (dgp, clipvp, obsvps, V_BF_LATEST_N);
    }
    else
        printf ("The graph cannot be set up properly.");
```

The following code fragment draws a data group, resizes it, and redisplays the context and original data:

```
VPdgdraw (dgp, clipvp, obsvps);
```

/* Resize the data group. */

```
VGdgvp (dgp, &vp);

vp.ur.x = 2*vp.ur.x;
vp.ur.y = 2*vp.ur.y;
VPdgvp (dgp, &vp);
```

/* Free the temporary storage for the previous display of the data group and reset the data group to its initial state. */

```
VPdgdfreset (dgp);
```

/* Set up the data group again and redisplay the previously displayed data with the new coordinates. */

```
if (DV_SUCCESS == VPdgsetup (dgp))
   {
      VPdgdrcontext (dgp, clipvp, obsvps, V_BF_DISP);
      VPdgdrdata (dgp, clipvp, obsvps, V_BF_DISP);
   }
   else
      printf ("The graph cannot be set up properly.");
```

VPdg	VPdgdfargs	VPvd	VPvdcontext
VPdgcolor	**VPdgdraw**	VPvdaccess	VPvdrange
VPdgcontext	VPdgvd	VPvdcolor	VPvdvarvalue
VPdgdf	VPdgviewport		

VPdgdraw Functions

VPdgcleanup	Deallocates the temporary storage for a data group.
VPdgdfcontext_only	Sets the context-draw flag.
VPdgdraw	Draws the context and data for a data group.
VPdgdrawnull	Draws a null representation of a data group.
VPdgdrcontext	Draws the context of a data group.
VPdgdrdata	Draws one or more iterations of data.
VPdgsetup	Sets up the layout for a data group.
VPdgtakedata	Takes one iteration of data from the data sources.

All routines that use the *clipvp* and *obsvps* parameters interpret them as defined below.

clipvp The clipping viewport. *clipvp* is a pointer to a rectangle structure that specifies a viewport in screen coordinates. The data group is clipped to this viewport. If *NULL*, the data group is clipped to its own viewport as returned by *VGdgscreenvp*.

obsvps The obscuring viewports. *obsvps* is a pointer to a *NULL*-terminated array of rectangle structures specifying viewports in screen coordinates that obscure the data group. If *NULL*, clipping to obscuring viewports is not required.

VPdgcleanup

Vpdgdraw Functions VP Routines

Deallocates the temporary storage for a data group.

```
void
VPdgcleanup (
       DATAGROUP dgp)
```

VPdgcleanup deallocates the internal structures of the data group, *dgp*, which were set up by *VPdgsetup*. Should only be called to clean up. If you need to reset the data group, use *VPdgdfreset*.

VPdgdfcontext_only

Sets the context-draw flag.

```
int
VPdgdfcontext_only (
       int flag)
```

VPdgdfcontext_only sets a flag that controls the initial drawing of the data group. This routine is used with both pre-9.0 and post-9.0 display formatters. It works in conjunction with *TdpDraw*, *VPdgdraw,* and *VPdgdisplay*. If *flag* is *YES*, a call to one of these routines draws only the context. If *flag* is *NO*, a call to one of these routines draws the context together with the first data values. The default value for the flag is *NO*. To determine the current value of the flag, set *flag* to any value other than *YES* or *NO*. Returns the old flag value.

VPdgdraw

Vpdgdraw Functions VP Routines

Draws the context and data for a data group.

```
BOOLPARAM
VPdgdraw (
       DATAGROUP dgp,
       RECTANGLE *clipvp,
       RECTANGLE **obsvps)
```

VPdgdraw sets up the data group, *dgp*, draws the context, and displays data. Draws the data group clipped to the appropriate viewports as specified by *clipvp* and *obsvps*. If the data group is already set up and the context is displayed, retrieves and displays the next iteration of data along with any other new data. Can be used with *VPdgdfcontext_only* to set up the data group and draw the context only. This routine combines most of the functionality of *VPdgsetup*, *VPdgdrcontext*, *VPdgtakedata*, and *VPdgdrdata*, but always draws the newest data. Returns *DV_SUCCESS* if successful, otherwise returns *DV_FAILURE*.

VPdgdrawnull

Vpdgdraw Functions VP Routines

Draws a null representation of a data group.

```
void
VPdgdrawnull (
       DATAGROUP dgp,
       RECTANGLE *clipvp,
       RECTANGLE **obsvps)
```

VPdgdrawnull draws a filled rectangle with the text string “Graph” in place of the data group, *dgp*. Clips to the appropriate viewports as specified by *clipvp* and *obsvps*.

VPdgdrcontext

Vpdgdraw Functions VP Routines

Draws the context of a data group.

```
BOOLPARAM
VPdgdrcontext (
       DATAGROUP dgp,
       RECTANGLE *clipvp,
       RECTANGLE **obsvps,
       int draw_flag)
```

VPdgdrcontext draws the context for the display formatter associated with the data group, *dgp*. Clips to the appropriate viewports as specified by *clipvp* and *obsvps*. When called to redisplay data, the labels in the context correspond to the iterations of data indicated by *draw_flag*. Returns *DV_SUCCESS* if the context is drawn; otherwise returns *DV_FAILURE*. Valid values for *draw_flag* are:

V_BF_DISP	Draw the context for the most recently displayed iterations.
V_BF_UNDISP	Draw the context for the iterations that haven't been displayed.
V_BF_LATEST_N	Draw the context for the latest *n* iterations, where *n* is the number of slots in the graph.

VPdgdrdata

Vpdgdraw Functions VP Routines

Draws one or more iterations of data.

```
BOOLPARAM
VPdgdrdata (
       DATAGROUP dgp,
       RECTANGLE *clipvp,
       RECTANGLE **obsvps,
       int draw_flag)
```

VPdgdrdata displays the iterations of data which correspond with the *draw_flag* indicated and updates the time axis. Draws the data group, *dgp*, clipped to the appropriate viewports as specified by *clipvp* and *obsvps*. Returns *DV_SUCCESS* if the data is displayed; otherwise returns *DV_FAILURE*. When this routine is called after *VPdgdrcontext* to redisplay data, both should use the same value for *draw_flag*. Valid values are:

V_BF_DISP	Draw the most recently displayed iterations.
V_BF_UNDISP	Draw the iterations that haven't been displayed.
V_BF_LATEST_N	Draw the latest *n* iterations, where *n* is the number of slots in the graph.

VPdgsetup

Vpdgdraw Functions VP Routines

Sets up the layout for a data group.

```
BOOLPARAM
VPdgsetup (
       DATAGROUP dgp)
```

VPdgsetup sets up the layout for the data group, *dgp*, including determining if the display formatter is valid, if the data group's variables meet the constraints of the display formatter, and if the graph can be drawn in the viewport. The layout information is attached to the data group, but is used by the display formatter to draw and update the data group. Returns *DV_SUCCESS* if the data group is set up; otherwise returns *DV_FAILURE*.

VPdgtakedata

Vpdgdraw Functions VP Routines

Takes one iteration of data from the data sources.

```
BOOLPARAM
VPdgtakedata (
       DATAGROUP dgp)
```

VPdgtakedata retrieves one iteration of data from the data sources associated with the data group, *dgp*. Returns *DV_SUCCESS* if the data is retrieved; otherwise returns *DV_FAILURE*. Note: You can call this routine several times without intervening calls to *VPdgdrdata* since the data group stores undisplayed data.

VPdgvd

Vpdgvd Functions VP Routines

Variable descriptor utilities.

See Also

VPdg, VPvd, VGdg, VGdgvd

Example

The following code fragment adds two newly created variable descriptors to a newly created data group then reverses the order of the variables in the data group.

```
DATAGROUP vdp1, vdp2, dgp;
float data1, data2;

dgp = VPdgcreate();
vdp1 = VPvdcreate ((ADDRESS) &data1, V_F_TYPE);
vdp2 = VPvdcreate ((ADDRESS) &data2, V_F_TYPE);
VPdgvdadd (dgp, vdp1);
VPdgvdadd (dgp, vdp2);

VPdgvdinsert (dgp, 1, VPdgvdremove (dgp, 2));
```

VPdg	VPdgdfargs	VPvd	VPvdcontext
VPdgcolor	VPdgdraw	VPvdaccess	VPvdrange
VPdgcontext	**VPdgvd**	VPvdcolor	VPvdvarvalue
VPdgdf	VPdgviewport		

VPdgvd Functions

VPdgvdadd	Adds a variable descriptor to the data group.
VPdgvdinsert	Inserts a variable descriptor in the list.
VPdgvdremove	Removes a variable descriptor from the list.
VPdgvdswitch	Swaps a variable descriptor within the list.

VPdgvdadd

Vpdgvd Functions VP Routines

Adds a variable descriptor to the data group.

```
void
VPdgvdadd (
       DATAGROUP dgp,
       VARDESC vdp)
```

VPdgvdadd adds a variable descriptor, *vdp*, to the end of the list of variable descriptors connected to the data group, *dgp*.

VPdgvdinsert

Vpdgvd Functions VP Routines

Inserts a variable descriptor in the list.

```
void
VPdgvdinsert (
       DATAGROUP dgp,
       int ndx,
       VARDESC vdp)
```

VPdgvdinsert inserts a variable descriptor, *vdp*, before the *ndx*-th variable descriptor in the list of variable descriptors connected to the data group, *dgp*.

VPdgvdremove

Vpdgvd Functions VP Routines

Removes a variable descriptor from the list.

```
VARDESC
VPdgvdremove (
       DATAGROUP dgp,
       int ndx)
```

VPdgvdremove removes the *ndx*-th variable descriptor in the list of variable descriptors connected to the data group, *dgp*, and returns its address.

VPdgvdswitch

Vpdgvd Functions VP Routines

Swaps a variable descriptor within the list.

```
VARDESC
VPdgvdswitch (
       DATAGROUP dgp,
       int ndx,
       VARDESC vdp)
```

VPdgvdswitch switches the variable descriptor, *vdp*, with the *ndx*-th variable descriptor in the list of variable descriptors connected to the data group, *dgp.* Returns the address of the previous *vdp*.

The first variable in the list has an index of 1. The index of the last variable is provided by *VGdgvd*.

VPdgviewport

Vpdgviewport Functions VP Routines

Sets the viewport of a data group using virtual, screen, or normalized device coordinates.

See Also

VGdgviewport

Examples

The following code fragment sets the data group viewport to be the bottom half of the screen.

```
RECTANGLE vvp;

vvp.ll.x = 0;
vvp.ll.y = 0;
vvp.ur.x = 32767;
vvp.ur.y = 32767 / 2;
VPdgvp (dgp, &vvp);
```

The following code fragment makes a copy of a data group so that you can see the same data displayed in two different ways in two different places.

```
DATAGROUP dgp, dgpnew;
GLOBALREF DISPFORM VDbar, VDline;
RECTANGLE vp1 = {0, 0, 32767, 16383}
RECTANGLE vp2 = {0, 16384, 32767, 32767}
RECTANGLE *clipvp1, *clipvp2, **obsvps;

clipvp1 = &vp1; clipvp2 = &vp2;
obsvps = NULL;

/* Convert the clipped viewports from virtual to screen coordinates. */
GRvcs_to_scs (&clipvp1.ll, &clipvp1.ll);
GRvcs_to_scs (&clipvp1.ur, &clipvp1.ur);
GRvcs_to_scs (&clipvp2.ll, &clipvp2.ll);
GRvcs_to_scs (&clipvp2.ur, &clipvp2.ur);

/* Display the original data group as a bar graph in the lower half of the screen. */
VPdgvp (dgp, &vp1);
VPdgdf (dgp, VDbar);
VPdgdraw (dgp, clipvp1, obsvps);

/* Display the original data group as a line graph in the upper half of the screen. */
dgpnew = VPdgclone (dgp);
VPdgvp (dgpnew, &vp2);
VPdgdf (dgpnew, VDline);
VPdgdraw (dgp, clipvp2, obsvps);
```

VPdg VPdgdfargs VPvd VPvdcontext
VPdgcolor VPdgdraw VPvdaccess VPvdrange
VPdgcontext VPdgvd VPvdcolor VPvdvarvalue
VPdgdf **VPdgviewport**

VPdgviewport Functions

VPdgNDCvp	Sets the viewport of a data group in normalized device coordinates.
VPdgscreenvp	Sets the viewport of a data group in screen coordinates.
VPdgvp	Sets the viewport of a data group in virtual coordinates.

VPdgNDCvp

Vpdgviewport Functions VP Routines

Sets the viewport of a data group in normalized device coordinates.

```
void
VPdgNDCvp (
       DATAGROUP dgp,
       FLOAT_POINT *ll,
       FLOAT_POINT *ur)
```

VPdgNDCvp defines the viewport that contains the data group, *dgp*, using normalized device coordinates, *ll* and *ur*. Normalized device coordinates are *float*s where (0.0, 0.0) corresponds to the lower left of the screen and (1.0, 1.0) corresponds to the upper right of the screen. For example, if the viewport was zoomed to twice the width and height of the screen, the viewport's normalized device coordinates could be *ll* = (0.0, 0.0) and *ur* = (2.0, 2.0).

VPdgscreenvp

Vpdgviewport Functions VP Routines

Sets the viewport of a data group in screen coordinates.

```
void
VPdgscreenvp (
       DATAGROUP dgp,
       RECTANGLE *scvp)
```

VPdgscreenvp defines the viewport that contains the data group, *dgp*, using screen coordinates. *scvp* is a pointer to a *RECTANGLE* data structure. In screen coordinates, (0, 0) corresponds to the lower left corner of the screen and the upper right corner depends on the size of the screen.

VPdgvp

Vpdgviewport Functions VP Routines

Sets the viewport of a data group in virtual coordinates.

```
void
VPdgvp (
       DATAGROUP dgp,
       RECTANGLE *vvp)
```

VPdgvp defines the viewport that contains the data group display using virtual coordinates. *vvp* is a pointer to a *RECTANGLE* data structure. In virtual coordinates, (0, 0) corresponds to the lower left corner of the screen and (32767, 32767) corresponds to the upper right corner.

VPvd

Vpvd Functions VP Routines

Manipulates the basic aspects of the variable descriptors.

See Also

VPdg, VPdgvd, VGdgvd, VGvd

Examples

The following code fragment creates a variable descriptor for a float variable called *data*.

```
VARDESC vdp;
LOCAL float data;
vdp = VPvdcreate ((ADDRESS) &data, V_F_TYPE);
```

VPvdclone is useful for applications where the same data is included in several different displays. The following code fragment illustrates this.

```
/* Create two data groups, dgp1, dgp2 */
...

/* Create a variable descriptor, vdp */
...

/* Add it to the two data groups */
VPdgvdadd (dgp1, VPvdclone (vdp));
VPdgvdadd (dgp2, vdp);
```

The following code fragment specifies the dimensions for several example variables.

```
{
LOCAL VARDESC scalar_vdp, vector_vdp, matrix_vdp, buffered_scalar_vdp;
LOCAL int scalar, vector[5], matrix[3][4], buffered_scalar[10];
. . .
scalar_vdp = VPvdcreate ((ADDRESS) &scalar, V_I_TYPE);
vector_vdp = VPvdcreate ((ADDRESS) vector, V_I_TYPE);
matrix_vdp = VPvdcreate ((ADDRESS) matrix, V_I_TYPE);
buffered_scalar_vdp = VPvdcreate ((ADDRESS) &buffered_scalar, V_I_TYPE);

VPvddim (scalar_vdp, 1, 1, 1);
VPvddim (vector_vdp, 1, 1, 5);
VPvddim (matrix_vdp, 1, 3, 4);
VPvddim (buffered_scalar_vdp, 10, 1, 1);
Describe (scalar_vdp);
Describe (vector_vdp);
Describe (matrix_vdp);
Describe (buffered_scalar_vdp);
. . .
}

Describe (vdp)
   VARDESC vdp;
{
   int d1, d2, d3;

   VGvddim (vdp, &d3, &d2, &d1);
```

```
   printf ("The variable is a");
   if (d3 > 1)
      printf (" time-buffered");
   if (d1 == 1)
      if (d2 == 1)
         printf (" scalar\n");
      else
         printf (" column vector\n");
   else if (d2 == 1)
      printf (" row vector\n");
   else
      printf (" matrix\n");
}
```

VPvd Functions

VPvdclone	Makes a copy of a variable descriptor.
VPvdcreate	Creates a variable descriptor.
VPvddelete	Deletes a variable descriptor, freeing the associated memory.
VPvdddereference	Decrements the variable descriptor's reference count.
VPvddim	Specifies the dimensions of a variable.
VPvdreference	Increments the variable descriptor's reference count.
VPvdtype	Defines the type of a variable descriptor.

VPvdclone

Vpvd Functions VP Routines

Makes a copy of a variable descriptor.

```
VARDESC
VPvdclone (
       VARDESC vdp)
```

VPvdclone allocates space for and makes a copy of the specified variable descriptor, *vdp*, without attaching the copy to any data structures. Returns the address of the copy. After the copy is made, changes to the original are not reflected in the copy.

VPvdcreate

Vpvd Functions VP Routines

Creates a variable descriptor.

```
VARDESC
VPvdcreate (
       ADDRESS var_address,
       int var_type)
```

VPvdcreate creates a new variable descriptor with appropriate default values. The routine selects a color from a table of default colors and assigns that color to the variable, while ensuring that consecutively created variable descriptors are not assigned the same color. The routine expects the base address of the variable and a flag describing its data type.

Valid data types are:

Flag	**Data Type**	**Size in bits**
V_C_TYPE	char	8
V_UC_TYPE	unsigned char, UBYTE	8
V_S_TYPE	short	16
V_US_TYPE	unsigned short	16
V_L_TYPE	int, LONG	32
V_UL_TYPE	unsigned int, ULONG	32
V_F_TYPE	float	32 (or 64 for some systems)
V_D_TYPE	double	64 (or 128 for some systems)
V_T_TYPE	NULL-terminated string	no set size

Returns a pointer to the newly created variable descriptor.

VPvddelete

Vpvd Functions VP Routines

Deletes a variable descriptor, freeing the associated memory.

```
void
VPvddelete (
       VARDESC vdp)
```

VPvddelete removes the variable descriptor from the list in which it resides and frees all allocated memory.

VPvddereference

Vpvd Functions VP Routines

Decrements the variable descriptor’s reference count.

```
void
VPvddereference (
       VARDESC vdp)
```

VPvddereference decrements the reference count for a variable descriptor. If the count reaches zero, it deletes the variable descriptor. The reference count starts at zero when the variable descriptor is created.

VPvddim

Vpvd Functions VP Routines

Specifies the dimensions of a variable.

```
void
VPvddim (
       VARDESC vdp,
       int dim3,
       int dim2,
       int dim1)
```

VPvddim specifies the dimensions of a variable as a scalar, vector, or matrix and specifies the vector or matrix size. *dim3* gives the number of time slices in the data. This allows buffering of the data before calling the display formatter and increases the routine's efficiency. *dim3* is typically set to 1. *dim2* specifies the number of rows in the variable; *dim1* specifies the number of columns.

VPvdreference

Vpvd Functions VP Routines

Increments the variable descriptor’s reference count.

```
VARDESC
VPvdreference (
       VARDESC vdp)
```

VPvdreference increments the reference count for a variable descriptor. The reference count starts at zero when the variable descriptor is created.

VPvdtype

Vpvd Functions VP Routines

Defines the type of a variable descriptor.

```
void
VPvdtype (
       VARDESC vdp,
       int var_type)
```

VPvdtype defines the type of the variable described by the variable descriptor. The type is defined when the variable descriptor is initially created using *VPvdcreate*. Valid data types are:

Flag	**Data Type**	**Size in bits**
V_C_TYPE	char	8
V_UC_TYPE	unsigned char, UBYTE	8
V_S_TYPE	short	16
V_US_TYPE	unsigned short	16
V_L_TYPE	int, LONG	32
V_UL_TYPE	unsigned int, ULONG	32
V_F_TYPE	float	32 (or 64 for some systems)
V_D_TYPE	double	64 (or 128 for some systems)
V_T_TYPE	NULL-terminated string	no set size

VPvdaccess

Vpvdaccess Functions VP Routines

Access routines for variable descriptors.

See Also

VPvd, VGvdaccess

Examples

The following code sets up a 10 element window in a 100 element vector. This window can move around in the vector to show different portions of it.

```
LOCAL int data[100], *datap;
datap = &data[0];
RECTANGLE *clipvp, **obsvps;
obsvps = NULL;

VGdgscreenvp (dgp, &clipvp);

/* datap initially points to beginning of array. */
vdp = VPvdcreate ((ADDRESS) &datap, V_I_TYPE);
VPvd_accmode (vdp, V_INDIR_ACCESS);
VPvddim (vdp, 1, 1, 10);

/* When the datagroup containing the variable descriptor is displayed, */
/* the display plots the first ten elements of the array data. */
VPdgdraw (dgp, clipvp, obsvps);
datap = &data[ 90 ];
```

```
/* The last 10 elements of the array are displayed. */
VPdgdraw (dgp, clipvp, obsvps);
```

The following code fragment is an access function that simulates a 20 by 20 identity matrix.

```
typedef struct
{
   int current_row, current_column;
   float LastValue;
}  ARG_BLOCK;

ADDRESS access_function (argp, i3, i2, i1)
   ARG_BLOCK *argp;
   int i3, i2, i1;
{

   /* Return address of the most recent actual value? */
   if (i3 == -2) return (ADDRESS) &argp->LastValue;

   /* Do we need to get the next entry? */
   if (i3 == -1)
      {
         /* Update the pointer to the current position in the array. */
         argp->current_column++;
         if (argp->current_column >= 20)
            {
               argp->current_column = 0;
               argp->current_row++;
            }
         i1 = argp->current_column;
         i2 = argp->current_row;
         i3 = 0;
      }
   if (i3 != 0 || i2 < 0 || i2 >= 20 || i1 < 0 || i1 >= 20)

      {

         /* Index out of range: return V_UNDEFINED. */
         argp->LastValue = -1;
         return (ADDRESS)-1;
      }

   else if (i1 == i2)
      {
      /* Along diagonal: return maximum value. */
         argp->LastValue = 1;
         return (ADDRESS)32767;
      }
   else
      {
      /* Return minimum value. */
         argp->LastValue = 0;
         return (ADDRESS)0;
      }
}

/* This section of code defines the variable descriptor.
 * Note that you don't need to specify a data address because
 * the access function simulates the data. */
VARDESC vdp;
ARG_BLOCK arg = { 0, 0, 0 };
```

```
vdp = VPvdcreate (NULL, V_L_TYPE);
VPvdaccess (vdp, (VGADDRACCESSFUNPTR) access_function, (ADDRESS) &arg,
             sizeof (ARG_BLOCK));
```

The following code fragment verifies that the variable descriptor base address is set properly.

```
float data, newdata;
VARDESC vdp;

RECTANGLE *clipvp, ** obsvps;
obsvps = NULL;

VGdgscreenvp (dgp, &clipvp);

vdp = VPvdcreate ((ADDRESS) &data, V_F_TYPE);

/* Change the variable being pointed to by variable descriptor */
VPvdbase (vdp, (ADDRESS) &newdata);

/* The last 10 elements of the array are displayed. */
VPdgdraw (dgp, clipvp, obsvps);
```

VPvdaccess Functions

VPvd_accmode	Sets the data access mode to direct or indirect.
VPvdaccess	Defines the data access function for the data described by a variable descriptor.
VPvdbase	Sets the base address of a variable descriptor.

VPvd_accmode

Vpvdaccess Functions VP Routines

Sets the data access mode to direct or indirect.

```
void
VPvd_accmode (
        VARDESC vdp,
        int accessmode)
```

VPvd_accmode specifies how to interpret the base address of the variable descriptor, *vdp*. If the access mode, *accessmode*, is direct, the base address is interpreted as the actual address of the data to be displayed. If *accessmode* is indirect, the address is interpreted as the address of a pointer to the data. The indirect mode allows moving the data without notifying DataViews and without resetting anything in the variable descriptor. By default, the variable descriptor is set to direct access.

The valid flag values are:

V_DIR_ACCESS	Direct access.
V_INDIR_ACCESS	Indirect access.
V_DS_BOUND	Indirect access through a DataViews data source variable.

VPvdaccess

Vpvdaccess Functions VP Routines

Defines the data access function for the data described by a variable descriptor.

```
void
VPvdaccess (
        VARDESC vdp,
        VGADDRACCESSFUNPTR fcnp,
        ADDRESS argp,
        int argsize)

    ADDRESS
    fcnp (
          ADDRESS argp,
          int i3,
          int i2,
          int i1)
```

VPvdaccess specifies an access function that is used by the dynamic update routines and the display formatter to get the value of the data associated with the variable descriptor, *vdp*. Novice users can disregard this function since the default access function is usually sufficient. The remaining information in this section is intended for sophisticated DataViews users who are writing their own access functions.

The access function returns a value in the range [0,32767], where 0 corresponds to the data's minimum value as set by a call to *VPvd_irange* or *VPvd_drange*, and 32767 corresponds to the data's maximum value. If the value is undefined, the routine returns -1.

Access functions used by display formatters assume that data being displayed has three dimensions, any of which can be of size one. Thus, a scalar has dimension (1,1,1). This dimension is set by calling *VPvddim*. The display formatter indexes through the data, calling the access function as follows:

```
data_value = access_function (argp, i3, i2, i1);
```

where *i1* is in the range [0,dim1], *i2* is in the range [0,dim2], and *i3* is in the range [0,dim3] as set by *VPvddim*.

Alternatively, *i3* can have special values that the access function must respond to:

If *i3* = -1, the access function increments to the next location and returns the value contained in that new location. In this case, the other index arguments are ignored. This optimizes the case where the display formatter is stepping through a matrix. The display formatter calls the access function with a non-negative value first to initialize the location.

If *i3* = -2, the access function returns a pointer to the most recently returned actual data value, instead of to the normalized value. The pointer points to a *float* or a *LONG*, depending on the type of the variable descriptor. This is for cases where the display formatter needs a more exact representation, such as the digits graph.

The access function can return an integer or an address, so it is declared to be of type *ADDRESS*, which is large enough to contain an *int*.

The argument block pointed to by *argp* is copied and saved as part of the variable descriptor. The pointer to this copy is passed to the access function when it is actually called.

This function is not intended for text variable descriptors of type *V_T_TYPE*.

VPvdbase

Vpvdaccess Functions VP Routines

Sets the base address of a variable descriptor.

```
void
VPvdbase (
       VARDESC vdp,
       ADDRESS newbase)
```

VPvdbase sets the base address of a variable in a variable descriptor, *vdp*. This replaces the base address defined when the variable descriptor was created using *VPvdcreate*. The variable's base address is its memory location.

VPvdcolor

Vpvdcolor Functions VP Routines

Utilities for specifying the variable color.

See Also

VPvd, VGvdctt

Example

The following code fragment sets up a color threshold table that displays the data in green if it is in the lower 90% of its range, and in red if it is in the top 10% of its range.

```
COLOR_THRESHOLD ct[2];

ct[0].threshcolor.rgb_rep.rgb_rep_flag = -1;
ct[0].threshcolor.rgb_rep.red = 0;
ct[0].threshcolor.rgb_rep.green = 255;
ct[0].threshcolor.rgb_rep.blue = 0;
ct[0].upperlimit = 9 * 32767 / 10;

ct[1].threshcolor.rgb_rep.rgb_rep_flag = -1;
ct[1].threshcolor.rgb_rep.red = 255;
ct[1].threshcolor.rgb_rep.green = 0;
ct[1].threshcolor.rgb_rep.blue = 0;
ct[1].upperlimit = 32767;

VPvdctt (vdp, 2, ct);
```

VPvdcolor Functions

VPvdclrndx	Sets the color using the lookup table index.
VPvdctt	Specifies the color threshold table.
VPvdcttscale	Specifies linear or log scale for a color threshold table.
VPvdrgb	Specifies the color using the RGB format.

VPvdclrndx

Vpvdcolor Functions VP Routines

Sets the color using the lookup table index.

```
void
VPvdclrndx (
       VARDESC vdp,
       int clrndx)
```

VPvdclrndx sets the color using the device-dependent color lookup table index.

VPvdctt

Vpvdcolor Functions VP Routines

Specifies the color threshold table.

```
void
VPvdctt (
       VARDESC vdp,
       int num_colors,
       COLOR_THRESHOLD *ctp)
```

VPvdctt specifies a color threshold table for the variable. This table associates colors with ranges of data values. It contains a list of color specifications in either RGB or color index form, together with associated normalized data values (thresholds). The display formatter uses the last color with an associated threshold greater than or equal to the data value. Thresholds are normalized in the range [0,32767], where 0 corresponds to the variable's minimum value and 32767 corresponds to its maximum value as set by a call to *VPvd_irange* or *VPvd_drange*.

A color threshold table has the following structure:

1: color, limit;
2: color, limit;
...
n: color, limit;

where limit[i] > limit[j] when i>j; limit[n] = 32767. The data is displayed using color[i] when the normalized data value is limit[i-1] < value <= limit[i]; and where limit[0] is defined as zero.

VPvdcttscale

Vpvdcolor Functions VP Routines

Specifies linear or log scale for a color threshold table.

```
void
VPvdcttscale (
       VARDESC vdp,
       int log_flag)
```

VPvdcttscale converts the limits of the color threshold table attached to *vdp* to log or linear, depending on the value of *log_flag*. *YES* indicates that the color threshold table limits should be log. This function is called automatically by *VPvdlog*, so the user should call it only to convert the limits of a color threshold table that is attached after the call to *VPvdlog*.

VPvdrgb

Vpvdcolor Functions VP Routines

Specifies the color using the RGB format.

```
void
VPvdrgb (
       VARDESC vdp,
       int r,
       int g,
       int b)
```

VPvdrgb sets the color in RGB format. RGB format specifies a color using three numbers in the range [0,255], where each number corresponds to the intensity of one of the additive primary colors: red, green, and blue. The display formatter selects the color closest to the specified color, given the color lookup table for the device.

VPvdcontext

Vpvdcontext Functions VP Routines

Manages the context for variable descriptors.

See Also

VPdgcontext, VPvd, VGvdcontext

Examples

The following code fragment illustrates how to set a value label.

```
VPvdvallabel (vdp, "Velocity in feet per second");
```

The following code fragment illustrates how to name a variable descriptor.

```
VARDESC vdp;
LOCAL float revenue;

vdp = VPvdcreate ((ADDRESS) &revenue, V_F_TYPE);
VPvdvarname (vdp, "REVENUE");
```

VPvdcontext Functions

VPvdlog	Specifies log or linear scaling for a variable descriptor.
VPvdltype	Sets the line type of a variable descriptor.
VPvdlwidth	Sets the line width of a variable descriptor.
VPvdsymbol	Defines the symbol used to encode a data value.
VPvdticlabfcn	Assigns the tick labeling function to a value axis.
VPvdvallabel	Specifies the value axis label for a variable.
VPvdvarname	Specifies the name of a variable.

VPvdlog

Vpvdcontext Functions VP Routines

Specifies log or linear scaling for a variable descriptor.

```
void
VPvdlog (
       VARDESC vdp,
       int flag)
```

VPvdlog specifies whether the variable is of log type. If the variable has a log flag of *YES*, the display formatter computes the log before displaying the variable.

VPvdltype

Vpvdcontext Functions VP Routines

Sets the line type of a variable descriptor.

```
void
VPvdltype (
       VARDESC vdp,
       int type)
```

VPvdltype sets the line type of the variable descriptor, *vdp*, to the line type, *type*. *type* is a number between 1 and 15 (inclusive) corresponding to one of 15 line types, which have device dependent interpretations. The default value of 1 corresponds to a solid black line.

VPvdlwidth

Sets the line width of a variable descriptor.

```
void
VPvdlwidth (
       VARDESC vdp,
       int width)
```

VPvdlwidth sets the width of the line of the variable descriptor, *vdp* to the width, *width*. The minimum width is 1; the maximum width is 255. Reasonable widths are in the range of 1 to 5, where 5 generates a line 5 pixels wide. The default width is 1.

VPvdsymbol

Vpvdcontext Functions VP Routines

Defines the symbol used to encode a data value.

```
void
VPvdsymbol (
       VARDESC vdp,
       int symbol)
```

VPvdsymbol sets the symbol field in the attributes section for the variable descriptor. *symbol* specifies the marker used to display the data defined by the variable descriptor. This symbol is not used by some display formatters.

The symbol flag can have one of the following values:

V_NULL_SYMBOL	’ ’	Default
V_ASTERISK	’*’	Asterisk
V_DOT	’.’	Dot
V_PLUS	’+’	Plus
V_CROSS	’x’	X
V_DIAMOND	’d’	Diamond
V_FILLED_DIAMOND	’D’	Filled Diamond
V_CIRCLE	’o’	Circle
V_FILLED_CIRCLE	’O’	Filled Circle
V_BOX	’r’	Box
V_FILLED_BOX	’R’	Filled Box
V_TRIANGLE	’t’	Triangle (apex up)
V_FILLED_TRIANGLE	’T’	Filled Triangle (apex up)
V_INVERTED_TRIANGLE	’v’	Triangle (apex down)
V_FILLED_INVERTED_TRIANGLE	’V’	Filled Triangle (apex down)
V_TRIANGLE_RIGHT	’)’	Triangle (apex right)
V_FILLED_TRIANGLE_RIGHT	’>’	Filled Triangle (apex right)
V_TRIANGLE_LEFT	’(’	Triangle (apex left)
V_FILLED_TRIANGLE_LEFT	’<’	Filled Triangle (apex left)
V_VERTICAL_LINE	’|’	Vertical Line
V_HORIZONTAL_LINE	’-’	Horizontal Line

If the symbol value is *NULL*, the default display formatter is used.

VPvdticlabfcn

Vpvdcontext Functions VP Routines

Assigns the tick labeling function to a value axis.

```
void
VPvdticlabfcn (
       VARDESC vdp,
       DV_TICLABELFUNPTR ticlabelfunc,
       char *argp,
       int argsize)

   void
   ticlabelfunc (
          ADDRESS argpcopy,
          double *value,
          ADDRESS output,
          TIC_DATA *tdp)
```

VPvdticlabfcn assigns a tick labeling function for the value axis, *ticlabelfunc*, to a variable descriptor, and allocates memory for a copy of the function's arguments, a structure of *argsize* bytes at address *argp*.

VPdgticlabfcn describes the tick labeling function *ticlabelfunc*, its arguments, and how it is called.

VPvdvallabel

Vpvdcontext Functions VP Routines

Specifies the value axis label for a variable.

```
void
VPvdvallabel (
       VARDESC vdp,
       char *label)
```

VPvdvallabel assigns a label to the value axis associated with a variable. The value axis label of a variable typically appears on the vertical axis of a display formatter using scalar data when that variable is the first one attached to the data group.

VPvdvarname

Vpvdcontext Functions VP Routines

Specifies the name of a variable.

```
void
VPvdvarname (
       VARDESC vdp,
       char *name)
```

VPvdvarname assigns the character string to be used as the name of the variable.

VPvdrange

Vpvdrange Functions VP Routines

The variable value range utilities.

See Also

VPvd, VGvdrange

Examples

The following calls are equivalent:

```
VPvd_drange (vdp, 0.0, 100.0);

VPvd_irange (vdp, 0, 100);
```

VPdg	VPdgdfargs	VPvd	VPvdcontext
VPdgcolor	VPdgdraw	VPvdaccess	**VPvdrange**
VPdgcontext	VPdgvd	VPvdcolor	VPvdvarvalue
VPdgdf	VPdgviewport		

Functions

VPvd_drange	Sets the range delimiters as double precision floats.
VPvd_irange	Sets the range delimiters as integers.

These routines define the highest and lowest values the specified variable can have. If the data value is outside this range, it is adjusted to the closest value by the display formatter.

You can use either of these routines independently of the variable type, since the routines handle the necessary conversions.

VPvd_drange

Vpvdrange Functions VP Routines

Sets the range delimiters as double precision floats.

```
void
VPvd_drange (
       VARDESC vdp,
       double low,
       double high)
```

VPvd_drange specifies the range delimiters as double precision floating point numbers.

VPvd_irange

Vpvdrange Functions VP Routines

Sets the range delimiters as integers.

```
void
VPvd_irange (
       VARDESC vdp,
       int low,
       int high)
```

VPvd_irange specifies the range delimiters as integers.

VPvdvarvalue

Vpvdvarvalue Functions VP Routines

Routines to set variables associated with variable descriptors.

See Also

VPvd, VGvdvarvalue

VPvdvarvalue Functions

VPvdDValue	Puts a double value in the base address.
VPvdIValue	Puts an integer value in the base address.
VPvdSValue	Puts a string value in the base address.
VPvdValue	Puts a value in the base address.

VPvdDValue

Vpvdvarvalue Functions VP Routines

Puts a double value in the base address.

```
void
void (
       VARDESC vdp,
       double value)
```

VPvdDValue puts a double value at the base address. If the destination type is a string, the routine formats the number in ASCII and copies the ASCII value to the destination.

VPvdIValue

Vpvdvarvalue Functions VP Routines

Puts an integer value in the base address.

```
void
VPvdIValue (
       VARDESC vdp,
       int value)
```

VPvdIValue puts an integer value at the base address. If the destination type is a string, the routine formats the number in ASCII and copies the ASCII value to the destination.

VPvdSValue

Vpvdvarvalue Functions VP Routines

Puts a string value in the base address.

```
void
VPvdSValue (
       VARDESC vdp,
       char *value)
```

VPvdSValue puts a string value at the base address. If the destination type is numeric, the routine tries to get the number from the string by scanning it. If it fails to scan it, *value* is not set.

VPvdValue

Vpvdvarvalue Functions VP Routines

Puts a value in the base address.

```
void
VPvdValue (
       VARDESC vdp,
       ADDRESS value)
```

VPvdValue puts a value at the position specified by the variable descriptor. The variable is assumed to be a scalar so it puts the value in the position specified by the base address. The type of the value argument depends on the type of the variable. If the variable is one of the integer types, then *value* is a pointer to an *int*. If the variable is a floating point type, *value* must be a pointer to a *double*. If the variable is text type, *value* must be a pointer to a *NULL*-terminated string of *char*s. With a text type variable, *VPvdValue* checks the space available before copying the string to the address. The available space is defined by the dimension of the variable. If the dimension is 1 (scalar), the available space is the length of the current string.

VT Routines

Hash and symbol table management routines.

VT Modules

All modules in the *VT* layer require the following headers:

```
#include "std.h"
#include "dvstd.h"
#include "VTfundecl.h"
```

VThash	Hashed symbol table management routines.
VTsymbol	Symbol table management routines.

VThash

Vthash Functions VT Routines

Hashed symbol table management routines. Hashed symbol tables are dynamic linear hash tables, which are incrementally expanded according to an algorithm described in the *Communications of the Association for Computing Machinery*, April 1988, Vol. 31, No. 4, pp. 446-457. A hash table is composed of a header and a list of hash table nodes pointed to by the header. Each hash table node is composed of a key, a key code, and a value.

The key is an unsigned long integer or a pointer to a user-defined data structure such as a string containing a symbol name. When the key is user-defined, the data structure must be maintained by the user and should not be changed while it is in the table.

The key code, which is always an unsigned long integer, is the number that is hashed to determine the position of the node in the table. The key code is a user-defined function of the key.

The value is an unsigned long integer or a pointer to a user-defined data structure. This is the entity associated with the key and retrieved when the key is referenced. The caller is responsible for managing the allocation of the data structures pointed to by the key or value. This means that symbol names must stay around as long as the keys that point to them are in a symbol table. Similarly, if the symbol node value is a pointer, you must make sure the symbol node value always points to something meaningful. When you delete a node, you must free any memory used to store the objects pointed to by the node.

The routines use the following naming conventions: *ht* for hash table, and *hn* for hash table node.

Example

This code segment creates a hash table for data areas in a program:

```
static int idata1, idata2, idata3;
typedef ADDRESS HASHTABLE, HASHNODE, HASHVALUE;
typedef char *HASHKEY;
HASHTABLE ht;
HASHNODE hn;

/* Create hash table for integer data. */
ht = VThtcreate ("Integer data",
             (VTHTCONVERTFUNPTR) VThtstrconvert, (VTHTCOMPAREFUNPTR) strcmp);
VThthninsert (ht, "idata1", &idata1);
VThthninsert (ht, "idata2", &idata2);
VThthninsert (ht, "idata3", &idata3);

/* Print the value for data location &idata1 */
hn = VThtvalfind (ht, NULL, &idata1);
printf ("The name of the location is: %s",
                                      VThnkey (hn));

/* Print the value associated with the name idata1 */
hn = VThtkeyfind (ht, "idata1");
```

```
printf ("The value associated with 'idata1' is %d.", *(int *) VThnvalue (hn));
. . .
VThtdestroy (ht, NULL, NULL);
```

VThash Functions

VThnkey	Returns specified hash table node key.
VThnsetvalue	Sets hash table node value.
VThnvalue	Returns hash table node value.
VThtcountval	Returns number of nodes with specific value.
VThtcreate	Creates hash table, no size specified.
VThtdestroy	Destroys hash table.
VThtget	Returns address of hash table.
VThthnget	Returns address of indexed node.
VThthninsert	Inserts node in hash table.
VThthnremove	Removes node from hash table.
VThtkeyfind	Returns address of specified key in hash table.
VThtlen	Returns number of nodes in hash table.
VThtstats	Prints statistics about the hash table.
VThtstrconvert	Converts string keys to key codes.
VThttraverse	Traverses hash table and calls specified function.
VThtvalfind	Finds hash table node with specified value.

For the purposes of this description the data structures are defined as follows:

```
typedef ADDRESS HASHTABLE;
typedef ADDRESS HASHNODE;
typedef ULONG KEY; or typedef ADDRESS KEY;
typedef ULONG VALUE; or typedef ADDRESS VALUE;
```

VThnkey

VThash Functions VT Routines

Returns the key associated with the specified hash table node.

```
KEY
VThnkey (
       HASHNODE hnp)
```

VThnsetvalue

VThash Functions VT Routines

Sets the value associated with the hash table node.

```
void
VThnsetvalue (
        HASHNODE hnp,
        VALUE newvalue)
```

VThnvalue

VThash Functions VT Routines

Returns the value associated with the hash table node.

```
VALUE
VThnvalue (
        HASHNODE hnp)
```

VThtcountval

VThash Functions VT Routines

Returns a count of the nodes in the hash table with the specific value.

```
int
VThtcountval (
       HASHTABLE htp,
       VALUE searchval)
```

VThtcreate

VThash Functions VT Routines

Creates hash table, no size specified.

```
HASHTABLE
VThtcreate (
       char *table_name,
       VTHTCONVERTFUNPTR convert_key,
       VTHTCOMPAREFUNPTR compare)

    ULONG
    convert_key (
           KEY newkey)

    int
    compare (
           KEY key1,
           KEY key2)
```

VThtcreate creates a new hash table with the specified *table_name*. If a hash table with that name already exists, returns the address of that hash table. Otherwise returns the address of the newly created hash table. If *table_name* is *NULL*, a table is created with no name.

When a table is created, two functions can be associated with it. The first is *convert_key*, which converts the key into an unsigned long integer key code. If this function is not specified, the key code is the same as the key. If the key is a pointer to a string, use *VThtstrconvert* to convert the string to a key code. The second function that can be specified is *compare*, which compares two keys. This function should be specified if the keys are pointers to user-defined data structures. It should return a zero if the keys are equal and non-zero if they are different. If the keys are pointers to strings, you can use the system function *strcmp*.

VThtdestroy

VThash Functions VT Routines

Destroys hash table.

