GE Fanuc DataViews
Headquarters
47 Pleasant Street
Northampton, MA 01060
U.S.A.
Telephone:(413) 586-4144
FAX:(413) 586-3805
e-mail:info@dvcorp.com
web:www.dvcorp.com

GE Fanuc DataViews
U.K. Sales Office
Unit 1, Mill Square
Featherstone Road
Wolverton Mill South
Milton Keynes, MK12 5BZ
United Kingdom
Telephone:+44 1243 527953
FAX:+44 1243 527253
e-mail:duncan.stewart@gefanuceur.ge.com
web:www.dvcorp.com

GE Fanuc DataViews
Japan Sales Office
16 Parale Mitsui Bldg
8 Higashida-cho
Kawasaki-ku, Kawasaki
Kanagawa, 210-0005, Japan.
Telephone:+81 44 221 9462
FAX:+81 44 221 9463
e-mail:dv-japan@dvcorp.com
web:www.dvcorp.com

Copyrights and Trademarks
Copyright © 2000 GE Fanuc DataViews
All Rights Reserved. This manual is subject to copyright protection.
No portion of the contents of this book may be reproduced in any form or by any means without prior
written consent from GE Fanuc DataViews.

DataViews, DV-Tools, and DV-Draw are registered trademarks of GE Fanuc DataViews.
DVX-Designer is a trademark of GE Fanuc DataViews.
All other trademarks are acknowledged as the property of their respective owners.

July 1995
Software Release Version 9.7
Document Number 820-00018-00
Document Revision B
Table of Contents

Chapter 1 | Before You Begin

Introduction ................................................................. 1-1
The Purpose of This Manual ......................................... 1-1
Learning Path ................................................................. 1-2
Setting the Environment ................................................ 1-3
DataViews Directory Structure ...................................... 1-3
How this User’s Guide is Organized ............................... 1-6
Conventions Used in this Manual ................................. 1-7

Chapter 2 | Concepts

Overview ................................................................. 2-1
The Role of the User Interface ....................................... 2-1
Structuring the Application and Interface ...................... 2-2
Planning the User Interface ......................................... 2-4
Implementing the User Interface ................................... 2-5
The Application Model ................................................. 2-8

Chapter 3 | Structural Overview

Overview ................................................................. 3-1
Migrating from DV-Draw to DV-Tools ............................. 3-1
DV-Tools Data Structure Types ...................................... 3-10
The Organization of Routines ........................................ 3-11
A DV-Tools Program Skeleton ...................................... 3-14

DV-Tools User’s Guide
Table of Contents

Chapter 4 | Display
Overview ................................................................. 4-1
Basic Function Calls ................................................. 4-2
Coordinate Systems .................................................. 4-3
Low-Level Graphics .................................................. 4-5
Screens ................................................................. 4-5
Drawports ................................................................. 4-8
Displaying Views ....................................................... 4-16
GRconfig: Controlling the Number of Devices .............. 4-19
Summary ................................................................. 4-21

Chapter 5 | Objects
Overview ................................................................. 5-1
Basic Function Calls ................................................. 5-2
Manipulating Objects ............................................... 5-3
Attributes Table ........................................................ 5-8
Common Attributes .................................................. 5-8
Graphical Objects ..................................................... 5-9
Subobjects of Graphical Objects ................................. 5-21
Non-Graphical Objects .............................................. 5-26
Summary ................................................................. 5-36

Chapter 6 | Dynamics
Overview ................................................................. 6-1
Basic Function Calls ................................................. 6-4
Data Sources ........................................................... 6-5
Variable Descriptors .................................................. 6-8
Graphs ................................................................. 6-10
Object Dynamics ....................................................... 6-22
The Internal Dynamics of Subdrawings ....................... 6-28
Blinking ................................................................. 6-33
Table of Contents