```
void
VThtdestroy (
        HASHTABLE htp,
        VTHTFREEKEYFUNPTR freekey,
        VTHTFREEVALFUNPTR freevalue)

    void
    freekey (
            KEY key)

    void
    freevalue (
            VALUE value)
```

VThtdestroy destroys the hash table and frees the memory required to store the hash table. In addition, specifying the functions *freekey* or *freevalue* calls the functions with the key or value as the node is freed. This lets you write a function to free the node and the data structures pointed to by the node at the same time.

VThtget

VThash Functions VT Routines

Returns the address of the hash table with the specified name.

```
HASHTABLE
VThtget (
       char *ht_name)
```

VThthnget

VThash Functions VT Routines

Returns address of indexed node.

```
HASHNODE
VThthnget (
       HASHTABLE htp,
       int index)
```

VThthnget returns the address of the *index*-th node in the specified hash table. Note that, as in C, indexing is zero based, which means the index of the first node is zero and the index of the last node is the hash table length (returned by *VThtlen*) minus one. It is inefficient to use this routine to index through a table since the hash table is not sorted in any predictable, useful way. This is different from the *VTs* * symbol table routines which are sorted and easily indexed. If you need to apply a function to the entries in a table it is better to use *VThttraverse*.

VThthninsert

VThash Functions VT Routines

Inserts a node in a hash table and returns the address of the inserted node.

```
HASHNODE
VThthninsert (
       HASHTABLE htp,
       KEY newkey,
       VALUE newvalue)
```

VThthnremove

VThash Functions VT Routines

Removes the specified node from a hash table.

```
void
VThthnremove (
       HASHTABLE htp,
       HASHNODE hnp)
```

VThtkeyfind

VThash Functions VT Routines

Returns address of specified key in hash table.

```
HASHNODE
VThtkeyfind (
       HASHTABLE htp,
       KEY searchkey)
```

VThtkeyfind returns the address of the hash table node that has the specified key. Returns *NULL* if *searchkey* is not associated with a node.

VThtlen

VThash Functions VT Routines

Returns number of nodes in the specified hash table.

```
int
VThtlen (
       HASHTABLE htp)
```

VThtstats

VThash Functions VT Routines

Prints statistics about the hash table.

```
void
VThtstats (
       HASHTABLE htp)
```

VThtstrconvert

VThash Functions VT Routines

Converts string keys to key codes.

```
ULONG
VThtstrconvert (
       char *s)
```

VThtstrconvert converts a key that is a pointer to a string into a key code. The routine scrambles the characters in the string into an unsigned long integer, cycling through the bytes in the key code and XORing the characters of the string into it. This generates a number that creates a good distribution of hash codes.

VThttraverse

VThash Functions VT Routines

Traverses hash table and calls specified function.

```
void
VThttraverse (
       HASHTABLE htp,
       VTHTTRAVERSEFUNPTR fcn,
       ADDRESS args)

    void
    fcn (
           KEY key,
           VALUE value,
           ADDRESS args)
```

VThttraverse traverses the hash table, calling the specified function with the key and value from each node as well as the *args* parameter.

VThtvalfind

VThash Functions VT Routines

Finds hash table node with specified value.

```
HASHNODE
VThtvalfind (
       HASHTABLE htp,
       HASHNODE hnp,
       VALUE searchval)
```

VThtvalfind finds the next hash table node that has the specified value. The routine expects a pointer to a hash table, a pointer to hash node in that table, and a value. The routine starts searching at the next node after the given node. If the node pointer is *NULL*, it starts at the beginning. Returns the address of the next node with the specified value. Returns *NULL* if there is no such node.

VTsymbol

Vtsymbol Functions   VT Routines

Symbol table management routines. A symbol table is composed of a header and a list of symbol table nodes pointed to by the header. Each symbol table node is composed of a key, which is usually a pointer to a character string (the symbol), and a value, which is usually a pointer to the named by the object. The list of symbol table nodes is sorted in increasing order by key, where the order of the keys is defined by a comparison function. A pointer to the comparison function is kept in the symbol table header. The default comparison function interprets the keys as addresses to strings and returns the lexicographic ordering of the two strings. For more information about the comparison function, see the description of *VTstcreate*.

The *VT* routines allocate space from the heap for the tables. The caller must manage the memory space for the objects pointed to by the symbol table nodes.This means that symbol names must stay around as long as the keys that point to them are in a symbol table. Similarly, if the symbol node is a pointer, you must make sure the symbol node value always points to something meaningful. When you delete a node, you must free any memory used to store the objects pointed to by the node.

The routines use the following naming conventions: *st* for symbol table; and *sn* for symbol table node. Note that the declarations refer to data types *SYMTABLE* (symbol table) and *SYMNODE* (symbol node). These types are defined in *dvstd.h*.

Example

This code fragment creates symbol tables for data areas in a program:

```
static float data1, data2, data3;
static int idata1, idata2, idata3;

SYMTABLE float_st, int_st;
SYMNODE sn;

/* Create the symbol table for floating point data. */
float_st = VTstcreate ("Float data", NULL);
VTstsninsert (float_st, "data1", (int *) &data1);
VTstsninsert (float_st, "data2", (int *) &data2);
VTstsninsert (float_st, "data3", (int *) &data3);

/* Create the symbol table for integer data. */
int_st = VTstcreate ("Integer data", NULL);
VTstsninsert (int_st, "idata1", &idata1);
VTstsninsert (int_st, "idata2", &idata2);
VTstsninsert (int_st, "idata3", &idata3);

/* Print the symbol for data location &idata1 */
printf ("The name of the location is: %s",
   VTsnkey (VTstvalfind (int_st, NULL, &idata1)));
```

Diagnostics

Since these routines use *NULL* keys to terminate a symbol table, do not use *NULL* as a key value. If you need to include *NULL* in symbol tables, make the keys pointers to a *NULL* object.

VTsymbol Functions

#include hashtypes.h

VTsnkey	Returns specified symbol table node key.
VTsnprint	Prints specified symbol table node contents.
VTsnsetvalue	Sets symbol table node value.
VTsnvalue	Returns symbol table node value.
VTstcountval	Returns number of nodes with specific value.
VTstcreate	Creates symbol table, no size specified.
VTstdestroy	Destroys symbol table.
VTstget	Returns address of symbol table.
VTstkeyfind	Returns address of specified key in symbol table.
VTstlen	Returns number of nodes in symbol table.
VTstsizecreate	Creates symbol table, specifies size.
VTstsnget	Returns address of indexed node.
VTstsninsert	Inserts node in symbol table.
VTstsnremove	Removes node from symbol table.
VTsttraverse	Traverses symbol table, calls specified function.
VTstvalfind	Finds symbol table node with specified value.

VTsnkey

Returns the key associated with the specified symbol table node.

```
char *
VTsnkey (
       SYMNODE snp)
```

VTsnprint

Vtsymbol Functions VT Routines

Prints specified symbol table node contents.

```
void
VTsnprint (
       char *key,
       int *value)
```

VTsnprint prints the contents of the specified symbol table node, assuming that *key* is a pointer to a string and *value* is an address.

VTsnsetvalue

Vtsymbol Functions VT Routines

Sets the value associated with the symbol table node.

```
void
VTsnsetvalue (
       SYMNODE snp,
       int *newvalue)
```

VTsnvalue

Vtsymbol Functions VT Routines

Returns the value associated with the symbol table node.

```
int *
VTsnvalue (
       SYMNODE snp)
```

VTstcountval

Vtsymbol Functions VT Routines

Returns a count of the nodes with the specified value in the symbol table.

```
int
VTstcountval (
       SYMTABLE stp,
       int *searchval)
```

VTstcreate

Vtsymbol Functions VT Routines

Creates symbol table, no size specified.

```
SYMTABLE
VTstcreate (
       char *table_name,
       VTSTCOMPAREFUNPTR compare_function)

    int
    compare_function (
          char *K1,
          char *K2)
```

VTstcreate and *VTstsizecreate* create a new symbol table with the specified *table_name*. If a symbol table with that name already exists, these routines return the address of that symbol table. Otherwise, they return the address of the newly created symbol table. These routines associate a compare function with the table. This function is used to order the keys in the table. It must work as follows: given two keys such as *k1* and *k2*, it must return a negative integer if $k1 < k2$, a zero if $k1 = k2$, and a positive integer if $k1 > k2$. If no compare function is specified, *VTstcreate* and *VTstsizecreate* assume that the keys are pointers to character strings and use a default compare function that compares the strings. This default compare function returns the result of comparing the strings lexicographically. *VTstsizecreate* differs from *VTstcreate* in that the former lets the caller specify an initial size for the symbol table. This saves memory allocations when you know that the symbol table is going to be large.

VTstdestroy

Vtsymbol Functions   VT Routines

Destroys the symbol table and frees the memory required to store the symbol table.

```
void
VTstdestroy (
       SYMTABLE stp)
```

VTstget

Vtsymbol Functions VT Routines

Returns the address of the symbol table with the specified name.

```
SYMTABLE
VTstget (
       char *st_name)
```

VTstkeyfind

Vtsymbol Functions VT Routines

Returns address of specified key in symbol table.

```
SYMNODE
VTstkeyfind (
       SYMTABLE stp,
       char *searchkey)
```

VTstkeyfind returns the address of the symbol table node that has the specified key. Returns *NULL* if *searchkey* is not associated with a node.

VTstlen

Vtsymbol Functions VT Routines

Returns the number of nodes in the specified symbol table.

```
int
VTstlen (
       SYMTABLE stp)
```

VTstsizecreate

Creates symbol table, specifies size.

```
SYMTABLE
VTstsizecreate (
       char *table_name,
       VTSTCOMPAREFUNPTR compare_function,
       int initial_size)

    int
    compare_function (
           char *K1,
           char *K2)
```

VTstsizecreate creates a symbol table, using a given initial size. See *VTstcreate* above.

VTstsnget

Vtsymbol Functions VT Routines

Returns address of indexed node.

```
SYMNODE
VTstsnget (
       SYMTABLE stp,
       int index)
```

VTstsnget returns the address of the *index*-th node in the specified symbol table. Note that, as in C, indexing is zero based, which means the index of the first node is zero and the index of the last node is the symbol table length (returned by *VTstlen*) minus one.

VTstsninsert

Vtsymbol Functions VT Routines

Inserts node in symbol table.

```
SYMNODE
VTstsninsert (
       SYMTABLE stp,
       char *newkey,
       int *newvalue)
```

VTstsninsert inserts a node in a symbol table. Insertion works fastest if the nodes are added in order because this routine performs a special check to see if the new item goes at the end of the list. The symbol table is sorted in increasing order according to the associated compare function. With the default compare function, the table is sorted in alphabetical order. This routine returns the address of the inserted node.

VTstsnremove

Removes the specified node from a symbol table.

```
void
VTstsnremove (
       SYMTABLE stp,
       SYMNODE snp)
```

VTsttraverse

Traverses symbol table, calls specified function.

```
void
VTsttraverse (
       SYMTABLE stp,
       VTSTTRAVERSEFUNPTR fcn,
       ADDRESS args)

   void
   fcn (
          char *key,
          int *value,
          ADDRESS args)
```

VTsttraverse traverses the symbol table, calling the specified function with the key and value from each node as well as the *args* parameter.

VTstvalfind

Vtsymbol Functions VT Routines

Finds symbol table node with specified value.

```
SYMNODE
VTstvalfind (
       SYMTABLE stp,
       SYMNODE snp,
       int *searchval)
```

VTstvalfind finds the next symbol table node that has the specified value. The routine expects a pointer to a symbol table, a pointer to symbol node in that table, and a value. The routine starts searching at the next node in the symbol table after the given node. If the node pointer is *NULL*, it starts at the beginning. Returns the address of the next node with the specified value, or *NULL* if there is no such node.

VU Routines

Vu Routines

Utility routines.

VU Modules

All modules in the *VU* layer require the following *#include* files:

```
#include "std.h"
#include "dvstd.h"
#include "dvtools.h"
#include "VUfundecl.h"
```

Any special *#include* files required by a particular module are listed in the synopsis section for that module.

VUaxis	Axis descriptor creation and drawing utilities.
VUcopyright	Displays the DataViews copyright notice in the center of the screen.
VUdebug	Prints data structure utilities for VP/VG layer.
VUdevice	Graphics device utility routines.
VUexit	Closes all open devices and exits cleanly.
VUpixrep	Routines to manage pixrep structures (*px*).
VUregistry	Routines to query the Windows Registry.
VUsearchpath	Utility routines.
VUstring	Routines for managing strings.
VUstrlist	Routines for managing lists of string pointers.
VUtextarray	Low-level functions for manipulating hardware text
VUticlabel	Axis tick mark labeling routine.
VUtraverse	Data group function utilities.
VUvplist	Routines for managing viewport lists.
VUwinevent	Reports window events at a specified level of detail.

VUaxis

VUaxis Functions VU Routines

Axis descriptor creation and drawing utilities. These routines are currently intended for use only by programmers writing their own display formatters. The axis descriptor, *AXISDESC*, is of type *ADDRESS*, and stores information about graph axis labels.

An axis has many attributes including labels, tick marks, grid lines, color, and start and end values. Major tick values are integer multiples of 1, 2, 5, or 10 × 10±n where n is called the **base exponent**. The number of divisions marked by minor ticks between the major ticks can be 1, 2, 5, or 10. Grid lines, when displayed, occur at major ticks. The axes are created, managed, and drawn using the routines below.

See Also

The flags are defined in the include file *dvaxis.h*. An example of *VUaxis* routines usage is found in the file *axis.c* in the *programs* directory. *GRbackcolor* can be called to change the background color before drawing.

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUaxis Functions

VUaxCreate	Creates and returns an axis descriptor.
VUaxDestroy	Destroys an axis descriptor, freeing its memory.
VUaxDraw	Draws an axis according to the axis descriptor.
VUaxDrawRange	Draws a portion of the axis.
VUaxGet	Gets axis descriptor attribute fields.
VUaxSet	Sets axis descriptor attribute fields.
VUaxSetupForDrawing	Prepares the axis for drawing.

```
#include "dvaxis.h"
```

VUaxCreate

VUaxis Functions VU Routines

Creates and returns an axis descriptor.

```
AXISDESC
VUaxCreate (
       double Start,
       double End)
```

VUaxCreate creates an axis descriptor given a start value, *Start*, and an end value, *End*. Returns a pointer to the axis descriptor.

VUaxDestroy

VUaxis Functions VU Routines

Destroys an axis descriptor, freeing the axis descriptor data structure memory, *axis*.

```
void
VUaxDestroy (
       AXISDESC axis)
```

VUaxDraw

VUaxis Functions VU Routines

Draws an axis according to the axis descriptor.

```
void
VUaxDraw (
       AXISDESC axis)
```

VUaxDraw draws the axis according to the flags defined by calling *VUaxSet*. Once the axis has been drawn, no attribute can be changed except for the start value, which is used for handling wrap-around, and grid and axis colors.

VUaxDrawRange

VUaxis Functions VU Routines

Draws a portion of the axis.

```
void
VUaxDrawRange (
       AXISDESC axis,
       double StartValue
       double EndValue)
```

VUaxDrawRange draws a portion of the axis. The portion drawn is determined by the given start and end values.

VUaxGet

VUaxis Functions VU Routines

Gets axis descriptor attribute fields.

```
void
VUaxGet (
       AXISDESC axis,
       int flag,
       ADDRESS arg)
```

VUaxGet gets certain attributes of an axis descriptor. This routine must be preceded by a call to *VUaxSetupForDrawing*, *VUaxDraw*, or *VUaxDrawRange*. The attribute field flags, defined in the include file *dvaxis.h*, are listed below, together with the pointer to the data type, *arg*.

Flag	*arg* Type	Comment
AXIS_BOUNDS	*RECTANGLE **	Rectangle containing offsets for ticks and labels to be added to axis start and end points.
BASE_EXPONENT	*int **	Base exponent (see above).
INITIAL_TICK_VALUE	*double **	Value associated with first tick.
INITIAL_TICK_POINT	*DV_POINT **	Position in screen coordinates of first tick.
MAJOR_PIXEL_GAP	*double **	Actual screen distance between major ticks.
MAJOR_VALUE_GAP	*double **	Actual value difference between major ticks.
MINOR_PIXEL_GAP	*double **	Actual screen difference between minor ticks.
MINOR_VALUE_GAP	*double **	Actual value difference between minor ticks.
MINOR_TICKS_PER_MAJOR	*int **	Number of minor ticks per major tick (1, 2, 5, or 10).
TICK_LABEL_EXTENT	*DV_POINT **	Size (in pixels) of largest tick label.

VUaxSet

Sets axis descriptor attribute fields.

```
void
VUaxSet (
        AXISDESC axis,
            int flag, <type> value,
            int flag, <type> value,
            ...,
        0)
```

VUaxSet sets the attributes of an axis descriptor. The attribute list must end in 0. The argument list begins with the axis descriptor, which is followed by flag-value pairs. *value* must correspond to the type of *flag* used. The flags that define the axis attribute fields are listed below. The flags are defined in the include file *dvaxis.h*. The first group of flags are required by the VUaxis routines and must be set by the programmer. The second group of flags are parameters that the programmer can change. The third group of flags lets the programmer bypass the routine's default settings to set tick spacing, values, and labels directly. Use care in modifying these flags since conflicts in tick spacing, values, and labeling can occur.

Required Flags	**Value Type**	**Comment**
AXIS_LENGTH	*int*	Axis length in screen coordinates.
AXIS_START_POINT	*DV_POINT **	Position in screen coordinates.

Optional Flags	**Value Type**	**Comment**
AXIS_COLOR	*int*	Color index of axis. Default: axis color, if specified; otherwise current foreground color.
AXIS_DIRECTION	*int*	*AXIS_UP, AXIS_DOWN*, *AXIS_LEFT*, *AXIS_RIGHT*. Default: *AXIS_UP*.
AXIS_IS_LOG	*int*	Use logarithmic scaling? (*YES*, *NO*). Default: *NO*.
AXIS_NEW_START_VALUE	*double*	Data value at start of axis. Used to redraw axis with a new start value, typically higher. Use with *HIGHEST_VALUE* (see below). No default.
DRAW_GRID	*int*	Display a grid? (*YES*, *NO*). Default: *NO*.
DRAW_LABELS	*int*	Label the ticks? (*YES*, *NO*). Default: *YES*.
DRAW_TICKS	*int*	Draw any ticks? (*YES*, *NO*). Default: *YES*.
DRAW_MINOR_TICKS	*int*	Draw minor ticks? (*YES*, *NO*). Default: *YES*.
GRID_COLOR	*int*	Color index of grid lines. Default: current foreground color.
GRID_EXCLUDE_ENDS	*int*	Exclude grid lines for first and last ticks? (*YES*, *NO*). Default: *NO*.
GRID_LENGTH	*int*	Length of grid lines in screen coordinates. No default.
GRID_LINE_TYPE	*int*	Line type index of grid lines. Default: solid.
GRID_SIDE	*int*	Grid lines on *LEFT_SIDE* or *RIGHT_SIDE* with respect to axis direction from the start point. Default: opposite of *LABEL_SIDE*, if any; otherwise opposite of *TICK_SIDE*.
HIGHEST_VALUE	*double*	Highest label value for using *AXIS_NEW_START_VALUE* to redraw repeatedly. Not effective if using *LABEL_FUNCTION* (see below). No default.
INTEGER_AXIS	*int*	Make axis values integers; base exponent is ≥ 0. (*YES*, *NO*). Default: *NO*.
LABEL_SIDE	*int*	Tick labels on *LEFT_SIDE* or *RIGHT_SIDE* of axis line. Defaults to values for *TICK_SIDE* (see below).
TICK_LENGTH	*int*	Length in screen coordinates of a major tick mark. Default: equal to one character width.

TICK_SIDE	*int*	Ticks on *LEFT_SIDE* or *RIGHT_SIDE* of axis. Default: left for axis up; right for axis right.

Advanced Optional Flags	**Value Type**	**Comment**
LABEL_DISTANCE	*int*	Distance in screen coordinates of tick labels from axis.
LABEL_TEXTSIZE	*int*	Character size index of tick labels (1 to 4).
LABEL_FORMAT_FUNCTION	ADDRESS,	Tick labeling function,
	ADDRESS,	argument block,
	int	argument size. (See *VPdgticlabfcn* and *VPvdticlabfcn.)*
MIN_MAJOR_PIXEL_GAP	*double*	Minimum screen distance between major ticks.
MIN_MAJOR_VALUE_GAP	*double*	Minimum value difference between major ticks. Do not use with *MIN_MAJOR_PIXEL_GAP*.
MIN_MINOR_PIXEL_GAP	*double*	Minimum screen distance between minor ticks.
MIN_MINOR_VALUE_GAP	*double*	Minimum value difference between minor ticks. Do not use with *MIN_MINOR_PIXEL_GAP*.

VUaxSetupForDrawing

VUaxis Functions VU Routines

Prepares the axis for drawing.

```
BOOLPARAM
VUaxSetupForDrawing (
       AXISDESC axis)
```

VUaxSetupForDrawing prepares the axis descriptor for drawing by filling undefined fields with defaults, positioning the tick marks, and determining tick values and labels. Normally called when information about the axis descriptor is needed before drawing.

VUcopyright

VUcopyright Functions VU Routines

By default, the DataViews copyright notice is displayed on all newly created screens and remains visible until you draw over the screen. The utilities described in this section let you change this behavior.

VUcopyright Functions

VUcopyright	Displays DataViews copyright notice.
VUoff_copyright	Turns off display of DataViews copyright notice.
VUon_copyright	Turns on display of DataViews copyright notice.

VUcopyright

VUcopyright Functions VU Routines

Displays DataViews copyright notice.

```
void
VUcopyright (void)
```

VUcopyright displays the DataViews copyright notice in the center of the screen. On color systems, the copyright logo should appear with yellow text on a blue background. If the background is red, your software may have been incorrectly validated. If you have questions, call DataViews Customer Support.

This routine is called by *VUopendev_set*, and indirectly by *TscOpenSet*. You can override the DataViews copyright notice with your own version if you don't want DataViews's notice to appear in your application. To do this, write your own *VUcopyright* routine using the same syntax. Your routine can be just:

```
void VUcopyright() {}
```

The DataViews routines then call your function instead of the DV-Tools version.

VUoff_copyright

VUcopyright Functions VU Routines

Turns off display of DataViews copyright notice.

```
void
VUoff_copyright (void)
```

VUoff_copyright sets a flag that tells DataViews not to display the DataViews copyright notice when new windows are opened.

VUon_copyright

VUcopyright Functions VU Routines

Turns on display of DataViews copyright notice.

```
void
VUon_copyright (void)
```

VUon_copyright sets a flag that tells DataViews to display the DataViews copyright notice when new windows are opened.

VUdebug

VUdebug Functions VU Routines

Prints data structure utilities for VP/VG layer. On some systems, these routines can be called directly by the debugger to they are not located in the library; instead, they occur as source modules in the *tooldebug* subdirectory of the *src* directory. In the following descriptions, all references to “print” refer to printing to the standard output.

See Also

VOdebug

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUdebug Functions

VUdbgCcf	Prints the context control flags in a data group.
VUdbgColor	Prints the contents of the *COLOR_SPEC* data structure.
VUdbgCtt	Prints the contents of the color threshold table.
VUdbgDgp	Prints the contents of a data group.
VUdbgVdp	Prints the contents of a variable descriptor.

VUdbgCcf

VUdebug Functions VU Routines

Prints the context control flags in a data group.

```
void
VUdbgCcf (
       DATAGROUP *datagroup)
```

VUdbgCcf prints all the context control flags in a given data group, *datagroup*. See *VPdgcontext* for a description of the flags.

VUdbgColor

VUdebug Functions   VU Routines

Prints the contents of the *COLOR_SPEC* data structure pointed to by *color*.

```
void
VUdbgColor (
       COLOR_SPEC *color)
```

VUdbgCtt

VUdebug Functions VU Routines

Prints the contents of the color threshold table, *ct*, containing *size* elements.

```
void
VUdbgCtt (
       int size,
       COLOR_THRESHOLD *ct)
```

VUdbgDgp

VUdebug Functions VU Routines

Prints the contents of a data group.

```
void
VUdbgDgp (
       DATAGROUP *datagroup)
```

VUdbgVdp

VUdebug Functions VU Routines

Prints the contents of a variable descriptor.

```
void
VUdbgVdp (
       VARDESC vdp)
```

VUdbgVdp prints the contents of a variable descriptor, *vdp*. Prints the variable’s type, name, size, range, and access mode.

VUdevice

VUdevice Functions VU Routines

Graphics device utility routines.

See Also

VPdgdevice, *VGdgdevice*, *GRopen*, *GRclose*, *GRrgbtoindex*, *GRindextorgb*

Example

The following code fragment demonstrates opening a device, finding its physical device number, finding the index in the color lookup table that best approximates white, displaying the corresponding color components, and closing the device.

```
int logdevice, white_index;
int red, green, blue;

logdevice = VUopendevice ("CONSOLE");

printf ("Physical device number = %d\n", VUgetdevnum (logdevice));

/* Get the index in the color lookup table that corresponds to white. */
white_index = VUrgbtoindex (logdevice, 255, 255, 255);

VUindextorgb (logdevice, white_index,
              &red, &green, &blue);

printf ("Components of white: ");
printf ("red = %d, green = %d, blue = %d.\n",
              red, green, blue);

VUclosedevice (logdevice);
```

VUdevice Functions

VUclosedevice	Closes specified display device.
VUctBestColors	Reduces a set of color tables to a single table.
VUctRGBtoIndex	Finds the closest match to a color in a color table.
VUctSort	Sorts the colors in a color table.
VUctTransform	Makes a transformation between two color tables.
VUgetdevindex	Returns logical device number for *VP*/*VU* routine use.
VUgetdevnum	Returns physical device number for *GR* routine use.
VUindextorgb	Sets RGB arguments to color lookup table values.
VUloadclut	Loads color lookup table from file.
VUopendev_clut	Opens device using a color lookup table.
VUopendev_set	Opens the device using the specified color lookup table and attributes.
VUopendevice	Opens specified display device.
VUrgbtoindex	Returns display device index, given RGB format.

VUclosedevice

VUdevice Functions VU Routines

Closes specified display device.

```
void
VUclosedevice (
       int logdevice)
```

VUclosedevice closes the device specified by *logdevice*. *logdevice* contains the logical device number, returned by *VUopendevice*.

VUctBestColors

VUdevice Functions VU Routines

Reduces a set of color tables to a single table.

```
BOOLPARAM
VUctBestColors (
       COLOR_TABLE **color_tables,
       int new_size,
       COLOR_TABLE *new_tablep)
```

VUctBestColors determines a set of colors that best matches all the colors in an array of color tables. *color_tables* is a *NULL*-terminated array of pointers to color tables to be matched. *new_size* specifies the maximum number of colors in the new color set and must be between 1 and 256. *new_tablep* is a pointer to the color table to contain the new set of colors. Returns *DV_SUCCESS* or *DV_FAILURE*.

VUctRGBtoIndex

VUdevice Functions VU Routines

Finds the closest match to a color in a color table.

```
BOOLPARAM
VUctRGBtoIndex (
       COLOR_TABLE *color_tablep,
       int r,
       int g,
       int b,
       int *indexp)
```

VUctRGBtoIndex determines the index of the “closest” match in the specified color table, *color_tablep*, to a color specified using the RGB values, *r*, *g*, and *b*. *indexp* is a pointer to the location to store the index value. Returns *DV_SUCCESS* or *DV_FAILURE*.

VUctSort

VUdevice Functions VU Routines

Sorts the colors in a color table.

```
void
VUctSort (
       COLOR_TABLE *color_tablep)
```

VUctSort reorders the colors in a color table based on hue, lightness, and saturation. *color_tablep* is a pointer to the color table. You can call this routine to sort the color table returned by *VOpmGet*.

VUctTransform

VUdevice Functions VU Routines

Makes a transformation between two color tables.

```
void
VUctTransform (
       COLOR_TABLE *from_colors,
       COLOR_TABLE *to_colors,
       COLOR_XFORM *transform)
```

VUctTransform makes a color transform from the source color table, *from_colors*, to the target color table, *to_colors*. Colors in the source color table are translated to the closest color in the target color table. Fills the empty color transform structure, *transform*, with the mappings of the source color indices to the target color indices.

VUgetdevindex

VUdevice Functions VU Routines

Returns logical device number for VP/VU routine use.

```
int
VUgetdevindex (
       int PhysicalDevice)
```

VUgetdevindex returns the logical device number when given the physical device number. The logical device number is expected by the VP and VU routines.

VUgetdevnum

VUdevice Functions VU Routines

Returns physical device number for GR routine use.

```
int
VUgetdevnum (
       int logdevice)
```

VUgetdevnum, given the logical device number (obtained by a previous call to *VUopendevice*), returns the physical device number expected by the GR select routine.

VUindextorgb

VUdevice Functions VU Routines

Sets RGB arguments to color lookup table values.

```
void
VUindextorgb (
       int logdevice
       int color_index,
       int *red,
       int *green,
       int *blue)
```

VUindextorgb, given a logical device number and a color lookup table index, sets the red, green, and blue arguments to the values in the color lookup table corresponding to the index. RGB format specifies a color using three numbers in the range [0,255], where each number corresponds to the intensity of one of the additive primary colors: red, green, and blue.

VUloadclut

VUdevice Functions VU Routines

Loads color lookup table from file.

```
void
VUloadclut (
       char *filename)
```

VUloadclut loads a color lookup table from a file. If the filename is *NULL*, loads the default table. The file must have the following format:

One line for each entry in the table. These lines should comprise triplets, giving the red, green, and blue components of that entry in the table.

Each component must be in the range [0,255]. If the first component in the line is a negative number, that entry in the table remains unchanged. For example, to change the first and last entries for a device with four planes to black and white, use the following table:

0 0 0
-1
-1
255 255 255

If the table has more entries than the device can handle, the extra ones are ignored. If the table has fewer entries, the ones not specified are not changed. Most devices have no more than 256 colors. Extra characters after the numbers are ignored, so you can add comments.

This routine must be called after *VUopendevice*.

VUopendev_clut

VUdevice Functions VU Routines

Opens device using a color lookup table.

```
int
VUopendev_clut (
       char *name,
       char *clutfile)
```

VUopendev_clut opens the display device and sets the color lookup table to the values defined in the file, *clutfile*. This file contains a list of red, green, and blue triplets, with one line per color index.

VUopendev_set

VUdevice Functions VU Routines

Opens the device using the specified color lookup table and attributes.

```
int
VUopendev_set (
       char *dev_name,
       char *clutfile,
          ULONG flag, <type> value,
          ULONG flag, <type> value,
          ...,
       V_END_OF_LIST)
```

VUopendev_set opens the device, *dev_name*, specifies the color lookup table, *clutfile*, sets device attributes, and returns the number representing that device. The device's attributes are set using a variable length argument list of attribute/value pairs. Each pair of parameters starts with an attribute flag that specifies the device attribute to be set. The second argument sets the value of the attribute. The list must terminate with *V_END_OF_LIST* or *0*.

Examples of attributes are window width and height, window icon, and for externally created windows, the window id. Attributes are specified as integer constant flags. For a list of the flags and their attributes, see the description of *TscOpenSet*. These flags, defined in the header file *dvGR.h*, are also used by *GRget*, *GRopen_set*, *GRset*, *TscOpenSet*, and *VOscOpenClutSet*

In the following example, a window with the dimensions 800x600 pixels is opened on an X11 window system:

```
device = VUopendev_set ("X", NULL, V_WINDOW_WIDTH, 800, V_WINDOW_HEIGHT, 600,
             V_END_OF_LIST);
```

Not all attribute flags work on all DataViews drivers. These attributes are device-dependent and can not be set on all devices.

VUopendevice

VUdevice Functions VU Routines

Opens specified display device.

```
int
VUopendevice (
       char *name)
```

VUopendevice opens a graphic display device for input/output. Returns a logical device number used when referring to the device. *VPdgdevice* expects this logical device number rather than the physical device number obtained using *GRopen*. *name* is a character string containing the name of the device. Note that it does not matter if you reopen a device that is already open, so this routine can be used to find the logical device number associated with an open device.

VUrgbtoindex

VUdevice Functions VU Routines

Returns display device index, given RGB format.

```
int
VUrgbtoindex (
       int logdevice,
       int red,
       int green,
       int blue)
```

VUrgbtoindex, given a logical device number and an RGB color specification, returns the index of the device's color lookup table closest to the specified color. RGB format specifies a color using three numbers in the range [0,255], where each number corresponds to the intensity of one of the additive primary colors: red, green, and blue.

VUexit

VUexit Functions   VU Routines

Closes all open devices and exits cleanly.

VUaxis	**VUexit**	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUexit Functions

VUexit

VUexit Functions   VU Routines

Closes all open devices and exits cleanly.

```
void
VUexit (
       int status)
```

VUexit exits cleanly, closing all open display devices and calling *exit(status)*. This is useful because calling *exit()* on some systems causes an exit but does not necessarily close open display devices.

VUpixrep

VUpixrep Functions VU Routines

Routines to manage pixrep structures (*px*). A pixrep is an abstract representation of pixel-based graphic data. The pixrep format is flexible enough to be a superset of many raster or pixel formats. It lets you handle diverse image formats in a single structure, which can then be used by pixmaps and the GR layer raster modules.

These routines and macros are useful for fast image input/output, image processing, directly accessing pixel data, and reading unsupported formats into the pixrep structure. The pixrep structure can then be used to create pixmaps.

The assumed pixel arrangement is a rectangular array; however, the layout of the pixels in the array and the interpretation of pixel values are flexible. The layout of pixreps is explained in the *General Description* later in this module.

A set of macros is also provided for reading the pixels in a pixrep regardless of its layout and the interpretation of its pixel values.

The layout of the pixel data array is controlled by certain fields in the pixrep structure. This section describes the allowable variations in the layout and the fields that control them.

Pixel values may be either indirect color or direct color. Indirect color pixel values are indices into a color table; direct color pixel values store actual RGB component values. If a pixrep points to a color table, the pixel values must be indirect color. Since a color table has no more than 256 entries, the color depth of the pixrep cannot exceed 8. A pixrep can contain a pointer to a boolean vector (of length 256), *color_used*, indicating which colors in the color table are actually used by the pixrep. Setting this field can speed up some pixrep operations such as converting pixreps to rasters.

If the color table pointer field is *NULL*, the pixel values must be direct color. In this case there is one mask for each color component indicating where the color value is stored in the pixel. Components can be located anywhere in the pixel, but each component must occupy consecutive bits in the pixel. For example, in a typical 24-bit color system, each pixel is 32 bits long. The most significant byte is unused. The next byte contains red intensity, the third byte contains the green, and the last byte contains the blue. A pixrep also stores the location of the right-most “1” bit of each mask to speed up pixel reading.

The pixrep structure contains fields giving the height and width of the data. The pixels are arranged in rows from left to right. The data can be arranged with the bottom row of the picture first in the pixel array (the standard DataViews row order) or with the top row first (the order used by X). Each pixel in the row takes up a certain number of bits. This number can be 1, 2, 4, 8, 16, or 32. If the color depth is less than the number of bits per pixel, the color data is stored in the least significant part of those bits. For example, if the pixrep has a depth of 1 but a byte is used for each pixel, the low-order bit of the byte contains the pixel value. An exception is direct-color pixreps, which store the colors directly in the pixrep as red, green, and blue intensities. The location of each value is determined by the masks.

Rows can be aligned on 8-, 16-, or 32-bit boundaries. If they are aligned on 8-bit boundaries, they are consecutive in memory. If they are aligned on 16-bit boundaries, each row starts on the next even address after the last byte of the previous row. A similar rule applies to 32-bit boundaries.

If pixels are less than 8 bits each, the data is packed into an 8-, 16-, or 32-bit unit, the *pack_unit*. For example, if there are 2 bits per pixel, 4 pixels are stored in each byte. Within the bytes of a unit, the pixel values can be stored in order from most-significant to least-significant bit, or vice versa. If MSB order is used, bits 7-6 contain the left-most pixel of the 5 pixels, 5-4 contain the next, 3-2 contain the third and 1-0 contain the right-most. Unused bits at the end of a row may have any value. If the *pack_unit* is 8, the unit packing order is irrelevant.

If pixels are more than 8 bits each, the byte order in each pixel is the native order.

In the macros, you can declare *pixptr* as *FAST* for more efficient reading and writing. The macros use a pointer to a pixel in the pixscan. *VUpxScanInit* initializes the pixscan pointer. This routine must be called before using the reading and writing macros. The pixscan contains the information necessary for reading a pixrep as a consecutive stream of pixel values. The next pixel in the stream is defined to be the next pixel to the right; however, macros are provided to read and write in other directions as well.

The fields that control the pixrep structure are:

Field Name	**Type**	**Description**
width, *height*	*int*	Width and height of the pixrep in pixels.
depth	*UBYTE*	Number of bits of color information.
bits_per_pixel	*UBYTE*	1, 2, 4, 8, 16, or 32 bits.
row_alignment	*UBYTE*	If *row_alignment* is 8, rows are byte-aligned; if 16, rows are short-aligned; if 32, rows are long-aligned.
origin_at_ll	*DV_BOOL*	*YES* if origin is in lower left. Otherwise, *NO*.
pack_unit	*UBYTE*	If fewer than 8 bits per pixel, packing unit. The packing unit is the 8, 16, or 32 bit unit into which the data is packed.
pack_msf_in_byte	*DV_BOOL*	If fewer than 8 bits per pixel, the order of pixels in the byte.
pack_msf_in_unit	*DV_BOOL*	If fewer than 8 bits per pixel, the order of bytes in the unit.
pixels_length	*LONG*	Length of the pixel array.
pixels	*UBYTE **	The array of pixels.
pclut	*COLOR_TABLE **	If (*pclut* != *NULL*), pixels are indexed into color table.
color_used	*DV_BOOL **	An array of type *DV_BOOL*. Specifies which colors are used by the pixrep. If *color_used[i]* is *TRUE*, the corresponding color in the color table is used in the pixrep. If *FALSE*, the color isn't used. If *color_used* is *NULL*, assumes all colors are used. This field is optional, but can speed up some operations if used.
red_mask	*ULONG*	Information for finding the red component
red_shift	*int*	of the pixel.
grn_mask	*ULONG*	Information for finding the green
grn_shift	*int*	component of the pixel.
blu_mask	*ULONG*	Information for finding the blue
blu_shift	*int*	component of the pixel.

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	**VUpixrep**	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUpixrep Functions

VUpxBytesPerRow	Gets the number of bytes per row of a pixrep.
VUpxCalcMaskInfo	Gets the color shift amount from the color mask.
VUpxChannelMerge	Merges three pixreps; each provides a primary color.
VUpxClip	Clips a pixrep.
VUpxCopy	Makes a copy of a pixrep.
VUpxDefault	Fills in the pixrep with default values.
VUpxFlip	Flips a pixrep.
VUpxFree	Frees storage used by a pixrep.
VUpxGetPixel	Reads a pixel from a pixrep.
VUpxMerge	Merges two pixreps.
VUpxNewColorTable	Copies a pixrep using a different color table.
VUpxResize	Resizes a pixrep.
VUpxRotate	Rotates a pixrep.
VUpxRowCompatible	Determines if rows can be copied from one pixrep to another.
VUpxScanInit	Initializes a pixscan pointer for fast reading and writing.
VUpxSetPixel	Writes a pixel value into a pixrep.
VUpxTransform	Transforms a pixrep from one layout to another.
VUpxValid	Determines if the data at an address is a valid pixrep.

VUpixrep Macros

```
#include "VUpixrep.h"
```

GETBLUPXRP	Gets the blue component from a direct-color pixel value.
GETGRNPXRP	Gets the green component from a direct-color pixel value.
GETREDPXRP	Gets the red component from a direct-color pixel value.
ISPIXSTD	Determines if the pixel value is in standard DataViews format.
PIXPXRP	Creates a pixel value from RGB components.
PIXSCALE	Scales a component to a different range.
PIXSTD	Creates a standard pixel value from RGB components.
PUTBLUPXRP	Puts the blue component into a direct-color pixel value.
PUTGRNPXRP	Puts the green component into a direct-color pixel value.
PUTREDPXRP	Puts the red component into a direct-color pixel value.
PXSCANPOINT	Specifies the next pixel to be read.
PXSCANREAD	Reads the current pixel and advances the pixscan pointer.
PXSCANREADD	Reads in decreasing row and increasing column order.
PXSCANREADL	Reads in increasing column and increasing row order.
PXSCANREADR	Reads in decreasing column and increasing row order.
PXSCANREADU	Reads in increasing row and increasing column order.
PXSCANWRITE	Writes to the current pixel and advances the pixscan pointer.
PXSCANWRITED	Writes in decreasing row and increasing column order.
PXSCANWRITEL	Writes in increasing column and increasing row order.
PXSCANWRITER	Writes in decreasing column and increasing row order.
PXSCANWRITEU	Writes in increasing row and increasing column order.

VUpxBytesPerRow

VUpixrep Functions   VU Routines

Gets the number of bytes per row of a pixrep.

```
int
VUpxBytesPerRow (
       PIXREP *pixrep)
```

VUpxBytesPerRow returns the number of bytes per row of a pixrep. *pixrep* is a pointer to the pixrep.

VUpxCalcMaskInfo

VUpixrep Functions VU Routines

Gets the color shift amount from the color mask.

```
void
VUpxCalcMaskInfo (
       ULONG mask,
       int *shift,
       int *size)
```

VUpxCalcMaskInfo determines how much a color component must be shifted to be in the correct location based on the mask. *mask* is a user-supplied mask for one of the color components. The amount of shift required is saved to *shift*. This routine also determines the number of bits in the component and saves this value to *size*.

VUpxChannelMerge

VUpixrep Functions VU Routines

Merges three pixreps, where each provides a primary color.

```
BOOLPARAM
VUpxChannelMerge (
       PIXREP *dest_pixrep,
       PIXREP *red_pixrep,
       PIXREP *green_pixrep,
       PIXREP *blue_pixrep)
```

VUpxChannelMerge combines the color information from three pixreps into a single target pixrep, *dest_pixrep*. *red_pixrep* is a pointer to the pixrep providing red information, *green_pixrep* provides green information, and *blue_pixrep* provides blue information. This function is useful for combining raw sensor data into a false-color representation. This routine only modifies the pixels in *dest_pixrep*; it does not allocate it or change its layout.

VUpxClip

VUpixrep Functions VU Routines

Clips a pixrep.

```
BOOLPARAM
VUpxClip (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       RECTANGLE *bounds)
```

VUpxClip copies a source pixrep to the target pixrep. *bounds* indicates the portion to copy. This routine reallocates storage for *dest_pixrep*, discarding unused pixels. If the rectangle is 10x20, the size of the copy is 10x20. This routine allocates storage for *dest_pixrep*. If successful, returns *YES*. Otherwise returns *NO*.

VUpxCopy

VUpixrep Functions VU Routines

Makes a copy of a pixrep.

```
void
VUpxCopy (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep)
```

VUpxCopy makes a deep copy of the pixrep structure *source_pixrep* to a new pixrep structure *dest_pixrep*. This routine allocates the storage for *dest_pixrep*.

VUpxDefault

VUpixrep Functions VU Routines

Fills in the pixrep with default values.

```
void
VUpxDefault (
       PIXREP *pixrep,
       int h,
       int w,
       COLOR_TABLE *color_table,
       ULONG red_mask,
       ULONG green_mask,
       ULONG blue_mask)
```

VUpxDefault initializes a pixrep, *pixrep*, with reasonable default values. The parameters *h* and *w* specify the height and width of the new pixrep. *color_table* is a pointer to the color table for the new pixrep. If *color_table* is *NULL*, use *red_mask*, *green_mask*, and *blue_mask* to specify the color.

This routine does all initialization except allocation for the pixel storage. This routine sets the *pixels_length* field. If you change a field that could affect the row length, such as *bits_per_pixel* or *row_alignment*, you must recalculate the pixel length using the formula:

```
pixels_length = height*VUpxBytesPerRow (pixrep)
```

VUpxFlip

VUpixrep Functions   VU Routines

Flips a pixrep.

```
BOOLPARAM
VUpxFlip (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       V_PX_FLIP_ENUM axis)
```

VUpxFlip copies the source pixrep, *source_pixrep*, to the target pixrep*, dest_pixrep,* flipping the pixrep around the horizontal or vertical axis. If *axis* is *V_PX_HORIZONTAL*, flips the pixrep along the horizontal axis; if axis is *V_PX_VERTICAL*, flips the pixrep along the vertical axis. The flipped pixrep is saved to the target pixrep *dest_pixrep*. This routine allocates storage for *dest_pixrep*. Returns *YES* if successful. Otherwise returns *NO*.

VUpxFree

VUpixrep Functions VU Routines

Frees storage used by a pixrep.

```
void
VUpxFree (
       PIXREP *pixrep)
```

VUpxFree frees the storage allocated for a pixrep. *pixrep* is a pointer to the pixrep.

VUpxGetPixel

VUpixrep Functions VU Routines

Reads a pixel from a pixrep.

```
ULONG
VUpxGetPixel (
        PIXREP *pixrep,
        int x,
        int y)
```

VUpxGetPixel returns the value of a pixel in the pixrep. *x* and *y* specify the coordinates of the pixel to read.

VUpxMerge

VUpixrep Functions VU Routines

Merges two pixreps.

```
BOOLPARAM
VUpxMerge (
       PIXREP *source_pixrep,
       RECTANGLE *bounds,
       PIXREP *dest_pixrep,
       DV_POINT *ll,
       V_PX_MERGEMODE_ENUM mode,
       PIXREP *mask,
       COLOR_XFORM *mask_transform)
```

VUpxMerge modifies an existing pixrep, *dest_pixrep*, by merging data from the source pixrep, *source_pixrep*, into it. *bounds* is the portion from the source pixrep to merge. *ll* indicates where to place the lower left corner of the source portion within the destination pixmap. *mode* indicates the method for merging the source and target. Valid flags for *mode* are:

V_PX_COPY	Replace the *target* pixel with the *source* pixel.
V_PX_AND	Bit-wise AND the target and source pixels.
V_PX_OR	Bit-wise OR the target and source pixels.
V_PX_XOR	Bit-wise XOR the target and source pixels.

The pixreps must either both be direct color or both be indirect color. The *AND*, *OR*, and *XOR* modes combine the color of a source pixel with the color of the corresponding pixel in the target pixrep.

For good results using indirect color, you must set up the color table of the target pixrep specifically for the merge mode. For information on setting up the color table, see the *Plane Masking* technical note. The merged pixrep uses the color table of the target pixrep; if the target and source pixrep have different color tables, the results may not be what you expect.

The pixreps must be using indirect color to use a mask. If *mask* is specified, only the pixels in the target pixrep whose corresponding pixels in *mask* have an index greater than 0 are actually merged with the source pixels. All others are unchanged. *mask_transform* specifies a color transform that changes the interpretation of *mask*. When *mask* is the target or source pixrep, you can only use *mask_transform* to merge certain colors in either the source or target. If *mask_transform* is *NULL*, the mask is used directly.

The mask and target pixreps should have the same dimensions. They should both have indirect color using the same color tables, or both have direct color using the same color masks.

This routine only modifies the pixels in *dest_pixrep*; it does not allocate it or change its layout. Returns *YES* if successful. Otherwise returns *NO*.

VUpxNewColorTable

VUpixrep Functions VU Routines

Copies a pixrep using a different color table.

```
BOOLPARAM
VUpxNewColorTable (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       COLOR_TABLE *color_table,
       BOOLPARAM do_dither)
```

VUpxNewColorTable copies the source pixrep to the target pixrep, *dest_pixrep*, replacing the color table of the source pixrep with a new color table. The *color_table* parameter is a pointer to the new color table. If a color in *source_pixrep* does not have an exact match in the new color table, the closest match is used. If *do_dither* is *TRUE*, a Floyd-Steinberg dither is applied when matching colors. This routine allocates storage for *dest_pixrep*. Returns *YES* if successful. Otherwise returns *NO*.

VUpxResize

VUpixrep Functions VU Routines

Resizes a pixrep.

```
BOOLPARAM
VUpxResize (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       int new_h,
       int new_w)
```

VUpxResize copies and resizes the source pixrep to the target pixrep, *dest_pixrep*. The pixrep is resized to the new height and width, *new_h* and *new_w*. If either *new_h* or *new_w* is negative, the corresponding dimension is not changed. This routine allocates storage for *dest_pixrep*. Returns *YES* if successful. Otherwise returns *NO*.

VUpxRotate

VUpixrep Functions VU Routines

Rotates a pixrep.

```
BOOLPARAM
VUpxRotate (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       int amount)
```

VUpxRotate copies and rotates the source pixrep to the target pixrep, *dest_pixrep*. *amount* specifies the degree of rotation. Rotation is clockwise and rounded down to the nearest multiple of 90 degrees. This routine allocates storage for *dest_pixrep*. Returns *YES* if successful. Otherwise returns *NO*.

VUpxRowCompatible

VUpixrep Functions VU Routines

Determines if rows can be copied from one pixrep to another.

```
BOOLPARAM
VUpxRowCompatible (
       PIXREP *pixrep1,
       PIXREP *pixrep2)
```

VUpxRowCompatible determines whether the formats of pixreps *pixrep1* and *pixrep2* are similar enough for a row of one pixrep to be copied directly into a row of the other pixrep using C routines such as *memcpy()*. If so, returns *YES*. Otherwise, returns *NO*.

VUpxScanInit

VUpixrep Functions VU Routines

Initializes a pixscan pointer for fast reading and writing.

```
void
VUpxScanInit (
       PIXREP *pixrep,
       PIXSCAN *pixscan,
       PIXPTR *pixptr,
       BOOLPARAM origin_at_ll)
```

VUpxScanInit initializes a pixscan structure based on a pixrep, *pixrep*. *pixscan* is a pointer to the pixscan being initialized; *pixptr* is the byte pointer being initialized. The pixscan can then be used by the macros for reading and writing the stream. If *origin_at_ll* is *TRUE*, the pixels in the pixscan are indexed in DataViews order, with the bottom row as row 0. If *origin_at_ll* is *FALSE*, the pixscan is indexed with the top row as row 0. The pixscan is initialized to point to pixel (0,0). The pixscan must be initialized before using the reading and writing macros.

VUpxSetPixel

VUpixrep Functions VU Routines

Writes a pixel value into a pixrep.

```
void
VUpxSetPixel (
       PIXREP *pixrep,
       int x,
       int y,
       ULONG pixval)
```

VUpxSetPixel writes a pixel value into the pixrep, *pixrep*. *pixval* specifies the pixel value to write. *x* and *y* specify the target location in the pixrep.

VUpxTransform

VUpixrep Functions VU Routines

Transforms a pixrep from one layout to another.

```
BOOLPARAM
VUpxTransform (
       PIXREP *dest_pixrep,
       PIXREP *source_pixrep,
       RECTANGLE *bounds,
       COLOR_XFORM *color_transform)
```

VUpxTransform transforms the data from the source pixrep to match the format specified by the target pixrep, *dest_pixrep*. The target pixrep must be properly initialized and pixel data allocated. This routine only modifies the pixels in *dest_pixrep*; it does not allocate it or change its layout. This routine is used primarily to create a copy of a pixrep with new row attributes.

If the target pixrep and the source pixrep are different sizes, the source data is resized to fit the target. If the *bounds* rectangle is supplied, only the part of the source pixrep within these boundaries is copied to the target.

The source and target pixreps may have different row attributes, but they should be either both direct or both indirect color. If both the source and target pixreps use indirect color, you can use *color_transform* to indicate how to map colors from one pixrep to the other. The contents of the pixels are otherwise unchanged. The *bits_per_pixel* field of the target should be greater than or equal to that of the source.

VUpxValid

VUpixrep Functions VU Routines

Determines whether the data at an address is a valid pixrep.

```
BOOLPARAM
VUpxValid (
       ADDRESS address)
```

VUpxValid determines whether or not the data at *address* is a valid pixrep. Return *YES* if valid; otherwise returns *NO*.

GETBLUPXRP

VUpixrep Functions VU Routines

Gets the blue component from a direct-color pixel value.

```
ULONG
GETBLUPXRP (
       ULONG pixel,
       PIXREP pixrep)
```

GETBLUPXRP gets the blue component from a direct-color pixel value, *pixel*. The parameter *pixrep* specifies the pixrep containing the pixel. Returns the blue component of the pixel value.

GETGRNPXRP

VUpixrep Functions VU Routines

Gets the green component from a direct-color pixel value.

```
ULONG
GETGRNPXRP (
       ULONG pixel,
       PIXREP pixrep)
```

GETGRNPXRP gets the green component from a direct-color pixel value, *pixel*. The parameter *pixrep* specifies the pixrep containing the pixel. Returns the green component of the pixel value.

GETREDPXRP

VUpixrep Functions VU Routines

Gets the red component from a direct-color pixel value.

```
ULONG
GETREDPXRP (
       ULONG pixel,
       PIXREP pixrep)
```

GETREDPXRP gets the red component from a direct-color pixel value, *pixel*. The parameter *pixrep* specifies the pixrep containing the pixel. Returns the red component of the pixel value.

ISPIXSTD

VUpixrep Functions VU Routines

Determines if the pixel value is in standard DataViews format.

```
BOOLPARAM
ISPIXSTD (
       ULONG pixel)
```

ISPIXSTD determines if the pixel value is in standard DataViews format. Returns *YES* if the pixel is in standard DataViews format. Otherwise, returns *NO*.

PIXPXRP

VUpixrep Functions VU Routines

Creates a pixel value from RGB components.

```
ULONG
PIXPXRP (
       ULONG r,
       ULONG g,
       ULONG b,
       PIXREP pixrep)
```

PIXPXRP creates a pixel value from RGB components, *r*, *g*, and *b*. This macro uses the color mask from the pixrep. Returns the combined pixel value.

PIXSCALE

VUpixrep Functions VU Routines

Scales a component to a different range.

```
ULONG
PIXSCALE (
       ULONG pixel,
       int bs,
       int bt)
```

PIXSCALE scales the color intensity to depth *bt* given the depth *bs*. Returns the new color intensity depth.

PIXSTD

VUpixrep Functions VU Routines

Creates a standard pixel value from RGB components.

```
ULONG
PIXSTD (
       ULONG r,
       ULONG g,
       ULONG b)
```

PIXSTD creates a standard pixel value from RGB components, *r*, *g*, and *b*. Returns the combined pixel value.

PUTBLUPXRP

VUpixrep Functions VU Routines

Puts the blue component into a direct-color pixel value.

```
void
PUTBLUPXRP (
       ULONG pixel,
       ULONG b,
       PIXREP pixrep)
```

PUTBLUPXRP puts the blue component specified by *b* into a direct-color pixel value, *pixel*, in the pixrep.

PUTGRNPXRP

VUpixrep Functions VU Routines

Puts the green component into a direct-color pixel value.

```
void
PUTGRNPXRP (
       ULONG pixel,
       ULONG g,
       PIXREP pixrep)
```

PUTGRNPXRP puts the green component specified by *g* into a direct-color pixel value, *pixel*, in the pixrep.

PUTREDPXRP

VUpixrep Functions VU Routines

Puts the red component into a direct-color pixel value.

```
void
PUTREDPXRP (
       ULONG pixel,
       ULONG r,
       PIXREP pixrep)
```

PUTREDPXRP puts the red component specified by *r* into a direct-color pixel value, *pixel*, in the pixrep.

PXSCANPOINT

VUpixrep Functions VU Routines

Specifies the next pixel to be read.

```
void
PXSCANPOINT (
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr,
       int x,
       int y)
```

PXSCANPOINT sets the pixscan pointer so the next pixel to be read or written is the pixel specified by *x* and *y*.

PXSCANREAD

VUpixrep Functions VU Routines

Reads the current pixel and advances the pixscan pointer.

```
void
PXSCANREAD (
       ULONG dest_pixel,
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr)
```

PXSCANREAD reads the next pixel from the pixscan pointer and puts the value in *dest_pixel*. Advances *pixscan* to the next pixel. The next pixel is the one to the right, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order).

PXSCANREADD

VUpixrep Functions VU Routines

Reads in decreasing row and increasing column order.

```
void
PXSCANREADD (
       ULONG dest_pixel,
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr)
```

PXSCANREADD reads in decreasing row and increasing column order. Reads the next pixel from the pixscan pointer and puts the value in *dest_pixel*. Advances *pixscan* to the next pixel. The next pixel is the next one in the column with a lower number (down if in standard DataViews row order), or if at the end of a column, the first pixel in the next column to the right.

PXSCANREADL

VUpixrep Functions VU Routines

Reads in increasing column and increasing row order.

```
void
PXSCANREADL (
       ULONG dest_pixel,
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr)
```

PXSCANREADL reads in increasing column and increasing row order. Reads the next pixel from the pixscan pointer and puts the value in *dest_pixel*. Advances *pixscan* to the next pixel. The next pixel is the one to the right, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order). This macro is the same as *PXSCANREAD*.

PXSCANREADR

Reads in decreasing column and increasing row order.

```
void
PXSCANREADR (
       ULONG dest_pixel,
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr)
```

PXSCANREADR reads in decreasing column and increasing row order. Reads the next pixel from the pixscan pointer and puts the value in *dest_pixel*. Advances *pixscan* to the next pixel. The next pixel is the one to the left, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order).

PXSCANREADU

VUpixrep Functions VU Routines

Reads in increasing row and increasing column order.

```
void
PXSCANREADU (
       ULONG dest_pixel,
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr)
```

PXSCANREADU reads in increasing row and increasing column order. Reads the next pixel from the pixscan pointer and puts the value in *dest_pixel*. Advances *pixscan* to the next pixel. The next pixel is the next one in the column with a higher number (up if in standard DataViews row order), or if at the end of a column, the first pixel in the next column to the right.

PXSCANWRITE

VUpixrep Functions VU Routines

Writes to the current pixel and advances the pixscan pointer.

```
void
PXSCANWRITE (
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr,
       ULONG source_pixel)
```

PXSCANWRITE writes the pixel value specified by *source_pixel* to the next pixel from the pixscan pointer. Advances *pixscan* to the next pixel. The next pixel is the one to the right, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order). The parameter *pixrep* specifies the pixrep containing the pixel.

PXSCANWRITED

VUpixrep Functions VU Routines

Writes in decreasing row and increasing column order.

```
void
PXSCANWRITED (
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr,
       ULONG source_pixel)
```

PXSCANWRITED writes in decreasing row and increasing column order. Writes the pixel value specified by *source_pixel* to next pixel from the pixscan pointer. Advances *pixscan* to the next pixel. The next pixel is the next one in the column with a lower number (down if in standard DataViews row order), or if at the end of a column, the first pixel in the next column to the right. The parameter *pixrep* specifies the pixrep containing the pixel.

PXSCANWRITEL

VUpixrep Functions VU Routines

Writes in increasing column and increasing row order.

```
void
PXSCANWRITEL (
        PIXREP pixrep,
        PIXSCAN pixscan,
        PIXPTR pixptr,
        ULONG source_pixel)
```

PXSCANWRITEL writes in increasing column and increasing row order. Writes the pixel value specified by *source_pixel* to next pixel from the pixscan pointer. Advances *pixscan* to the next pixel. The next pixel is the one to the right, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order). The parameter *pixrep* specifies the pixrep containing the pixel.

PXSCANWRITER

VUpixrep Functions VU Routines

Writes in decreasing column and increasing row order.

```
void
PXSCANWRITER (
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr,
       ULONG source_pixel)
```

PXSCANWRITER writes in decreasing column and increasing row order. Writes the pixel value specified by *source_pixel* to next pixel from the pixscan pointer. Advances *pixscan* to the next pixel. The next pixel is the one to the left, or if at the end of a row, the first pixel in the next row with a higher number (up if in standard DataViews row order). The parameter *pixrep* specifies the pixrep containing the pixel.

PXSCANWRITEU

VUpixrep Functions VU Routines

Writes in increasing row and increasing column order.

```
void
PXSCANWRITEU (
       PIXREP pixrep,
       PIXSCAN pixscan,
       PIXPTR pixptr,
       ULONG source_pixel)
```

PXSCANWRITEU writes in increasing row and increasing column order. Writes the pixel value specified by *source_pixel* to next pixel from the pixscan pointer. Advances *pixscan* to the next pixel. The next pixel is the next one in the column with a higher number (up if in standard DataViews row order), or if at the end of a column, the first pixel in the next column to the right. The parameter *pixrep* specifies the pixrep containing the pixel.

VUregistry

VUregistry Functions VU Routines

Routines to query the Windows Registry.

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	**VUregistry**	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUregistry Functions

VURegQueryDVHome	Finds and returns a string representing the DataViews Home directory.
VURegQueryVal	Searches the Windows registry for a value.

VURegQueryDvHome

VUregistry Functions VU Routines

Finds and returns a string representing the DataViews Home directory.

```
long
VURegQueryDvHome (
       LPSTR* lpDvHome)
```

This function searches the Windows registry for the DataViews Home directory. If successful, it assigns a buffer containing the directory information to *lpDvHome*.

Note: It is up to the user to free the memory allocated for the buffer.

Returns ERROR_SUCCESS if the DataViews home directory was found or an error value if it failed. The return values are the same as the Win32 function RegQueryValueEx(). See your Microsoft Devloper Studio On-line Documentation for more information about the return values.

VURegQueryVal

VUregistry Functions VU Routines

Searches the Windows registry for a value.

```
long
VURegQueryVal (
       LPCTSTR lpSubkey,
       LPCTSTR lpValueName,
       LPBYTE lpData,
       LPDWORD lpSize,
       LPDWORD lpType)
```

This routine searches the Windows registry for the subkey, *lpSubkey* and returns the value, *lpValueName*, in the buffer, *lpData*. It calls the win32 function *RegQueryValueEx()* opening first HKEY_CURRENT_USER then HKEY_LOCAL_MACHINE during its search. The search stops if the subkey is found in HKEY_CURRENT_USER.

If you set *lpSize* and the returned data is larger than this size, the function returns ERROR_MORE_DATA and changes *lpSize* to the correct size for the returned data. If *lpSize* is null, the data is returned successfully and *lpSize* is set to the size of the data.

lpType is either the address of a DWORD containing the data type returned, or NULL if you do not care about the type information. The data types are the same as those returned for *RegQueryValueEx()*. See the Microsoft Developer Studio On-line Documentation for more information.

If the key is found, the function returns ERROR_SUCCESS, otherwise it returns an error value.

You use this routine the same way you would use *RegQueryValueEx().* If you do not know the size or type of the data that will be returned, set *lpData* to NULL before calling this function. If the key *lpSubkey* is found, *lpSize* and *lpType* are set to the size and type of the key's value. Knowing this information, you allocate an appropriately sized buffer for *lpData,* then call this routine again with *lpData* pointing to that buffer and *lpSize* set to the size returned in the first call. Be sure to make the successive calls to this function quickly to avoid the key changing out from under you.

VUsearchpath

VUsearchpath Functions VU Routines

Utility routines.

VUsearchpath Functions

VUaddSearchPath	Adds a new path to the search path.
VUgetSearchPath	Gets the search path.
VUsetSearchPath	Sets the search path to the specified string.

VUaddSearchPath

VUsearchpath Functions VU Routines

Adds a new path to the search path.

```
BOOLPARAM
VUaddSearchPath (
       char *Path,
       BOOLPARAM Append)
```

VUaddSearchPath adds a new pathname, *Path*, to the search path. If *Append* is *YES*, *Path* is added to the end of the search path, if *NO*, *Path* is added at the beginning. Returns *DV_SUCCESS* or *DV_FAILURE*.

VUgetSearchPath

VUsearchpath Functions VU Routines

Gets the search path.

```
BOOLPARAM
VUgetSearchPath (
       char **SearchPath)
```

VUgetSearchPath gets the search path. *SearchPath* is a pointer to an internal data structure that should not be modified. Returns *DV_SUCCESS* or *DV_FAILURE*.

VUsetSearchPath

VUsearchpath Functions VU Routines

Sets the search path to the specified string.

```
BOOLPARAM
VUsetSearchPath (
       char *SearchPath)
```

VUsetSearchPath sets the search path to the specified string. Returns *DV_SUCCESS* or *DV_FAILURE*.

VUstring

VUstring Functions VU Routines

VUaxis	VUexit	**VUstring**	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUstring Functions

VUstrClone *Creates a copy of a string.*

VUstrClone

VUstring Functions VU Routines

Creates a copy of a string.

```
char *
VUstrClone (
       char *string)
```

VUstrClone allocates space for and copies *string* and returns a pointer to the copy. If there is no input string, returns *NULL*.

VUstrlist

VUstrlist Functions VU Routines

Module for managing lists of string pointers. Two common parameters for these routines are:

sl_row The position of the string pointer within the string list. It must be greater than zero and less than or equal to the length of the string list.

sl_col The position of a character within a string. It must be less than or equal to the length of the string.

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	**VUstrlist**	VUvplist
VUdebug	VUregistry	VUtextarray	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUstrlist Functions

VUslAddString	Adds a string to a string list.
VUslClone	Copies a string list.
VUslConvertToString	Converts a string list to a single string.
VUslCreate	Creates a string list.
VUslCreateFromString	Creates a string list from a string.
VUslCutString	Cuts the end of a string.
VUslDeleteString	Deletes a string from a string list.
VUslDeleteSubstring	Deletes a substring from a string.
VUslDestroy	Destroys a string list.
VUslInsertString	Inserts a string into a string list.
VUslInsertSubstring	Inserts a substring into a string.
VUslJoinStrings	Joins two consecutive strings.
VUslLength	Returns the length of a string list.
VUslList	Returns a pointer to the list of string pointers.
VUslLongest	Returns the length of the longest string in a string list.
VUslPadList	Pads a list with strings to achieve the specified length.
VUslPadString	Pads a string with characters to achieve the specified length.
VUslSort	Sorts the list of strings.
VUslSplitString	Splits a string into two.
VUslTraverse	Applies a user-defined function to every string in a string list.

VUslAddString

VUstrlist Functions VU Routines

Adds a string to the end of a string list.

```
void
VUslAddString (
       ADDRESS StringList,
       char *string)
```

VUslClone

VUstrlist Functions VU Routines

Copies a string list and returns the address of the copy.

```
ADDRESS
VUslClone (
       ADDRESS StringList)
```

VUslConvertToString

VUstrlist Functions VU Routines

Converts a string list to a single string.

```
char *
VUslConvertToString (
       ADDRESS StringList)
```

VUslConvertToString converts *StringList* to a string. Creates a string and fills it with the strings from the string list, using *\n* as the line separator in the output string. If *\n* appears in a string in the string list, it is copied into the output string, so it is the user's responsibility to check for *\n* in the strings of the string list. The space for the string is allocated internally using *S_ALLOC*, so the user is responsible for freeing the output string using *S_FREE*. Returns the filled string.

VUslCreate

VUstrlist Functions VU Routines

Creates a string list.

```
ADDRESS
VUslCreate (
       int InitialSize)
```

VUslCreate creates a string list with the number of slots equal to *InitialSize*. Returns the address of the new string list.

VUslCreateFromString

VUstrlist Functions VU Routines

Creates a string list from a string.

```
ADDRESS
VUslCreateFromString (
       char *string)
```

VUslCreateFromString creates a string list and fills it with lines from the input string, using *\n* in the input string to determine the line separations for the string list. The input string can be empty. Returns the address of the new string list.

VUslCutString

VUstrlist Functions VU Routines

Cuts the end of a string.

```
BOOLPARAM
VUslCutString (
       ADDRESS StringList,
       int sl_row,
       int sl_col)
```

VUslCutString cuts the end of a string, *sl_row*, starting at the *sl_col* position. If *sl_col* is less than zero, deletes all the characters from the string except *EOS*. If *sl_col* and *sl_row* are not valid positions, does not cut the string and returns *DV_FAILURE*. Otherwise returns *DV_SUCCESS*.

VUslDeleteString

VUstrlist Functions VU Routines

Deletes a string at position *sl_row* from the string list.

```
void
VUslDeleteString (
       ADDRESS StringList,
       int sl_row)
```

VUslDeleteSubstring

VUstrlist Functions VU Routines

Deletes a substring from a string.

```
int
VUslDeleteSubstring (
       ADDRESS StringList,
       int sl_row,
       int sl_col,
       int count)
```

VUslDeleteSubstring deletes *count* characters from the string at *sl_row* of *StringList*, starting with the *sl_col* position. If *count* is negative or larger than the number of characters in *sl_row*, deletes everything up to but not including *EOS*. If *sl_col* and *sl_row* are not valid positions in *StringList*, does not delete any characters. This routine never deletes the *EOS* character so it cannot be used to join strings; see *VUslJoinStrings* instead. Returns the number of deleted characters.

VUslDestroy

VUstrlist Functions VU Routines

Destroys a string list.

```
void
VUslDestroy (
       ADDRESS StringList)
```

VUslInsertString

VUstrlist Functions VU Routines

Inserts a string into a string list at the position *sl_row*.

```
void
VUslInsertString (
       ADDRESS StringList,
       int sl_row,
       char *string)
```

VUslInsertSubstring

VUstrlist Functions VU Routines

Inserts a substring into a string.

```
BOOLPARAM
VUslInsertSubstring (
       ADDRESS StringList,
       int sl_row,
       int sl_col,
       char *substr)
```

VUslInsertSubstring inserts a substring, *substr*, at the *sl_col* position in the string located at *sl_row*. If *sl_col* and *sl_row* are not valid positions, does not insert the substring and returns *DV_FAILURE*. Otherwise returns *DV_SUCCESS*.

VUslJoinStrings

VUstrlist Functions VU Routines

Joins two consecutive strings.

```
BOOLPARAM
VUslJoinStrings (
       ADDRESS StringList,
       int sl_row)
```

VUslJoinStrings joins two consecutive strings into one and deletes the second one from the string list. If *sl_row* is not a valid row or is the last string in *StringList*, does not join the strings and returns *DV_FAILURE*. Otherwise returns *DV_SUCCESS*.

VUslLength

VUstrlist Functions VU Routines

Returns the number of filled slots in the string list.

```
int
VUslLength (
       ADDRESS StringList)
```

VUslList

VUstrlist Functions VU Routines

Returns a pointer to the list of string pointers.

```
char **
VUslList (
       ADDRESS StringList)
```

VUslList returns a pointer to the list of string pointers. This pointer is valid until the next call to any of these functions: *VUslAddString*, *VUslInsertString*, *VUslSplitString*, *VUslDeleteString*, *VUslJoinStrings*, or *VUslPadList*.

VUslLongest

VUstrlist Functions VU Routines

Returns the length of the longest string in a string list.

```
int
VUslLongest (
       ADDRESS StringList)
```

VUslPadList

VUstrlist Functions VU Routines

Pads a list with strings to achieve the specified length.

```
void
VUslPadList (
       ADDRESS StringList,
       int length,
       char *string)
```

VUslPadList adds identical strings to the end of the string list to achieve the specified length, *length*. *string* is used as the added string. If *string* is *NULL*, adds empty strings.

VUslPadString

VUstrlist Functions VU Routines

Pads a string with characters to achieve the specified length.

```
void
VUslPadString (
       ADDRESS StringList,
       int length,
       int sl_row,
       int ch)
```

VUslPadString adds identical characters to the end of the string located at *sl_row* to achieve the specified length, *length*. *ch* is used as the added character. If *char* is *NULL*, adds blank spaces.

VUslSort

VUstrlist Functions VU Routines

Sorts the list of strings in *StringList* using *strcmp()* to define the order.

```
void
VUslSort (
       ADDRESS StringList)
```

VUslSplitString

VUstrlist Functions VU Routines

Splits a string into two.

```
BOOLPARAM
VUslSplitString (
       ADDRESS StringList,
       int sl_row,
       int sl_col)
```

VUslSplitString splits a string into two. Splits the string located at *sl_row* at position indicated by *sl_col*, placing the second portion of the split string in a new string directly after *sl_row*. If the split point is not a valid position, does not split the string and returns *DV_FAILURE*. Otherwise returns *DV_SUCCESS*.

VUslTraverse

VUstrlist Functions VU Routines

Applies a user-defined function to every string in a string list.

```
int
VUslTraverse (
       ADDRESS StringList,
       VUSLTRVRSFUNPTR fun,
       ADDRESS args)

    int
    fun (
          char *string,
          int index,
          ADDRESS args)
```

VUslTraverse applies a function to every string in the list. Stops when the function returns a non-*NULL* value. The function is called with the string, its index in the list, and argument block. Returns the integer result of the function, if any. Otherwise returns *0*.

VUtextarray

VUtextarray Functions VU Routines

This module provides low-level functions for manipulating hardware text within a rectangular region of the screen. This rectangular region is a two-dimensional array of text characters. Some applications where these routines would be useful are terminal emulators, spreadsheet programs, and message display.

Handling a large block of text in a text array is memory-intensive. A more efficient way to handle a large block of text is to use string lists in conjunction with a text array. In this case, the text array displays a portion of the text, and the string list stores the entire block of text. To display the text, call *VUtaFillWithStringList*, which fills the text array with text from the string list. To scroll the text, just refill the text array starting with a different point in the string list. Edits to the text are made in the string list using VUsl routines and displayed using *VUtaFillWithStringList*. See also the *VUstrlist* module.

The creation and modification of a text array are separate from drawing operations. Changes made to a text array do not appear on the screen until after a call to *VUtaDraw* to draw the changes or to *VUtaRedraw* to draw the entire array.

The *VUtextarray* module works with screen coordinates and character coordinates. *VUtaCreate* specifies the text array size in either character coordinates, screen coordinates, or both. Note that if the text array size is given in screen coordinates, the size of the region may not be evenly divisible by the character size. Any extra space at the edges of the text array is referred to as **slop**. Text arrays must be less than or equal to 256 characters in width; there is no limitation on height.

Two structures let you manipulate the text array in character coordinates: *TA_POSITION* specifies the position of a character in the text array, and *TA_RECT* specifies a rectangular region of the text array. All positions specified by these structures are zero-based. You can manipulate the structures using macros provided in *VUtextarray.h*.

The text array maintains a cursor showing the current position. The default position for the cursor is outside the text array, so it is not visible unless you move it into the text array using *VUtaSetCursorPos*. You can set the cursor style and color using *VUtaSetCursorStyle*.

Colors for the text array are specified using a 16-element color table containing color indices into the device's color table. Note that if the device color table changes, subsequent writes may appear in different colors.

The colors of text array characters are stored as packed colors. A packed color contains the foreground and background colors packed together. The macros *V_PACK_COLOR* and *V_UNPACK_COLOR* let you combine foreground and background indices into packed color format and retrieve these indices from the packed format.

Two pre-packed text colors are provided:

V_TA_NORMAL foreground == color[1], background == color[0]
V_TA_INVERSE foreground == color[0], background == color[1]

On a color system with the default color table, *V_TA_NORMAL* appears white on black, and *V_TA_INVERSE* appears black on white. However, on a black-and-white systems the color sense is reversed, so *V_TA_NORMAL* sometimes appears black on white and *V_TA_INVERSE* sometimes appears white on black.

Text array clipping is provided by the drawing functions *VUtaDraw* and *VUtaRedraw*. These routines take a *NULL*-terminated list of clipping viewports which you can create using *VUvlCreate*.

If the attributes of the display device change after the text array has been created, the text array is affected. It is the programmer's responsibility to ensure that the current device is set to the device on which the text array was created. If it is set differently, the following effects can result:

If the operation is a draw or redraw, the output appears on the current device instead of the device on which the text array was created.

If the current device has a different set of fonts from the creation device, the text may not appear in the correct size. This can change the size of the entire text array.

If the current device has a different color table from the creation device, the text can appear in unexpected colors.

If the window size changes, you should create a new text array, copy the contents of the old text array into the new one, and destroy the old text array.

A text array displays a tab character as a single space.

Examples

Text array creation: The following code fragment creates a text array with an orientation point at the lower left corner anchored to the screen coordinate (0,0). The size of the text array is given both in character coordinates and screen coordinates. The larger of the two sizes is chosen and any slop is discarded. *ColorMapping* is a sixteen-element array of color indexes. The constant *V_TA_NUM_COLORS* is defined in *VUtextarray.h* to be 16.

```
TEXTARRAY TextArray;
DV_POINT AnchorPoint = { 0, 0 }
DV_POINT ScreenRectSize = { 20, 25 } /* size in screen coordinates */
TA_POSITION CharRectSize = { 2, 2 } /* size in character coordinates */
int TextSize = 2;
ColorMapping[V_TA_NUM_COLORS];

TextArray = VUtaCreate ((ULONG) (V_OP_LL|V_RSLVE_GREATER|V_SLOP_SHRINK),
             &AnchorPoint, &ScreenRectSize, &CharRectSize, TextSize, ColorMapping);
```

Getting the text array's color: The following code fragment shows how to get colors from the text array's mini-color table:

```
int fgcolor, bgcolor;
fgcolor = VUtaGetColor (TextArray, 0);
bgcolor = VUtaGetColor (TextArray, 1);
```

Setting the text array's mini-color table: The following code fragment shows how to change the colors in the text array's mini-color table. The call sets the third element of the text array's color table to the index of the thirty-first element of the device's color table and returns the old value of the third element of the text array's mini-color table.

```
oldcolor = VUtaSetColor (TextArray, 3, 31);
```

Selecting a character with the mouse: The following code fragment translates a screen position obtained through a mouse pick to a character position in the text array. *inside* is set to *YES* if mouse pick was inside the text array. Otherwise *inside* is set to *NO*. The character position of the selected character is returned in *CharPos*.

```
TA_POSITION CharPos;
OBJECT location;
DV_POINT ScreenCoords;
DV_BOOL inside;
location = TloPoll (WAIT_PICK);
ScreenCoords = VOloScpGet (location);
inside = VUtaScreenToChar (TextArray, ScreenCoords, CharPos);
```

Scrolling the text array: The following code fragment scrolls text in a text array. The *V_TRSET* macro sets the upper left and lower right corners of *trect* to (*1, 0)* and (*height-1, width-1)* respectively. Note that because the text array is zero-based, 1 is subtracted from the width and height. The *DownDist* parameter of *-1* means to move *trect* up one row. *VUtaMoveRect* does not erase the old line so we call *VUtaFillRect* to fill the old line with *blank_char* in color, *fgcolor*.

```
Scroll (t, blank_char)
    TEXTARRAY t;
    char blank_char;

{
    int height, width;
    TA_RECT trect;
    height = VUtaGetHeight (t);
    width = VUtaGetWidth (t);
    V_TRSET (&trect, 1, 0, height-1, width-1);
    VUtaMoveRect (t, &trect, -1, 0); /* Move trect up one row */

    V_TRSET (&trect, height-1, 0, height-1, width-1);
    VUtaFillRect (t, &trect, blank_char, fgcolor);
}
```

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	**VUtextarray**	VUwinevent
VUdevice	VUsearchpath	VUticlabel	

VUtextarray Functions

```
#include "VUtextarray.h"
```

VUtaBox	Returns the bounding box of the text array.
VUtaCharToScreen	Converts character coordinates to screen coordinates.
VUtaCopyRect	Copies a rectangle of text from one text array to another.
VUtaCrAreaSort	Sorts the points of a *TA_RECT*.
VUtaCreate	Creates a text array.
VUtaCrSort	Sorts the coordinates of a *TA_RECT*.
VUtaDestroy	Destroys a text array.
VUtaDraw	Draws the changes in a text array.
VUtaFillRect	Fills a rectangular region of a text array with a character.
VUtaFillWithStringList	Fills a text array with strings from a string list.
VUtaGetCharSize	Returns the current character size of text array.
VUtaGetColor	Returns the color associated with a given index.
VUtaGetCursorPos	Gets the position of the cursor.
VUtaGetCursorStyle	Gets the cursor style and color.
VUtaGetHeight	Returns the height of a text array in character coordinates.
VUtaGetMaxWidth	Returns the maximum width of a text array.
VUtaGetString	Gets a text string from a text array.
VUtaGetWidth	Returns the width of a text array in character coordinates.
VUtaMoveRect	Moves and copies a rectangle of text within a text array.
VUtaPutChar	Writes a character one or more times to a text array.
VUtaPutString	Writes a text string to a text array.
VUtaRecolor	Changes the fore/background color of one or more columns.
VUtaRecolorArea	Changes the fore/background color of a region in a text array.
VUtaRedraw	Redraws a text array.
VUtaScreenToChar	Converts screen coordinates to character coordinates.
VUtaSetColor	Sets a color in the color table of a text array.
VUtaSetCursorPos	Sets a new cursor position.
VUtaSetCursorStyle	Sets the style of the cursor.
VUtaSwapColor	Swaps fore/background colors for one or more columns.

VUtextarray Macros

V_PACK_COLOR	Packs fore/background color indices together.
V_TPADD	Adds values to fields of a *TA_POSITION*.
V_TPCOPY	Copies values from one *TA_POSITION* to another.
V_TPSET	Assigns new values to a *TA_POSITION*.
V_TRADD	Adds values to fields of a *TA_RECT*.
V_TRCOPY	Copies values from one *TA_RECT* to another.
V_TRHEIGHT	Returns the height of a *TA_RECT*.
V_TRSET	Assigns new values to a *TA_RECT*.
V_TRWIDTH	Returns the width of a *TA_RECT*.
V_UNPACK_COLOR	Unpacks packed colors into separate color indices.

VUtaBox

VUtextarray Functions VU Routines

Returns the bounding box of the text array.

```
BOOLPARAM
VUtaBox (
       TEXTARRAY TextArray,
       RECTANGLE *ScreenRect)
```

VUtaBox gets the bounding box of *TextArray* and puts it in *ScreenRect*. The bounding box includes any slop. If a text array was created without slop, calling *VUtaBox* is equivalent to calling *VUtaCharToScreen* with *CharRect* set to *NULL*. Returns *DV_FAILURE* if either *TextArray* or *ScreenRect* is *NULL*. Otherwise returns *DV_SUCCESS*.

VUtaCharToScreen

VUtextarray Functions VU Routines

Converts character coordinates to screen coordinates.

```
BOOLPARAM
VUtaCharToScreen (
       TEXTARRAY TextArray,
       TA_RECT *CharRect,
       RECTANGLE *ScreenRect)
```

VUtaCharToScreen converts a rectangular area of *TextArray* to screen coordinates. The rectangular area is specified by *CharRect* and passed back in *ScreenRect*. If *CharRect* is *NULL*, *ScreenRect* contains the entire region of the text array minus any slop. Returns *DV_SUCCESS* if *CharRect* is within the text array. Otherwise returns *DV_FAILURE*.

VUtaCopyRect

VUtextarray Functions VU Routines

Copies a rectangle of text from one text array to another.

```
BOOLPARAM
VUtaCopyRect (
       TEXTARRAY DestTextArray,
       TA_RECT *DestCharRect,
       TEXTARRAY SrcTextArray,
       TA_RECT *SrcCharRect)
```

VUtaCopyRect copies a rectangular region from one text array to another. *SrcCharRect* specifies the region of the source text array, *SrcTextArray*; *DestCharRect* specifies the region of the destination text array, *DestTextArray*. If the source and destination text arrays are the same, this routine is equivalent to *VUtaMoveRect*. If the size of the two rectangles differs, *VUtaCopyRect* begins the copy in the upper left corner of the source text array, and stops when it reaches the edge of the rectangular region of either the source or destination text array. Returns *DV_FAILURE* if both the source and destination text arrays are *NULL*, or if either *SrcCharRect* or *DestCharRect* are entirely outside the bounds of their respective text arrays. Otherwise returns *DV_SUCCESS*.

VUtaCrAreaSort

VUtextarray Functions VU Routines

Sorts the points of a *TA_RECT*.

```
TA_RECT *
VUtaCrAreaSort (
       TA_RECT *CharRect)
```

VUtaCrAreaSort sorts the points of the *TA_RECT* structure to which *CharRect* points, ensuring that the *CharRect->ul* is above *CharRect->lr*. Use this routine to sort a *TA_RECT* for an area of text and use *VUtaCrSort* to sort a *TA_RECT* for a rectangle of text. See *VUtaRecolorArea* for a figure showing the different ways to interpret a *TA_RECT*. Returns the address of the sorted *CharRect*.

VUtaCreate

VUtextarray Functions VU Routines

Creates a text array.

```
TEXTARRAY
VUtaCreate (
       ULONG SpecFlag,
       DV_POINT *AnchorPoint,
       DV_POINT *ScreenRectSize,
       TA_POSITION *CharRectSize,
       int CharSize,
       int *ColorMapping)
```

VUtaCreate creates and returns a text array for the current device. This routine only allocates and initializes the data structure; use *VUtaDraw* or *VUtaRedraw* to draw the text array to the device. It is the programmer's responsibility to free the text array with a call to *VUtaDestroy*.

The size of the text array can be specified in either screen coordinates, *ScreenRectSize*, or character coordinates, *CharRectSize*, or both. If the width of the text array exceeds 256 characters, the text array is not created and *NULL* is returned. The position on the screen is specified in screen coordinates by *AnchorPoint*. *CharSize* is the hardware font size in the range [1,4] for the text array's characters.

ColorMapping is a 16-element array of color indices. If *ColorMapping* is *NULL*, a default color table is used. Once the text array's mini-color table is set up, you can use *VUtaSetColor* to change colors.

SpecFlag is a bit mask flag that sets three characteristics of the text array. It determines where the text array's orientation point is, how to resolve any conflicts between character and screen regions, and what to do with any slop. To construct *SpecFlag*, select one flag from each of the three categories below using a bitwise OR.

The orientation flag specifies which text array orientation point is mapped to the anchor point. If the text array falls partially or completely off the screen, the text array is clipped to the screen boundaries when drawn. Valid orientation flags are:

V_OP_BITS	All the orientation point bits.
V_OP_TOP	Top of rectangle mapped to the anchor point.
V_OP_BOTTOM	Bottom of rectangle mapped to the anchor point.
V_OP_LEFT	Mid-left side of rectangle mapped to the anchor point.
V_OP_RIGHT	Mid-right side of rectangle mapped to the anchor point.
V_OP_LL	V_OP_BOTTOM | V_OP_LEFT
V_OP_LR	V_OP_BOTTOM | V_OP_LEFT
V_OP_UL	V_OP_TOP | V_OP_LEFT
V_OP_UR	V_OP_TOP | V_OP_RIGHT
V_OP_CENTERED	Center of rectangle mapped to the anchor point.

The rect size flag indicates how to resolve conflicts between *ScreenRectSize* and *CharRectSize*. Valid rect size flags are:

V_RSLVE_BITS	All the resolution bits.
V_RSLVE_X_GREATER	Use the greater of two in x direction.
V_RSLVE_Y_GREATER	Use the greater of two in y direction.

V_RSLVE_X_LESSER	Use the lesser of two in x direction.
V_RSLVE_Y_LESSER	Use the lesser of two in y direction.
V_RSLVE_GREATER	Use the greater of the two x directions and the greater of the two y directions.
V_RSLVE_LESSER	Use the lesser of the two x directions and the lesser of the two y directions.

The slop flag determines how to handle any slop in the text array. When slop is present, it is drawn in *color[0]* from the color table. The orientation point determines where the slop is drawn relative to the text. If the orientation point is *V_OP_CENTERED*, the slop is distributed equally on all four sides of the text array. Otherwise, the slop is drawn opposite the orientation point. For example, if the orientation point is *V_OP_LEFT*, the slop is distributed to the right, top, and bottom sides. Valid slop flags are:

V_SLOP_BITS	All the slop bits.
V_SLOP_X_SHRINK	Discard the slop in the *x* direction.
V_SLOP_Y_SHRINK	Discard the slop in the *y* direction.
V_SLOP_X_LEAVE	Leave the slop in the *x* direction.
V_SLOP_Y_LEAVE	Leave the slop in the *y* direction.
V_SLOP_X_EXPAND	Expand the slop in the *x* direction by one character.
V_SLOP_Y_EXPAND	Expand the slop in the *y* direction by one character.
V_SLOP_SHRINK	*V_SLOP_X_SHRINK | V_SLOP_Y_SHRINK*
V_SLOP_LEAVE	*V_SLOP_X_LEAVE | V_SLOP_Y_LEAVE*
V_SLOP_EXPAND	*V_SLOP_X_EXPAND | V_SLOP_Y_EXPAND*

Default values for the text array:

If the anchor point is *NULL*, the upper left corner of the text array is placed in the upper left corner of the screen.

A *SpecFlag* of *(ULONG)0* centers the text array with respect to its anchor point. If the anchor point is non-*NULL*, this flag leaves any slop and resolves any size conflict between screen and character specification of the region towards the smaller size.

If both *ScreenRectSize* and *CharRectSize* are *NULL*, a text array 24 characters high by 80 characters wide is created.

If *CharSize* is *0*, the default hardware font size of *1* is used.

All character cells are filled with spaces of the background color, *color[0]*, and the foreground color, *color[1]*.

The default color table matches the following table as closely as possible:

index	**name**	**red**	**green**	**blue**
0	black	0	0	0
1	white	255	255	255
2	red	255	0	0
3	green	0	255	0
4	yellow	255	255	0
5	dk red	127	0	0
6	dk grn	0	127	0
7	cyan	0	255	255
8	blue	0	0	255
9	magenta	255	0	255
10	gray	127	127	127
11	lt blue	127	127	255

12	purple	12	0	127
13	dk blue	0	0	0
14	khaki	127	127	0
15	lt blue	127	127	255

VUtaCrSort

VUtextarray Functions VU Routines

Sorts the coordinates of a *TA_RECT*.