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Input Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>7-1</td>
</tr>
<tr>
<td>Basic Function Calls</td>
<td>7-3</td>
</tr>
<tr>
<td>Input Objects</td>
<td>7-5</td>
</tr>
<tr>
<td>Input Technique Object</td>
<td>7-6</td>
</tr>
<tr>
<td>Interaction Handlers</td>
<td>7-7</td>
</tr>
<tr>
<td>Input Handling</td>
<td>7-9</td>
</tr>
<tr>
<td>Making Input Objects Do Useful Work</td>
<td>7-10</td>
</tr>
<tr>
<td>Complex Interaction Handlers</td>
<td>7-10</td>
</tr>
<tr>
<td>Interaction Summary</td>
<td>7-12</td>
</tr>
<tr>
<td>Summary</td>
<td>7-13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>Event Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>8-1</td>
</tr>
<tr>
<td>Basic Function Calls</td>
<td>8-4</td>
</tr>
<tr>
<td>Polling for Events</td>
<td>8-5</td>
</tr>
<tr>
<td>Types of Events</td>
<td>8-8</td>
</tr>
<tr>
<td>The Event Handling Loop</td>
<td>8-9</td>
</tr>
<tr>
<td>Posting Event Requests</td>
<td>8-11</td>
</tr>
<tr>
<td>Managing Requests</td>
<td>8-19</td>
</tr>
<tr>
<td>Service Routines</td>
<td>8-20</td>
</tr>
<tr>
<td>How Input Objects Use the Event Handler</td>
<td>8-26</td>
</tr>
<tr>
<td>Posting Service Result Requests</td>
<td>8-28</td>
</tr>
<tr>
<td>Echo Functions</td>
<td>8-31</td>
</tr>
<tr>
<td>Handling Window Expose and Resize</td>
<td>8-34</td>
</tr>
<tr>
<td>Explicit Event Handling</td>
<td>8-35</td>
</tr>
<tr>
<td>Handling Device-Specific Window Events</td>
<td>8-37</td>
</tr>
<tr>
<td>Setting Up Location Objects</td>
<td>8-38</td>
</tr>
<tr>
<td>Building the Control Loop</td>
<td>8-40</td>
</tr>
<tr>
<td>Summary</td>
<td>8-42</td>
</tr>
</tbody>
</table>

Master Data Source Lists .............................................................. 6-36
Dynamic Data Flow Example .......................................................... 6-38
Summary ................................................................. 6-39

DV-Tools User’s Guide

FANUC CNC Parts
# Table of Contents

## Chapter 9  Coding Environment

- Overview ................................................................. 9-1
- Include Files ............................................................ 9-1
- Compiling and Linking under UNIX .......................... 9-2
- Compiling and Linking under OpenVMS .................... 9-4
- Linking with the DataViews Shareable Image for OpenVMS 9-6
- Required DataViews Object Files ............................... 9-7
- Coding Tips ............................................................. 9-8
- Debugging Tips ......................................................... 9-9
- Linking with the Optimized Library ............................ 9-11

## Chapter 10  Tips for Improving DataViews Performance

- Introduction ............................................................. 10-1
- Configuration Variables Affecting Speed .................... 10-2
- General Optimizations ............................................. 10-2
- Optimizing Load Speed ........................................... 10-4
- Optimizing Draw Speed .......................................... 10-4
- Optimizing Event Handling Speed ............................ 10-7
- Reducing the Size of Your Executable ....................... 10-8
- Suggestions for OpenVMS ....................................... 10-9
- Suggestions for OPEN LOOK ..................................... 10-11

## Glossary

Glossary-1
# Chapter 1

## Before You Begin

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1-1</td>
</tr>
<tr>
<td>What Comes with DV-Tools</td>
<td>1-1</td>
</tr>
<tr>
<td>The Purpose of This Manual</td>
<td>1-1</td>
</tr>
<tr>
<td>Intended Audience</td>
<td>1-2</td>
</tr>
<tr>
<td>Learning Path</td>
<td>1-2</td>
</tr>
<tr>
<td>Setting the Environment</td>
<td>1-3</td>
</tr>
<tr>
<td>DataView Directory Structure</td>
<td>1-3</td>
</tr>
<tr>
<td>Directory Descriptions</td>
<td>1-5</td>
</tr>
<tr>
<td>How this User’s Guide is Organized</td>
<td>1-6</td>
</tr>
<tr>
<td>DV-Tools User’s Guide Chapter Summary</td>
<td>1-6</td>
</tr>
<tr>
<td>Conventions Used in this Manual</td>
<td>1-7</td>
</tr>
</tbody>
</table>
This chapter discusses the following topics:

- The Purpose of this Manual
- Learning Path for DV-Tools
- DataViews Directory Structure
- Organization of this Manual
- Conventions Used in this Manual

**Introduction**

DV-Tools is a library of graphics routines, written in C, that works in conjunction with the DV-Draw editor. The routines range from low-level drawing functions to utilities that manipulate complex dynamic views. With DV-Tools subroutines, you can integrate DV-Draw views into powerful graphical user interfaces for a wide range of applications.

**What Comes with DV-Tools**

Your copy of DV-Tools includes a distribution tape and a standard set of manuals.

The distribution tape contains a script for running DV-Draw, the DV-Tools routine library, source code for the demonstration and example programs, as well as READ_ME files and link scripts. It also contains scripts for installing, validating, and authorizing your copy of the software. The DV-Tools directory structure is described at the end of this chapter.

**The Purpose of This Manual**

The *DV-Tools User’s Guide* introduces the concepts and structures of DV-Tools. It explains the data structures, how to use the routines to manipulate these data structures, and how to build an interface for your application. It also discusses the use of DV-Draw together with DV-Tools to achieve maximum power, efficiency, and flexibility.
**Intended Audience**

DV-Tools is intended for experienced programmers with a knowledge of the C language. Familiarity with the concepts of DV-Draw is necessary since DV-Tools routines manipulate graphical objects and data structures created in DV-Draw. Knowledge of general programming tools, such as a debugger and the program analyzer, *Lint*, is also helpful.

---

**Learning Path**

Reading and understanding the *DV-Tools User’s Guide* is an important step in learning the DataViews products. We recommend the following path to learn DV-Tools:

**Step 1.** Run the demos.

Notice how information is organized and displayed and how users can interact with the interface. Return to the demos at any point in the learning path to review their features.

**Step 2.** Use DV-Draw.

Become familiar with DV-Draw features and terms by using DV-Draw and reading the *DV-Draw User’s Guide* and the DV-Draw tutorial in the *DataViews Tutorial Guide*.

**Step 3.** Read the DV-Tools tutorial in the *DataViews Tutorial Guide*.

Learn the basic concepts and methods of DV-Tools programming by stepping through six example programs.

**Step 4.** Read the *DV-Tools User’s Guide*.

Learn the concepts and structures required to build an interface using DV-Tools and DV-Draw.

**Step 5.** Run example programs.

Learn to code specific features. Using the provided source, you can examine the context of working programs. Step through the examples using a debugger. The DV-Tools routine library contains debugging routines for examining DataViews objects.
Additional help is available from training courses, technical support, and other documents, particularly the *DV-Tools Reference Manual* and *DV-Tools Quick Reference Manual*. Consulting services and the *Technical Notes* are also available to assist with sophisticated uses of our products.

**Setting the Environment**

DataViews lets you set many environment variables that affect the execution of your DV-Tools programs. The most common are the search path and the display device. You can specify the directories to be searched by DataViews and their order of precedence. You can specify information about the display device on which your application will run. You can also customize the display with information such as dimensions or the color palette to be used.

DataViews lets you set your environment in three different ways. You can use a configuration file, operating system commands, or command line options. Use multiple configuration files or a combination of these methods to achieve a high level of flexibility and control. For more information about the environment options and their use see the *Setting the DataViews Environment* appendix in the *DV-Draw User’s Guide*.

**DataViews Directory Structure**

The following outline shows the DataViews directory structure. Your DataViews directory is referred to as the `dataviews` directory. Logical names used in OpenVMS are shown after the corresponding UNIX names.