```
TA_RECT *
VUtaCrSort (
       TA_RECT *CharRect)
```

VUtaCrSort sorts the coordinates of the *TA_RECT* structure to which *CharRect* points, ensuring that the *CharRect->ul* is above and to the left of *CharRect->lr*. Returns the address of the sorted *CharRect*. See also *VUtaCrAreaSort*.

VUtaDestroy

VUtextarray Functions VU Routines

Destroys a text array.

```
BOOLPARAM
VUtaDestroy (
       TEXTARRAY TextArray)
```

VUtaDestroy destroys the given text array. Frees the data structure only. It is the programmer's responsibility to clean up the screen. For example, you can call *VUtaBox* to determine what portion of the screen has been affected, then clean up that portion of the screen with *GRf_rectangle*, *TscRedraw*, or *GRrasdraw*.

VUtaDraw

VUtextarray Functions VU Routines

Draws the changes in a text array.

```
BOOLPARAM
VUtaDraw (
       TEXTARRAY TextArray,
       RECTANGLE **Clipvps)
```

VUtaDraw draws the changed parts of *TextArray* to the screen. *VUtaDraw* clips the text array to any obscuring viewports if you pass a *NULL*-terminated list of clipping viewports in *ClipVpList*. Use *VUvlCreate* in the *VUvpList* module to create the clipping viewport list. If *ClipVpList* is *NULL*, no clipping occurs. All text is marked as drawn after a call to this routine, even if part of the text array is clipped. The first time you call *VUtaDraw* for a particular text array, any slop is drawn in *color[0]*. Returns *DV_FAILURE* if any lower-level graphics calls fail. Otherwise returns *DV_SUCCESS*. See also *VUtaRedraw*.

VUtaFillRect

VUtextarray Functions VU Routines

Fills a rectangular region of a text array with a character.

```
BOOLPARAM
VUtaFillRect (
       TEXTARRAY TextArray,
       TA_RECT *CharRect,
       int chr,
       int PackedColor)
```

VUtaFillRect fills the rectangular region of *TextArray* pointed to by *CharRect* with the character, *chr*, in *Color*. *NULL* is not a valid *chr* value. If *CharRect* is *NULL*, the entire *TextArray* is filled with the specified character. Returns *DV_FAILURE* if *CharRect* is not within the bounds of the text array. Otherwise returns *DV_SUCCESS*.

VUtaFillWithStringList

VUtextarray Functions VU Routines

Fills a text array with strings from a string list.

```
void
VUtaFillWithStringList (
       TEXTARRAY TextArray,
       ADDRESS StringList,
       TA_POSITION *ta_pos,
       int anch_row,
       int anch_col,
       int color)
```

VUtaFillWithStringList fills a text array with strings of a string list. If *ta_pos* is *NULL*, places the strings in *TextArray* starting with *anch_row* and *anch_col*. Fills every line of *TextArray* with the corresponding string of *StringList* until it reaches *EOS* or the right border of *TextArray*. If the string does not fill the row, fills the rest of *the* row with spaces. If the number of strings in *StringList* is less than the height of *TextArray*, fills the rest of the *TextArray* with spaces. In one-line mode (*ta_pos != NULL*) follows the same procedure, but fills only one row, *anch_row*, starting with *anch_col*.

VUtaGetCharSize

VUtextarray Functions VU Routines

Returns the current character size of text array.

```
int
VUtaGetCharSize (
       TEXTARRAY TextArray)
```

VUtaGetCharSize returns the current character size associated with *TextArray*. The character size is device-dependent. See also *GRch_Size*.

VUtaGetColor

VUtextarray Functions VU Routines

Returns the color associated with a given index.

```
int
VUtaGetColor (
       TEXTARRAY TextArray,
       int Index)
```

VUtaGetColor returns the *Index*-th element of the text array's mini-color table, which is a index into the device's color table. Returns *-1* if passed an illegal index.

VUtaGetCursorPos

VUtextarray Functions VU Routines

Gets the position of the cursor.

```
TA_POSITION *
VUtaGetCursorPos (
       TEXTARRAY TextArray,
       TA_POSITION *ta_pos)
```

VUtaGetCursorPos returns the cursor position. *ta_pos* is a *TA_POSITION* structure passed to the routine, which fills it with the current cursor position and returns it. This lets you pass in an old cursor position and reuse it for the new cursor position.

VUtaGetCursorStyle

VUtextarray Functions VU Routines

Gets the cursor style and color.

```
V_UTA_CURSOR_ENUM *
VUtaGetCursorStyle (
       TEXTARRAY TextArray,
       TA_PACKED_COLOR *cursor_color)
```

VUtaGetCursorStyle returns the cursor style as the return value and the colors in *cursor_color*. Cursor styles are *V_UTA_UNDERSCORE*, *V_UTA_REVERSE*, and *V_UTA_COLOR*. The colors apply when the cursor style is *V_UTA_COLOR*.

VUtaGetHeight

VUtextarray Functions VU Routines

Returns the height of a text array in character coordinates.

```
int
VUtaGetHeight (
       TEXTARRAY TextArray)
```

VUtaGetHeight returns the number of characters that fit vertically in the text portion of the text array.

VUtaGetMaxWidth

VUtextarray Functions VU Routines

Returns the maximum width of a text array.

```
int
VUtaGetMaxWidth (void)
```

VUtaGetString

VUtextarray Functions VU Routines

Gets a text string from a text array.

```
char *
VUtaGetString (
       TEXTARRAY TextArray,
       char *Buf,
       TA_PACKED_COLOR *LeadingCharColor,
       TA_POSITION *CharPos,
       int MaxCols)
```

VUtaGetString returns the string, not longer than *MaxCols*, from *TextArray* starting at *CharPos*. Also places the string in *Buf* and the packed color of the leading character in *LeadCharColor*. If *MaxCols* is negative, *VUtaGetString* returns the string from *CharPos* to the right edge of the text array. If *MaxCols* is non-negative, the length of the *Buf* must be at least *MaxCols+1* to allow for the string terminator. If *MaxCols* is negative, the buffer must be large enough for a string that extends from *CharPos* to the right edge of the text array. *VUtaGetWidth* routine helps calculate the buffer size.

VUtaGetWidth

VUtextarray Functions VU Routines

Returns the width of a text array in character coordinates.

```
int
VUtaGetWidth (
       TEXTARRAY TextArray)
```

VUtaGetWidth returns the number of characters that fit horizontally in the text portion of the *TextArray*.

VUtaMoveRect

VUtextarray Functions VU Routines

Moves and copies a rectangle of text within a text array.

```
BOOLPARAM
VUtaMoveRect (
       TEXTARRAY TextArray,
       TA_RECT *CharRect,
       int DownDist,
       int RightDist)
```

VUtaMoveRect moves and copies a rectangular region of text within a text array. *CharRect* specifies the region; *DownDist* and *RightDist* specify the number of character spaces to move the region. Negative *DownDist* and *RightDist* values indicate movement up and to the left respectively. The application must explicitly erase any characters left on the screen. For an illustration of scrolling the text array, see the examples section.

VUtaPutChar

VUtextarray Functions VU Routines

Writes a character one or more times to a text array.

```
BOOLPARAM
VUtaPutChar (
       TEXTARRAY TextArray,
       int chr,
       int PackedColor,
       TA_POSITION *CharPos,
       int MaxCols)
```

VUtaPutChar writes a single character, *chr*, repeatedly to the text array. Writing begins at the specified character position, *CharPos*, and stops after writing *MaxCols* number of columns or when it reaches the right edge of the *TextArray*, whichever happens first. If *MaxCols* is negative, writing continues to the right edge.

VUtaPutString

VUtextarray Functions VU Routines

Writes a text string to a text array.

```
BOOLPARAM
VUtaPutString (
       TEXTARRAY TextArray,
       char *Str,
       int PackedColor,
       TA_POSITION *CharPos,
       int MaxCols)
```

VUtaPutString puts a string, *Str*, of the packed color, *PackedColor*, into the text array. If the column plus the length of *Str* is greater than the *width* of the text array, writing stops at right edge.

VUtaRecolor

VUtextarray Functions VU Routines

Changes the fore/background color of one or more columns.

```
BOOLPARAM
VUtaRecolor (
       TEXTARRAY TextArray,
       int PackedColor,
       TA_POSITION *CharPos,
       int MaxCols)
```

VUtaRecolor changes the foreground and/or background color of one or more columns in a text array row. Starting at *CharPos* in *TextArray*, *VUtaRecolor* changes the color of the number of columns specified in *MaxCols* to the packed colors, *PackedColor*. If *MaxCols* is negative, the change starts at *CharPos* and continues to the right edge of the text array, or to the end of the string.

VUtaRecolorArea

VUtextarray Functions VU Routines

Changes the fore/background color of a region in a text array.

```
BOOLPARAM
VUtaRecolorArea (
       TEXTARRAY TextArray,
       int PackedColor,
       TA_RECT *Region,
       V_UTA_AREA_ENUM Mode)
```

VUtaRecolorArea changes the foreground and/or background color of region, specified by *Region*, of a text array. *PackedColor* specifies the foreground and background colors. *Mode* indicates whether the recolored region is a rectangle or an area. If either or both points in *Region* are outside the defined text array, *VUtaRecolorArea* reinterprets them as points at the edge of the text array. The reinterpreted position depends on *MODE.* Valid *Mode* values are:

VUtaRedraw

VUtextarray Functions VU Routines

Redraws a text array.

```
BOOLPARAM
VUtaRedraw (
       TEXTARRAY TextArray,
       RECTANGLE **Clipvps)
```

VUtaRedraw redraws entire text array to the screen. *VUtaRedraw* clips the text array to any obscuring viewports if you pass a *NULL*-terminated list of clipping viewports in *ClipVpList*. Use *VUvlCreate* in the *VUvplist* module to create the clipping viewport list. If *ClipVpList* is *NULL*, no clipping occurs. Any slop is redrawn in *color[0]*. For a discussion of slop, see *VUtaCreate*. All text is marked as drawn after a call to this routine, even if part of the text array is clipped. Returns *DV_FAILURE* if it is unable to draw the text array. See also *VUtaDraw*.

VUtaScreenToChar

VUtextarray Functions VU Routines

Converts screen coordinates to character coordinates.

```
BOOLPARAM
VUtaScreenToChar (
       TEXTARRAY TextArray,
       DV_POINT *ScreenCoords,
       TA_POSITION *CharPos)
```

VUtaScreenToChar determines what character position is associated with a given screen position, *ScreenCoords*, and passes it back in *CharPos*. Returns *YES* if position is within the text array. Otherwise returns *NO*.

VUtaSetColor

VUtextarray Functions VU Routines

Sets a color in the color table of a text array.

```
int
VUtaSetColor (
       TEXTARRAY TextArray,
       int Index,
       int Color)
```

VUtaSetColor sets the *Index*-th position in the text array's color table to the device's color index, *Color*. To convert an RGB value to the closest corresponding color index for the *Color* parameter, use *GRrgbtoindex*. Affected areas are marked to be redrawn by the next *VUtaDraw*. Returns the old color if successful. Otherwise returns *-1*.

VUtaSetCursorPos

VUtextarray Functions VU Routines

Sets a new cursor position.

```
void
VUtaSetCursorPos (
       TEXTARRAY TextArray,
       TA_POSITION *new_pos)
```

VUtaSetCursorStyle

VUtextarray Functions VU Routines

Sets the style of the cursor.

```
void
VUtaSetCursorStyle (
       TEXTARRAY TextArray,
       V_UTA_CURSOR_ENUM cursor_style,
       int cursor_color)
```

VUtaSetCursorStyle sets the cursor style. Valid cursor styles are *V_UTA_UNDERSCORE*, *V_UTA_REVERSE*, and *V_UTA_COLOR*. If you specify *V_UTA_COLOR*, you must specify packed colors using *cursor_color*. The default cursor style is *V_UTA_UNDERSCORE*.

VUtaSwapColor

VUtextarray Functions VU Routines

Swaps fore/background colors for one or more columns.

```
BOOLPARAM
VUtaSwapColor (
       TEXTARRAY TextArray,
       TA_POSITION *CharPos,
       int MaxCols)
```

VUtaSwapColor swaps the foreground and background colors for one or more cells in the text array. Starts swapping at *CharPos* and continues for *MaxCols* number of columns or until it reaches the edge of the text array. If *MaxCols* is negative, swapping continues until it reaches the right edge of the text array. This routine can be used for highlighting with inverse video and for cursor display. See also *VUtaRecolor*.

V_PACK_COLOR

VUtextarray Functions VU Routines

Packs fore/background color indices together.

```
TA_PACKED_COLOR
V_PACK_COLOR (
       int Foreground,
       int Background)
```

V_PACK_COLOR packs the foreground and background colors into a *TA_PACKED_COLOR*. Returns the packed colors.

V_TPADD

VUtextarray Functions VU Routines

Adds values to fields of a *TA_POSITION*.

```
TA_POSITION *
V_TPADD (
       TA_POSITION *TextPos,
       int Row,
       int Col)
```

V_TPADD takes a pointer, *TextPos*, to a *TA_POSITION* structure, adds the value of *Row* to *TextPos->Row* and the value of *Col* to *TextPos->Col*, and returns *TextPos*.

V_TPCOPY

VUtextarray Functions VU Routines

Copies values from one *TA_POSITION* to another.

```
TA_POSITION *
V_TPCOPY (
       TA_POSITION *DestTextPos,
       TA_POSITION *SourceTextPos)
```

V_TPCOPY copies the value of the *TA_POSITION* structure, *SourceTextPos*, to *DestTextPos* and returns *DestTextPos*.

V_TPSET

VUtextarray Functions VU Routines

Assigns new values to a *TA_POSITION*.

```
TA_POSITION *
V_TPSET (
       TA_POSITION *TextPos,
       int Row,
       int Col)
```

V_TPSET takes a pointer, *TextPos*, to a *TA_POSITION* structure, sets *TextPos->row* to *Row* and *TextPos->col* to *Col*, and returns *TextPos*.

V_TRADD

VUtextarray Functions VU Routines

Adds values to fields of a *TA_RECT*.

```
TA_RECT *
V_TRADD (
       TA_RECT *CharRect,
       int ulRow,
       int ulCol,
       int lrRow,
       int lrCol)
```

V_TRADD takes a pointer, *CharRect*, to a *TA_RECT* structure and adds the upper left and lower right coordinates to *CharRect*. Return *CharRect*.

V_TRCOPY

VUtextarray Functions VU Routines

Copies values from one *TA_RECT* to another.

```
TA_RECT *
V_TRCOPY (
       TA_RECT *DestTextRect,
       TA_RECT *SourceTextRect)
```

V_TRCOPY copies the *SourceTextRect* to the *DestTextRect* and returns *DestTextRect*.

V_TRHEIGHT

VUtextarray Functions VU Routines

Returns the height of a *TA_RECT*.

```
int
V_TRHEIGHT (
       TA_RECT *TextRect)
```

V_TRSET

VUxxx Functions VUer Routines

Assigns new values to a *TA_RECT*.

```
TA_RECT *
V_TRSET (
       TA_RECT *CharRect,
       int ulRow,
       int ulCol,
       int lrRow,
       int lrCol)
```

V_TRSET sets the upper left corner and lower right corner of the *TA_RECT* structure, *CharRect*, to the values *ulRow*, *ulCol*, *lrRow*, *lrCol* and returns *CharRect*.

V_TRWIDTH

Returns the width of the *TA_RECT* structure, *TextRect*.

```
int
V_TRWIDTH (
       TA_RECT *TextRect)
```

V_UNPACK_COLOR

VUtextarray Functions VU Routines

Unpacks packed colors into separate color indices.

```
TA_PACKED_COLOR
V_UNPACK_COLOR (
       TA_PACKED_COLOR Color,
       int Foreground,
       int Background)
```

V_UNPACK_COLOR unpacks a packed color into the parameters *Foreground* and *Background*. Returns the packed color.

VUticlabel

VUticlabel Functions

VU Routines

See Also

VPdgticlabfcn

Example

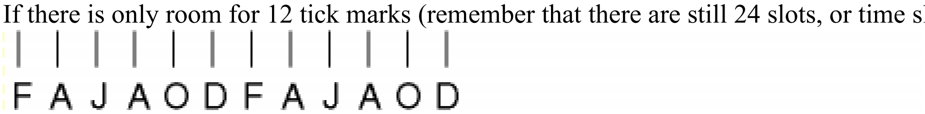
The following code fragment assigns the table *months* as time axis tick labels.

```
static char *months[] = {
   "J", "F", "M", "A", "M", "J",
   "J", "A", "S", "O", "N", "D"
   };

VPdgslots (dgp, 24);
VUdgticlabtab (dgp, V_TIME_AXIS, months, 12);
```

This yields the following labeling of the time axis (if there is room for all 24 tick marks):

If there is only room for 12 tick marks (remember that there are still 24 slots, or time slices), it yields:



VUticlabel Functions

VUdgticlabtab	Axis tick mark labeling routine.

VUdgticlabtab

VUticlabel Functions VU Routines

Axis tick mark labeling routine.

```
void
VUdgticlabtab(
       ADDRESS dgp,
       int axis_type,
       char *(*table)[],
       int size)
```

VUdgticlabtab assigns a table of tick label strings to the time axis or to one of the two spatial axes. These strings are used to label the tick marks on the specified axis. These axes are discrete, meaning that they can only have integral values. The label table should have one entry for each possible tick value. If there are fewer entries in the table than possible values along an axis, the table is treated as a cyclic table. Valid arguments are:

dgp is the address of the data group being assigned the tick labeling table.

axis_type tells which axis is to get these labels. Acceptable values are: *V_FIRST_AXIS* for the first spatial dimension axis, used to indicate the columns of a matrix variable; *V_SECOND_AXIS* for the second spatial dimension axis, used to indicate the rows of a matrix variable; *V_TIME_AXIS* for the time axis.

table is the address of a table of pointers to strings used to label the tick marks.

size is the number of labels in the table.

Diagnostics

The routine creates a tick labeling function that maps tick #1 to the first element in the table. To control how the tick marks are mapped to the table entries, use *VPdgticlabfcn*. To control the labeling of the value axis ticks, use *VPvdticlabfcn*.

VUtraverse

VUtraverse Functions VU Routines

Data group function utilities.

Examples

If *PrintAddressOfThing* is a function that prints an address, the following code fragment demonstrates how to print the addresses of all variable descriptors associated with a data group:

```
/* Where vdp is the first variable descriptor in the data group. */
VUvdtraverse (vdp, PrintAddressOfThing);
```

VUtraverse Functions

VUdgtraverse	Traverses data groups, applies specified function.
VUvdtraverse	Traverses variable descriptors, applies specified function.

VUdgtraverse

VUtraverse Functions VU Routines

Traverses data groups, applies specified function.

```
void
VUdgtraverse (
        ADDRESS dgp,
        VUDGTRVRSFUNPTR function)

    void
    function (
            DATAGROUP dgp)
```

VUdgtraverse traverses linked lists of data groups, performing the function specified by *function* on each of them. The caller specifies the first data group in the linked list, and the address of the function. The addressed function is called with a single argument which is the address of a data group.

VUvdtraverse

VUtraverse Functions VU Routines

Traverses variable descriptors, applies specified function.

```
void
VUvdtraverse (
       ADDRESS vdp,
       VUVDTRVRSFUNPTR function)

    void
    function (
           VARDESC vdp)
```

VUvdtraverse performs the function specified by *function* on every variable descriptor in the linked list specified by *vdp*. The caller specifies the first variable descriptor in the linked list, and the address of the function. The addressed function is called with a single argument which is the address of a variable descriptor in the list.

VUvplist

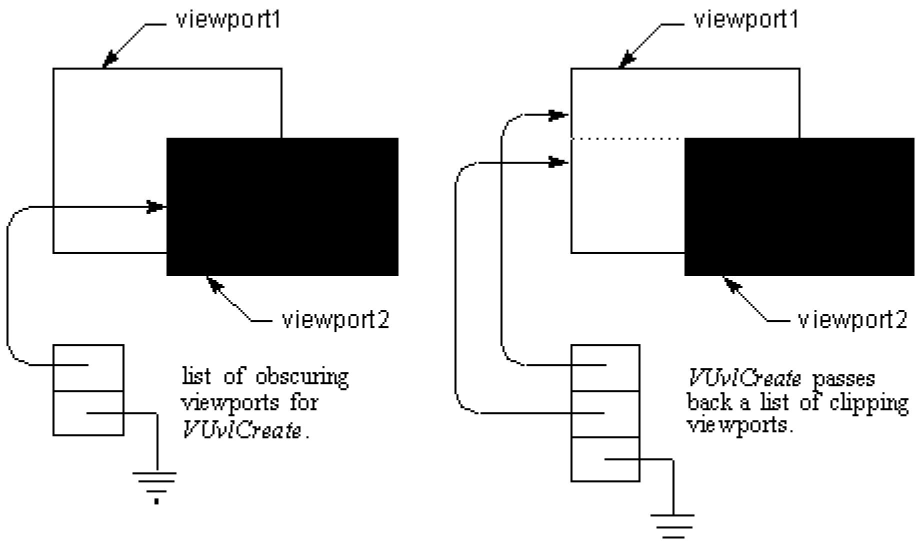
VUvplist Functions VU Routines

Manages viewport lists. This module includes routines for creating a *NULL*-terminated list of viewports from a clipping viewport and a *NULL*-terminated list of obscuring viewports, copying those lists, and destroying those lists.

The list returned by *VUvlCreate* is allocated by this module. After the caller is finished with this list, it should be freed by calling *VUvlDestroy*.

Examples

The following code fragment demonstrates the use of *VUvlCreate* to create a list of clipping viewports given that *viewport2* is obscuring *viewport1* as shown in the following illustration.




```
ADDRESS viewport1, viewport2;
RECTANGLE viewport1,viewport2, *obvplist[2], **clipvplist;
```

These lines construct a *NULL*-terminated list of obscuring viewports:

```
obvplist[0]=&viewport2;
obvplist[1]=(RECTANGLE *)NULL
```

And these lines send the resulting list to *VUvlCreate* to return the list of clipping viewports:

```
clipvplist=VUvlCreate (&viewport1, obvplist);
```

clipvplist must be freed with a call to *VUvlDestroy*.

VUvplist Functions

VUvlCopy	Makes a copy of an existing viewport list.
VUvlCreate	Creates and returns a clipping viewport list.
VUvlDestroy	Destroys a viewport list.

VUvlCopy

VUvplist Functions VU Routines

Makes a copy of an existing viewport list.

```
RECTANGLE **
VUvlCopy (
       RECTANGLE **clipvps)
```

VUvlCreate

VUvplist Functions VU Routines

Creates and returns a clipping viewport list.

```
RECTANGLE **
VUvlCreate (
       RECTANGLE *invp,
       RECTANGLE **outvps)
```

VUvlCreate creates and returns a clipping viewport list given a viewport and an obscuring viewport list. The obscuring viewport list is a *NULL*-terminated list of pointers to viewports that the viewport should be outside of. If *outvps* is *NULL*, then there are no viewports that occlude this one.

VUvlDestroy

VUvplist Functions VU Routines

Destroys a *NULL*-terminated list of pointers to viewports.

```
void
VUvlDestroy (
       RECTANGLE **clipvps)
```

VUwinevent

VUwinevent Functions VU Routines

VUaxis	VUexit	VUstring	VUtraverse
VUcopyright	VUpixrep	VUstrlist	VUvplist
VUdebug	VUregistry	VUtextarray	**VUwinevent**
VUdevice	VUsearchpath	VUticlabel	

VUwinevent Functions

VUweReportEvent Reports window events at a specified level of detail.

VUweReportEvent

VUwinevent Functions VU Routines

Reports window events at a specified level of detail.

```
void
VUweReportEvent (
       WINEVENT *we,
       int level)
```

VUweReportEvent reports window events at a specified level of detail. The *WINEVENT* structure is defined in the header file *dvGR.h*. *we* is the window event pointer. *level* specifies the level of detail to be reported. Valid levels of detail are:

4 Report every field in the window event structure, *we*.
3 Report information relevant to the event type, plus the *eventdata*, count, and *state* fields of the *WINEVENT* structure.
2 Report information relevant to the event type, plus the exposed rectangle list, *rectlist*.
1 Report only information relevant to the event type.
0 Report only the event type.

GR Routines

GR Routines

The *GR* routines are the lowest level of device-independent graphics routines.

The routines expect the screen coordinates, which are device-dependent. If you want a routine to be device-independent, you can use *GRvcs_to_scs* to convert virtual coordinates, in the range 0 <= *x, y* < 32768, into corresponding screen coordinates. In virtual coordinates, the point (0,0) corresponds to the lower left corner of the screen, and (32767,32767) corresponds to the upper right corner. These routines use *DV_POINT* structures to pass the coordinates of a point. Polar coordinates are in a *PLR_POINT* data structure. These types are defined in *dvstd.h*.

GR Modules

All modules in the *GR* layer require the following *#include* files:

```
#include "std.h"
#include "dvstd.h"
#include "dvGR.h"
#include "GRfundecl.h"
```

GRcolor	Manages the color table and device foreground and background colors.
GRcursor	Manages locator cursor and picking.
GRcurve	Routines for calculating and drawing curves.
GRdevice	Device setup and management.
GRdraw	Manages drawing and positioning.
GRinquiry	Routines for getting information about the display device.
GRpalette	Routines for using the color palette.
GRraster	Routines for handling raster operations.
GRrqpcurve	Routines for calculating and drawing rational quadratic parametric (rqp) curves.
GRtext	Routines for drawing hardware text.
GRtransform	Converts between screen and virtual coordinates.
GRvtext	Routines for managing vector text.
GRwinevent	Routines for managing window events.

GRcolor

GRcolor Functions GR Routines

Utilities for setting up and editing the color table, and for selecting colors for drawing.

Each device has a separate color table. All routines that manipulate color tables use the color table associated with the current device. The maximum size of the color table is determined at the time the device is opened and cannot be changed.

Utilities are provided for converting indices in the color table to RGB format and vice versa. RGB format specifies a color using three numbers in the range [0,255], where each number corresponds to the intensity of one of the additive primary colors: red, green, and blue.

Diagnostics

GRrgbtoindex may not return the best approximation of the desired color, since it uses Euclidean distance in RGB space as a measure of the proximity of the index to the specified color. Sometimes a closer match results from measuring the distance in a different space, such as Hue-Saturation-Value (HSV) space.

The color index is device-dependent. If a color must be saved for use on other devices, save its RGB components instead of its index.

GRs_color_table is device-dependent, which means it depends on the number of colors available in the device and whether or not the colors can be modified.

Except when contiguous planes are used under X, all pixels for color tables created by DV-Tools window creation routines are read-only. If the pixel referred to by an index in a color table is read-only and a color change is requested for that index (by *GRs_index_color* or by replacing the entire table with *GRs_color_table*), the color change may not appear in the display until objects that use that color are redrawn. Until objects are redrawn, there may be no color change or an unpredictable change in the display.

Examples

Setting the color table. The following code fragment sets the color table to five colors: black, white, yellow, green, and red. It then draws the color palette that represents this table. Only these five colors are included in the palette.

```
static RECTANGLE palette_vp = {{ 0,0 }, { 600, 450 }};

COLOR_TABLE new_ct =          /* New color table with its values. */
    {
    5,                        /* Number of entries in the table. */
    {                         /*
    {-1,   0,   0,   0},      /* 0 = Black */
    {-1, 255, 255, 255},      /* 1 = White */
    {-1, 255, 255,   0},      /* 2 = Yellow */
    {-1,   0, 255,   0},      /* 3 = Green */
    {-1, 255,   0,   0},      /* 4 = Red */
    }};

/* Make new_ct the new color table. */
GRs_color_table (&new_ct);
```

Examining the color table. The following code fragment inspects the color table of the current device and prints the RGB values of each color.

```
int i;
COLOR_TABLE *ctp;       /* ctp is a pointer to the table. */

GRg_color_table (&ctp);

for (i = 0; i < ctp- >ctsize; i++)
   printf ("#%d: red =%d green =%d blue =%d \n", i, ctp->ct[i].red, ctp-
             >ct[i].green, ctp->ct[i].blue);
```

Setting foreground and background color. The following code fragment writes text on the screen. The text string has a red foreground and yellow background, as defined by *GRcolor* and *GRbackcolor* respectively.

```
int color_index;
DV_POINT p;

/* Get index from RGB values. Set foreground color to red. */
GRrgbtoindex (255, 0, 0, &color_index);
GRcolor (color_index);

/* Get index from RGB values. Set background color to yellow. */
GRrgbtoindex (255, 255, 0, &color_index);
GRbackcolor (color_index);

/* Move to the point (200, 300). */
p.x = 200;
p.y = 300;
GRmove (&p);

/* Draw the text at that point. */
GRtext ("This red text has a yellow background.");
```

GRcolor Functions

GRappend_color	Appends a color to the color table.
GRbackcolor	Selects the background color.
GRcolor	Selects the foreground color.
GRdrop_color	Drops the last color from the color table.
GRg_color_table	Gets the size and contents of the current lookup table.
GRg_pixel	Gets the device-dependent color value.
GRg_real_color_tab	Gets the actual color table that the device is using.
GRindextorgb	Converts a color table index to an RGB value.
GRpixeltorgb	Gets the RGB values corresponding to a device-dependent color value.
GRrgbtoindex	Converts an RGB value to a color table index.
GRs_color_table	Sets up the color lookup table.
GRs_index_color	Sets the *index*-th entry in the color table.
GRs_index_rw	Sets the pixel indicated by the *index*-th entry in the color table to read-write or read-only.

GRappend_color

GRcolor functions GR Routines

Appends a color to the color table.

```
BOOLPARAM
GRappend_color (
       RGB_SPEC *rgb)
```

GRappend_color appends the color specified in *rgb* to the end of the current color table. Fails if the current table is already at its maximum allowable size or if the device doesn't support this operation. Returns *DV_SUCCESS* or *DV_FAILURE*.

Note that DV-Tools always creates color tables with the maximum number of slots. Therefore, *GRappend_color* fails unless preceded by one or more calls to *GRdrop_color*.

GRbackcolor

GRcolor functions GR Routines

Selects the background color.

```
BOOLPARAM
GRbackcolor (
       int color_index)
```

GRbackcolor selects the background color to be used for subsequent drawing operations, using *color_index*, an index in the color table. This color is used for erasing and as the background color for text.

If *color_index* is larger than the largest color table array index, the index is adjusted in a device-dependent way, usually by using index mod *color_table_size*.

Returns *DV_SUCCESS* or *DV_FAILURE.*

GRcolor

GRcolor functions GR Routines

Selects the foreground color.

```
BOOLPARAM
GRcolor (
       int color_index)
```

GRcolor selects the foreground color to be used for subsequent drawing operations, using *color_index*, an index in the color table. The foreground color is used to draw all the graphics primitives.

If *color_index* is larger than the largest color table array index, the index is adjusted in a device-dependent way, usually by using *index* mod *color_table_size*.

Returns *DV_SUCCESS* or *DV_FAILURE*.

GRdrop_color

GRcolor functions GR Routines

Drops the last color from the color table.

```
BOOLPARAM
GRdrop_color (void)
```

GRdrop_color drops the color at the end of the current color table. Fails if driver doesn't support this operation. Returns *DV_SUCCESS* or *DV_FAILURE*.

GRg_color_table

GRcolor functions GR Routines

Gets the size and contents of the current lookup table.

```
BOOLPARAM
GRg_color_table (
       COLOR_TABLE **color_table)
```

GRg_color_table gets the address of the current color lookup table in *color_table*. This includes the size of the table. The argument *color_table* must be the address of a pointer to a structure of type *COLOR_TABLE*. Do not modify the structure whose address is returned because it is used internally by the *GR* routines. To get the actual color table on some devices such as X, you must call *GRg_real_color_tab*. Returns *DV_SUCCESS* or *DV_FAILURE*.

GRg_pixel

GRcolor functions GR Routines

Gets the device-dependent color value.

```
ULONG
GRg_pixel (
       ULONG index)
```

GRg_pixel gets the device-dependent color value that corresponds to the color index, *index*. This color value is useful when you need the device-dependent representation of a color. Returns the device-dependent color in a *ULONG*. In X, this is the pixel value.

GRg_real_color_tab

GRcolor functions GR Routines

Gets the actual color table that the device is using.

```
BOOLPARAM
GRg_real_color_tab (
       COLOR_TABLE **color_table)
```

GRg_real_color_tab gets the address of the color table that the device is actually using. Returns the address in *color_table*. On some devices such as X that reserve or limit colors, this color table may differ from the color table that was set. Returns *DV_SUCCESS* or *DV_FAILURE*.

GRindextorgb

GRcolor functions GR Routines

Converts a color table index to an RGB value.

```
BOOLPARAM
GRindextorgb (
       int color_index,
       int *red,
       int *green,
       int *blue)
```

GRindextorgb converts a color table index, *color_index*, to its equivalent RGB representation, *red*, *green*, *blue*. Returns *DV_SUCCESS* or *DV_FAILURE*.

color_index must contain a value in the range of the color table array. *red*, *green*, and *blue* are set to the red, green, and blue components of the color in the color lookup table.

GRpixeltorgb

GRcolor functions GR Routines

Gets the RGB values corresponding to a device-dependent color value.

```
BOOLPARAM
GRpixeltorgb (
       ULONG pixel,
       UBYTE *red,
       UBYTE *green,
       UBYTE *blue)
```

GRpixeltorgb gets the RGB values corresponding to the device-dependent color value, *pixel*. The RGB values are returned in *red*, *green*, and *blue*, and are in the range [0,255]. They can be used to set colors in DataViews. Device-dependent color values are returned by *GRg_pixel*. Returns *DV_SUCCESS* or *DV_FAILURE*.

GRrgbtoindex

GRcolor functions GR Routines

Converts an RGB value to a color table index.

```
BOOLPARAM
GRrgbtoindex (
       int red,
       int green,
       int blue,
       int *color_index)
```

GRrgbtoindex, given a color in RGB format, *red*, *green*, *blue*, returns the index of the color nearest it in the color table in *color_index*. Returns *DV_SUCCESS* or *DV_FAILURE*.

red, *green*, and *blue* must each contain a value in the range [0,255], with 255 being the most intense.

color_index contains an integer value which represents the number of an array element in the color lookup table. The particular array element represented by this index contains a combination of RGB values which are closest to those values passed to *GRrgbtoindex*.

GRs_color_table

GRcolor functions GR Routines

Sets up the color lookup table.

```
BOOLPARAM
GRs_color_table (
       COLOR_TABLE *color_table)
```

GRs_color_table sets up the color table for the current device. After calling *GRopen* to open the device, call *GRs_color_table* to set up the color table. You can pass a *color_table* setting of *NULL* to initialize the color table to device-dependent default values, or you can set up your own color table structure, as described below, and pass its address. Returns *DV_SUCCESS* or *DV_FAILURE*.

To create a new color table, follow these three steps:

1. Define the color table data structure and declare a variable of that type. The structure is:

   ```
   typedef struct
       {
       int ctsize;            /* size of color table */
       RGB_SPEC ct[256];      /* array of no more than 256 RGB values */
       } COLOR_TABLE;

   COLOR_TABLE new_color_table;
   ```

   Set *ctsize* to the actual number of elements in the new color table, which must be less than or equal to 256.
2. Initialize each *RGB_SPEC* in the table to the desired RGB value for that color. (See *RGB_SPEC* data structure in *dvstd.h*).
3. Call *GRs_color_table* with a pointer to *new_color_table*.

You can call *GRs_color_table* on a device that already has a color table. Doing this changes the color table for the device, and consequently changes the foreground and background colors of the device. To reset the foreground and background colors after calling *GRs_color_table*, call *TscDefForecolor* and *TscDefBackcolor* or *GRcolor* and *GRbackcolor*.

GRs_index_color

GRcolor functions GR Routines

Sets the *index*-th entry in the color table.

```
BOOLPARAM
GRs_index_color (
       int index,
       RGB_SPEC *rgb)
```

GRs_index_color changes the *index*-th color in the table to the given RGB value. Returns *DV_SUCCESS* or *DV_FAILURE*.

In X, this routine works most smoothly if the pixel indicated by the given index is read-write. If the pixel is read-write, it is reset to the new RGB value, and the change appears in the display immediately. If the pixel is read-only, the color change may not appear until objects are redrawn. Until objects are redrawn, they may show an unpredictable color change.

GRs_index_rw

GRcolor functions GR Routines

Sets the pixel indicated by the *index*-th entry in the color table to read-write or read-only.

```
BOOLPARAM
GRs_index_rw (
       int index,
       BOOLPARAM rw)
```

GRs_index_rw makes the pixel indicated by the index-th entry in the color table read-write (*rw* = *TRUE*) or read-only (*rw* = *FALSE*). Returns *DV_SUCCESS* or *DV_FAILURE*.

This routine works only for X drivers whose colormaps support read-write color cells.

GRcursor

GR Functions GR Routines

Manages locator cursor and picking.

Diagnostics

Depending on the device, mouse button presses can have a different priority than key presses, so the “button queue” may be emptied first, regardless of the order in which key presses entered the queues.

Before using *GRcr_poll*, you must open the locator cursor or the keyboard for polling by calling *GRcr_open_poll*. To free the keyboard for normal use, call *GRcr_close_poll*.

GRlocate may not return when certain keys are pressed, depending on the operating system and the device. The <*Spacebar*> always works. For example, some devices use the numeric keypad to move the cursor.

To move the locator cursor on non-mouse systems, it is necessary to close polling with *GRcr_close_poll*, call *GRmove*, and reopen polling with *GRcr_open_poll*.

Examples

Getting position and pick information. The following code fragment prints the cursor position until user presses the <*q*> key.

```
DV_POINT pt;

GRcr_open_poll();
while ('q'!= GRcr_poll (&pt))
   printf ("current coordinates are (%d, %d)\n", pt.x, pt.y);

GRcr_close_poll();
```

Blocking for picks. The following code fragment waits for user to choose a position on the screen:

```
int key;
DV_POINT pt;

key = GRlocate (&pt);
printf ("keycode:%d at (%d, %d)\n", key, pt.x, pt.y);

GRunlocate (key, &pt);            /* To undo the pick. */
```

GRcursor Functions

GRcr_close_poll	Turns off the graphics cursor.
GRcr_define	Sets the graphical representation of the cursor.
GRcr_event	Sets an event flag.
GRcr_open_poll	Turns on the graphics cursor.
GRcr_poll	Polls the cursor.
GRcr_status	Returns the status of the cursor.
GRlocate	Reads the cursor position.
GRunlocate	Pushes the cursor-event stack.

Unless otherwise noted, these routines return *YES* if successful, *NO* if not.

GRcr_close_poll

GRcursor functions GR Routines

Turns off the graphics cursor.

```
BOOLPARAM
GRcr_close_poll (void)
```

GRcr_close_poll turns off the graphics cursor on the selected device and sets the current position (CP) to the last cursor position.

GRcr_define

GRcursor functions GR Routines

Sets the graphical representation of the cursor.

```
BOOLPARAM
GRcr_define (
       ADDRESS pattern)
```

GRcr_define sets the graphics cursor for the current device to the bit pattern pointed to by *pattern*. Currently supported for X platforms, but not MS Windows. See the device-specific notes for more information.

GRcr_event

GRcursor functions GR Routines

Sets an event flag.

```
BOOLPARAM
GRcr_event (
       int new_eventflag,
       int *current_eventflag)
```

GRcr_event sets the polling mode of *GRcr_poll* to *new_eventflag* and returns the old mode in *current_eventflag*. The four possible cases, defined in *dvGR.h*, are:

V_LOC_CHANGE_WAIT	Wait for a change in the state of the locator; either a move, a button, or a key press.
V_LOC_PICK_WAIT	Wait for a button or key press. This is the same as the *GRlocate* event.
V_LOC_NO_WAIT	Return immediately and get the current position. Returns *NULL* if there was no key or button press. This is the default.
V_LOC_PICK_NO_WAIT	Return immediately; but unlike the previous flag, do NOT get the valid current position. Returns *NULL* if there was no key or button press. This saves the overhead of asking the device for its current position.

If *new_eventflag* is *NULL*, the current polling mode value is returned without change.

GRcr_open_poll

GRcursor functions GR Routines

Turns on the graphics cursor.

```
BOOLPARAM
GRcr_open_poll (void)
```

GRcr_open_poll turns on the graphics cursor for the selected device at the current position.

GRcr_poll

GRcursor functions GR Routines

Polls the cursor.

```
int
GRcr_poll (
       DV_POINT *pt)
```

GRcr_poll polls the cursor for input, returns an *int* containing information about key or button presses since the last call to *GRlocate* or *GRcr_poll*, and gets the most recent cursor position *pt*. The macros *GR_BUTTON* and *GR_KEY*, defined in the include file *dvGR.h*, can be used to extract the information returned. *GRcr_poll* returns the first key or button pressed and queues up the remaining calls. Successive calls to *GRcr_poll* return queued keys and buttons. See the *Diagnostics* section at the end of this module.

GRcr_status

GRcursor functions GR Routines

Returns the status of the cursor.

```
BOOLPARAM
GRcr_status (
       DV_BOOL *onoff,
       DV_POINT *pt,
       ADDRESS *raster)
```

GRcr_status gets information about the polled cursor and returns the status of the graphics cursor. *onoff* indicates whether the cursor is open for polling or not. *pt* points to the current cursor position. *raster* is not used currently. To get the device-dependent representation of the cursor, see *GRget*.

GRlocate

GRcursor functions GR Routines

Reads the cursor position.

```
int
GRlocate (
       DV_POINT *p)
```

GRlocate waits for a key press then reads the cursor position in screen coordinates. Returns the ASCII code of the key that was pressed and the location of the cursor in *p*. This lets the user move the cursor with a joystick or mouse before pressing a key. If the device has a mouse, pressing a mouse button returns the number of the button. This routine does not require a preceding call to *GRcr_open_poll*, nor must *GRcr_close_poll* be closed to free the keyboard. See also *Diagnostics*.

GRunlocate

GRcursor functions GR Routines

Pushes the cursor-event stack.

```
BOOLPARAM
GRunlocate (
       int key,
       DV_POINT *location)
```

GRunlocate pushes a screen location and key press onto the cursor-event stack. The next time *GRlocate* or *GRcr_poll* is called, the result is the same as if the user had made the key press. The event stack is checked before the cursor playback. The stack has a fixed size. If the stack is full, the routine returns *DV_FAILURE*. Otherwise returns *DV_SUCCESS*. The event being pushed is a locate event and must have a key press associated with it. If key press is *NULL*, then the routine sets it to button number 1. The key press must be in the correct format. The *GR_SET_KEY* and *GR_SET_BUTTON* macros can be used to convert the key press to the correct format. If the key press comes from a previous call to *GRlocate*, it is already in the correct format.

GRcurve

GRcurve Functions   GR Routines

Routines to calculate and draw curves.

These routines manipulate and draw parametric cubic curves based on cubic polynomials of the form:

```
p(t) = a0 * t3 + a1 * t2 + a2 * t + a3
```

where *p(t), a0, a1, a2*, and *a3* are coordinate pairs. (For a discussion of cubic curves, see any computer graphics textbook.) This module handles the following types of cubic curves:

- **Cubic** polynomials, which are curves represented in the above polynomial representation.
- **Bezier** curves, which use four control points to define cubic curves. Bezier representations of curves are easy for users to manipulate graphically.
- Uniform cubic **B-splines**, which use four or more control points to define series of smoothly connected cubic curves. This type of curve approximates the B-spline control polygon, which is the set of lines that joins the control points of the curve.

GRcolor	GRdraw	GRraster	GRtransform
GRcursor	GRinquiry	GRrqpcurve	GRvtext
GRcurve	GRpalette	GRtext	GRwinevent
GRdevice			

GRcurve Functions

GRbezsplit	Splits a cubic Bezier curve in half.
GRbeztocub	Converts cubic Bezier to coefficients for cubic curve.
GRbspcubics	Gets the cubic curves that are the B-spline.
GRbspdraw	Draws a B-spline.
GRbsptocub	Converts one 4-pt cubic B-spline to coefficients.
GRcubdraw	Draws a cubic curve.
GRcubprecision	Specifies how precisely to draw the cubic curve.
GRcubpts	Gets the points on a cubic curve.
GRcubsize	Gets number of points needed for cubic curve.
GRcubtobez	Converts cubic curve coefficients to cubic Bezier.

GRbezsplit

GRcurve functions GR Routines

Splits a cubic Bezier curve in half.

```
void
GRbezsplit (
       DV_POINT inbez[4],
       DV_POINT outbez0[4],
       DV_POINT outbez1[4])
```

GRbezsplit splits a cubic Bezier curve, defined by the four control points *inbez[4]*, in half, generating two smaller Bezier curves with control points *outbez0[4]* and *outbez1[4]*.

GRbeztocub

GRcurve functions GR Routines

Converts cubic Bezier to coefficients for cubic curve.

```
void
GRbeztocub (
       DV_POINT bez[4],
       DV_POINT a[4])
```

GRbeztocub converts a cubic Bezier curve defined by the four control points *bez[4]* to the cubic polynomial form defined by the coefficients *a[4]*. These coefficients correspond to the polynomial equation shown above. These coefficients are calculated with the following formulae:

```
a[0] =     -   bez[0]   +         3 * bez[1] -3 * bez[2] + bez[3];
a[1] =     3 * bez[0]   -         6 * bez[1] +3 * bez[2];
a[2] =   - 3 * bez[0]   +         3 * bez[1];
a[3] =         bez[0];
```

GRbspcubics

GRcurve functions GR Routines

Gets the cubic curves that are the B-spline.

```
int
GRbspcubics (
        DV_POINT bsp[],
        int numcps,
        int end_conditions,
        DV_POINT a[][4])
```

GRbspcubics converts a uniform cubic B-spline curve defined by the control points *bsp[numcps]* to an array of cubic curves *a[numcps][4]*. The B-spline can have one of the following three *end_conditions*:

OPEN_ENDS	Open, with the curve going through the two end points of the control polygon.
CLOSED_ENDS	Closed, with the curve forming a loop like the snake eating its own tail.
FLOATING_ENDS	Floating, with the end points of the curve not attached to the control polygon.

The B-spline must have at least four control points. *GRbspcubics* returns the number of cubic curves actually created. The number varies depending on the end conditions, but there are never more than *numcps*.

GRbspdraw

GRcurve functions GR Routines

Draws a B-spline.

```
int
GRbspdraw (
       DV_POINT bsp[],
       int numcps,
       int end_conditions,
       int linepattern,
       int linewidth)
```

GRbspdraw draws the B-spline defined by the control points *bsp[numcps]* and *end_conditions*, using a series of vectors with the *linepattern* and *linewidth* attributes. The end conditions are described above. The degree of precision of the vector approximation is controlled by *GRcubprecision*, as described below.

GRbsptocub

GRcurve functions GR Routines

Converts one 4-pt cubic B-spline to coefficients.

```
void
GRbsptocub (
       DV_POINT bsp[4],
       DV_POINT a[4])
```

GRbsptocub converts a 4-point B-spline, *bsp[4]*, to its cubic polynomial representation, *a[4]*. These coefficients are calculated with the following formulae:

```
a[0] = (-bsp[0] + 3 * bsp[1] - 3 * bsp[2] + bsp[3])/6;
a[1] = (3 * bsp[0] - 6 * bsp[1] + 3 * bsp[2])/6;
a[2] = (-3 * bsp[0] + 3 * bsp[2])/6;
a[3] = (bsp[0] + 4 * bsp[1] + bsp[2])/6;
```

GRcubdraw

GRcurve functions GR Routines

Draws a cubic curve.

```
void
GRcubdraw (
       DV_POINT a[4],
       int linepattern,
       int linewidth)
```

GRcubdraw draws a cubic curve, described by the coefficients *a[4]*, using a series of vectors, and drawn with the attributes *linepattern* and *linewidth*. The degree of precision of the vector approximation is controlled by *GRcubprecision*, described below.

GRcubprecision

GRcurve functions GR Routines

Specifies how precisely to draw the cubic curve.

```
int
GRcubprecision (
       int max_deviation)
```

GRcubprecision specifies the precision for use in approximating a cubic curve with straight lines. The precision value is the maximum deviation allowed between the drawn curve and the ideal curve. Therefore, a value of zero for *max_deviation* gives the maximum precision and larger values give less precision. Returns the old precision value. A negative precision value returns the current precision with no change.

GRcubpts

GRcurve functions GR Routines

Gets the points on a cubic curve.

```
int
GRcubpts (
       DV_POINT a[4],
       DV_POINT ptbuf[],
       int bufsize)
```

GRcubpts converts a cubic polynomial curve defined by the coefficients *a[4]* into a vector approximation, *ptbuf[bufsize]*. Returns the number of points added to the points buffer.

GRcubsize

GRcurve functions GR Routines

Gets number of points needed for cubic curve.

```
int
GRcubsize (
       DV_POINT a[4])
```

GRcubsize returns the estimated maximum number of points that would be required to represent a specified cubic curve at a given level of precision. Representing the curve might actually require fewer points. See also *GRcubprecision*.

GRcubtobez

GRcurve functions GR Routines

Converts cubic curve coefficients to cubic Bezier.

```
void
GRcubtobez (
       DV_POINT a[4],
       DV_POINT bez[4])
```

GRcubtobez is the inverse of *GRbeztocub*; it converts the cubic curve defined by the coefficients *a[4]* into the equivalent Bezier representation defined by the control points *bez[4]*. These control points are calculated with the following formulae:

```
bez[0] =                                a[3];
bez[1] =                  a[2]/3     +  a[3];
bez[2] =         a[1]/3  +  2*a[2]/3   +  a[3];
bez[3] = a[0]    +       a[1]          +  a[2]       +       a[3].
```

GRdevice

GRdevice Functions GR Routines

Routines for device setup and management.

Since these routines are device-dependent, not all device drivers support them. They return *DV_SUCCESS* when they are implemented successfully, and *DV_FAILURE* when they cannot be implemented or when passed an invalid flag for the current driver.

See Also

GRcolor, GRinquiry

Examples

Drawing to the device. The following code fragment displays a filled square whose color corresponds to the red, green, and blue values entered by the user.

```
static DV_POINT llp = { 200, 200 }, urp = { 400, 400 };
int red, green, blue;
int color_index;

/* Prompt user for input. */
printf ("Enter red, green and blue values. ^D to quit. \n");
printf ("Press <RETURN> after each. \n");
printf ("Enter a CTL-D to quit. \n");

while (scanf ("%d %d %d", &red, &green, &blue) != EOF)
   {
      GRrgbtoindex (red, green, blue, &color_index); /* index. */
      GRcolor (color_index);       /* Sets foreground color. */
      GRf_rectangle (&llp, &urp);     /* Draws a filled rectangle. */
      GRflush();
   }
GRindextorgb (color_index, &red, &green, &blue);
printf ("The closest color index, %d, \n", color index);
printf ("Corresponds to red=%d, green=%d, blue=%d \n", red, green, blue);
```

Erasing the device. The following code fragment erases the device to an amber background color. Any displays previously left on the device no longer appear.

```
int color_index;

/* Erase screen to an amber background. */
GRrgbtoindex (200, 90, 0, &color_index);     /* specify amber */
GRbackcolor (color_index);
GRerase();     /* erase screen */
```

Planemasking. The following code fragment draws a red circle on one plane and a green square on the other, with a black background, and with squares having priority over circles. *GRmaskplanes* then erases the green square and the whole circle becomes visible, undamaged by the erase. The device is assumed to have only 2 planes.

```
/* The color table has been set up as follows */
/* color #0: black */
/* color #1: red */
/* color #2: green */
/* color #3: green */
DV_POINT p1 = { 100,100 }, p2 = { 200,200 };
LONG oldmask;
```

```
/* Set color to all bits ON.
/* The actual color is the result of ANDing with the mask */
GRcolor (3);

/* Draw the circle */
oldmask = GRmaskplanes ((LONG)1);
GRf_circle (&p1, 100);

/* Draw the square */
GRmaskplanes (2);
GRf_rectangle (&p1, &p2);

/* Erase the square */
GRcolor (0);
GRf_rectangle (&p1, &p2);

/* Restore the mask */
GRmaskplanes (oldmask);
```

Planemasking under X. The following code fragment shows how set up planemasking in a color table and X colormap simultaneously. The color table has 128 entries. The lowest 64 entries are shades of red. The upper 64 entries constitute the overlay plane, and are all a single shade of blue. The colormap has 256 entries, so it can accommodate the colors for other applications.

```
unsigned long pixels[256], planes[256];
COLOR_TABLE clut;
XColor x_colors[256];

XAllocColorCells (display, colormap, False, planes, 1, pixels, 64);

clut.ctsize = 128;

/* 64 shades of red in the lower layer. To save space, setting the other color components isn't shown. */
for (i = 0; i < 64; i++)
    {
    clut.ct[i].red = i;  /* set DV color */

    /* Set the X pixels. */
    x_colors[i].pixel = pixel[i];
    x_colors[i].red = clut.ct[i].red << 8;  /* X uses short */
    x_colors[i].flags = DoRed | DoGreen | DoBlue;
    }

/* Set up the blue overly plane. */
for (i = 64; i < 128; i++)
    {
    clut.ct[i].blue = 255;  /* set DV color */

    /* Set the X pixels. */
    x_colors[i].pixel = planes[0] | pixel[i-64];
    x_colors[i].blue = clut.ct[i].blue << 8;  /* X uses short */
    x_colors[i].flags = DoRed | DoGreen | DoBlue;
    }

/* Set the DV color table. */
GRs_color_table (&clut);

/* Set the X colormap. */
XStoreColors (display, colormap, x_colors, 128);

/* To draw in the lowest layer: */
GRmaskplanes (AllPlanes);  /* AllPlanes is defined by X */
GRcolor (3);  /* or whatever index in [0,63] has color you want */
```

```
/* To draw in the overlay plane (higher layer) */
GRmaskplanes (planes[0]);
GRcolor (64);  /* ANY color in [64,127]: they all come out blue anyway */
```

GRdevice Functions

GRclose	Closes a graphics device.
GRdraw_background	Repairs all or part of the device by drawing with the background color.
GRerase	Erases the device by drawing with the background color.
GRflush	Flushes display buffers.
GRget	Gets information about parameters from a driver.
GRg_viewport	Gets viewport boundaries.
GRmaskplanes	Sets the write mask for the device.
GRopen	Opens a graphics device.
GRopen_set	Opens a device and returns the device number.
GRreset	Resets all internal variables of the driver to the current device attributes.
GRselect	Selects the current device.
GRset	Resets device attributes.
GRviewport	Defines a drawing viewport.

GRclose

GRdevice functions GR Routines

Closes a graphics device.

```
BOOLPARAM
GRclose (
       int dev_num)
```

GRclose closes the graphics device specified by *dev_num*.

GRdraw_background

GRdevice functions GR Routines

Repairs all or part of the device by drawing with the background color.

```
BOOLPARAM
GRdraw_background (
       RECTANGLE *svp)
```

GRdraw_background draws over the portion of the display device specified by *svp* using the background color. This has the effect of erasing the specified region. If *svp* is *NULL*, this routine is equivalent to *GRerase*. If *svp* is not *NULL* and a viewport has been set using *GRviewport*, erases the intersection of *svp* and the viewport. The current position (CP) is not changed by this routine.

GRerase

GRdevice functions GR Routines

Erases the device by drawing with the background color.

```
BOOLPARAM
GRerase (void)
```

GRerase erases by drawing all pixels in the current device in the background color. The device can be erased to any color in the color table. The background color is set by *GRbackcolor*. If a viewport has been set using *GRviewport*, erases only the viewport. The current position (CP) is not changed by this routine.

GRflush

GRdevice functions GR Routines

Flushes display buffers.

```
BOOLPARAM
GRflush (void)
```

GRflush flushes any pending graphics instructions from the internal display buffers of all selected devices.

GRget

GRdevice functions GR Routines

Attribute Flags
MS Windows Specific DataFlags
Window System Data Structures
X11-Specific Data Stuctures
DataViews Pre- Defined Cursors

Gets information about parameters from a driver.

```
BOOLPARAM GRget (
          ULONG flag, <type> value,
          ULONG flag, <type> value,
          ...,
       V_END_OF_LIST)
```

GRget gets information about the parameters, or attributes, of the current device. These attributes are device-dependent and may not be supported on all devices. Attributes include the input file descriptor, window id, cursor, window dimensions. Attributes are specified using zero-terminated parameter lists of attribute-value pairs. Each pair of parameters starts with an attribute flag which specifies the particular attribute of the device being queried. The second argument is the address of a variable in which to return the value of the attribute. The list must terminate with *V_END_OF_LIST* or *0*.

For example, to get the dimensions of a window specified in pixels, you can call:

```
GRget (V_WINDOW_WIDTH, &x, V_WINDOW_HEIGHT, &y, V_END_OF_LIST);
```

Many of the following attribute flags, defined in the include file *dvGR.h*, are also used by *GRopen_set*, *GRset*, *VUopendev_set*, *VOscOpenClutSet*, *VOscOpenSet*, and *TscOpenSet* to set device attributes. Some of the flags are used only by *GRget* to get information about the device attributes; some are used by the open-set functions for the initial setup of the device and cannot be reset using *GRset*.

Attribute Flags	**Description**
V_WINDOW_WIDTH	Width of window in pixels. Takes an *int* argument. (open/set/get)
V_WINDOW_HEIGHT	Height of window in pixels. Takes an *int* argument. (open/set/get)
V_WINDOW_NAME	Title of window for window systems which have a title bar. Takes a *char ** argument. (open/set/get)
V_WINDOW_X	The system-dependent *x* coordinate position of the window's upper left corner. Takes an *int* argument. (open/set/get)
V_WINDOW_Y	The system-dependent *y* coordinate position of the window's upper left corner. Takes an *int* argument. (open/set/get)
	Determining window position involves your window system, window manager, and specific configuration. Therefore, when using *V_WINDOW_X* and *V_WINDOW_Y*, the value you get may not be the value you set. Because of this system dependency, *GRset* should be tested in your specific environment.
V_CLUT_DEPTH	Depth of DataViews color lookup table (i.e. log2 of number of colors). For monochrome systems, or if DataViews is in monochrome mode, this is 1. Takes an *int* argument. (get)
V_RASTER_DEPTH	Depth of the rasters in pixels. This is not always the same as *V_CLUT_DEPTH*. For example, a device with 8 bit planes might be running DataViews with only 128 colors. Takes an *int* argument. (get)
V_DRAW_FUNCTION	Drawing mode. Valid values are *V_COPY* (normal draw) and *V_XOR* (draw by reversing bits, applicable to rubberbanding). Takes a *LONG* argument. (open/set/get)

V_EVENTS_REPORTED	A DataViews event mask containing all event types supported by the current device. See *GRwe_mask* for the event types. Takes a *ULONG* argument. (get)

Window System Data Structures:

V_INPUT_FD	UNIX file descriptor on which events arrive for the current screen. This is useful for UNIX system calls such as "select" which activates the program when an event happens on the window. Takes an *int* argument. (get)
V_WINDOW_ID	Identifier or "handle" for the window maintained by the current screen. Takes a *Window* argument for X11. (open/get)
V_DISPLAY	The id or data structure for maintaining the network connection for window systems with network-based display (currently only X11). Takes a *Display* * argument. (open/get)
V_ICON_NAME	Title of the icon for systems with an icon title bar. Takes a *char* * argument. (open/set/get)
V_MOTION_COLLAPSE	Collapses all successive motion notify events to a single event. Default is *YES*. Takes a *BOOLPARAM* argument. (open/set)
V_EXPOSE_COLLAPSE	Collapses all successive expose events to a single event. Default is *YES*. Takes a *BOOLPARAM* argument. (open/set)

DataViews Pre-Defined Cursors:

If using *WINEVENT* polling routines, DataViews cursors must be switched explicitly.

V_ACTIVE_CURSOR	Sets the DataViews active cursor, the arrow. Doesn't take an argument. (open/set)
V_INITIAL_CURSOR	Sets the DataViews initial cursor, the DV logo. Doesn't take an argument. (open/set)

Queries About Capabilities of the Driver and System:

V_HAS_WINEVENTS	True if device driver supports the window event routines such as *GRwe_mask*, *GRwe_poll*, and *GRwe_state*. Takes a *BOOLPARAM* argument. (get)
V_HAS_PLANE_MASKING	True if device driver supports the plane masking. Takes a *BOOLPARAM* argument. (get)
V_HAS_XOR	True if device driver supports *V_XOR* drawing mode. Takes a *BOOLPARAM* argument. (get)
V_IS_BLACK_AND_WHITE	True if device driver is black-and-white (single bit plane). Takes a *BOOLPARAM* argument. (get)
V_IS_WINDOW_SYSTEM	True if the device driver is operating in a window system. Takes a *BOOLPARAM* argument. (get)
V_NUM_FONTS	The number of fonts available on the system. Takes an *int* argument. (get)

Queries About the System-Specific Masks:

V_XWINDOW_MASK	The X Window mask which results from combining *mask* and *altmask*. Takes a *ULONG* argument. (get)

Microsoft Windows-Specific Data Flags:

These flags are also discussed in the *DataViews Installation and System Administration Manual*.

V_WIN32_WINDOW_HANDLE	Window handle. Takes an *HWND* * argument. (open/get)
V_WIN32_NEWFONT	Specifies the four DataViews hardware fonts. The fonts

increase in size; the smallest is associated with *1*, the largest with *4*. Indices that are not set programmatically use the fonts specified in the *DV.INI* file if there is one. To maintain consistent sizes and styles, set all four fonts. Takes two arguments: an *int* specifying the index and an *HFONT*. (open/set)

V_WIN32_DOUBLE_BUFFER Double-buffering status of the window. Default is *YES*. Takes an *int* argument (*YES* or *NO*). (open/set/get)

V_WIN32_ICON_NAME Identification of the icon. Takes a *char* * argument. (open/set/get)

V_WIN32_XORFLAG Win32 raster-operation code for XOR objects. Default is *R2_XORPEN*. Takes an *int* argument. For a list of valid values, see the Win32 documentation for *SetROP2*. (open/set/get)

V_WIN32_IS_DV_DEVICE Returns a value >= 0 if this window is a DataViews device; else returns -1. Takes two arguments: an *HWND* and an *int* * for the result. (get)

V_WIN32_WINDOWPROC Gets the DataViews internal window procedure. Takes one argument: a variable to hold the function pointer. Declare the variable this way:
*LRESULT (CALLBACK * dv_proc)()*. (get)

V_WIN32_HPALETTE Handle to a logical palette. Lets you pass the Windows equivalent of a color table. The logical palette must have 256 colors or less. Takes an *HPALETTE* argument. (open/set/get)

X11-Specific Data Structures:

Some of these flags are discussed in more detail in the *DataViews and the View Widget in the X Environment Manual*.

V_X_WINDOW_ID Same as *V_WINDOW_ID*. Takes a *Window* argument. (open/get)

V_X_DISPLAY Same as *V_DISPLAY*. Takes a *Display* * argument. (open/get)

V_X_DISPLAY_NAME Character string giving the name of an X11 remote display, for opening an X11 window on a remote server. The string has the form:

UNIX: *hostname:server.screen*
OpenVMS: *hostname::server.screen*

where *hostname* is the network name of the remote machine, *server* is the server number and *screen* is the screen number on which to display the window. These last two numbers are usually zero. Takes a *char* * argument. (open/get)

V_X_APPLIC_CONTEXT The application context for the device. Ignored when widgets are passed. Within an application, all devices use the application context of the first device. Takes an *XtAppContext* argument. (open/get)

V_X_DRAW_WIDGET The widget passed to display DataViews. Can be a form widget or a widget of any other composite widget subclass. Takes a *Widget* argument. (open/get)

V_X_CURSOR X Window system representation of the current cursor. Takes a *Cursor* argument. (open/set/get)

V_X_APPLIC_CLASS The generic application class for this application. The application class of the first device is assigned to all subsequent devices. Takes a *char* * argument. (open/get)

V_X_APPLIC_NAME The specific application name for this device. Controls which set of defaults the window reads from the resource database and X defaults files. Takes a *char ** argument. (open/get)

V_X_SHELL — The shell widget used by the current DataViews device. Takes a *Widget* argument. (get)

V_X_ICON — X Window system representation for the current icon in the X bitmap format. Requires that you set *V_X_ICON_WIDTH* and *V_X_ICON_HEIGHT*. Takes a *char ** argument. (open/set/get)

V_X_ICON_WIDTH — Width of the X icon. Takes an *int* argument. (open/set/get)

V_X_ICON_HEIGHT — Height of the X icon. Takes an *int* argument. (open/set/get)

V_X_ICON_X,
V_X_ICON_Y — Control the *x* and *y* position of the iconified window, though the window manager may override the settings. Each flag takes an *int* argument. (open)

V_X_ICONIC — Controls whether the window is drawn initially in an iconified state. Default is *NO*. Takes a *BOOLPARAM* argument. (open)

V_X_EXPOSURE_BLOCK — Controls whether the open-set routine blocks (waits for) the expose event before returning. Applies only to the initial expose event for internally created windows. If *YES*, the device is ready for drawing when the routine returns. If *NO*, your application should wait for an expose event before drawing on the device. Default is *NO*. Takes a *BOOLPARAM* argument. (open/set/get)

V_X_RESIZE_BLOCK — Controls whether *GRset* blocks (waits for) the resize and expose events before returning after an explicit resize. If *YES*, your application should follow up immediately with calls to *TscReset* and *TscRedraw*. If *NO*, your application should wait for resize and expose events before drawing on the device. Default is *NO*. Takes a *BOOLPARAM* argument. (open/set/get)

V_X_FONTSTRUCT — Specifies the font corresponding to a 1-based index of fonts used for text. The fonts increase in size; the smallest is associated with *1*, the largest with *4*. Indices that are not set programmatically use the fonts specified in resource files, or the *DVfonts* file if there is one. To maintain consistent sizes and styles, set all four indices. Takes two arguments: an *int* argument specifying the index and an *XFontStruct **. For example:

GRset (V_X_FONTSTRUCT, 1,
small_fontstr_ptr ...

(open/set/get)

V_X_DOUBLE_BUFFER — If *YES*, graphics are written to an off-screen pixmap which is copied to the screen whenever *GRflush* is called. Reduces flicker but may slow down drawing speed. Default is *NO*. Takes a *BOOLPARAM* argument. (open/set/get) If you are using double buffering with the OPEN LOOK server, you should also set *V_X_RAS_SYNC* to *YES*. (open/set/get)

V_X_RAS_SYNC — If *YES*, forces an *XSync* call after every raster drawing. Ensures that all raster draws occur when many are done in rapid succession. Default is *NO*. Takes a *BOOLPARAM* argument. (open/set/get)

V_X_POLY_HINT — Specifies the shape of polygons so the X driver can optimize its performance. If all polygons in the application are non-self-intersecting, specify *Nonconvex* to achieve faster drawing. If all polygons are both non-self-intersecting and convex, specify *Convex* for even faster drawing. Default is *Complex*. Takes an *int* argument. (open/set/get)

V_X_IMAGE_STRING	If *YES*, text is drawn on a filled rectangle drawn in the background color. If *NO*, the text is drawn directly on top of the existing graphics. Default is *YES*. Takes a *BOOLPARAM* argument. (open/set/get)
V_X_DASH_STYLE	Specifies how gaps in a dashed line are drawn. Valid values are: *LineOnOffDash* (gaps are not drawn, so the underlying graphics are visible) or *LineDoubleDash* (the gaps are drawn using the current background color). Default is *LineOnOffDash*. Takes an *int* argument. (open/set/get)
V_X_GC	The graphics context used for drawing. Use *XChangeGC* with caution since changes in the *GC* can adversely affect DataViews graphics. The following fields of the *GC* might be overwritten immediately: *plane_mask*, *foreground*, *background*, *line_width*, *line_style*, *clip_x_origin*, *clip_y_origin*, *clip_mask*, *dash_offset*, and *dashes*. Takes a *GC* argument. (get)
V_X_COLORMAP	The X colormap for the device. Lets you supply a shared colormap to avoid color swapping problems. For more information, see the discussion after the flags. Takes a *Colormap* argument. (open/set/get)
V_X_PIXELS	Array of X pixels corresponding to the indices in the color table. Forces use of these pixels, taking precedence over any other method for setting colors. For more information, see the discussion after the flags. Takes two arguments: an *int* argument specifying the number of pixels and an *unsigned long[]*. For example: *GRset (V_X_PIXELS, 128, pixels* ... (open/set/get)
V_X_PLANES	Array of X plane masks corresponding to the color planes of the pixels. You must supply these masks if you are planemasking with pixels supplied using *V_X_PIXELS*. For more information, see the discussion after the flags. Takes two arguments: an *int* argument specifying the number of masks and an *unsigned long[]*. For example: *GRset (V_X_PLANES, 7, masks* ... (open/set/get)

V_X_COLORMAP, *V_X_PIXELS*, and *V_X_PLANES* give more control over the X structures that the X driver uses. In general, you don't have to pass the X colormap, pixels, or plane masks to DataViews. Instead, the X driver makes X calls to allocate the RGB values based on the DataViews color table. If it cannot allocate all the colors, it maps the additional colors in the color table to the closest color in the colormap. The colormap is private if you specify the *:p* or *:nd* device name option; otherwise the default colormap is used.

V_X_COLORMAP lets you supply a shared colormap for the DataViews display device. This lets you avoid the swapping encountered when using private colormaps for different applications running at the same time. Using the *V_X_COLORMAP* flag ensures only the use of the same colormap; it does not ensure that DataViews will use the colors you want within the colormap. When DataViews receives the colormap, it tries to allocate the colors it needs (up to 128 colors) using any free cells remaining in the colormap. If it cannot allocate all the colors it needs, it finds the best match among the existing colors. For the best color match, you should supply a colormap with an adequate number of free color cells. A colormap with few free cells may result in poor color matches for your view. For example, the colormap may not contain any yellow, so a yellow object may be drawn in the nearest green instead.

When you do not want DataViews to allocate new colors, but instead want it to use certain colors already allocated in the colormap, you should use the *V_X_COLORMAP* flag, but should also use the *V_X_PIXELS* flag, which lets you specify the exact X pixels from the colormap. The following code fragment shows how to pass the pixels using *V_X_PIXELS*:

```
unsigned long pixels[128];
```

```
/* User-defined function that determines which pixels to use. */
pixel[0] = AllocatePixelFromColormap (colormap);
...
GRset (V_X_PIXELS, 128, pixels, V_END_OF_LIST);
```

When you use *V_X_PIXELS*, DataViews uses the pixels you supply as though they were in the DataViews color table. For example, any place that it would use *color[1]* from the DataViews color table, it will use *pixel[1]* from the array you supply. Therefore, it is your responsibility to supply pixels that are a good match to the colors in the color table, which in turn should be a good match for the colors requested in your view. You must maintain the correspondence between the RGB values of the pixels and the RGB values in the color table. For the best results, create a color table with exactly the same RGB values as the pixels in the array, and pass this color table when you open the device. If you later change the RGB values of pixels, you must also change the RGB values in the color table.

The correspondence is important DataViews uses both the RGB values of the pixels and RGB values in the color table, but it uses them for different functions. The RGB values of the pixels determine the drawing colors. The RGB values in the color table are used during view loading: the colors in the view are mapped to the closest RGB value in the color table. If correspondence between the pixels and color table is not maintained, views may display wildly incorrect colors instead of closely matched colors.

Note that you normally use these flags when you first open the device so that they will be in effect before you draw any graphics. Anytime you use *V_X_COLORMAP*, *V_X_PIXELS*, and *V_X_PLANES,* you can reset the internal structures they control only by using these flags again. Calls to *GRs_color_table*, or other routines that normally would cause the X driver to modify these X structures, no longer have that effect.

These flags also let you do planemasking with a shared colormap or the default colormap. You can use either of two methods. For the simpler method, use the following call to set up contiguous planes and specify a color map:

```
TscOpenSet ("x:p", "planemask.clut", V_X_COLORMAP, DefaultColormap (display,
             screen), ...)
```

With this method, as with all planemasking in DataViews, it is your responsibility to set up the color table correctly and set the write mask using *TdpMaskPlanes* or *GRmaskplanes*. However, the X driver makes the calls that set up the colormap for planemasking.

If you have set up your own colormap for planemasking, perhaps because another application is also using planemasking, these additional steps are required:

Allocate the colors using *XAllocColorCell*s. This returns the pixels and plane masks required for *TscOpenSet* or *GRset*.

Open the DataViews device with the *:p* option for contiguous planes and pass the colormap, pixels, and planes:

```
unsigned long pix_arg[npixels];
unsigned long plane_arg[npixels];

screen = TscOpenSet ("x:p", "planemask.clut",
         V_X_COLORMAP, (Colormap)cmap_arg,
         V_X_PIXELS, npixels, pix_arg,
         V_X_PLANES, nplanes, plane_arg, ...)
```

GRg_viewport

GRdevice functions GR Routines

Gets viewport boundaries.

```
BOOLPARAM
GRg_viewport (
       DV_POINT *llp,
       DV_POINT *urp)
```

GRg_viewport gets the current viewport boundaries. This subroutine call is not added to the log file.

GRmaskplanes

GRdevice functions GR Routines

Sets the write mask for the device.

```
LONG
GRmaskplanes (
       LONG mask)
```

GRmaskplanes sets the write mask for the device. For example, if the device has eight planes (256 colors), this routine allows selection of any subset of those eight planes for writing. Any graphics primitives (lines, circles, etc.) drawn after a call to *GRmaskplanes* use a bit-wise AND of the current color and *mask* to determine their drawing color.

The allowed ranges for *mask* depend on the number of display planes. *mask* must be in the range [1,n-1] (inclusive), where n is the number of colors supported by the device.

GRmaskplanes is not supported on all devices. The routine also requires some care in setting up the color table, so that when a zero is written in the higher level planes, it doesn't obscure graphics in the lower level planes.

Returns the old mask value. If *mask* is *NULL*, returns the current mask value without changing the mask. If the device doesn't support masked writes, the routine always returns *NULL*.

For examples showing how to set up a color table and draw when planemasking, see the *Examples* section of this module.

GRopen

GRdevice functions GR Routines

Opens a graphics device.

```
BOOLPARAM
GRopen (
       char dev_name[],
       int *dev_num)
```

GRopen opens the graphics device specified by *dev_name* for I/O. *dev_name* is a character string that names the device, and *dev_num* is the user-specified location in which the device number is placed. The device number is used to refer to the device in *GRclose* and *GRselect*. Note that opening a device that is already open has no effect on the device: *GRopen* simply sets the device number. Valid device names for your system are listed in the *READ_ME* file in the DataViews home directory.

GRopen_set

GRdevice functions GR Routines

Opens a device and returns the device number.

```
BOOLPARAM
GRopen_set (
       char *dev_name,
       int *dev_num,
          ULONG flag, <type> value,
          ULONG flag, <type> value,
          ...,
       V_END_OF_LIST)
```

GRopen_set opens a new device, *dev_name*, and sets the device attributes. The routine returns the device number in *dev_num*. The device attributes are set using a variable length argument list of attribute/value pairs. Each pair of parameters starts with an attribute flag which specifies the particular attribute of the device to be set. The second argument sets the value of the attribute. The list must terminate with *V_END_OF_LIST* or *0*.

Examples of attributes that can be set are window width and height, window icon, and for externally created windows, the window id. The attributes are specified as integer constants flags; see the description of *GRget* for the list of the flags and the attributes they set. These flags, defined in the *#include* file *dvGR.h*, are also used by *GRset*, *VUopendev_set*, *TscOpenSet*, *VOscOpenClutSet* and *VOscOpenSet*.

The following code opens a DataViews device with the dimensions 800x600 pixels, with an upper left position of (100, 100) relative to the screen origin, on an X11 Window system:

```
GRopen_set ("X1", &devnum, V_WINDOW_X, 100, V_WINDOW_Y, 100, V_WINDOW_WIDTH, 800,
             V_WINDOW_HEIGHT, 600, V_END_OF_LIST);
```

Not all attribute flags work on all DataViews drivers. These attributes are device-dependent and can only be set on certain devices.

To set the color table on the device, select the device using *GRselect*, then call *GRs_color_table*.

GRreset

GRdevice functions GR Routines

Resets all internal variables of the driver to the current device attributes.

```
BOOLPARAM
GRreset (void)
```

GRreset resets DataViews to reflect the current attributes of the device. The most important of these attributes are the screen dimensions for the windows. Note that this routine is not implemented for terminals that do not let you change window size.

GRselect

GRdevice functions GR Routines

Selects the current device.

```
BOOLPARAM
GRselect (
       int dev_num)
```

GRselect selects the device specified by *dev_num* and defines it as the current device.

GRset

GRdevice functions GR Routines

Resets device attributes.

```
BOOLPARAM
GRset (
          ULONG flag, <type> value,
          ULONG flag, <type> value,
          ...,
       V_END_OF_LIST)
```

GRset resets attributes of the current device using a variable-length list of attribute/value parameter pairs. For an example of setting device attributes, see *GRopen_set*. For descriptions of the attributes that can be set, see *GRget*.

GRviewport

GRdevice functions GR Routines

Defines a drawing viewport.

```
BOOLPARAM
GRviewport (
       DV_POINT *llp,
       DV_POINT *urp;
```

GRviewport defines the drawing viewport. Objects are clipped to the viewport boundaries. Calling this with a *llp* setting of *NULL* sets the viewport to the full screen.

GRdraw

GRdraw Functions GR Routines

Routines for drawing and positioning graphical objects.

CP is the current position. Objects are drawn using the current foreground color as set by *GRcolor*.

All routines return *DV_SUCCESS* or *DV_FAILURE*.

See Also

GRcolor and *GRcur_point* in *GRinquiry*

Examples

Drawing circles. The following code fragment draws a circle near the center of the screen with a smaller filled circle inside it:

```
DV_POINT p;

p.x = 300;     /* Position center of circle near */
p.y = 300;     /* center of screen. */
GRcircle (&p, 100);          /* Draw a circle of radius 100. */
GRf_circle (&p, 50);         /* Draw a filled circle of radius 50. */
```

Drawing concatenated vectors. The following code fragment draws a series of concatenated vectors which form a triangle. The first and fourth elements of the array represent the same point on the screen, thereby closing the triangle.

```
DV_POINT pt_list[4] = {{ 200, 200 },{ 300,300},
    { 300, 200 },{ 200, 200 }};

GRconcat_vector (pt_list, 4);
```

Drawing polygons. The following code fragment draws a quadrilateral on the screen with a boundary in a different color:

```
static DV_POINT pt_list[] =
{{ 250, 150 },{ 300, 400 },{ 400, 300 },{ 350, 150 }};

GRcolor (1);
GRf_polygon (pt_list, 4);
GRcolor (2);
GRpolygon (pt_list, 4);
```

Drawing rectangles. The following code fragment draws a filled rectangle in one color and its boundary in a different color:

```
static DV_POINT  llp = { 200, 200 },
   urp = { 500, 400 };
GRcolor (1);
GRf_rectangle (&llp, &urp);
GRcolor (2);
GRrectangle (&llp, &urp);
```

Drawing sectors. The following code fragment draws a filled sector which sweeps out a quarter of a circle. The negative value of *delta* indicates that the sector fills the fourth quadrant of the circle. It then draws the arc edge in a different color.

```
static DV_POINT p = { 300, 200 };
```

```
GRcolor (1);
GRf_sector (p, 100, 0, -90);
GRcolor (2);
GRsector (&p, 100, 0, -90);
```

Drawing vectors. The following code fragment draws two line segments on the screen. The CP is moved after drawing the first line segment so that the second one can be drawn in a different location. The first line segment is drawn from left to right. The second is drawn from right to left.

```
DV_POINT p;

p.x = 150;      /* Declare starting location of first line segment. */
GRmove (&p);  /* Move CP to that location. */
p.x += 200;    /* Declare end location. */
GRvector (&p);     /* Draw first line segment from left to right. */
p.y += 100;
GRmove (&p);  /* Move CP up 100 units. */
p.x -= 200;    /* Declare end location of second line segment. */
GRvector (&p);     /* Draw second line segment from right to left. */
```

The following code fragment draws a vector from the CP to a point specified by *end_pt*:

```
DV_POINT p, end_pt;

p.x = 200;
p.y = 200;
GRmove (&p);  /* Reposition CP. */
end_pt.x = 450;                /* Set end point. */
end_pt.y = 400;
GRvector (&end_pt);            /* Draw vector from CP to end point. */
```

Equivalently, the *GRmove* and *GRvector* calls could be replaced by a single call to *GR_move_and_vector* at the end of the code fragment:

```
GRmove_and_vector (&p, &end_pt);
```

Drawing different line types. The following code fragment draws 16 different line types:

```
DV_POINT startp, endp;       /* startp represents CP */
int type;

startp.x = 150;
startp.y = 100;
endp.x = 450;
endp.y = 100;

/* Reposition CP for each line type drawn */
for (type = 1;
     type <= 7;
     type++, startp.y += 15, endp.y += 15)
    {

      /* Move CP to new starting position. */
      GRmove (&startp);
      GRline (&endp, type, 1);
    }

for (type = 8;
     type <= 16;
     type++, startp.y += 15, endp.y += 15)
    GRmv_and_line (&startp, &endp, type, 1);
```

Drawing polar vectors. The following code fragment draws a vector based on a polar coordinate system. After the vector is drawn, a dot is drawn at the origin of the coordinate system.

```
DV_POINT center, startp, endp;
PLR_POINT p0, p1;

p0.radius = 100;
p0.angle = 100;
p1.radius = 250;
p1.angle = 270;
center.x = 300;                /* Coordinates of center of circle. */
center.y = 250;

/* Draw polar coordinate vector */
GRplrvector (&center, &p0, &p1);
GRf_rectangle (&center, &center);
```

GRcolor GRdraw GRraster GRtransform
GRcursor GRinquiry GRrqpcurve GRvtext
GRcurve GRpalette GRtext GRwinevent
GRdevice

GRdraw Functions

GRcircle	Draws an unfilled circle.
GRconcat_line	Draws concatenated patterned lines.
GRconcat_vector	Draws a series of concatenated vectors.
GRf_circle	Draws a filled circle.
GRf_polygon	Draws a filled polygon.
GRf_rectangle	Draws a filled rectangle.
GRf_sector	Draws a filled arc sector.
GRline	Draws a line to a point.
GRmove	Moves the current position (CP).
GRmove_and_vector	Draws a vector between two points.
GRmv_and_line	Draws a line between two points.
GRplrvector	Draws a linear curve in a polar coordinate system.
GRpolygon	Draws an unfilled polygon.
GRrectangle	Draws an unfilled rectangle.
GRsector	Draws an unfilled arc sector.
GRvector	Draws a vector to a point.

GRcircle

GRdraw functions GR Routines

Draws an unfilled circle.

```
BOOLPARAM
GRcircle (
       DV_POINT *center,
       int radius)
```

Draws an unfilled circle of radius, *radius*, around a central point, *center*. *center* must be a pointer to the desired location, in screen coordinates, of the center of the circle. *radius* must be a positive integer representing the distance in screen coordinates from *center* to the edge of the circle. The CP is set to the center of the circle.

GRconcat_line

GRdraw functions GR Routines

Draws concatenated patterned lines.

```
BOOLPARAM
GRconcat_line (
       DV_POINT pt_list[],
       int numpts,
       int type,
       int width)
```

GRconcat_line draws concatenated lines on the selected device. Draws patterned lines, starting with the first point in the array *pt_list*, and ending with the last point in *pt_list*. The number of points in the array is specified by *numpts*. The CP is set to the last point in *pt_list*. *type* and *width* indicate the pattern and width of the concatenated lines.

GRconcat_vector

GRdraw functions GR Routines

Draws a series of concatenated vectors.

```
BOOLPARAM
GRconcat_vector (
       DV_POINT pt_list[],
       int num)
```

Draws a series of concatenated vectors starting at the first point in the points array, *pt_list*. The number of points in the array is specified by *num*. The points must be in screen coordinates. The CP is set to the position represented by the last element of *pt_list*.

GRf_circle

GRdraw functions GR Routines

Draws a filled circle.

```
BOOLPARAM
GRf_circle (
       DV_POINT *center,
       int radius)
```

Draws a filled circle of radius, *radius*, around a central point, *center*. *center* must be a pointer to the desired location, in screen coordinates, of the center of the circle. *radius* must be a positive integer representing the distance in screen coordinates from *center* to the edge of the circle. The CP is set to the center of the circle.

GRf_polygon

GRdraw functions GR Routines

Draws a filled polygon.

```
BOOLPARAM
GRf_polygon (
       DV_POINT pt_list[],
       int num)
```

GRf_polygon draws a filled polygon with *num* vertices, starting at the first point in the points array, *pt_list*, and connecting the last point to the first point.

Each value in *pt_list* must be a point in screen coordinates. These points represent the locations of the vertices of the polygon. *num* must be the number of elements in the array, *pt_list*. The CP is set to the first point in the polygon, which is represented by the value in the first element of the array, *pt_list*.

GRf_rectangle

GRdraw functions GR Routines

Draws a filled rectangle.

```
BOOLPARAM
GRf_rectangle (
       DV_POINT *p1,
       DV_POINT *p2)
```

GRf_rectangle draws a filled rectangle with a lower left corner specified by *p1* and an upper right corner specified by *p2*. *p1* and *p2* must be pointers to points containing screen coordinates. The CP is set to the lower left point, *p1*.

GRf_sector

GRdraw functions GR Routines

Draws filled arc sector.