Most directories have `READ_ME` files that contain information about the directory and its contents. The information in the `READ_ME` files is more current than in the printed manuals.
Note that only the major directories are shown; individual files are not listed.

```
dataviews
  DVtools
  demos
  widget_demos
  examples
DVdraw
  demos
DVproto
DVdemo
DVdrivers
user
  include (DV$INCLUDE)
bin (DV$BIN)
etc (DV$ETC)
    SHARE (OpenVMS only)
lib (DV$LIB)
  fonts (DV$FONTS)
  images (DV$IMAGES)
  icons (DV$ICONS)
  templates (DV$TEMPLATES)
    drawings (DV$TEMPLATES:[DRAWINGS])
lint (not a directory in OpenVMS)
views (DV$VIEWS)
  isa (DV$ISA)
clip_art
src
  names
  tables
tooldebug
unsupported
```
### Directory Descriptions

The contents of the DataViews directories and significant files are summarized below:

**bin**  
DataViews programs. Scripts, including `DVdraw`, `DVplay`, `DVproto`, `DVlink`, `DVlint`, `RunDemos`, `ProtoDemo` (if you purchase a remote DV-Tools license), `DVwhich` (locates files in the DataViews environment), and `DVversion` (displays the DataViews version number).  
On OpenVMS, `DVADDTOPATH.COM`, which adds new directories to the `DV$PATH` definition.

**etc**  
Administrative files and programs, including `Validate`, `Authorize`, and `license.dv`.  
The default configuration file, `.DVconfig` (UNIX) and `DVCONFIG.DAT` (OpenVMS).  
The files `ih.stb` and `dispforms.stb` specify which interaction handlers and display formaters to load.  
The `Makefile` for compiling DataViews executables. You can use this `Makefile` as a template for creating a makefile specific to your application.  
The `Plotqueue` script. This script is executed when you select “Plot” in DV-Draw. For information on editing `Plotqueue` for use with your printer or plotter, see the DataViews Installation and System Administration Manual.  
The DataViews standard color table files: `default.clut`, `pastel.clut`, `grey16.clut`, and `grey128.clut`.  
The `default.dat` and `default.prc` files. These are default data sources.  
On OpenVMS, the DCL command files for defining logical names.  
All files related to the license manager.

**lib**  
The DataViews libraries.  
Object files used when linking your DataViews application programs to the libraries.  
Views for the DataViews logos, icons, and input object templates.  
Views from the Instrumentation Society of America (ISA) drawing library.  
Vector text fonts.  
Sample pixmap files in GIF format for creating icons and images.  
UNIX lint libraries.  
On OpenVMS, the shareable image, `LIBDVTSHR.EXE`, and the link options file.
How this User's Guide is Organized

This manual teaches the concepts and implementation of a DataViews interface. The discussion of concepts covers many topics, including the roles of the interface, integration of the interface with your application, and a model for using DV-Draw and DV-Tools. Implementation includes an overview of the data structures and routines used in DataViews. Subsequent chapters focus on specific areas of functionality in order of increasing complexity followed by a discussion of the DV-Tools programming environment.

The chapters cover the following material:

Concepts - describes the role of the interface within the application and the model for using DV-Tools and DV-Draw effectively to create an interface.
Structural Overview - describes the DV-Tools data structures, the organization of the routines, and presents a DV-Tools skeleton program.

Display - describes how views are manipulated and displayed on the physical screen.

Objects - describes the structure and behavior of each of the graphical and non-graphical objects.

Dynamics - describes how data from sources in the application can control changes in the graphical objects.

Input Objects - describes the structure and behavior of input objects.

Event Handling - describes how to gather input from the user and act on it.

Coding Environment - describes coding and debugging procedures.

Tips for Improving DataViews Performance - describes many techniques for increasing the speed and reducing the size of your executable.

Glossary - lists common DV-Tools terms.

Before You Begin Conventions Used in this Manual

Code in the text.

All routine names, parameter names, filenames, and script names are in Times Italic font:

To open multiple screens, call TscOpenSet repeatedly using the same generic device name, such as \textit{X}.

Code fragments.

Code fragments are in Courier font. Comments are in Times font.

/* Set up the master dsl using the dsl from the top view */
masterdsl = TviGetDataSourceList (topview);
TviMergeAddDataSources (view1, masterdsl, DS_EXACTMATCH);

Titles.

References to specific sections of the manual and titles of other manuals are in italic font:

Structural Overview

Place holders.