```
BOOLPARAM
GRf_sector (
       DV_POINT *center,
       int radius,
       int start,
       int delta)
```

GRf_sector draws a filled arc sector of a circle, resembling a pie slice.

center and *radius* define the circle, in screen coordinates, in which the arc is embedded. The CP is set to the location of center.

start specifies the start angle of the arc in degrees counter-clockwise from the horizontal. The allowed range for *start* is [0,359].

delta specifies the number of degrees subtended by the arc. The allowed range for *delta* is [-359,+359]. A positive value for *delta* creates the sector in a counter-clockwise direction. A negative value creates the sector in a clockwise direction.

GRline

GRdraw functions GR Routines

Draws a line to a point.

```
BOOLPARAM
GRline (
       DV_POINT *p,
       int type,
       int width)
```

GRline uses a line pattern specified by *type* to draw a line segment *width* pixels wide from the CP, which can be set using *GRmove*, to a point, *p*.

GRmove

GRdraw functions GR Routines

Moves the current position (CP).

```
BOOLPARAM
GRmove (
       DV_POINT *p)
```

GRmove moves the CP to the point *p*, in screen coordinates, without drawing.

GRmove_and_vector

GRdraw functions GR Routines

Draws a vector between two points.

```
BOOLPARAM
GRmove_and_vector (
       DV_POINT *p1,
       DV_POINT *p2)
```

GRmove_and_vector moves the CP and draws a vector from *p1* to *p2*. Points must be specified in screen coordinates. After vector is drawn, the CP is set to the end point.

GRmv_and_line

GRdraw functions GR Routines

Draws a line between two points.

```
BOOLPARAM
GRmv_and_line (
       DV_POINT *p1,
       DV_POINT *p2,
       int type,
       int width)
```

GRmv_and_line uses a line pattern specified by *type* to draw a line segment *width* pixels wide from point *p1* to point *p2.* The CP is set to the end of the line segment.

Both *width* and *type* should be positive. The interpretation of *type* is device-dependent. Line types 0 and 1 are always solid. There are usually no more than 16 line types.

GRplrvector

GRdraw functions GR Routines

Draws a linear curve in a polar coordinate system.

```
BOOLPARAM
GRplrvector (
       DV_POINT *center,
       PLR_POINT *p0,
       PLR_POINT *p1)
```

GRplrvector draws a linear curve in a polar coordinate system. The curve equation has this form:

```
r = m * theta + b
```

where *theta* is the angle and *r* is the radius. The routine uses this equation to draw the curve in polar coordinates around the point specified by *center*, given a start angle and radius, *p0*, and an end angle and radius, *p1*. The curve connects the two points (*p0* and *p1*) in such a way that the radius varies linearly with the angle. The curve is drawn counter-clockwise from the start angle specified by *p0*, to the end angle specified by *p1*. *center* defines the center of the polar coordinate system in screen coordinates.

The angle portion of the *PLR_POINT* structure is specified in degrees. The *radius* portion of the *PLR_POINT* structure must be in screen coordinates. The curve is drawn in a counter-clockwise direction regardless of the signs of the angles. The CP is set to the position corresponding to *p1*, the end point of the curve.

GRpolygon

GRdraw functions GR Routines

Draws an unfilled polygon.

```
BOOLPARAM
GRpolygon (
       DV_POINT pt_list[],
       int num)
```

GRpolygon draws an unfilled polygon with *num* vertices, starting at the first point in the points array, *pt_list*, and connecting the last point to the first point.

Each value in *pt_list* must be a point in screen coordinates. These points represent the locations of the vertices of the polygon. *num* must be the number of elements in the array, *pt_list*. The CP is set to the first point in the polygon, which is represented by the value in the first element of the array, *pt_list*.

GRrectangle

GRdraw functions GR Routines

Draws an unfilled rectangle.

```
BOOLPARAM
GRrectangle (
       DV_POINT *p1,
       DV_POINT *p2)
```

GRrectangle draws an unfilled rectangle with a lower left corner specified by *p1* and an upper right corner specified by *p2*. *p1* and *p2* must be pointers to points containing screen coordinates. The CP is set to the lower left point, *p1*.

GRsector

GRdraw functions GR Routines

Draws unfilled arc sector.

```
BOOLPARAM
GRsector (
       DV_POINT *center,
       int radius,
       int start,
       int delta)
```

GRsector draws an unfilled arc sector of a circle.

center and *radius* define the circle, in screen coordinates, in which the arc is embedded. The CP is set to the end point of the sector.

start specifies the start angle of the arc in degrees counter-clockwise from the horizontal. The allowed range for *start* is [0,359].

delta specifies the number of degrees subtended by the arc. The allowed range for *delta* is [-359,+359]. A positive value for *delta* creates the sector in a counter-clockwise direction. A negative value creates the sector in a clockwise direction.

GRvector

GRdraw functions GR Routines

Draws a vector to a point.

```
BOOLPARAM
GRvector (
       DV_POINT *p)
```

GRvector draws a vector from the CP to the point, *p*, in screen coordinates. The CP can be set by *GRmove*. Points must be specified in screen coordinates. After vector is drawn, the CP is set to the end point.

GRinquiry

Routines that get information from or about the display device.

See Also

GRdevice

GRinquiry Functions

GRaspect_ratio	Gets x and y pixel-count.
GRcur_point	Gets the current drawing position.
GRcurrent_dev	Gets the current display device number.
GRdepth	Gets the number of bits per pixel.
GRdevname	Gets the current display device name.
GRdevnum	Gets the ordinal number of the current device.
GRisdevopen	Determines if the current device is open.

Unless otherwise noted, all routines return *DV_SUCCESS* or *DV_FAILURE.*

GRaspect_ratio

Gets x and y pixel-count.

```
BOOLPARAM
GRaspect_ratio (
       int *x,
       int *y)
```

GRaspect_ratio gets the number of pixels in the horizontal direction, *x*, and the number of pixels in the vertical direction, *y*, that can be displayed on the current device.

GRcur_point

GRinquiry functions GR Routines

Gets the current drawing position.

```
BOOLPARAM
GRcur_point (
       DV_POINT *pt)
```

GRcur_point gets the current position (CP) for the graphics device. The CP is set by drawing routines such as *GRline*, *GRvector*, and *GRmove*.

GRcurrent_dev

GRinquiry functions GR Routines

Gets the current display device number.

```
BOOLPARAM
GRcurrent_dev (
       int *curr_device)
```

GRcurrent_dev gets the device number of the current device and returns it in *curr_device*.

GRdepth

GRinquiry functions GR Routines

Gets the number of bits per pixel.

```
BOOLPARAM
GRdepth (
       int *depth)
```

GRdepth gets the number of bits per pixel, *depth*, for the screen. The maximum number of colors that can be represented on the device is 2 to the *depth* power.

GRdevname

GRinquiry functions GR Routines

Gets the current display device name.

```
BOOLPARAM
GRdevname (
       int device_ordinal,
       char **device_name)
```

GRdevname gets the device name that corresponds to the given device number and returns it in *device_name*. If there is no device with the given device number, the routine returns *NO* and sets the device name pointer to *NULL*. Note that this routine returns a pointer to an internal name string which should not be modified.

GRdevnum

GRinquiry functions GR Routines

Gets the ordinal number of the current device.

```
BOOLPARAM
GRdevnum (
       char *device_name,
       int *device_ordinal)
```

GRdevnum gets the device number of the named device and returns it in *device_ordinal*. If there is no device with the given name, the routine returns *NO* and sets the device number to -1.

GRisdevopen

GRinquiry functions GR Routines

Determines if the current device is open.

```
BOOLPARAM
GRisdevopen (
       char *device_name)
```

GRisdevopen returns a Boolean value indicating if the named device has been opened yet. Returns *YES* if the device is open and *NO* if it is not.

GRpalette

GRpalette Functions GR Routines

Routines for using the color palette.

Diagnostics

Setting the palette viewport does not affect the viewport set by *GRviewport*; they are different entities.

GRpalpick may set the CP to the location that was picked.

Examples

Drawing the palette. The following code fragment draws the default color palette in the specified viewport.

```
DV_POINT p;
RECTANGLE palette_vp;

/* Specify coordinates of color palette viewport, and draw palette. */
palette_vp.ll.x = 100;
palette_vp.ll.y = 30;
palette_vp.ur.x = 500;
palette_vp.ur.y = 450;
GRpaldraw (&palette_vp);

/* Move CP and write text starting at CP. */
p.x = 100;
p.y = 15;
GRmove (&p);
GRtext ("The color table contains the above colors.");
```

Picking in the palette. The following code fragment displays a color palette and an unfilled rectangle, then asks the user to fill the rectangle with any six colors from the palette. Each color selected fills 1/6th of the rectangle. In each iteration of the loop, the call to *GRpalcrmove* places the cursor in the middle of the color patch which was chosen in the previous iteration. As the user moves the cursor, the current color selection is displayed in the echo viewport.

```
LONG color_index;
static RECTANGLE echovp = {{ 0, 301 },{ 300, 400 }};
static RECTANGLE palette_vp = {{ 300, 200 }, { 600,450 }};
DV_POINT llp, urp;              /* lower left and upper right */
int i;

GRpaldraw (&palette_vp);        /* Draw palette */

/* Draw unfilled rectangle next to palette. Prompt user for colors. */
llp.x = 0;
llp.y = 200;
urp.x = 300;
urp.y = 300;
GRrectangle (&llp, &urp);

printf ("Choose six colors from palette to fill the rectangle. \n");
printf ("Position cursor at a color in palette and press <space> \n");

for (i = 1, urp.x = 50; i < 7; i++, llp.x += 50, urp.x += 50)
   {
   GRpalpick (&echovp, &color_index);    /* User picks a color. */
   GRcolor ((int) color_index);        /* Set desired foreground color. */
   GRf_rectangle (&llp, &urp);         /* Draw small rectangle, said color. */
   GRpalcrmove (color_index);          /* Position cursor at previous choice. */
```

```
    GRflush();                          /* Flush internal display buffers. */
    }
```

Echoing. The following code fragment lets the user move the graphics cursor over the color palette, echoing each color the cursor moves over in a filled circle in the lower left corner of the screen. If the cursor moves off the color palette, the previous color appears in the circle. The program terminates when the user presses any key or mouse button.

```
LONG color_index;
static RECTANGLE palette_vp = {{ 150, 200 },{ 600, 450 }};
static DV_POINT p = { 100, 100 };
int keypress = 0;

GRpaldraw (&palette_vp);            /* Draw color palette. */
GRcr_open_poll();                   /* Turn on graphics cursor. */

/* Let user move the graphics cursor throughout the color palette. */
/* Echo each color patch color in the filled circle. */
while ((keypress = GRpalpoll (&color_index)) == 0)
    {
        GRcolor (color_index);  /* Reset current foreground color. */
        GRf_circle (&p, 50);    /* Draw filled circle with color. */
    }

GRcr_close_poll();              /* Turn off graphics cursor. */
```

GRcolor	GRdraw	GRraster	GRtransform
GRcursor	GRinquiry	GRrqpcurve	GRvtext
GRcurve	GRpalette	GRtext	GRwinevent
GRdevice			

GRpalette Functions

GRpalcrmove	Moves the graphics cursor to the palette color patch corresponding to the specified color.
GRpaldraw	Draws the color palette for the current device in the specified palette viewport.
GRpalhas_pt	Determines if the passed point is inside the drawn palette.
GRpalloc	Gets the color at a given location in the palette.
GRpalpick	Returns a color palette pick.
GRpalpoll	Gets the color currently pointed to by the cursor, and returns any key or button that was pressed.

Unless otherwise noted, all routines return *DV_SUCCESS* or *DV_FAILURE*.

GRpalcrmove

GRpalette functions GR Routines

Moves the graphics cursor to the palette color patch corresponding to the specified color.

```
BOOLPARAM
GRpalcrmove (
       LONG color_index)
```

GRpalcrmove moves the cursor to the center of the color patch that corresponds to the specified color in *color_index*. *color_index* must be an index into the device's color lookup table. For example, if the color lookup table has *n* indices, *color_index* must be in the range 0 to *n-1*.

Must be called after *GRpaldraw*, which draws the palette in which the cursor is to be placed.

GRpaldraw

GRpalette functions GR Routines

Draws the color palette for the current device in the specified palette viewport.

```
BOOLPARAM
GRpaldraw (
       RECTANGLE *palette_vp)
```

GRpaldraw draws the color palette of the current device in the viewport specified by *palette_vp*, and initializes variables that describe the palette's characteristics.

Only one palette can be active at a time. Drawing a second palette supersedes all references to the initial palette, rendering it useless.

Palette_vp must contain two points with values represented in screen coordinates. Note that the palette is adjusted when drawn to ensure that all color boxes are the same size. See *GRpalhas_pt*.

GRpalhas_pt

GRpalette functions GR Routines

Determines if the passed point is inside the drawn palette.

```
LONG
GRpalhas_pt (
       DV_POINT *pt)
```

GRpalhas_pt determines if the point, *pt*, is inside the drawn palette. When a palette is drawn using *GRpaldraw*, the palette is adjusted to ensure that all color boxes are the same size. Therefore, a palette may be drawn smaller than the requested size by a few pixels. Use this routine to determine if your pick is within the drawn palette.

Returns *YES* or *NO*.

GRpalloc

GRpalette functions GR Routines

Gets the color at a given location in the palette.

```
void
GRpalloc (
       DV_POINT *pt,
       LONG *color_index)
```

GRpalloc is passed the address of a point, *pt*, and uses *color_index* to return the color at the location of *pt* within the palette.

GRpalpick

GRpalette functions GR Routines

Returns a color palette pick.

```
int
GRpalpick (
       RECTANGLE *echovp,
       LONG *color_index)
```

GRpalpick lets the user select a color from the color palette. A color can be chosen by moving the cursor to the color patch that represents the desired color, then pressing any key or mouse button. *GRpalpick* waits for the key or button press, gets the color selected in the color palette, and returns the key or button that was pressed.

If an echo viewport is used, it echoes each color the cursor moves over. After a key or button is pressed, the echo viewport echoes only the color selected and does not change until the routine is called again. However, if the cursor moves beyond the boundaries of the color palette, the echo viewport displays the color that corresponds to the value of *color_index* at the time *GRpalpick* was called and echoes this color until the cursor is repositioned inside the palette. If a key or button is pressed while the cursor is outside the palette, the echo viewport displays this original color until the routine is called again, and the pick is not serviced. Therefore, the calling program should determine if there is a pick to be serviced after each call to *GRpalpick*.

GRpaldraw must be called before *GRpalpick* so that the color palette can be displayed on the screen.

The use of an echo viewport is optional. If it is not needed, a *NULL* pointer should be passed to the routine in place of the *echovp* argument. If an echo viewport is used, *echovp* must contain two points, in screen coordinates, which determine the location of the echo viewport.

color_index behaves as both an entry and an exit parameter, containing the original color on entry and the new color on exit (or the original color on exit if the cursor was outside the palette viewport when the key or button was pressed).

Returns the key or button that was pressed.

GRpalpoll

GRpalette functions GR Routines

Gets the color currently pointed to by the cursor, and returns any key or button that was pressed.

```
int
GRpalpoll (
       LONG *color_index)
```

GRpalpoll sets *color_index* to the color currently pointed to by the graphics cursor and returns any key or button that was pressed.

This routine allows color selection from the color palette as drawn by *GRpaldraw*. However, unlike *GRpalpick*, *GRpalpoll* does not wait for a key or button press before returning the color selected. Instead, it immediately returns the color being pointed to by the graphics cursor.

This routine assumes that the graphics cursor is already open, which means that *GRcr_open_poll* must be called before calling *GRpalpoll*. The graphics cursor must also be closed before the main program terminates, so *GRcr_close_poll* must be called to terminate. If the graphics cursor is not on the palette when *GRpalpoll* is called, *color_index* is set to the most recent color index.

Returns any key or button pressed; otherwise *NULL*.

GRraster

GRraster Functions GR Routines

Routines that handle raster operations (rasterops) to and from the display surface. Some terminals do not support rasterops. *GRrasquery* lets you query the device using to determine what raster operations are supported.

Rasters let you set pixels on the display device to specific colors. They also let you take a snapshot of part of a screen. DataViews rasters have their origin in the lower left. The origin (*ll*), width, and height parameters for rasters should be specified in screen coordinates and should indicate valid positions within the window.

To create a raster, either use *GRrascreate* to create an empty raster or use *GRrasget* to create a raster that contains a copy from the screen; don't use both on the same raster.

GRrasgpxrp, *GRrassmaskpxrp*, and *GRrasspxrp* handle raster operations using pixreps. A pixrep is a description of a rectangular block of pixels arranged in a flexible layout. Pixreps are explained in detail in the *VUpixrep* section of the *VU Routines* chapter.

Unless otherwise noted, these routines return *DV_SUCCESS* or *DV_FAILURE*. *DV_FAILURE* can indicate either that there was an invalid parameter or that the routine is not supported for the current device. To determine which routines are supported, use *GRrasquery*.

GRraster Functions

GRrascreate	Creates a new raster array.
GRrasdraw	Draws a raster array.
GRrasdrawpart	Draws a portion of a raster array.
GRrasfree	Frees the raster array storage area.
GRrasget	Gets a raster array.
GRrasgpix	Gets a pixel value in a raster array.
GRrasgpxrp	Fills in a pixrep to look like a raster.
GRrasmove	Copies and moves a raster array.
GRrasquery	Asks the selected device about rasterop capabilities.
GRrassize	Gets raster size information.
GRrassmask	Sets the draw mask for the raster.
GRrassmaskpxrp	Sets the draw mask for the raster using a pixrep.
GRrasspix	Sets a pixel value in a raster array.
GRrasspixels	Sets all of the raster's pixels at once.
GRrasspxrp	Sets all of the raster's pixels at once using a pixrep.
GRrasvalid	Determines whether or not an address contains a valid raster.

GRrascreate

GRraster functions GR Routines

Creates a new raster array.

```
BOOLPARAM
GRrascreate (
       int height,
       int width,
       ADDRESS *raster)
```

GRrascreate creates a new raster array compatible with the current device. *width* and *height* determine the size in screen coordinates. Returns the raster pointer in *raster*. The newly created raster array contains random values for the pixels. To set the pixel values, call *GRrasspix*. The raster must be destroyed by calling *GRrasfree* when it is no longer needed. To reuse the raster, call *GRrasfree* before calling *GRrasget*.

GRrasdraw

GRraster functions GR Routines

Draws a raster array.

```
BOOLPARAM
GRrasdraw (
       ADDRESS raster,
       DV_POINT *ll)
```

GRrasdraw draws the raster array to the current device starting at the lower left origin, *ll*. *ll* is in screen coordinates.

GRrasdrawpart

GRraster functions GR Routines

Draws a portion of a raster array.

```
BOOLPARAM
GRrasdrawpart (
       ADDRESS raster,
       DV_POINT *ll,
       RECTANGLE *portion)
```

GRrasdrawpart draws part of the raster array to the current device. *raster* is a device-dependent raster pointer and *portion* is the part of the raster to draw. *portion* is relative to the origin of *raster*, which is specified by *ll*.

GRrasfree

GRraster functions GR Routines

Frees the raster array storage area.

```
BOOLPARAM
GRrasfree (
       ADDRESS raster)
```

GRrasfree frees the storage area that was allocated for the raster array.

GRrasget

GRraster functions GR Routines

Gets a raster array.

```
BOOLPARAM
GRrasget (
       DV_POINT *ll,
       int width,
       int height,
       ADDRESS *raster)
```

GRrasget creates and gets the raster array of a viewport from the current device. The viewport is specified by the origin, *ll*, and *width* and *height*. Returns the raster pointer in *raster*. The raster must be destroyed by calling *GRrasfree* when it is no longer needed. To reuse the raster, call *GRrasfree* before calling *GRrasget*.

GRrasgpix

GRraster functions GR Routines

Gets a pixel value in a raster array.

```
LONG
GRrasgpix (
       ADDRESS raster,
       DV_POINT *point)
```

GRrasgpix returns the index of the color at a pixel in the raster array. On some monochrome devices, the normal color sense of 0 = white is reversed so 0 = black.

GRrasgpxrp

GRraster functions GR Routines

Fills in a pixrep to look like a raster.

```
BOOLPARAM
GRrasgpxrp (
       PIXREP *pixrep,
       ADDRESS raster)
```

GRrasgpxrp allocates storage for a pixrep and fills in the pixrep to look like the raster in *raster*. This routine does not affect the raster itself, but copies pixel values from the raster into the pixrep structure. This routine is usually much faster than using *GRrasgpix* for all the pixels in the raster.

GRrasmove

GRraster functions GR Routines

Copies and moves a raster array.

```
BOOLPARAM
GRrasmove (
       DV_POINT *ll,
       DV_POINT *ur,
       DV_POINT *dest)
```

GRrasmove copies the specified raster array on the current device to the position where *dest* is the lower left corner.

GRrasquery

GRraster functions GR Routines

Asks the selected device about rasterop capabilities.

```
BOOLPARAM
GRrasquery (
       int question)
```

GRrasquery queries the current device about its rasterop capabilities. The flags, defined in *dvGR.h*, determine if the corresponding routines exist in the driver. The valid flags are:

RAS_CREATE	GRrascreate
RAS_DRAW	GRrasdraw
RAS_DRAWPART	GRrasdrawpart
RAS_GET	GRrasget
RAS_GPIX	GRrasgpix
RAS_GPXRP	GRrasgpxrp
RAS_MOVE	GRrasmove
RAS_SMASK	GRrassmask
RAS_SMASKPXRP	GRrassmaskpxrp
RAS_SPIX	GRrasspix
RAS_SPIXELS	GRrasspixels
RAS_SPXRP	GRrasspxrp

If the query returns *NO*, the corresponding GR routine is not implemented. For example, if *GRrasquery (RAS_MOVE)* returns *NO*, *GRrasmove* does not work.

GRrassize

GRraster functions GR Routines

Gets raster size information.

```
BOOLPARAM
GRrassize (
       ADDRESS raster,
       int *width,
       int *height,
       int *depth)
```

GRrassize gets information about the size of the specified raster. If a particular argument is *NULL*, that information is not provided. Most devices have a fixed raster depth, so it is not necessary to specify a raster in order to determine depth. Therefore, you can determine the depth of a raster on a device by using the following call:

```
GRrassize (NULL, NULL, NULL, &depth);
```

GRrassmask

GRraster functions GR Routines

Sets the draw mask for the raster.

```
BOOLPARAM
GRrassmask (
       ADDRESS raster,
       ADDRESS values)
```

GRrassmask assigns a two-dimensional draw mask to the raster using the value array, *values*. The values in the raster draw mask indicate which pixels of the raster to draw. If the value is 1, the corresponding pixel in the raster is drawn in the next call to *GRrasdraw* or *GRrasdrawpart*. Since the mask values must be 0 or 1, *values* must be an array of bytes. The size of the array should correspond to the number of pixels in the raster.

GRrassmaskpxrp

GRraster functions GR Routines

Sets the draw mask for the raster using a pixrep.

```
BOOLPARAM
GRrassmaskpxrp (
       ADDRESS raster,
       PIXREP *pixrep,
       COLOR_XFORM *xform)
```

GRrassmaskpxrp assigns a two-dimensional draw mask to the raster using *pixrep*. The pixrep data is scaled to the size of the raster. The values in the draw mask indicate which pixels of the raster to draw. The pixrep must be using indirect color. Pixels with a color index of *0* are not drawn, pixels with any other index are drawn. The color indices in the pixrep can be transformed using an optional user-supplied *xform* when the raster is created. *xform* specifies a color transform that changes the interpretation of the mask.

GRrasspix

GRraster functions GR Routines

Sets a pixel value in a raster array.

```
BOOLPARAM
GRrasspix (
       ADDRESS raster,
       DV_POINT *point,
       LONG value)
```

GRrasspix sets a pixel specified by *point* in the raster array to a color index, *value*. *point* is specified in screen coordinates with the origin at the lower left.

GRrasspixels

GRraster functions GR Routines

Sets all of the raster's pixels at once.

```
BOOLPARAM
GRrasspixels (
       ADDRESS raster,
       ADDRESS values,
       int value_unit)
```

GRrasspixels sets the raster's pixels to the color index values in the value array, *values*. The size of the array should correspond to the number of pixels in the raster. *value_unit* indicates the size of the individual values. If the values are bytes, use *1* for *value_unit*. If the values are *LONG*s, use *4* for *value_unit*.

GRrasspxrp

GRraster functions GR Routines

Sets all of the raster's pixels at once using a pixrep.

```
BOOLPARAM
GRrasspxrp (
       ADDRESS raster,
       PIXREP *pixrep,
       COLOR_XFORM *xform)
```

GRrasspxrp modifies the raster to look like the pixrep by setting the raster's pixels to the color values in the *pixrep*. For pixreps using indirect color, the color indices in the pixrep can be transformed using an optional user-supplied *xform*. *xform* specifies a color transform that changes the interpretation of the colors in the pixrep. *xform* is ignored by pixreps using direct color.

The raster size may change. If the colors in the pixrep are not all available to the device, this function applies various methods to get a close match to the pixrep.

GRrasvalid

GRraster functions GR Routines

Determines whether or not an address contains a valid raster.

```
BOOLPARAM
GRrasvalid (
       ADDRESS raster)
```

GRrasvalid determines whether or not the address, *raster*, contains a valid raster. Returns *DV_SUCCESS* if *raster* points to a valid raster. Otherwise returns *DV_FAILURE*.

GRrqpcurve

GR Functions GR Routines

Routines for calculating the points on rational quadratic parametric (rqp) curves and drawing them. Rqp curves can represent any conic section.

These routines manipulate and draw rational quadratic parametric curve based on the form:

$$x(t) = \frac{a_x t^2 + b_x t + c_x}{a_w t^2 + b_w t + c_w} \qquad y(t) = \frac{a_y t^2 + b_y t + c_y}{a_w t^2 + b_w t + c_w}$$

The coefficients are defined by 3 points and a “fullness factor”, k. If the fullness factor is 1, the curve is a section of a parabola and the rqp representation becomes identical to the Bezier formulation. When k>1 the curve is a section of an ellipse; when k<1 the curve is a section of a hyperbola. The curve is entirely contained in the convex hull of the three points for parameter values t in the range [0,1]. For more information on rqp curves, see *Computational Geometry for Design and Manufacture*, by I.D. Faux and M.J. Pratt.

See Also

GRcurve

Example

Given that array *cp[3]* contains three points in screen coordinates, the following code fragment draws a portion of an ellipse on the screen using a precision value of 1.

```
float k;
k = 1.0;
GRrqpprecision(1);
GRrqpdraw (cp, &k, 0, 0);
```

GRrqpcurve Functions

GRrqpdraw	Draws an rqp curve.
GRrqpprecision	Specifies how precisely to draw the rqp curve.
GRrqppts	Get the points on an rqp curve
GRrqpsize	Gets number of points needed for an rqp curve
GRrqpsplit	Splits an rqp curve in half.

GRrqpdraw

GRrqpcurve functions GR Routines

Draws an rqp curve.

```
void
GRrqpdraw (
       DV_POINT cp[3],
       float *k,
       int linepattern,
       int linewidth)
```

GRrqpdraw draws the portion of the rqp curve that is inside the three control points specified by *cp[3]*. The parameter *k* is described above. The rqp curve is drawn using the attributes *linepattern* and *linewidth*. If *linepattern* and *linewidth* are *NULL*, a single-width solid line is drawn.

GRrqpprecision

GRrqpcurve functions GR Routines

Specifies how precisely to draw the rqp curve.

```
int
GRrqpprecision (
       int max_deviation)
```

GRrqpprecision specifies the precision for use in approximating an rqp curve with straight lines. The precision value is the maximum deviation allowed between the drawn curve and the ideal curve. Therefore, a value of zero (0) for *max_deviation* gives the maximum precision and larger values give less precision. Returns the old precision value. A negative precision value returns the current precision with no change.

GRrqppts

GRrqpcurve functions GR Routines

Get the points on an rqp curve

```
int
GRrqppts (
       DV_POINT cp[3],
       float *k,
       DV_POINT *ptbuf,
       int bufsize)
```

GRrqppts calculates the points on the curve for the parameter in the range [0,1] given the parametric equation for a rqp 2D curve. The points calculated are in screen coordinates. *GRrqppts* returns the number of points added to the points buffer.

GRrqpsize

GRrqpcurve functions GR Routines

Gets number of points needed for an rqp curve

```
int
GRrqpsize (
       DV_POINT cp[3],
       float *k)
```

GRrqpsize returns the estimated maximum number of points that would be required to represent a specified rqp curve. It may actually take fewer points. This estimates the number of points for a parabola where *k*=1. Representing the curve might actually require fewer points.

GRrqpsplit

GRrqpcurve functions GR Routines

Splits an rqp curve in half.

```
void
GRrqpsplit (
       DV_POINT incp[3],
       float *ink,
       DV_POINT outcp1[3],
       float *outk1,
       DV_POINT outcp2[3],
       float *outk2)
```

GRrqpsplit splits an rqp curve in half. *incp[3]* is an array of control points for the input rqp, and *ink* is the address of its fullness factor. *outcp1[3]* is the array of control points for the first output rqp, and *outk1* is its fullness factor. *outcp2[3]* is the array of control points for the second output rqp, and *outk2* is its fullness factor.

GRtext

GRtext Functions

GR Routines

Routines for writing text on the current device and controlling the character size. Character size is given in both the horizontal dimension (*xsize*) and the vertical dimension (*ysize*). For most devices, *xsize* and *ysize* must be the same.

The allowed ranges for *xsize* and *ysize* are device-dependent. Usually the ranges for both arguments are about [1,4]. Larger values yield larger characters, but *xsize* and *ysize* do not translate directly to a scale factor.

Diagnostics

Character size values are not directly related to the size of the text, and produce different scaling factors for different devices. For example, changing text size from 1 to 2 does not necessarily make the text twice as wide. Also, some values may have no effect on the scaling factors on some devices. For example, on a particular device, 1 and 3 might produce small and large characters respectively, but 2 might not change the size at all.

The rectangle mentioned above should not be confused with the one created by *GRrectangle* and *GRf_rectangle*. The rectangle associated with a text string is created by *GRtext* and appears on the screen as a delimiter around the height and length of the text string.

Examples

Different text sizes. The following code fragment writes two text strings of different sizes to the screen, each at a different current position (CP).

```
DV_POINT p;

p.x = 10;
p.y = 200;
GRmove (&p);  /* Move CP to the above position on screen. */
GRch_size (1, 1);
GRtext ("This string's characters are of a certain width and height.");
p.y = 300;
GRmove (&p);  /* Move CP to a higher position on screen. */
GRch_size (3, 3);                  /* Change size of characters. */
GRtext ("These characters are larger, so the string is longer.");
```

Clipping of text strings. The following code fragment writes two text strings to the screen. One starts at the top left of the viewport; the other starts near the middle and is partially blocked by the viewport boundaries. This example illustrates the importance of positioning text inside viewport boundaries:

```
DV_POINT llp, urp, ulp;

/* Set the viewport. */
llp.x = 200;
llp.y = 200;
urp.x = 500;
urp.y = 400;
GRviewport (&llp, &urp);

/* Reposition cursor and write text. */
ulp.x = llp.x;
ulp.y = urp.y - 10;       /* Leave room at top of viewport for characters. */
textp = "This string is located at the top of viewport";
GRmove (&ulp);
GRtext (textp);       /* Write text to screen. */

ulp.x = 300;
ulp.y = 300;
```

```
GRmove (&ulp);
GRtext ("Part of this string is hidden because of viewport boundaries");       /* Write
        new string. */
```

Getting text size in screen coordinates. The following code fragment defines a string, *textp*, determines its size in screen coordinates, and prints the size on the screen.

```
static char *textp = "This string has a certain height and width";
int xsize, ysize;
DV_POINT p;

p.x = 50;
p.y = 250;
GRmove (&p);           /* move CP to location where string should start */
GRch_size (3, 3);      /* create a large string */
GRtext (textp);        /* write string to screen */
GRtextsize (textp, &xsize, &ysize);

/* get size of string in x- and y-coordinates */
printf ("The screen coordinates for the length and \n");
printf ("height of this string are as follows: \n");
printf ("length (xsize) = %d; height (ysize) = %d \n", xsize, ysize);
```

GRcolor	GRdraw	GRraster	GRtransform
GRcursor	GRinquiry	GRrqpcurve	GRvtext
GRcurve	GRpalette	**GRtext**	GRwinevent
GRdevice			

GRtext Functions

GRch_size	Sets the scaling factors of characters in a text string.
GRg_ch_size	Gets the current character size for a device.
GRtext	Writes a string of text to the screen on the selected device.
GRtextsize	Returns the size of a text string in screen coordinates.

All routines return *DV_SUCCESS* or *DV_FAILURE*.

GRch_size

GRtext functions GR Routines

Sets the scaling factors of characters in a text string.

```
BOOLPARAM
GRch_size (
        int xsize,
        int ysize)
```

GRch_size sets the character scaling factors for graphics text where the horizontal factor is defined by *xsize* and the vertical factor is defined by *ysize*.

Any change in the scaling factors of a string affects all subsequent drawings of the text.

GRch_size is usually called before calling *GRtext*. However, using *GRch_size* is optional. If this routine is not used, the scaling factors of a string are automatically set to device-dependent default values when *GRtext* is called.

GRg_ch_size

GRtext functions GR Routines

Gets the current character size for a device.

```
BOOLPARAM
GRg_ch_size (
       int *xsize,
       int *ysize)
```

GRg_ch_size gets the current *x* and *y* character scaling factors for the current device.

GRtext

Writes a string of text to the screen on the selected device.

```
BOOLPARAM
GRtext (
       char *textp)
```

GRtext writes the string of text specified by *textp* at the current position (CP).

GRtext creates a rectangular boundary around the written text. This boundary is the size of the character height and the text length, and acts as a backdrop for the text. The CP is located at the lower left corner of this rectangle and moves to the lower right corner after the text string is written, so that any subsequent text is appended to the end of the string. The rectangular area is the color most recently specified in *GRbackcolor*. The text string is the color most recently specified in *GRcolor*.

Calling *GRch_size* before *GRtext* lets you specify the height and width of the characters in the string. This is optional, however. The default size is *xsize* = 1 and *ysize* = 1 (see *GRch_size*).

The displayed text string is clipped to the current viewports.

GRtextsize

GRtext functions GR Routines

Returns the size of a text string in screen coordinates.

```
BOOLPARAM
GRtextsize (
       char *textp,
       int *xsize,
       int *ysize)
```

GRtextsize returns the size of a text string, *textp*, in screen coordinates, *xsize* and *ysize*. The size of a string is the height (*ysize*) and width (*xsize*) of the text's rectangular boundary, in screen coordinates.

GRtransform

GRtransform Functions GR Routines

Converts screen coordinates to virtual coordinates and vice versa.

Screen coordinates are device-dependent. Virtual coordinates are in the range [0,32767], where the point (0,0) is the lower left corner of the screen and (32767,32767) is the upper right corner. Therefore, the entire virtual coordinate system space corresponds to the visible part of the bitmap. Note that a rectangle that is square in the virtual coordinate system is not generally square in the screen coordinates system.

GRtransform Functions

GRscs_to_vcs	Converts screen coordinates to virtual coordinates.
GRvcs_to_scs	Converts virtual coordinates to screen coordinates.

Both return *DV_SUCCESS* or *DV_FAILURE*.

GRscs_to_vcs

GRtransform functions GR Routines

Converts screen coordinates to virtual coordinates.

```
BOOLPARAM
GRscs_to_vcs (
       DV_POINT *input_p,
       DV_POINT *virtual_p)
```

GRvcs_to_scs

GRtransform functions GR Routines

Converts virtual coordinates to screen coordinates.

```
BOOLPARAM
GRvcs_to_scs (
       DV_POINT *input_p,
       DV_POINT *screen_p)
```

GRvtext

GRvtext Functions GR Routines

Routines to manipulate vector text.

GRvtext Functions

GRfreevfont	Frees memory allocated to a vector font.
GRgetvfont	Gets the current vector font index.
GRgetvfontname	Returns the font filename of a font index.
GRgetvheight	Gets the height vector of a vector text string.
GRgetvmaxwidth	Gets the width of the widest character in the vector string after transformation.
GRgetvnorm	Gets the vector text size normalization factor.
GRgetvspace	Gets the inter-character and inter-line spacing.
GRgetvwidth	Gets vector text string width vector.
GRvfont	Sets the current vector font and loads into memory.
GRvfontname	Assigns and returns the vector font index.
GRvspace	Sets inter-character and inter-line spacing.
GRvtext	Draws vector text at current position.
GRvtmatrix	Sets vector text transformation matrix.

GRfreevfont

GRvtext functions GR Routines

Frees memory allocated to a vector font.

```
BOOLPARAM
GRfreevfont (
       int nfont)
```

GRfreevfont frees the memory allocated to a vector font. Makes the font index available for newly loaded fonts. Returns *DV_SUCCESS* if the font memory is freed successfully. Returns *DV_FAILURE* if the font index is invalid or the font has already been freed. The *VO* level does not currently call this function. In other words, the *VOvt* routines do not currently free a loaded font even if it is no longer referred to by any active vector texts.

GRgetvfont

GRvtext functions GR Routines

Gets the current vector font index.

```
int
GRgetvfont (void)
```

GRgetvfont returns the index of the current font, or *-1* if no font is currently set.

GRgetvfontname

GRvtext functions GR Routines

Returns the font filename of a font index.

```
char *
GRgetvfontname (
       int nfont)
```

GRgetvfontname returns the font filename of the font referred to by the font index. Every font indexed using *GRvfontname* retains its font filename. This prevents the user from opening identical font files. Returns the font filename character string pointer. This function also returns an internal pointer to the string which should be modified with care.

GRgetvheight

GRvtext functions GR Routines

Gets the height vector of a vector text string.

```
void
GRgetvheight (
       int nlines,
       int *x,
       int *y)
```

GRgetvheight gets information about the height of a vector text block in the current transform, font, and spacing, given the number of lines of text, *nlines*. The variables *x* and *y* are the coordinates of the “text up vector” after transformation. The text up vector begins at the lower left corner of first character body in the bottom line of text and ends at the upper left corner of the first character body in the top line of text. If *nlines* is 0, the height returned is just the interline spacing (after transformation). If *nlines* is -1, the return values represent the height of the one line of text plus the transformed interline spacing.

GRgetvmaxwidth

GRvtext functions GR Routines

Gets the width of the widest character in the vector string after transformation.

```
void
GRgetvmaxwidth (
       char *str,
       int *x,
       int *y)
```

GRgetvmaxwidth gets information about the width of the widest character in a vector text string, in the current font and spacing after transformation. The variables *x* and *y* are the coordinates of the text baseline vector after the transformation. The text baseline is a vector that begins at the lower left corner of the first character body and ends at the lower right corner of the last character body of the string.

GRgetvnorm

GRvtext functions GR Routines

Gets the vector text size normalization factor.

```
void
GRgetvnorm (
       int pixheight,
       float *normfactor)
```

GRgetvnorm gets the normalization factor of the current font. The normalization factor is the ratio of the screen coordinate height to the actual height, *pixheight*. The actual height of a font is defined in the font file. *GRgetvnorm* is primarily useful for reducing all fonts to a standard size, which is system-defined at the *VO* level as *DEF_VFONT_SIZE*. The following example demonstrates its use:

```
float normfactor;
int x, y;

GRgetvnorm (DEF_VFONT_SIZE, &normfactor);
GRgetvheight (1, &x, &y);
newx = normfactor * x;
newy = normfactor * y;
```

GRgetvspace

GRvtext functions GR Routines

Gets the inter-character and inter-line spacing.

```
void
GRgetvspace (
       float *charspace,
       float *linespace)
```

GRgetvwidth

GRvtext functions GR Routines

Gets vector text string width vector.

```
BOOLPARAM
GRgetvwidth (
       char *str,
       int *x,
       int *y)
```

GRgetvwidth inquires about the width of a vector text string in the current font and spacing after transformation. The variables *x* and *y* are the coordinates of the text baseline vector after the transformation. The text baseline is a vector that begins at the lower left corner of the first character body and ends at the lower right corner of the last character body of the string. A tab string (“\t”) returns the transformed inter-character spacing. A null pointer, *NULL*, returns the transformed average character width of the font. A null character (“” or “\0”) returns just the slant vector. Returns *YES* if the text is backslanted. Otherwise returns *NO*.

GRvfont

GRvtext functions GR Routines

Sets the current vector font and loads into memory.

```
BOOLPARAM
GRvfont (
       int nfont)
```

GRvfont sets the current font using the font index. The font index is assigned using *GRvfontname*. If the font is not yet loaded into memory, *GRvfont* tries to read it in from a font file using the font name passed to *GRvfontname*. *GRvfont* does not reload a font that is already loaded into memory. Returns *DV_SUCCESS* if the font index is valid and the font has been or can be loaded. Returns *DV_FAILURE* if the font index is invalid or if an error is encountered when reading in the file. See also *GRvfontname*.

GRvfontname

GRvtext functions GR Routines

Assigns and returns the vector font index.

```
int
GRvfontname (
       char *fontname)
```

GRvfontname stores vector text font filename, *fontname*, then assigns and returns a unique font index. The index is used to refer to the font in *GRvfont*. Does not load the font into memory. Returns the font index. See also *GRvfont*.

GRvspace

GRvtext functions GR Routines

Sets inter-character and inter-line spacing.

```
void
GRvspace (
       double charspace,
       double linespace)
```

GRvspace sets inter-character and inter-line spacing. The inter-character spacing is specified as a fraction of the font's average character height and equal the spacing added between adjacent characters before transformation. The default value is 0.0. The inter-line spacing is also specified as a fraction of the font's height and equals the spacing added between two lines of text. The default value is 0.0.

GRvtext

GRvtext functions GR Routines

Draws vector text at current position.

```
void
GRvtext (
       char *string)
```

GRvtext draws vector text at the current position (CP) using the current font and spacing after applying the transformation set by *GRvtmatrix*. The current font is set by *GRvfont* and the current spacing is set by *GRvspace*. The current position is set by *GRmove*. If the vector definition of a character does not exist, it is not drawn. See also *GRvfont*, *GRvspace*, *GRmove*, *GRvtmatrix*.

GRvtmatrix

GRvtext functions GR Routines

Sets vector text transformation matrix.

```
void
GRvtmatrix (
       float tmatrix[2][2])
```

GRvtmatrix sets a two-by-two transformation matrix for transforming vector text, where the matrix is the product of the scaling, rotation, and shearing matrix. The transformation matrix is stored internally using fixed point arithmetic.

GRwinevent

GRwinevent Functions GR Routines

Routines that facilitate the use of system windowing features within DV-Tools applications. Since these routines are device-dependent, not all device drivers support them. If not supported, these routines return *DV_FAILURE*. Routines are also provided at the *VO* level for handling window events, in the *VOlo* and *VOsc* sections.

The *WINEVENT* structure contains information about events such as key strokes, mouse motion, and resizing that occur in windowing systems. A listing of the structure is located in *DataViews Public Types* in the *Include Files* chapter.

GRcolor	GRdraw	GRraster	GRtransform
GRcursor	GRinquiry	GRrqpcurve	GRvtext
GRcurve	GRpalette	GRtext	**GRwinevent**
GRdevice			

GRwinevent Functions

GRwe_convert	Converts a system-dependent event to a *WINEVENT.*
GRwe_gmask	Gets the window event mask
GRwe_mask	Sets the window event mask.
GRwe_poll	Returns the next window event in the event queue.
GRwe_state	Returns information about the last polled event.

These routines are not implemented by all device-drivers; therefore, they return *DV_SUCCESS* when they are implemented, and *DV_FAILURE* when they cannot be implemented.

GRwe_convert

GRwinevent functions GR Routines

Converts a system-dependent event to a *WINEVENT.*

```
BOOLPARAM
GRwe_convert (
       ADDRESS event,
       WINEVENT *we)
```

GRwe_convert converts a system-dependent event structure to a *WINEVENT* structure. Fills the fields of a *WINEVENT* structure with information from the system-dependent event, including filling the *eventdata* field with the address of the system-dependent event structure. *event* is the address of the system-dependent event structure and *we* is a pointer to the *WINEVENT* structure that is filled.

GRwe_gmask

GRwinevent functions GR Routines

Gets the window event mask.

```
BOOLPARAM
GRwe_gmask (
       ULONG *mask,
       ULONG *altmask)
```

GRwe_gmask gets the window event mask, which specifies which of the possible DataViews window event types is returned by *VOloWinEventPoll*, *VOscWinEventPoll*, or *GRwe_poll*, and passes it to *mask*. The mask is an unsigned long integer in which each bit represents a different type of window event. The types of events are represented by a set of constants defined in *dvGR.h*. The window system-dependent mask is returned in *altmask*. *mask* or *altmask* can be bitwise-ANDed together (using the & symbol in C) with the desired mask to determine if the mask is set correctly.

To get the actual system-dependent mask which results from the combination of *mask* and *altmask*, use *GRget* with the *V_XWINDOW_MASK*.

GRwe_mask

GRwinevent functions GR Routines

Sets the window event mask.

```
BOOLPARAM
GRwe_mask (
       ULONG mask,
       ULONG altmask)
```

GRwe_mask sets the current window’s event mask, *mask*, which specifies which DataViews window event types are returned by *GRwe_poll*. The mask is an unsigned long integer where each bit represents a different type of window event. The mask can be constructed by bitwise-OR’ing the *WINEVENT type* flags representing the events to be noted. The mask acts as a positive filter which passes only the desired events occurring in that window to the event queue. For example, the following call:

```
GRwe_mask (V_KEYPRESS | V_MOTIONNOTIFY, (ULONG)0);
```

lets *GRwe_poll* report only key press and mouse motion events.

Certain event type flags require additional information to be specified in *altmask*. *altmask* is an unsigned long integer that is interpreted with a special flag in *mask*. For example, when the flag *V_XWINDOW_MASK* is OR’ed into *mask,* it tells *GRwe_mask* to look in *altmask* for an X11 event mask. This allows any X Window event to be returned. If the event does not fall into one of the standard DataViews event types, it is returned in the *WINEVENT type* field as *V_NON_STANDARD_EVENT*.

To interpret a system-dependent event, you can access the *eventdata* field of the *WINEVENT* structure, where the windowing system’s event data structure is copied. For example, under X the *XEvent* structure is copied into the *eventdata* field. For more information about how it handles events, including flags for *altmask* and the system-specific event data structure, refer to your windowing system manual.

Normally, *GRwe_mask* replaces the previous window event mask. However, if the *V_ADD_TO_MASK* flag is OR’ed into *mask*, the events are added to the existing mask. See also *GRwe_gmask*, which you can use to get the current mask and altmask.

The following *WINEVENT type* flags can be used to construct the *mask* parameter:

V_KEYPRESS	Any key press, including modifier keys (shift, control, etc.) and function keys.
V_KEYRELEASE	Any key release, including modifier keys (shift, control, etc.) and function keys.
V_BUTTONPRESS	Any mouse button press.
V_BUTTONRELEASE	Any mouse button release.
V_MOTIONNOTIFY	Any motion of the mouse, with or without the mouse buttons down.
V_ENTERNOTIFY	The mouse has entered the window.
V_LEAVENOTIFY	The mouse has left the window.
V_WINDOW_ICONIFY	User requests a window iconify.
V_EXPOSE	Some portion of the window has been exposed and needs to be redrawn. The *rectlist* field of the *WINEVENT* structure contains a pointer to an array of the exposed rectangular regions, and is currently only implemented for X.

V_RESIZE The window size has changed.
V_WINDOW_QUIT User requests a window quit.
The following modifiers can be OR'ed with the window event mask:

V_EVENTS_OFF	Turns off all events, regardless of events that have been OR'ed into the mask.
V_ADD_TO_MASK	Indicates that the flags should be added to the current mask, not replace it.
V_XWINDOW_MASK	Indicates that *altmask* is an X11 event mask.

GRwe_poll

GRwinevent functions GR Routines

Returns the next window event in the event queue.

```
BOOLPARAM
GRwe_poll (
       int mode,
       int source,
       WINEVENT *we)
```

GRwe_poll returns the next window event in the event queue. This information is copied into the *WINEVENT* structure, *we*. Only event types that have been specified in the call to *GRwe_mask* are returned. If no mask is set, the default mask passes key press, key release, button press, button release, motion notify, window quit, enter notify, leave notify, iconify, expose, and resize events to the event queue. If the window contains widgets, the event queue may contain non-DataViews events. These events are always passed onto the queue, regardless of the event mask.

mode specifies which of the following types of polling modes is used. When the event queue is empty and *mode* is *V_WAIT*, *GRwe_poll* does not return until an event specified by *mask* or *altmask* in *GRwe_mask* is generated. If *mode* is *V_NO_WAIT*, *GRwe_poll* does not wait until an event is generated, but returns *V_NO_EVENT* as the type of event.

source determines whether events from other windows are reported. If *source* is *V_CURRENT_WINDOW*, only events from the current window are reported. If *source* is *V_MULTIPLE_WINDOW*, all events in the event queue are reported, regardless of their window origin. This flag is effective only where windows of the same device type share a single event queue.

we must be a pointer to a *WINEVENT* structure, which is a DataViews public type. For more information about the *WINEVENT* structure fields, see *dvGR.h* and the *Include Files* chapter. When *altmask* is specified in *GRwe_mask* for handling device-specific events, these events are returned in the *WINEVENT type* field as the flag *V_NON_STANDARD_EVENT*. The system event structure for interpreting the event can be accessed through the *eventdata* field of the *WINEVENT* structure.

GRwe_state

GRwinevent functions GR Routines

Returns information about the last polled event.

```
BOOLPARAM
GRwe_state (
       WINEVENT *we)
```

GRwe_state returns information about the last polled event. This information is copied into the *devnum*, *loc*, *maxpoint,* and *state* fields of the *WINEVENT* structure, *we*.

The *state* field is returned in an unsigned long integer where each bit represents the state of different modifier keys or mouse buttons. The state can be interpreted using the list of modifier keys and mouse buttons state flags, which are OR'ed together to reflect the combination of modifier keys and mouse buttons. These flags are found in *VOloState*.

Include Files

Include files contain typedefs for public types, defined constants, and function declarations for the DV-Tools routines. The include files necessary to call routines in a layer are listed in the introduction to that layer; the include files necessary for a particular module in the layer are listed in the synopsis for that module. Below is a summary of the contents of each include file. For more details, the files themselves may be examined.

Tfundecl.h	function declarations for *T* routines (formerly *dvtoolsfuns.h*)
VOfundecl.h	function declarations for *VO* and *VOob* routines (formerly *VOfuns.h*)
VUerfundecl.h	function declarations for *VUer* routines
VGfundecl.h	function declarations for *VG* routines
VPfundecl.h	function declarations for *VP* routines
VTfundecl.h	function declarations for *VT* routines
VUfundecl.h	function declarations for *VU* routines
GRfundecl.h	function declarations for *GR* routines
std.h	standard macros and constants (includes *stdio.h*)
dvGR.h	constants used by *GR* and window management routines
GRkeysymdef.h	key symbols used in *WINEVENT* structure
GRkeysym.h	defines which group of key symbols in *GRkeysymdef.h* are used as defaults
GRlink.h	indices into link tables used by *GR* graphics routines
VUpixrep.h	structures and macros for use with pixreps
VUtextarray.h	macros, constants, and public types used by *VUta* routines
dvstd.h	constants, public types used by *VP/VG/V*U routines
dstypes.h	data source type constants
dvmarker.h	constants representing markers for graphs
dvdatatypes.h	data type constants
VOstd.h	constants, public types used by the *VO* and *VOob* routines
dvtools.h	constants used by *T* routines
dvinteract.h	constants used by the event handler and input objects
dvaxis.h	constants used by the *VUax* routines
dvrule.h	definitions for event, condition, and action constants that an application can use to define rules
dvruletab.h	contains several tables to help interpret conditions and actions.
dvfds.h	macros, enums, and defines for function descriptor sets and data sources.
hashtypes.h	constants and public types for *VT* hash table routines.
ringbuf.h	macros and typedef for creating ring buffers
dvenv.h	device-specific defines
FDSeval.h, FDSevallex.h FDSevalfuns.h	include files for use with the Function Descriptor Set *FDSeval*

Include Files

Defined Constants

The include files contain definitions for the following DV-Tools constants. By convention, they are all upper case. Most are flags that indicate various messages to DV-Tools routines. The first column contains the constant name and the second contains its defined value. The third may contain a brief description, valid values for flag-value pairs, the type of the value for flag-value pairs, or other information.

Standard I/O File Descriptors *(std.h)*

```
STDIN                   0
STDOUT                  1
STDERR                  2
```

Boolean Values *(std.h)*

```
YES                     1
NO                      0
```

Looping Macros *(std.h)*

```
FOREVER                 for (;;)
```

Useful I/O Constants *(std.h)*

```
BUFSIZE                 512
BWRITE                  -1
READ                    "r"
WRITE                   "w"
READ_WRITE              "????"
APPEND                  "a"
BYTMASK                 0377
```

Return Value of DV-Tools Functions *(dvstd.h* and *VOstd.h)*

```
DV_SUCCESS              YES
DV_FAILURE              NO
```

World Coordinate Ranges *(VOstd.h)*

XMAX	16383	upper right corner
YMAX	16383	
XMIN	-16384	lower left corner
YMIN	-16384	

Coordinate Type Range *(VOstd.h)*

```
MAXCOORD                 32767
MINCOORD                -32768
```

ForEach and Traversal Flags *(dvstd.h)*

```
V_CONTINUE_TRAVERSAL    0
V_HALT_TRAVERSAL        1
```

Data Type Flags *(dvdatatypes.h)*

```
V_C_TYPE           1    char
V_UC_TYPE          2    unsigned char, UBYTE
V_S_TYPE           3    short
V_US_TYPE          4    unsigned short
```

`V_L_TYPE`	`5`	`LONG`
`V_I_TYPE`	`5`	`int`
`V_UL_TYPE`	`6`	`ULONG`
`V_UI_TYPE`	`6`	`unsigned int`
`V_F_TYPE`	`7`	`float`
`V_D_TYPE`	`8`	`double`
`V_T_TYPE`	`9`	text string, *NULL*-terminated
`V_DSV_TYPE`	`10`	data source variable
`V_NULL_TYPE`	`0`	list terminator

Attribute Fields Enumerated Constants *(VOstd.h)*

`FOREGROUND_COLOR`	`1`
`BACKGROUND_COLOR`	`2`
`LINE_WIDTH`	`3`
`LINE_TYPE`	`4`
`FILL_STATUS`	`5`
`TEXT_DIRECTION`	`6`
`TEXT_POSITION`	`7`
`TEXT_FONT`	`8`
`TEXT_SIZE`	`9`
`ARC_DIRECTION`	`10`
`CURVE_TYPE`	`11`
`TEXT_FONTNAME`	`12`
`TEXT_WIDTH`	`13`
`TEXT_HEIGHT`	`14`
`TEXT_ANGLE`	`15`
`TEXT_SLANT`	`16`
`TEXT_CHARSPACE`	`17`
`TEXT_LINESPACE`	`18`
`PROP_FILL`	`22`
`FILL_AMOUNT`	`23`
`TEXT_UNDERLINE`	`24`
`TEXT_WEIGHT`	`25`
`TEXT_PTSIZE`	`26`

Attributes Structure Values *(VOstd.h)*

General attribute field defined constants:

`EMPTY_FIELD`	`-2`	indicates empty field
`EMPTY_FLOAT_FIELD`	`-99999.0`	indicates empty field
`DONT_SET_THE_VALUE`	`((OBJECT)-2)`	indicates not to change field

Line type attribute field defined constant:

`SOLID_LINE`	`1`

Fill status attribute field defined constants:

`FILL`	`’f’`	
`EDGE`	`’u’`	
`EDGE_WITH_FILL`	`0xBF`	
`FILL_WITH_EDGE`	`0xFB`	
`DV_TRANSPARENT`	`’t’`	
`FILLED_OBJECT`	`FILL`	maintained for compatibility
`UNFILLED_OBJECT`	`EDGE`	maintained for compatibility

Text direction attribute field defined constants:

`HORIZONTAL_TEXT`	`’h’`

```
VERTICAL_TEXT           'v'
```

Text position attribute field defined constants:

```
AT_TOP_EDGE             0x1
AT_BOTTOM_EDGE          0x2
AT_LEFT_EDGE            0x4
AT_RIGHT_EDGE           0x8
CENTERED                0x10
POSITION_FLAGS_MASK     0x1F      bitwise OR of above flags to make 9 positions
```

Text weight attribute defined constants:

```
NORMAL_WEIGHT           400
BOLDFACE_WEIGHT         700
```

Arc direction attribute field defined constants:

```
CLOCKWISE               'r'
COUNTER_CLOCKWISE       'l'
```

Proportional fill attribute field defined constants:

```
PROP_FILL_NONE          ((char)0)
PROP_FILL_RIGHT         ((char)1)
PROP_FILL_UP            ((char)2)
PROP_FILL_LEFT          ((char)3)
PROP_FILL_DOWN          ((char)4)
```

Polygon curve attribute field defined constants:

```
FLOATING_ENDS           'f'
OPEN_ENDS               'o'
CLOSED_ENDS             'c'
```

Attributes Structure Default Values *(VOstd.h)*

```
DEF_BACKGROUND_COLOR    NULL
DEF_FOREGROUND_COLOR    NULL
DEF_LINE_WIDTH          1
DEF_LINE_TYPE           SOLID_LINE
DEF_FILL_STATUS         UNFILLED_OBJECT
DEF_TEXT_DIRECTION      HORIZONTAL_TEXT
DEF_TEXT_POSITION       CENTERED
DEF_TEXT_FONT           0
DEF_TEXT_SIZE           2
DEF_ARC_DIRECTION       COUNTER_CLOCKWISE
DEF_CURVE_TYPE          NULL
DEF_VFONT_SIZE          1024
DEF_TEXT_CHARSPACE      0.0
DEF_TEXT_LINESPACE      0.0
DEF_TEXT_WIDTH          1.0
DEF_TEXT_HEIGHT         1.0
DEF_TEXT_ANGLE          0.0
DEF_TEXT_SLANT          0.0
DEF_TEXT_FONTNAME       "roman.vf"
DEF_PROP_FILL           PROP_FILL_NONE
DEF_FILL_AMOUNT         SHORT_MAX
```

DEF_SFTEXT_FONTNAME	"Times New Roman"
DEF_SFTEXT_WIDTH	0.0
DEF_SFTEXT_HEIGHT	0.0

* This value is in tenths of a point.

These attributes replace *DEF_TEXT_FONTNAME*, *DEF_TEXT_WIDTH*, and *DEF_TEXT_HEIGHT* for scalable font text objects. They have no meaning for objects that are not scalable font text objects.

VOxxStatistic Flags *(VOstd.h)*

`OBJECT_COUNT`	`'c'`	returns number of given object types

Display Formatter Entry Points *(dvstd.h)*

```
V_INITIAL_DISPLAY      0
V_CLEANUP_ALLOCS       1
V_UPDATE_DISPLAY       2
V_CANT_DISPLAY         3
V_QUERY_DISPLAY        4
V_SETUP_DISPLAY        5
V_DRAW_CONTEXT         6
V_DRAW_DATA            7
V_TAKE_DATA            8
V_RECV_MESSAGE         9
V_DFTABLE_SIZE         11
```

Datum Type Flags *(VOstd.h)*

```
FLOAT_DATUM            (DATUM_TYPE) 'f'
INT_DATUM              (DATUM_TYPE) 'i'
TEXT_DATUM             (DATUM_TYPE) 't'
OBJECT_DATUM(obtype)   (DATUM_TYPE)('O')|
                       (DATUM_TYPE)(obtype<<8)
```

Datum Type Macros *(VOstd.h)*

```
IS_FLOAT_DATUM(datype)  ((datype)==FLOAT_DATUM)
IS_INT_DATUM(datype)    ((datype)==INT_DATUM)
IS_OBJECT_DATUM(datype) ((datype)&0xFF)=='O'))
DATUM_O_TYPE(datype)    ((datype)>>8)&0xFF)
IS_TEXT_DATUM(datype)   ((datype)==TEXT_DATUM)
```

Undefined Values *(dvparams.h* and *dvstd.h)*

`V_UNDEFINED`	`-1`	
`UNDEFINED_COLOR_INDEX`	`0x7fffffff`	undefined color specification

Window Attribute Flags *(dvGR.h)*

`V_END_OF_LIST`	`0`		
`V_DRAW_FUNCTION`	`0x55570011`	`V_XOR` or `V_COPY`	open/set/get
`V_WINDOW_WIDTH`	`0x55572001`	`int`	open/set/get
`V_WINDOW_HEIGHT`	`0x55572011`	`int`	open/set/get
`V_WINDOW_X`	`0x55572021`	`int`	open/set/get
`V_WINDOW_Y`	`0x55572031`	`int`	open/set/get
`V_WINDOW_NAME`	`0x55572041`	`char *`	open/set/get
`V_CLUT_DEPTH`	`0x55522051`	`int`	get

V_RASTER_DEPTH	0x55522061	int	get
V_EVENTS_REPORTED	0x55522071	ULONG	get

Window system data structures:

V_INPUT_FD	0x55521001	int	get
V_DISPLAY	0x55531021	Display *	open/get
V_ICON_NAME	0x55571071	char *	open/set/get
V_MOTION_COLLAPSE	0x55551081	BOOLPARAM	open/set
V_EXPOSE_COLLAPSE	0x55551091	BOOLPARAM	open/set

DataViews pre-defined cursors:

V_ACTIVE_CURSOR	0x55553000	No Value	open/set
V_INITIAL_CURSOR	0x55553010	No Value	open/set

Queries about capabilities of the driver and system:

V_HAS_WINEVENTS	0x55524001	BOOLPARAM	get
V_HAS_PLANE_MASKING	0x55524011	BOOLPARAM	get
V_HAS_XOR	0x55524021	BOOLPARAM	get
V_IS_BLACK_AND_WHITE	0x55524031	BOOLPARAM	get
V_IS_WINDOW_SYSTEM	0x55524041	BOOLPARAM	get
V_NUM_FONTS	0x55524051	int	get

Queries about the system-specific masks:

V_XWINDOW_MASK	0x1000000	ULONG	get
V_WINNT_MASK	0x8000000	ULONG	get

Microsoft Windows-specific data structures:

V_WIN32_NEWFONT	0x55559052	int, HFONT	open/set
V_WIN32_WINDOW_HANDLE	0x55579061	HWND	open/set/get
V_WIN32_WINDOWPROC	0x55529081	function ptr	get
V_WIN32_DOUBLE_BUFFER	0x55579091	int	open/set/get
V_WIN32_XORFLAG	0x555790A1	int	open/set/get
V_WIN32_IS_DV_DEVICE	0x555290B2	HWND, int *	get
V_WIN32_HPALETTE	0x555390C1	HPALETTE	open/get
V_WIN32_ICON_NAME	0x555790D1	char *	open/set/get

X11-specific data structures:

V_X_WINDOW_ID	0x55536001	Window	open/get
V_X_DISPLAY	0x55536011	Display *	open/get
V_X_DISPLAY_NAME	0x55536021	char *	open/get
V_X_APPLIC_CLASS	0x55536031	char *	open/get
V_X_APPLIC_NAME	0x55536041	char *	open/get
V_X_CURSOR	0x55576071	Cursor	open/set/get
V_X_ICON	0x55576081	char *	open/set/get
V_X_ICON_WIDTH	0x55576091	int	open/set/get
V_X_ICON_HEIGHT	0x555760A1	int	open/set/get
V_X_SHELL	0x555360B1	Widget	get
V_X_DRAW_WIDGET	0x555360C1	Widget	open/get
V_X_FONTSTRUCT	0x555760D2	int, XFontStruct *	open/set/get
V_X_APPLIC_CONTEXT	0x555360E1	XtAppContext	open/get
V_X_RAS_SYNC	0x555760F1	BOOLPARAM	open/set/get
V_X_EXPOSURE_BLOCK	0x55576131	BOOLPARAM	open/set/get
V_X_RESIZE_BLOCK	0x55576141	BOOLPARAM	open/set/get
V_X_DOUBLE_BUFFER	0x55576151	BOOLPARAM	open/set/get
V_X_COLORMAP	0x55576161	Colormap	open/set/get
V_X_PIXELS	0x55576172	int, unsigned long[]	

			open/set/get
`V_X_PLANES`	`0x55576182`	`int, unsigned long[]`	open/set/get
`V_X_GC`	`0x55526191`	`GC`	get
`V_X_POLY_HINT`	`0x555761A1`	`int`	open/set/get
`V_X_IMAGE_STRING`	`0x555761B1`	`BOOLPARAM`	open/set/get
`V_X_DASH_STYLE`	`0x555761C1`	`LineDoubleDash` or `LineOnOffDash`	open/set/get
`V_X_ICON_X`	`0x555161D1`	`int`	open
`V_X_ICON_Y`	`0x555161E1`	`int`	open
`V_X_ICONIC`	`0x555161F1`	`BOOLPARAM`	open

WINEVENT type Flags *(dvGR.h)*

```
V_KEYPRESS             0x1
V_KEYRELEASE           0x2
V_BUTTONPRESS          0x4
V_BUTTONRELEASE        0x8
V_MOTIONNOTIFY         0x10
V_ENTERNOTIFY          0x20
V_LEAVENOTIFY          0x40
V_EXPOSE               0x80
V_RESIZE               0x100
V_WINDOW_QUIT          0x200
V_WINDOW_ICONIFY       0x400

V_NON_STANDARD_EVENT   0x800
V_NON_DV_WINDOW_EVENT  0x1000

V_EVENTS_OFF           0x10000
V_NO_EVENT             0x20000
V_ADD_TO_MASK          0x40000
V_XWINDOW_MASK         0x1000000
```

WINEVENT state Flags *(dvGR.h)*

```
V_STATE_SHIFT          0x1
V_STATE_LOCK           0x2
V_STATE_CONTROL        0x4
V_STATE_MOD1           0x8
V_STATE_MOD2           0x10
V_STATE_MOD3           0x20
V_STATE_MOD4           0x40
V_STATE_MOD5           0x80
V_STATE_BUTTON1        0x100
V_STATE_BUTTON2        0x200
V_STATE_BUTTON3        0x400
V_STATE_BUTTON4        0x800
V_STATE_BUTTON5        0x1000
```

WINEVENT Polling Modes *(dvGR.h)*

```
V_WAIT                 1
V_NO_WAIT              2
```

GRbspcubics, GRbspdraw end_conditions Flags *(VOstd.h)*

```
OPEN_ENDS              'o'
CLOSED_ENDS            'c'
```

FLOATING_ENDS	'f'

GRcr_event Flags *(dvGR.h)*

V_LOC_CHANGE_WAIT	1
V_LOC_PICK_WAIT	2
V_LOC_NO_WAIT	3
V_LOC_PICK_NO_WAIT	4

GRrasquery Flags *(dvGR.h)*

RAS_CREATE	14
RAS_DRAW	12
RAS_DRAWPART	19
RAS_GET	13
RAS_GPIX	16
RAS_GPXRP	22
RAS_MOVE	11
RAS_SMASK	20
RAS_SMASKPXRP	23
RAS_SPIX	17
RAS_SPIXELS	18
RAS_SPXRP	21

GRwe_poll source Flags *(dvGR.h)*

V_CURRENT_WINDOW	1
V_MULTIPLE WINDOW	2

TdlSave, TdsSave, and TviFileSave access_mode Flags *(VOstd.h)*

WRITE_EXPANDED	'W'	ASCII write
WRITE_COMPACT	'w'	binary write

TdpGetXform Flags *(VOstd.h)*

DR_TO_SCREEN	3	drawing to screen xform
SCREEN_TO_DR	4	screen to drawing xform

TdrGetSelectedObject Flags *(dvtools.h)*

NAMED_SEARCH	0	search view for selection of named object
FULL_SEARCH	1	search entire view for selected object

TdsEditAttributes, TdsvEditAttributes Error! Reference source not found.**Flags** *(dvtools.h)*

NOCHANGE	-1	for attributes to remain unchanged

TdsvEditAttributes, TdsvGetAttributes delimiter Flags *(dvstd.h)*

V_SINGLE_QUOTED	'\001'	two single quotes separate text strings.
V_DOUBLE_QUOTED	'\002'	two double quotes separate text strings.

TdsEditAttributes, TdsGetAttributes Type and Format Flags *(dstypes.h)*

DSPROCESS	1	process data source type
DSFILE	2	file data source type
DSCONSTANT	3	constant data source type
DSFUNCTION	5	function data source type

`DSMEMORY`	`6`	memory data source type
`DSASCII`	`2`	ASCII file or process data source format
`DSBINARY`	`3`	binary file or process data source format

TdsvGetGlobalFlag, TdsvSetGlobalFlag Flags *(dvstd.h)*

`V_LOCAL`	`1`
`V_GLOBAL`	`2`

TloPoll Flags *(dvtools.h)*

`LOC_POLL`	`0`	return valid location object in any event
`WAIT_PICK`	`1`	block until selection key or button
`WAIT_CHANGE`	`2`	block until cursor movement or key press
`PICK_POLL`	`3`	does not block, returns location object

TprotoHandleInput Return Flag *(dvtools.h)*

`V_TPROTO_QUIT`	`-1`

TscPrintSet Flags *(dvGR.h* and *dvstd.h)*

`VP_PRINT_ORIENTATION`	`0x555490E1`
`VP_PRINT_SCALE`	`0x555490F1`
`VP_PRINT_QUALITY`	`0x55549111`
`VP_PRINT_DEVICE`	`0x55549131`
`VP_PRINT_PORT`	`0x55549141`
`VP_PRINT_DRIVER`	`0x55549151`
`VP_PRINT_NO_WARNING`	`0x55549161`
`VP_PRINT_DOCUMENT_NAME`	`0x55549171`

`DV_PORTRAIT`	`1`
`DV_LANDSCAPE`	`2`
`DV_DRAFT`	`-1`
`DV_LOW`	`-2`
`DV_MEDIUM`	`-3`
`DV_HIGH`	`-4`

TviMergeAddDataSources, TviMergeDataSources, TdsMerge Flags *(dvtools.h)*

`DS_EXACTMATCH`	`2`	match ds's exactly when merging views
`DS_SUBSETMATCH`	`3`	one ds must be a subset of a current ds
`DS_NAMEMATCH`	`4`	only ds names must match when merging

VOcoCreate, VOcoSubtype Flags *(VOstd.h)*

`COLOR_COMPONENTS`	`'c'`	three color primaries in range [0,255]
`COLOR_INDEX`	`'i'`	color look-up-table index
`COLOR_NAME`	`'n'`	color name character string
`COLOR_REFERENCE`	`'r'`	referenced color object
`COLOR_STRUCTURE`	`'s'`	pointer to a *COLOR_SPEC* structure

VOdqAdd, Error! Reference source not found.**VOdqAddDq Flags** *(VOstd.h)*

`TOP`	`'t'`	top of deque
`BOTTOM`	`'b'`	bottom of deque

VOdrBackcolor Flags *(VOstd.h)*

`NO_BACKGROUND`	`-1`	transparent drawing background

VOdrOffcolor Flags *(VOstd.h)*

`NO_OFF_DRAWING_COLOR`	`-1`	off-drawing region is transparent

VOdynamics Dynamic Action Flags *(VOstd.h)*

`V_DYN_ROTATE`	`30`
`V_DYN_PATH_MOVE`	`31`
`V_DYN_REL_MOVE_X`	`32`
`V_DYN_REL_MOVE_Y`	`33`
`V_DYN_ABS_MOVE_X`	`34`
`V_DYN_ABS_MOVE_Y`	`35`
`V_DYN_SCALE`	`36`
`V_DYN_SCALE_X`	`37`
`V_DYN_SCALE_Y`	`38`
`V_DYN_SUBDRAWING`	`39`
`V_DYN_FILL_RIGHT`	`40`
`V_DYN_FILL_UP`	`41`
`V_DYN_FILL_LEFT`	`42`
`V_DYN_FILL_DOWN`	`43`
`V_DYN_TEXT`	`44`
`V_DYN_VISIBILITY`	`45`

See also Attribute Field Enumerated Constants that are used for attribute dynamics.

VOdyGetEraseMethod, VOdySetEraseMethod Flags *(VOstd.h)*

`V_DYN_ERASE_REDRAW_DELAY`	`1`
`V_DYN_ERASE_RASTER`	`2`
`V_DYN_ERASE_OBJECT`	`3`
`V_DYN_ERASE_XOR`	`4`
`V_DYN_ERASE_NONE`	`5`
`V_DYN_ERASE_REDRAW_IMMEDIATE`	`6`
`V_DYN_ERASE_BOX`	`7`

VOinGetInternal Flags *(dvinteract.h)*

`TRANSFORM`	`0`	layout to screen transform
`AREA_DEQUE`	`1`	deque of menu item bounding rectangles
`OBJECT_TRANS`	`2`	transform for drawing embedded input objects
`INOBJS_DEQUE`	`3`	deque of embedded input objects
`OBJECT_DEQUE`	`4`	deque of menu objects
`ITEM_DEQUE`	`5`	deque of menu text objects
`INITIAL_VALUE`	`6`	original value of the variable descriptor
`INITIAL_XVALUE`	`7`	original value of the *x* variable descriptor
`INITIAL_YVALUE`	`8`	original value of the y variable descriptor
`ECHO_VIEWPORT`	`9`	primary echo area

VOinPutFlag, VOinGetFlag Flags *(dvinteract.h)*

`SAVE_RASTER`	`1`	save overwritten portion of screen (*YES/NO*)
`ERASE_METHOD`	`2`	how to erase interaction area when done
`DRAW_LAYOUT_BOUND`	`3`	draw layout boundary (*YES/NO*)
`DRAW_ECHO_BOUND`	`4`	draw echo viewport boundary (*YES/NO*)
`REDRAW_ON_UPDATE`	`6`	redraw obscuring objects (*YES/NO*)

VOinPutFlag, VOinGetFlag ERASE_METHOD Flags *(dvinteract.h)*

`RESTORE_RASTER`	`0`	restore the saved raster, if possible
`CALL_REDRAW`	`1`	repair damage by calling *VOscRedraw*
`ERASE_RECTANGLE`	`2`	erase the viewport to the background

`NO_ERASE`	3	don’t erase, just cleanup the data

VOinState Flags *(dvinteract.h)*

`ACTIVE`	1
`INACTIVE`	2

VOitKeyOrigin Flags *(dvinteract.h)*

`LOCAL_KEYS`	0
`GLOBAL_KEYS`	1

VOitPutEchoFunction Flags *(dvinteract.h)*

`INITIAL_DRAW`	0	Called when drawn
`TAKE_INPUT`	1	Called when input is taken
`UPDATE_DRAW`	2	Called when explicitly updated
`ERASE`	3	called when erased
`CONTEXT_REDRAW`	4	called when redrawn
`SETUP_FOR_DRAW`	5	sub-action for setting draw information
`CONTEXT_DRAW`	6	sub-action for drawing static portion
`CLEANUP_DATA`	7	sub-action for clearing data
`DATA_RESET`	8	sub-action for resetting data

VOitPutList, VOitGetList Flags *(VOstd.h)*

`TEXT_LIST`	`’t’`	pickable item list of text strings
`OBJECT_LIST`	`’o’`	pickable item list of objects
`NO_LIST`	`NULL`	no pickable item list

VOobType Object Type Flags *(VOstd.h)*

`LOWEST_TYPE_CODE`	1	
`HIGHEST_TYPE_CODE`	41	
`OT_ARC`	1	arc object
`OT_CIRCLE`	3	circle object
`OT_COLOR`	4	color object
`OT_DEQUE`	7	deque object
`OT_DG`	8	data group object
`OT_DRAWING`	9	drawing object
`OT_DYNAMIC`	37	dynamic control object
`OT_EDGE`	30	edge object
`OT_ELLIPSE`	32	ellipse object
`OT_ICON`	13	icon object
`OT_IMAGE`	5	image object
`OT_INPUT`	27	input object
`OT_INPUT_TECHNIQUE`	28	input technique object
`OT_LINE`	11	line object
`OT_LOCATION`	12	location object
`OT_NODE`	31	node object
`OT_PIXMAP`	2	pixmap object
`OT_POINT`	15	point object
`OT_POLYGON`	16	polygon object
`OT_RECTANGLE`	17	rectangle object
`OT_REFCOLOR`	38	reference color object
`OT_RGB`	18	RGB color object
`OT_RULE`	36	rule object
`OT_SCREEN`	19	screen object

OT_SLOTKEY	33	slotkey object
OT_SUBDRAWING	21	subdrawing object
OT_TEXT	22	text object
OT_THRESHTABLE	23	threshold table object
OT_VD	24	variable descriptor object
OT_VTEXT	29	vector text object
OT_XFORM	26	transform object

VOptCreate, VOptFCreate Flags *(VOstd.h)*

PIXEL_COORDINATES	'p'	screen coordinate pt
SCREEN_COORDINATES	'p'	screen coordinate pt
WORLD_COORDINATES	'w'	world coordinate pt

VOptMove Flags *(VOstd.h)*

DV_ABSOLUTE	'A'	move absolute pt by absolute amount
DV_RELATIVE	'a'	move absolute pt by relative amount
ADJUST_OFFSET_WORLD	'r'	adjust relative pt in world coords
ADJUST_OFFSET_SCREEN	'p'	adjust relative pt in screen coords

VOruGetInfo, VOruSetInfo Flags *(dvrule.h)*

Rule components:

V_R_EVENT	1
V_R_CONDITION	2
V_R_ACTION	3

Rule Events:

V_RE_PICK	1	
V_RE_DONE	2	
V_RE_ACCEPT	3	superseded by *V_RE_EVENT_USED*
V_RE_CANCEL	4	
V_RE_DRAW	5	
V_RE_UPDATE	6	
V_RE_EVENT_USED	7	
V_R_NUM_EVENTS	7	

Rule Conditionals Operands:

V_RC_ALWAYS	1
V_RC_PICK_BUTTON	2
V_RC_PICK_ASCII	3
V_RC_DSV_VALUE	4
V_RC_DSV_DSV	5
V_RC_OBJ_VAR_VALUE	6
V_R_NUM_CONDITIONS	6

Rule Conditionals Operators:

V_RC_EQUAL	1
V_RC_NOT_EQUAL	2
V_RC_LESS_THAN	3
V_RC_LESS_EQUAL_THAN	4
V_RC_GREATER_THAN	5
V_RC_GREATER_EQUAL_THAN	6
V_RC_NUM_OPERATORS	6

Rule Actions:

V_RA_NEXT	1

`V_RA_PREVIOUS`	`2`
`V_RA_OVERLAY_VIEW`	`3`
`V_RA_DEL_OVERLAY_VIEW`	`4`
`V_RA_OVERLAY_OBJ`	`5`
`V_RA_DEL_OBJECT`	`6`
`V_RA_POPUP_AT`	`7`
`V_RA_ERASE_ALL_POPUP_AT`	`9`
`V_RA_REDRAW`	`10`
`V_RA_QUIT`	`11`
`V_RA_NOTHING`	`12`
`V_RA_SYSTEM_CALL`	`13`
`V_RA_ERASE_ALL_OVERLAYS`	`14`
`V_RA_START_DYNAMICS`	`15`
`V_RA_STOP_DYNAMICS`	`16`
`V_RA_INC_UPDATE_RATE`	`17`
`V_RA_DEC_UPDATE_RATE`	`18`
`V_RA_SET_DSV`	`19`
`V_RA_INC_DSV`	`20`
`V_RA_DEC_DSV`	`21`
`V_R_NUM_ACTIONS`	`21`

VOsdGetDynamicFlag, VOsdSetDynamicFlag Flags *(VOstd.h)*

`SD_DYN_NONE`	`0`	get
`SD_DYN_DISABLED`	`1`	get/set
`SD_DYN_ENABLED`	`2`	get/set
`SD_DYN_RESET`	`3`	set

VOsdGetSelectedObject Flags *(dvtools.h)*

`NAMED_SEARCH`	`0`	search view for selection of named object
`FULL_SEARCH`	`1`	search entire view for selected object

VOskDeclare, VOskGetType Flags *(VOstd.h)*

`VOSK_EXTERNAL_TYPE`	`((int)'x')`
`VOSK_INT_ARRAY_TYPE`	`((int)'I')`
`VOSK_INT_TYPE`	`((int)'i')`
`VOSK_STRING_TYPE`	`((int)'t')`
`VOSK_FLOAT_ARRAY_TYPE`	`((int)'F')`
`VOSK_FLOAT_TYPE`	`((int)'f')`
`VOSK_OBJECT_TYPE`	`((int)'o')`

VOuObMove Flags *(VOstd.h)*

`RELATIVE_MOVE`	`'r'`	move by a relative amount
`ABSOLUTE_MOVE`	`'a'`	move to an absolute position

VOvdCreate, VOvdType Flags *(VOstd.h)*

`COLOR`	`'c'`	color type variable descriptor; obsolete
`NUMBER`	`'n'`	number type variable descriptor
`DV_TEXT`	`'t'`	text type variable descriptor

VGdgdfstatus *(dvstd.h)*

`V_DGDF_CANT_DRAW`	`0x1`	did the setup fail?
`V_DGDF_SETUP_DONE`	`0x2`	did the setup succeed?
`V_DGDF_CONTEXT_DRAWN`	`0x4`	was the context drawn?
`V_DGDF_ALL`	`0x7`	all

VPdgaxlabel, VGdgaxlabel, VGdgticlabfcn, VPdgticlabfcn, VUdgticlabtab Flags *(dvstd.h)*

`V_FIRST_AXIS`	`'1'`	first spatial dimension
`V_SECOND_AXIS`	`'2'`	second spatial dimension
`V_TIME_AXIS`	`'t'`	time dimension

VPdgcontext, VGdgcontext, VUdbgCcf Flags *(dvstd.h)*

`V_FPRE_ERASE`	`0x1`	erase before drawing?
`V_FCONTEXT`	`0x2`	draw context?
`V_FLEGEND`	`0x4`	draw legend?
`V_FVPBOX`	`0x8`	draw box around graph?
`V_FT_TICS`	`0x10`	draw time axis ticks?
`V_FT_MINTICS`	`0x20`	minimum time ticks?
`V_FT_LABEL_TICS`	`0x40`	label time axis ticks?
`V_FD1_TICS`	`0x80`	draw d1 axis ticks?
`V_FD1_MINTICS`	`0x100`	minimum d1 ticks?
`V_FD1_LABEL_TICS`	`0x200`	label d1 axis ticks?
`V_FD2_TICS`	`0x400`	draw d2 axis ticks?
`V_FD2_MINTICS`	`0x800`	minimum d2 ticks?
`V_FD2_LABEL_TICS`	`0x1000`	label d2 axis ticks?
`V_FV_TICS`	`0x2000`	draw value axis ticks?
`V_FV_MINTICS`	`0x4000`	minimum value ticks?
`V_FV_LABEL_TICS`	`0x8000`	label value axis ticks?
`V_FV_MULT_RANGE`	`0x10000`	multiple value ranges?
`V_FV_GRID`	`0x20000`	draw value axis grid?
`V_FT_GRID`	`0x40000`	draw time axis grid?
`V_FPITCH_TICS`	`0x80000`	draw pitch axis ticks?
`V_FPITCH_LABEL_TICS`	`0x100000`	label pitch axis ticks?
`V_FROLL_TICS`	`0x200000`	draw roll axis ticks?
`V_FROLL_LABEL_TICS`	`0x400000`	label roll axis ticks?
`V_F_ALL`	`0x7fffff`	all the flags

VPdgdfquery Flags *(dvstd.h)*

`V_Q_DATA_SAMPLE`	`12`	gets the number of the closest sample
`V_Q_DATA_SLOTSIZE`	`7`	gets the size of an element in spectro graphs
`V_Q_DATA_VALUE`	`13`	gets the closest data value
`V_Q_DATAVP`	`0`	gets the area devoted to encoding
`V_Q_DOES_CLIPPING`	`5`	determines whether the formatter clips
`V_Q_FLOOR_VALUE`	`14`	gets the underlaying value in stacked graphs
`V_Q_LEGSIZE`	`6`	gets the size of the legend
`V_Q_SAMPLE_AT_LOCATION`	`11`	gets the interpolated sample at a point
`V_Q_SECTOR_AT_LOCATION`	`15`	gets the sector in radial graphs
`V_Q_SLOT_AT_LOCATION`	`8`	gets the slot number at a point
`V_Q_SLOTSIZE`	`1`	gets the size of a slot
`V_Q_VALUE_AT_LOCATION`	`10`	gets the value at a point
`V_Q_VDPS_AT_LOCATION`	`9`	gets the vdps displaying data at a point
`V_Q_VDTITLE_CHARSIZE`	`4`	gets title size from the *VDtext* display
`V_Q_VDTITLE_TEXTVP`	`3`	gets title size from the *VDtext* display

VPdgdrcontext, VPdgdrdata *(dvstd.h)*

`V_BF_LATEST_N`	`0`	draw the recent n iterations
`V_BF_UNDISP`	`1`	draw the undisplayed data
`V_BF_DISP`	`2`	redraw the displayed data

VPvd_accmode, VGvd_accmode Flags *(dvstd.h)*

V_DIR_ACCESS	0
V_INDIR_ACCESS	1
V_DS_BOUND	3

VPvdsymbol, VGvdsymbol Flags *(dvmarker.h)*

V_NULL_SYMBOL	' '	default
V_ASTERISK	'*'	asterisk
V_DOT	'.'	dot
V_PLUS	'+'	plus
V_CROSS	'x'	x
v_DIAMOND	'd'	diamond
V_FILLED_DIAMOND	'D'	filled diamond
V_CIRCLE	'o'	circle
V_FILLED_CIRCLE	'O'	filled circle
V_BOX	'r'	box
V_FILLED_BOX	'R'	filled box
V_TRIANGLE	't'	triangle (apex up)
V_FILLED_TRIANGLE	'T'	filled triangle (apex up)
V_INVERTED_TRIANGLE	'v'	triangle (apex down)
V_FILLED_INVERTED_TRIANGLE	'V'	filled triangle (apex down)
V_TRIANGLE_RIGHT	')'	triangle (apex right)
V_FILLED_TRIANGLE_RIGHT	'>'	filled triangle (apex right)
V_TRIANGLE_LEFT	'('	triangle (apex left)
V_FILLED_TRIANGLE_LEFT	'<'	filled triangle (apex left)
V_VERTICAL_LINE	'|'	vertical line
V_HORIZONTAL_LINE	'-'	horizontal line

VUaxGet Flags *(dvaxis.h)*

AXIS_BOUNDS	26	RECTANGLE *
BASE_EXPONENT	36	int *
INITIAL_TICK_VALUE	27	double *
INITIAL_TICK_POINT	28	DV_POINT *
MAJOR_PIXEL_GAP	29	double *
MAJOR_VALUE_GAP	30	double *
MINOR_PIXEL_GAP	31	double *
MINOR_VALUE_GAP	32	double *
MINOR_TICKS_PER_MAJOR	33	int *
TICK_LABEL_EXTENT	38	DV_POINT *

VUaxSet Flags *(dvaxis.h)*

AXIS_COLOR	1	int
AXIS_DIRECTION	2	int
AXIS_IS_LOG	3	int
AXIS_LENGTH	4	int
AXIS_NEW_START_VALUE	5	double
AXIS_START_POINT	6	DV_POINT *
DRAW_GRID	7	int
DRAW_LABELS	8	int
DRAW_MINOR_TICKS	35	int
DRAW_TICKS	9	int
GRID_COLOR	11	int
GRID_EXCLUDE_ENDS	12	int
GRID_LENGTH	13	int
GRID_LINE_TYPE	14	int
GRID_SIDE	15	int
HIGHEST_VALUE	37	double
INTEGER_AXIS	34	int

```
LABEL_DISTANCE            16     int
LABEL_FORMAT_FUNCTION     41     ADDRESS, ADDRESS, int
LABEL_SIDE                18     int
LABEL_TEXTSIZE            19     int
MIN_MAJOR_PIXEL_GAP       20     double
MIN_MAJOR_VALUE_GAP       21     double
MIN_MINOR_PIXEL_GAP       22     double
MIN_MINOR_VALUE_GAP       23     double
TICK_LENGTH               24     int
TICK_SIDE                 25     int
```

VUaxSet Direction Flags *(dvaxis.h)*

```
AXIS_RIGHT                1
AXIS_UP                   2
AXIS_LEFT                 3
AXIS_DOWN                 4
LEFT_SIDE                 AXIS_LEFT
RIGHT_SIDE                AXIS_RIGHT
```

VUerBoundaryEventPost, VUerBoundaryEventDpPost Flags *(dvinteract.h)*

```
VUER_POS_EVENT            0
VUER_SE_EVENT             1
VUER_BRE_EVENT            2
VUER_DOE_EVENT            3
VUER_SRR_EVENT            4
VUER_OPOS_EVENT           5
```

VUer*Post InOut Flags *(dvinteract.h)*

```
V_OUTSIDE                 0
V_INSIDE                  1
```

VUerHandleLocEvent, VUerServiceResultPost Service Result Flags *(dvinteract.h)*

```
INPUT_ACCEPT      0x0001
INPUT_DONE        0x0002
INPUT_CANCEL      0x0004
INPUT_USED        0x0008
INPUT_UNUSED      0x0010
```

VUerHandler Termination Flags *(dvinteract.h)*

`ER_STOP_ON_ANY_EDGE`	`0x001`	any key press or release
`ER_STOP_ON_LEAD_EDGE`	`0x002`	reserved for future enhancements
`ER_STOP_ON_ANY_USE`	`0x008`	result `!=` *INPUT_UNUSED*
`ER_STOP_ON_UNUSED`	`0x010`	result == *INPUT_UNUSED*
`ER_STOP_ON_DONE`	`0x020`	result == *INPUT_DONE*
`ER_STOP_ON_ACCEPT`	`0x040`	result == *INPUT_ACCEPT*
`ER_STOP_ON_CANCEL`	`0x080`	result == *INPUT_CANCEL*
`ER_STOP_ON_USED`	`0x100`	result == *INPUT_USED*

VUerPutKeys, VUerGetKeys, VOitPutKeys, and VOitGetKeys Action Type Flags *(dvinteract.h)*

```
SELECT_KEYS          0
CANCEL_KEYS          1
DONE_KEYS            2
RESTORE_KEYS         3
```

```
CLEAR_KEYS               4
TOGGLE_POLLING_KEYS      8
```

VUerWinEventPost Flags *(dvinteract.h)*

```
VUER_RESIZE_EVENT        6
VUER_WINQUIT_EVENT       7
VUER_ICONIFY_EVENT       8
VUER_EXPOSE_EVENT        9
VUER_WIN_ENTER_EVENT     10
VUER_WIN_LEAVE_EVENT     11
```

VUtaCreate spec_flag Flags *(VUtextarray.h)*

Text array orientation flags:

```
V_OP_BITS                0x0F
V_OP_TOP                 0x01
V_OP_BOTTOM              0x02
V_OP_LEFT                0x04
V_OP_RIGHT               0x08
V_OP_LL                  (V_OP_BOTTOM|V_OP_LEFT)
V_OP_LR                  (V_OP_BOTTOM|V_OP_RIGHT)
V_OP_UL                  (V_OP_TOP|V_OP_LEFT)
V_OP_UR                  (V_OP_TOP|V_OP_RIGHT)
V_OP_CENTERED            0x00
```

Flags defining how to resolve size:

```
V_RSLVE_BITS       0x30
V_RSLVE_X_GREATER  0x10
V_RSLVE_Y_GREATER  0x20
V_RSLVE_X_LESSER   0x00
V_RSLVE_Y_LESSER   0x00
V_RSLVE_GREATER    (V_RSLVE_X_GREATER|V_RLVE_Y_GREATER)
V_RSLVE_LESSER     (V_RSLVE_X_LESSER|V_RLVE_Y_LESSER)
```

Flags defining what to do with slop:

```
V_SLOP_BITS        0x3C0
V_SLOP_X_SHRINK    0x040
V_SLOP_Y_SHRINK    0x080
V_SLOP_X_LEAVE     0x000
V_SLOP_Y_LEAVE     0x000
V_SLOP_X_EXPAND    0x100
V_SLOP_Y_EXPAND    0x200
V_SLOP_SHRINK      V_SLOP_X_SHRINK | V_SLOP_Y_SHRINK
V_SLOP_LEAVE       V_SLOP_X_LEAVE | V_SLOP_Y_LEAVE
V_SLOP_EXPAND      V_SLOP_X_EXPAND | V_SLOP_Y_EXPAND

V_TA_NUM_COLORS    16
V_TA_NORMAL        0x10
V_TA_INVERSE       0x01
```

Enums

V_FDS_FCN_ENUM *(dvfds.h)*

```
typedef enum
    {
    V_FDS_FCN_DS_START = 0,
    V_FDS_FCN_OPEN,
    V_FDS_FCN_READ,
    V_FDS_FCN_CLOSE,
    V_FDS_FCN_DS_CREATE,
    V_FDS_FCN_DS_DESTROY,
    V_FDS_FCN_DS_SAVE,
    V_FDS_FCN_DS_RESTORE,
    V_FDS_FCN_WRITE,

    /* flags for internal use */
    V_FDS_FCN_SELECT,
    V_FDS_FCN_DSV_CREATE,
    V_FDS_FCN_DSV_DESTROY
    V_FDS_FCN_SELECT_WRITE,

    /* flags for internal use */
    } V_FDS_FCN_ENUM;
```

V_IC_ATTR_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_IC_ATTR_ARGEND = 0,
    V_IC_HEIGHT,                height in screen coordinates
    V_IC_WIDTH,                 width in screen coordinates
    V_IC_PIXMAP,                icon is based on this pixmap
    V_IC_PIXMAP_XFORM,          how to transform pixmap colors to device's
    V_IC_MASK_PIXMAP,           pixmap used for icon mask
    V_IC_MASK_PIXMAP_XFORM,     transform mask colors to draw/no draw
    V_IC_RASTER                 actual raster used to draw icon
    } V_IC_ATTR_ENUM;
```

V_IM_ATTR_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_IM_ATTR_ARGEND = 0,
    V_IM_PIXMAP,                image is based on this pixmap
    V_IM_PIXMAP_XFORM,          how to transform pixmap colors to device's
    V_IM_MASK_PIXMAP,           pixmap used for image mask
    V_IM_MASK_PIXMAP_XFORM,     transform mask colors to draw/no draw
    V_IM_RASTER                 actual raster used to draw image
    } V_IM_ATTR_ENUM;
```

V_PM_ATTR_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_PM_ATTR_ARGEND = 0,
    V_PM_HEIGHT,              height of pixmap in pixels
    V_PM_WIDTH,               width of pixmap in pixels
    V_PM_DEPTH,               color depth
    V_PM_COLOR_TABLE,         colors used by pixmap
```

```
    V_PM_INCLUDE_PIXELS,      pixmap type is include or reference
    V_PM_FILENAME,            name of external file for referenced pixmaps
    V_PM_RAW_DATA,            data (and length) for included pixmaps
    V_PM_BOUNDS,              creating raster: portion of pixmap to use
    V_PM_COLOR_XFORM,         creating raster: color indices xform
    V_PM_VERSION,             number of changes since creation
    V_PM_PIXREP_DATA          pixrep used by the pixmap
    } V_PM_ATTR_ENUM;
```

See Also VOpmGet, VOpmSet, VOpmSetRasterMask, VOpmToRaster

V_PM_FLIP_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_PM_HORIZONTAL = 0,
    V_PM_VERTICAL
    } V_PM_FLIP_ENUM;
```

V_PM_FORMAT_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_PM_GIF,                 Graphics Interchange Format
    V_PM_PPM,                 Portable Pixmap
    V_PM_RASTER,              DataViews raster data
    V_PM_TIFF,                Tag Interchange File Format
    V_PM_PIXREP               DataViews device-independent format
    } V_PM_FORMAT_ENUM;
```

See Also VOpmWrite

V_PM_MERGEMODE_ENUM *(VOstd.h)*

```
typedef enum
    {
    V_PM_COPY,              source color index replaces target index
    V_PM_AND,               new target = source index AND old target
    V_PM_OR,                new target = source index OR old target
    V_PM_XOR                new target = source index XOR old target
    } V_PM_MERGEMODE_ENUM;
```

See Also VOpmMerge

V_PX_FLIP_ENUM *(VUpixrep.h)*

```
typedef enum
    {
    V_PX_HORIZONTAL,          flip around horizontal axis
    V_PX_VERTICAL             flip around vertical axis
    } V_PX_FLIP_ENUM;
```

V_PX_MERGEMODE_ENUM *(VUpixrep.h)*

```
typedef enum
    {
    V_PX_COPY,                copy source to target
    V_PX_AND,                 new target = source AND old target
    V_PX_OR,                  new target = source OR old target
    V_PX_XOR                  new target = source XOR old target
    } V_PX_MERGEMODE_ENUM;
```

V_UTA_AREA_ENUM *(VUtextarray.h)*

```
typedef enum
    {
    V_UTA_RECTANGLE=1,
    V_UTA_AREA
    } V_UTA_AREA_ENUM;
```

V_UTA_CURSOR_ENUM *(VUtextarray.h)*

```
typedef enum
    {
    NULL_ENUM=0,
    V_UTA_UNDERSCORE,
    V_UTA_REVERSE,
    V_UTA_COLOR
    } V_UTA_CURSOR_ENUM;
```

DataViews Private Types

These DataViews private types are defined in the following include files:

dvtools.h:

```
DRAWPORT              ADDRESS
VIEW                  ADDRESS
DATASOURCELIST        ADDRESS
DATASOURCE            ADDRESS
DSVAR                 ADDRESS
OBJECT                LONG
INHANDLER             ADDRESS
PROTO_ENV             ADDRESS
```

dvstd.h:

```
DATAGROUP             ADDRESS
DISPFORM              ADDRESS
SYMNODE               ADDRESS
SYMTABLE              ADDRESS
VARDESC               ADDRESS
```

dvinteract.h:

```
EVENT_REQUEST         ADDRESS
```

dvaxis.h:

```
AXISDESC              ADDRESS
```

hashtypes.h:

```
HASHNODE              struct
HASHTABLE             struct
```

VUpixrep.h:

```
PIXSCAN               struct
PIXPTR                union
```

VUtextarray.h:

```
TEXTARRAY             ADDRESS
```

DataViews Public Types

ANYTYPE typedef *(dvstd.h)*

```
typedef union ANYTYPE
    {
    char c;
    UBYTE uc;
    short s;
    unsigned short us;
    LONG l;
    ULONG ul;
    float f;
    double d;
    ADDRESS ptr;
    union ANYTYPE *ap;
    } ANYTYPE;
```

ATTRIBUTES typedef *(VOstd.h)*

```
typedef struct ATTRIBUTES
    {
    OBJECT foreground_color;  foreground color object
    OBJECT background_color;  background color object
    char line_width;          integer specifying width of line in pixels [1,3]
    char line_type;           integer specifying style of line
    char fill_status;         whether object is filled
    char text_direction;      direction of text (VERTICAL or HORIZONTAL)
    char text_position;       one of nine possible positions of text anchor point
    char text_font;           integer hardware text font (currently unused)
    char text_size;           integer specifying size for hardware text [1,4]
    char arc_direction;       arc drawing direction
    char curve_type;          polygon curve type for B-splines
    char *text_fontname;      filename of vector text font
    float text_width;         horizontal vector text expansion factor
    float text_height;        vertical vector text expansion factor
    float text_angle;         counter-clockwise angle of text baseline from horizontal
    float text_slant;         clockwise angle of text slant from vertical
    float text_charspace;     inter-character spacing fraction
    float text_linespace;     inter-line spacing fraction
    char *name;               node or edge object name
    RECTANGLE *node_bounds;   bounding box of node
    char edge_type;           type of edge
    char prop_fill;           proportional fill
    short fill_amount;        proportion of area to fill [0,32K]
    } ATTRIBUTES;
```

COLOR_SPEC typedef *(dvstd.h)*

```
typedef union COLOR_SPEC
    {
    LONG color_index;         should always be >= 0
    RGB_SPEC rgb_rep;
    } COLOR_SPEC;
```

COLOR_TABLE typedef *(dvstd.h*)

```
typedef struct
    {
    int ctsize;               size of color table
    RGB_SPEC ct[256];         array of RGB values
    } COLOR_TABLE;
```

COLOR_THRESHOLD typedef *(dvstd.h)*

```
typedef struct COLOR_THRESHOLD
    {
    short upperlimit;
    COLOR_SPEC threshcolor;
    } COLOR_THRESHOLD;
```

COLOR_XFORM typedef *(dvstd.h)*

```
typedef struct
    {
    int size;
    int new_index[256];
    } COLOR_XFORM;
```

DATUM typedef *(VOstd.h)*

```
typedef LONG DATUM;
```

DATUM_DESC typedef *(VOstd.h)*

```
typedef union
    {
    DATUM DATUM_alias;
    float f;
    LONG i;
    OBJECT O;
    char *t;
    } DATUM_DESC;
```

DATUM_TYPE typedef *(VOstd.h)*

```
typedef int DATUM_TYPE;
```

DRAWPORT_ATTRIBUTES typedef *(dvtools.h)*

```
typedef struct DRAWPORT_ATTRIBUTES
    {
    RECTANGLE *vvp;           where on the screen in virtual coordinates
    RECTANGLE *wvp;           portion of the view in world coords
    DV_BOOL stretch_flag;     TRUE: use TdpCreateStretch;  FALSE: use TdpCreate
    } DRAWPORT_ATTRIBUTES;
```

DV_COORD typedef *(dvstd.h)*

```
typedef LONG DV_COORD;
```

DV_POINT typedef *(dvstd.h)*

```
typedef struct DV_POINT
    {
    DV_COORD x;
    DV_COORD y;
    } DV_POINT;
```

FLOAT_POINT typedef *(dvstd.h)*

```
typedef struct FLOAT_POINT
    {
```

```
    float x, y;
    } FLOAT_POINT;
```

LABEL_SIZE typedef *(dvstd.h)*

```
typedef struct
    {
    int StringLength;          number of characters in the string
    short NumLines;            number of lines in the string
    short LongestLine;         number of characters in the longest line
    } LABEL_SIZE;
```

NAME_VALUE_PAIR typedef *(dvstd.h)*

```
typedef struct
    {
    char *name;
    char *value;
    } NAME_VALUE_PAIR;
```

PIXREP typedef *(dvstd.h)*

```
typedef struct
    {
    int width, height;           width and height of the pixrep in pixels
    UBYTE depth;                 number of bits of color information
    UBYTE bits_per_pixel;        1, 2, 4, 8, 16,or 32 bits
    UBYTE row_alignment;         If row_alignment is 8, rows are aligned on char; if 16, rows are aligned on
                                 short; if 32, rows are aligned on LONG.
    DV_BOOL origin_at_ll;        YES if origin is in lower left. Otherwise, NO.
    UBYTE pack_unit;             If fewer than 8 bits per pixel, packing unit. The packing unit is the 8-, 16-, or
                                 32-bit unit into which the data is packed.
    DV_BOOL pack_msf_in_byte;    If fewer than 8 bits per pixel, the order of pixels in the byte.
    DV_BOOL pack_msf_in_unit;    If fewer than 8 bits per pixel, the order of bytes in the unit.
    LONG pixels_length;          length of the pixel array
    UBYTE *pixels;               the array of pixels
    COLOR_TABLE *pclut;          If (pclut != NULL), pixels are indexed into color table.
    DV_BOOL *color_used;         An array of type DV_BOOL. Specifies which colors are used by the pixrep. If
                                 color_used[i] is TRUE, the corresponding color in the color table is used in the
                                 pixrep. If FALSE, the color isn't used. If color_used is NULL, assumes all
                                 colors are used. This field is optional, but can speed up some operations if used.
    ULONG red_mask;              information for finding the red
    int red_shift;               component  of the pixel
    ULONG grn_mask;              information for finding the green
    int grn_shift;               component  of the pixel
    ULONG blu_mask;              information for finding the blue
    int blu_shift;               component of the pixel
    } PIXREP;
```

PLR_POINT typedef *(dvstd.h)*

```
typedef struct PLR_POINT
    {
    short radius;
    short angle;
    } PLR_POINT;
```

RECTANGLE typedef *(dvstd.h)*

```
typedef struct
    {
    DV_POINT ll;             lower left corner of the rectangle
```

```
    DV_POINT ur;            upper right corner of the rectangle
    } RECTANGLE;
```

RGB_SPEC typedef *(dvstd.h)*

```
typedef struct _RGB_SPEC        for byte order 1234
    {
    char rgb_rep_flag;          should be -1 when used in COLOR_SPEC
    UBYTE red, green, blue;
    } RGB_SPEC;

typedef struct                  for byte order 4321
    UBYTE blue, green, red;
    char rgb_rep_flag;          should be -1 when used in COLOR_SPEC
    } RGB_SPEC;
```

RULE_ARG typedef *(dvrule.h)*

```
typedef LONG RULE_ARG;
```

TA_PACKED_COLOR typedef *(VUtextarray.h)*

```
typedef UBYTE TA_PACKED_COLOR;
```

TA_POSITION typedef *(VUtextarray.h)*

```
typedef struct TA_POSITION
    {
    short row, col;
    } TA_POSITION;
```

TA_RECT typedef *(VUtextarray.h)*

```
typedef struct TA_RECT
    {
    TA_POSITION ul, lr;
    } TA_RECT;
```

V_Q_PICK_VDP typedef *(dvstd.h)*

```
typedef struct V_Q_PICK_VDP
    {
    DV_POINT location;
    V_Q_VDP *vdp;
    } V_Q_PICK_VDP;
```

V_Q_VDP typedef *(dvstd.h)*

```
typedef struct V_Q_VDP
    {
    VARDESC vdp;
    int index;
    } V_Q_VDP;
```

V_Q_VDP_LIST typedef *(dvstd.h)*

```
typedef struct V_Q_VDP_LIST
    {
    int count;
    V_Q_VDP vdps[V_Q_PICKED_VDP_MAX];
```

```
} V_Q_VDP_LIST;

#define V_Q_PICKED_VDP_MAX 64
```

WINEVENT typedef *(dvGR.h)*

`typedef struct _winevent`	
`{`	
`int devnum;`	device number of window where event occurred
`ULONG type;`	*WINEVENT* type flag showing the type of event.
`ULONG time;`	server's recorded time of event in milliseconds
`LONG count;`	number of events in the event queue
`ADDRESS eventdata;`	copy of the window system's event data structure
`DV_POINT loc;`	location of cursor relative to window
`RECTANGLE region;`	exposed region
`DV_POINT maxpoint;`	new size of window
`ULONG state;`	*WINEVENT* state flag showing the state of the keyboard
`ULONG button;`	button code for mouse button events
`ULONG keycode;`	physical key code (device-dependent)
`ULONG keysym;`	virtual key symbol code, which are listed in the header files *GRkeysymdef.h* and *GRkeysym.h*
`char *keystring;`	key string
`LONG nchars;`	length of key string
`UBYTE firstchar;`	ASCII equivalent for the first character of key string
`RECTANGLE *rectlist;`	array of exposed regions in screen coordinates
`DV_POINT root_loc;`	location of cursor relative to the root window (not implemented for all drivers)
`} WINEVENT;`	

DataViews FUNPTR Types

ADDRFUNPTR typedef *(std.h)*

```
typedef ADDRESS (*ADDRFUNPTR) ();
```

BOOLFUNPTR typedef *(std.h)*

```
typedef BOOLPARAM (*BOOLFUNPTR) ();
```

CHARFUNPTR typedef *(std.h)*

```
typedef char (*CHARFUNPTR) ();
```

DV_TICLABELFUNPTR typedef *(dvtypes.h)*

```
typedef void (*DV_TICLABELFUNPTR) (
       ADDRESS argpcopy,
       double *value
       ADDRESS output
       TIC_DATA *tdp);
```

GRPALPICKFUNPTR typedef *(dvstd.h)*

```
typedef int (*GRPALPICKFUNPTR) (
       LONG fbcolor,
       RECTANGLE *echovp);
```

INTFUNPTR typedef *(std.h)*

```
typedef int (*INTFUNPTR) ();
```

LONGFUNPTR typedef *(std.h)*

```
typedef LONG (*LONGFUNPTR) ();
```

SHORTFUNPTR typedef *(std.h)*

```
typedef short (*SHORTFUNPTR) ();
```

TDLFOREACHDSFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDLFOREACHDSFUNPTR) (
       DATASOURCE ds
       ADDRESS argblock);
```

TDLFOREACHDSVFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDLFOREACHDSVFUNPTR) (
       DATASOURCE ds,
       DSVAR dsv
       ADDRESS argblock);
```

TDPTRAVERSEFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDPTRAVERSEFUNPTR) (
       DRAWPORT drawport,
       ADDRESS redraw_vp);
```

TDRFOREACHNAMEDOBJFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDRFOREACHNAMEDOBJFUNPTR) (
	OBJECT obj
	char *name,
	ADDRESS argblock);
```

TDSFOREACHVARFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDSFOREACHVARFUNPTR) (
	DSVAR dsv,
	ADDRESS argblock);
```

TDSFREEFUNPTR typedef *(dvtools.h)*

```
typedef void (*TDSFREEFUNPTR) (
	ADDRESS data);
```

TDSVFOREACHREFFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDSVFOREACHREFFUNPTR) (
	VARDESC vdp,
	int type
	ADDRESS argblock);
```

TDSVFOREACHVDPFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TDSVFOREACHVDPFUNPTR) (
	VARDESC vdp,
	ADDRESS argblock);
```

TDSVFREEFUNPTR typedef *(dvtools.h)*

```
typedef void (*TDSVFREEFUNPTR) (
	ADDRESS data);
```

TOBFOREACHSUBOBJFUNPTR typedef *(VOstd.h)*

```
typedef ADDRESS (*TOBFOREACHSUBOBJFUNPTR) (
	OBJECT subobj,
	ADDRESS argblock);
```

TOBFOREACHVDPFUNPTR typedef *(VOstd.h)*

```
typedef ADDRESS (*TOBFOREACHVDPFUNPTR) (
	OBJECT subobj,
	VARDESC vdp,
	ADDRESS argblock);
```

TVIFOREACHDSFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TVIFOREACHDSFUNPTR) (
	DATASOURCE ds,
	ADDRESS argblock);
```

TVIFOREACHVARFUNPTR typedef *(dvtools.h)*

```
typedef ADDRESS (*TVIFOREACHVARFUNPTR) (
	DATASOURCE ds,
	DSVAR dsv
	ADDRESS argblock);
```

ULONGFUNPTR typedef *(std.h)*

```
typedef ULONG (*ULONGFUNPTR) ();
```

VGADDRACCESSFUNPTR typedef *(dvstd.h)*

```
typedef ADDRESS (*VGADDRACCESSFUNPTR) (
       ADDRESS argp,
       int i3,
       int i2,
       int i1);
```

VGDOUBLEACCESSFUNPTR typedef *(dvstd.h)*

```
typedef double *(*VGDOUBLEACCESSFUNPTR) (
       ADDRESS argp,
       int i,
       int j,
       int k);
```

VGLONGACCESSFUNPTR typedef *(dvstd.h)*

```
typedef LONG (*VGLONGACCESSFUNPTR) (
       ADDRESS argp,
       int i,
       int j,
       int k);
```

VODQADDFUNPTR typedef *(VOstd.h)*

```
typedef OBJECT (*VODQADDFUNPTR) (
       OBJECT entity);
```

VODQCOMPAREFUNPTR typedef *(VOstd.h)*

```
typedef int (*VODQCOMPAREFUNPTR) (
       OBJECT entity1,
       OBJECT entity2);
```

VODQDELFUNPTR typedef *(VOstd.h)*

```
typedef void (*VODQDELFUNPTR) (
       OBJECT entity);
```

VODQEQUALFUNPTR typedef *(VOstd.h)*

```
typedef BOOLPARAM (*VODQEQUALFUNPTR) (
       OBJECT entity1,
       OBJECT entity2);
```

VODRNAMETRVRSFUNPTR typedef *(VOstd.h)*

```
typedef ADDRESS (*VODRNAMETRVRSFUNPTR) (
       int position,
       OBJECT object,
       char *name);
```

VOGDRAWFUNPTR typedef *(VOstd.h)*

```
typedef void (*VOGDRAWFUNPTR) (
       ADDRESS drawargs);
```

VOIDFUNPTR typedef *(std.h)*

```
typedef VOID (*VOIDFUNPTR) ();
```

VOITECHOFUNPTR typedef *(VOstd.h)*

```
typedef void (*VOITECHOFUNPTR) (
      OBJECT Input,
      int Origin,
      int State,
      double *Value,
      VARDESC Vdp,
      RECTANGLE *EchoVP,
      ADDRESS args);
```

VOOBTRAVERSEFUNPTR typedef *(VOstd.h)*

```
typedef BOOLPARAM (*VOOBTRAVERSEFUNPTR) (
      OBJECT subobj,
      ADDRESS testargs);
```

VPDGDFENTRYFUNPTR typedef *(dvstd.h)*

```
typedef int (*VPDGDFENTRYFUNPTR) ();
```

VTHTCOMPAREFUNPTR typedef *(dvstd.h)*

```
typedef int (*VTHTCOMPAREFUNPTR) (
      ADDRESS key1,
      ADDRESS key2);
```

VTHTCONVERTFUNPTR typedef *(dvstd.h)*

```
typedef ULONG (*VTHTCONVERTFUNPTR) (
      ADDRESS newkey);
```

VTHTFREEKEYFUNPTR typedef *(dvstd.h)*

```
typedef void (*VTHTFREEKEYFUNPTR) (
      ADDRESS key);
```

VTHTFREEVALFUNPTR typedef *(dvstd.h)*

```
typedef void (*VTHTFREEVALFUNPTR) (
      ADDRESS value);
```

VTHTTRAVERSEFUNPTR typedef *(dvstd.h)*

```
typedef void (*VTHTTRAVERSEFUNPTR) (
      ADDRESS key,
      ADDRESS value,
      ADDRESS args);
```

VTSTCOMPAREFUNPTR typedef *(dvstd.h)*

```
typedef int (*VTSTCOMPAREFUNPTR) (
      ADDRESS searchkey,
      ADDRESS key);
```

VTSTTRAVERSEFUNPTR typedef *(dvstd.h)*

```
typedef void (*VTSTTRAVERSEFUNPTR) (
       ADDRESS key,
       ADDRESS value,
       ADDRESS args);
```

VUDGTRVRSFUNPTR typedef *(dvtools.h)*

```
typedef void (*VUDGTRVRSFUNPTR) (
       DATAGROUP dgp);
```

VUSLTRVRSFUNPTR typedef *(dvstd.h)*

```
typedef int (*VUSLTRVRSFUNPTR) (
       char *string,
       int index,
       ADDRESS argblock);
```

VUVDTRVRSFUNPTR typedef *(dvtools.h)*

```
typedef void (*VUVDTRVRSFUNPTR) (
       VARDESC vdp);
```

Error Messages

Introduction

When an error occurs in a DV-Tools routine, a central error message processing routine is called that manages the formatting and display of error messages. If the optimized DV-Tools library is used, some error messages will be suppressed. A message is made up of four parts which are printed in the following format:

<<<<<Severity of error>>>>> Type of error

Routine name Additional explanation

Severity of error: There are two levels of severity for error messages that are issued by DV-Tools.

warning: A warning notifies you that an error occurred, but the error was not severe enough to prevent the entire system from continuing to function. A message is printed, but execution of the program continues.

severe: A severe error notifies you that an error occurred and that the system cannot continue execution. A message is printed and execution is halted.

Type of error: A message detailing the general nature of the problem. The possible messages are listed below, under the Generic Error Messages heading, along with some suggestions as to the kind of problem that may have caused the error.

Routine name: This is the name of the routine that detected the error. When opening a file, this will be the name of the file that the system was attempting to open.

Additional explanation: A specific message giving a more complete description of the particular error. These are listed below under the Specific Error Messages heading.

Error *Messages*

Generic Error Messages

Error	**Cause**
Can't open device.	The specified device cannot be opened. The device may be configured incorrectly.
Can't open the file.	The file does not exist, you do not have permission to access the file, or the pathname is mistyped.
Data group has been deleted already.	A subroutine has tried to delete a data group that has already been deleted.
Data type of variable in descriptor is unknown.	A variable type other than *float* has been specified.
Data table or array is full.	Internal tables are full.
Data value out of range.	You have specified too narrow a range for a variable.
DataViews internal coding error.	Internal error.
Display formatter cannot handle data group.	You have used a display format that is inappropriate; for example, using an inappropriate number or shape (dimension) of variables for the data type, or specifying than one time slot for a graph type that can more display only one time slot.
Display formatter not specified for data group.	You have attempted to run the system without specifying a display format for a particular data group.
Ill defined Input Object.	The input object cannot be used given its current context. For example, the object is not attached to a variable descriptor of the proper type.
Ill defined Template.	A template is not supplied when required, or the supplied template is not of the proper type.
Ill defined Input Technique.	The input technique is being used improperly. The specific error message should give further information.
Illegal argument in call to routine.	You have specified a bad file name or a structure that does not exist; generally this is a typing error.
IMS linking error.	Internal error.
Invalid action.	The action cannot be performed or completed. The specific error message should give further information.
Invalid data group structure.	Data group not defined correctly.
Invalid error code.	Internal error in the error handling mechanism.
Invalid variable descriptor structure.	Variable descriptor not defined correctly.
No such device exists.	You have specified a device name or number that is unknown to your system; this may be a typing error, the device has been detached or may not yet be connected.
NOT IMPLEMENTED	You have asked for a feature which is not yet operational.
Text cannot be split to fit in viewport.	Text in your display is too long or the viewport you specified is too small.
Text too tall to fit in viewport.	The viewport you specified for the display is too small.
Text too wide to fit in viewport	Text in your display is too long or the viewport you specified is too small.
Variable descriptor has been deleted already.	You have attempted to delete a variable descriptor that has already been deleted.
Viewport too small for display formatter.	The viewport is too small for the format you want to use.

Specific Error Messages

Error	Cause	Routines
Access function not saved.	Tried to save a view that had an access function associated with a variable descriptor.	TviSave
Axis needs to have its length specified.	Didn't specify axis length before drawing.	VUaxSetupForDrawing
Bad format name.	Format argument must be one of *COLOR_COMPONENT*, *COLOR_INDEX*, *COLOR_NAME*, *COLOR_REFERENCE*, or *COLOR_STRUCTURE*.	VOcoCreate
Bad pointer to function.	Passed in the address of something that wasn't a function.	VPdgdfentry
Boundary Points.	Either the lower left or upper right points are invalid point objects.	VOdgCreate
Can't change attribute after axis has been drawn.	Tried to change an attribute of an axis after it had been drawn.	VUaxSet
Can't delete last two points.	In order for an object to be a polygon it must have at least two points.	VOpyPtDelete
Can't fit time display.	Not enough room allocated for the graph.	VDclock
Can't fit value labels.	Not enough room allocated for the graph.	VDdial360, VDdial, VDdigits, VDface, VDhistdial, VDknob, VDrects
Can't use object lists.	*VNtoggle* is not implemented for lists of objects.	VNtoggle
Cannot embed Combiner or Multiplexor in a Combiner.	To prevent recursive use of composite input objects, combiner and multiplexor input objects may not be used as components.	VNcombine
Cannot embed Combiner or Multiplexor in a Multiplexor.	To prevent recursive use of composite input objects, combiner and multiplexor input objects may not be used as components of a multiplexor.	VNmulti
Color index too big; used low 16-bits.	Tried to create a color object by specifying color index and the color index was too large.	VOcoCreate
Control Point index out of range.	Specified the index of a non-existent point.	VOpyPtAdd, VOpyPtDelete, VOobPtGet, VOobPtSet
Couldn't find data source variable corresponding to graph variable.	Couldn't match up data source variables in the view with the variables attached to the data group.	VDdrawing
Couldn't fit tic labels in context.	Not enough room allocated for the graph.	VDfader
Couldn't fit title in context.	Not enough room allocated for the graph.	VDfan
Couldn't set string variable. Incompatible destination.	Variable pointed to by variable descriptor was not of type text.	VPvdSValue
Display Format not linked.	Display formatter referred to in	TInit

	dispforms.stb has not been linked into the DV-Tools application.	
DRAWING already contains the object.	Tried to add an object to a drawing already containing that object.	VOdrObAdd
Drawing doesn't contain the object.	Tried to refer to an object that doesn't exist in the drawing.	VOdrAddName, VOdrObBottom, VOdrObDelete, VOdrObErase, VOdrObReplace, VOdrObTop
Drawing has component that references itself.	Drawing could not be loaded because a subdrawing in the drawing refers to itself or a parent drawing.	VOuDrRetrieve
Event Posted with no bounding rect. and no keys.	Not enough information specified when posting an event.	VUerRectEdgePost
First argument must be a data source variable or a variable descriptor.	First argument wasn't of the appropriate type.	VOvdCreate
First argument must be DRAWING or a DEQUE.	First argument was not a drawing or deque.	VOuGetInList
Function not valid for object.	Function does not apply to an object of that type.	all VOob routines
Illegal access mode; no change made.	Tried to define an access mode for the variable that was invalid.	VPvd_accmode
Illegal logical device code specified.	Specified logical device number for an unopened or non-existent device.	VUgetdevnum, VUindextorgb, VUrgbtoindex
Index out of range.	The index argument specifying which control point was to be set indicates a non-existent point. For deque objects, indicates a non-existent deque entry.	VOobPtSet, VOdqGetEntry
Input file does not contain a valid DATASOURCE.	Tried to load a file that was not the result of a successful call to *TdsSave*.	TdsLoad
Input file does not contain a valid DATASOURCELIST.	Tried to load a file that was not the result of a successful call to *TdlSave*.	TdlLoad
Input file does not contain a valid DSVAR.	Tried to load a file that was not the result of a successful call to *TdsSave*.	TdsvLoadList
Input file does not contain a valid VIEW.	Tried to load a file that doesn't contain a view.	TviFileLoad, TviLoad
Interaction needs a number, not a text string.	The variable descriptor should not be of type text string.	VNpalette
Interaction needs a text string.	The variable descriptor should be of type text string.	VNtext
Invalid control point.	Tried to set control point to something that is not a point object.	VOobPtSet
Logical device not open.	Specified logical device number for an unopened or non-existent device.	VUgetdevnum
Logical device table full.	Ran out of space for storing device information.	VUopendev_clut
Multiple windows not available on this device.	Non-*NULL* window id used.	TscOpenWindow
Must call VUaxSetupForDrawing	Tried to get axis bounds before calling *VUaxSetupForDrawing*.	VUaxGet

before getting axis bounds.		
Must have at least two variables.	Only one variable is associated with the data group.	VDimpulse, VDscatter, VDweb
Must specify drawing or filename.	When creating a subdrawing you must specify a drawing or a filename containing a drawing.	VOsdCreate
No current SCREEN.	There is no currently active screen for display.	VOscClose, VOscDraw, VOscLocate, VOscPoll, VOscRedraw, VOscReset, VOscClosePoll, VOscLoSet, VOscOpenPoll
No drawing specified. The graph's title should be the drawing filename.	The title field of the data group did not contain the name of a valid viewfile.	VDdrawing
No more available windows.	Limited number of windows available on most devices.	TscOpenWindow
No room for clock hands.	Not enough room allocated for the graph.	VDanclock
No room for knob needle.	Not enough room allocated for the graph.	VDknob
No room for needle.	Not enough room allocated for the graph.	VDmeter
Non-NULL access function.	Tried to load a view that had an access function associated.	TviLoad
Not a valid DATASOURCE VARIABLE.	The data source variable parameter passed to this routine was invalid.	TdsAddDsVar, TdsDeleteDsVar
Not a valid DATASOURCE.	The data source parameter passed to this routine was invalid.	TdlAddDataSource, TdlDeleteDataSource, TdsAddDsVar, TdsClone, TdsCloseData, TdsDeleteDsVar, TdsDestroy, TdsEditAttributes, TdsForEachVar, TdsGetAttributes, TdsGetName, TdsMoveDataSource, TdsOpenData, TdsReadData, TdsSave
Not a valid DATASOURCELIST.	The data source list variable parameter passed to this routine was invalid.	TdlAddDataSource, TdlClone, TdlCloseData, TdlDeleteDataSource, TdlDestroy, TdlForEachDataSource, TdlForEachVar, TdlOpenData, TdlReadData, TdlSave, TviMergeAddDataSources, TviMergeDataSources, TviPutDataSourceList

Not a valid DRAWPORT.	The drawport parameter passed to this routine was invalid.	TdpBack, TdpDestroy, TdpDraw, TdpDrawNext, TdpDrawNextObject, TdpDrawObject, TdpErase, TdpEraseObject, TdpFront, TdpGetDrawingVp, TdpGetScale, TdpGetScreen, TdpGetScreenVp, TdpGetXform, TdpIsDrawn, TdpPan, TdpRedraw, TdpZoom
Not a valid DSVAR.	The data source variable passed to this routine was invalid.	TdsvAttachVariableDescriptor, TdsvClone, TdsvDestroy, TdsvDetachVariableDescriptor, TdsvEditAttributes, TdsvGetAttributes, TvdPutDataSourceVariable
Not a valid object.	Tried to replace an object in a drawing with something that was an invalid object.	VOdrObReplace
Not a valid VARDESC.	The variable descriptor parameter passed to this routine was invalid.	TvdPutDataSourceVariable
Not a valid VIEW.	The view parameter passed to this routine was invalid.	TdpCreate, TdpCreateStretch, TviASCIISave, TviDestroy, TviSave, TviExciseDrawing, TviFileLoad,TviFileSave, TviGetDataSourceList, TviGetDrawing, TviMergeAddDataSources, TviMergeDataSources, TviMergeDrawing, TviPutDataSourceList, TviPutDrawing
Not a valid xx object.	The function was called with an object that was not of the appropriate type.	TloSetup VOobAtGet VOobAtSet VOobBox VOobClone VOobDereference VOobDraw VOobErase VOobIntersect VOobPtGet VOobPtSet VOobReference VOobTraverse VOcoNdxGet VOcoRgbGet VOdqAdd VOdqDelete VOdqSize VOdqGetEntry VOdqHasEntry VOdqReplaceEntry VOdgAddress VOdgReset VOdgUpdate VOdrBackcolor VOdrForecolor VOdrAddName VOdrDeleteName VOdrGetName VOdrNameTraverse VOdrObAdd VOdrObBottom VOdrObDelete VOdrObErase VOdrObReplace VOdrObTop VOdrGetNamedObject VOinTechnique VOinUpdate VOinGetFlag VOinPutFlag VOinGetVarList VOinPutVarList

		VOitGetInteraction VOitGetKeys
		VOitGetTemplate VOitGetList
		VOitKeyOrigin VOitListStart VOitPutKeys
		VOitPutList VOitGetEchoFunction
		VOitGetListValues VOitPutEchoFunction
		VOitPutListValues VOloKey VOloScpGet
		VOloWcpGet VOptMove VOpyPtAdd
		VOpyPtDelete VOscSelect
		VOscDeviceName VOsdRotate VOsdScale
		VOsdDrGet
		VOsdDrKeep VOsdDrSet VOsdFilename
		VOtxGetString VOtxSetString
		VOttReset
		VOttScale VOttSize VOttUpdate
		VOttVd VOttAddThresh VOttDelThresh
		VOttGetThresh VOttLastGet VOttTypeGet
		VOvdAddress VOvdChanged
		VOvdReset
		VOvdSwitch VOvdType VOvdDvGet
		VOvdSvGet VOvdSvPut VOvtGetBound
		VOvtGetString VOvtSetString
		VOxfPoint
		VOxfScale VOxfCatCreate VOxfDpPoint
		VOxfInvCreate VOxfMatGet
		VOxfMatCreate VOxfStCreate
Not drawing value labels.	Not enough room allocated for the graph.	VDpie
Not enough room for dial needle.	Not enough room allocated for the graph.	VDdial360
Not enough toggle items.	There must be at least two items in a toggle.	VNtoggle
Null or missing list of input objects.	No input objects specified for the interaction handler. Combiners and multiplexors require a list of input objects.	VNcombine, VNmulti
Number of values must equal number of options.	There must be a one to one correspondence between the list of values and the list of options.	VNtoggle
Number of variable descriptors does not match number of embedded objects.	There isn't a one to one correspondence between the variable descriptors and the list of input objects.	VNcombine, VNmulti
Object not in DEQUE.	Tried to delete an object that wasn't in the list.	VOdqDelete
Out of xx heap space.	There is no more room to store objects of that type. Destroy unneeded objects to make room.	VOobCreate, VOscOpen, VOscOpenClut
Physical device not open.	Specified physical device number for an unopened or non-existent device.	VUgetdevindex
Pie would be too small.	Not enough room allocated for the graph.	VDpie
Pixel space coords require a reference point.	When creating a point with pixel space offset you must specify a reference point.	VPptCreate
Polling not opened.	In order to call *VOscPoll* you must call	VOscPoll

	VOscOpenPoll first.	
Reference Point.	The reference point specified was invalid.	VOptCreate
Scale out of range.	Tried to create a transform object with scale factor that was either too large or too small.	VOxfStCreate
SCREEN still has attached viewports.	Destroy all drawports before closing screen.	VOscClose
Stroke font file does not exist.	Stroke file cannot be found.	VOobAtSet, VOvtCreate
The Filled Lines chart must have a samples count greater than one.	The slot count is one.	VDline
The strip chart must have a samples count greater than one.	The slot count is one.	VDstrip
Threshold out of range.	Tried to add a threshold that was outside of the range of thresholds.	VOttAddThresh
THRESHOLD TABLE index out of range.	Tried to refer to a threshold index that did not exist.	VOttGetThresh
Type should be 'c', 'n', or 't'.	Flag type was invalid.	VOvdCreate
Unknown attribute flag.	Specified an invalid attribute flag.	VUaxGet, VUaxSet
Unknown axis type.	Axis flag must be *TIME_AXIS*, *FIRST_AXIS*, or *SECOND_AXIS*.	VGdgaxlabel, VGdgticlabfcn, VPdgaxlabel, VPdgticlabfcn
Unknown color name.	Specified an unknown color name when trying to create a color.	VOcoCreate
Unknown data type.	Variable descriptor referred to an unknown data type.	VPvdValue, VPvdDValue, VPvdIValue, VPvdSValue
Unknown data type; assumed to LONG integer.	Tried to specify a data type flag that was invalid.	VPvdCreate
Unknown Key Origin.	Key origin flag not in set of valid choices.	VOitKeyOrigin
Unknown Key Type.	Key type flag is not in the set of valid choices. See *dvinteract.h*.	VOitGetKeys, VOitKeyOrigin, VOitPutKeys
Unknown object type.	Object argument can't be recognized by the program.	VOobPtGet, VOobPtSet
Variable descriptor already part of a data group.	Tried to add a variable descriptor to a data group that already contained a variable descriptor.	VPdgvdadd, VPdgvinsert
Variable unreadable.	Could not read the data source file.	TdlRead, TdsRead
Viewport not changed.	Viewport had values that were out of range.	VPdgvp