Instructions to enter information may contain *Times italicized* text that you should replace with your own information. For example:

```
DVplay -d your_device -v your_viewfile
```

indicates that you should enter the script `DVplay -d -v`, replacing `your_device` and `your_viewfile` with your specific information.

UNIX and OpenVMS.

Differences between UNIX and OpenVMS requirements are noted where necessary. Files, directories, and scripts are usually shown in UNIX format.
### Chapter 2 Concepts

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>2-1</td>
</tr>
<tr>
<td>The Role of the User Interface</td>
<td>2-1</td>
</tr>
<tr>
<td>Structuring the Application and Interface</td>
<td>2-2</td>
</tr>
<tr>
<td>Advantages to Separating Interface and Application</td>
<td>2-3</td>
</tr>
<tr>
<td>Structuring the Program</td>
<td>2-3</td>
</tr>
<tr>
<td>Planning the User Interface</td>
<td>2-4</td>
</tr>
<tr>
<td>Implementing the User Interface</td>
<td>2-5</td>
</tr>
<tr>
<td>Implementation Path</td>
<td>2-5</td>
</tr>
<tr>
<td>The Role of DV-Draw in Building Your User Interface</td>
<td>2-5</td>
</tr>
<tr>
<td>The Role of DV-Tools in Building Your User Interface</td>
<td>2-7</td>
</tr>
<tr>
<td>The Application Model</td>
<td>2-8</td>
</tr>
<tr>
<td>Model Program Description</td>
<td>2-9</td>
</tr>
<tr>
<td>Control Flow in the Model Program</td>
<td>2-10</td>
</tr>
</tbody>
</table>
This chapter describes the following topics:

- The Role of the User Interface in Your Application
- How to Use DataViews within your Application
- Planning a User Interface
- Efficient Integration of DV-Tools and DV-Draw
- A Conceptual Application

Overview

DataViews is a set of tools that lets you build graphical user interfaces for complex software applications. DataViews consists of DV-Draw, the graphical editor for building the visual interface, and DV-Tools, the library of routines for building the programmatic control of the interface.

This chapter presents a conceptual framework for implementing an application interface using DV-Tools and DV-Draw. Details of the implementation process are discussed in subsequent chapters.

DataViews is a flexible product that you can use in many ways. The model presented in this chapter may not apply directly to your application, but the concepts will help you plan a useful alternative model.

The Role of the User Interface

A user interface is the graphical representation of the application to the user. It displays information gathered or generated by the application, displays the state of the application, and lets the user control the application. The interface acts as an intermediary, handling the dialog between the user and the application, as illustrated in the following figure:
In this role, the interface handles four aspects of the dialog:

- **Application ⇒ Interface**: The interface receives input from the application in the form of application data.
- **Interface ⇒ User**: The interface delivers output to the user in the form of graphical data display.
- **User ⇒ Interface**: The interface receives input from the user in the form of key strokes, mouse picks, or other locator events.
- **Interface ⇒ Application**: The interface delivers output to the application in the form of application control.

In this model, the term **application** refers to all parts of the software except those that implement the interface.

---

**Structuring the Application and Interface**

Your program probably needs to perform many complex tasks which requires code for both the application and the user interface. You must also manage the connections between the application and the interface in an efficient manner that facilitates orderly development, maintenance, revision, and extension. The recommended approach is to separate the code for the application from the code for the interface as much as possible, and use discrete modules to handle the connections.
Advantages to Separating Interface and Application

Coding the application and interface separately lets you build, test, and verify your program in stages. It simplifies maintenance and increases program flexibility for extensions or revisions.

DataViews lets you build your interface separately from the rest of your application because the connections between the DataViews interface and the underlying application occur at defined modules within the program. Connecting modules typically handle application data being transferred to the interface for display, or user input being sent to control the application.

DataViews makes it easy to code, test, and verify the various parts of your application separately and incrementally. When creating the views to be used by your application, you don’t have to wait until you have written the modules that supply your application with data before testing the views. You can design and run your views in DV-Draw using the default data sources supplied by DataViews. You can then use DV-Tools to write modules to display, update, and clean up. After completing the interface modules, you can write modules to gather actual application data and to handle user control of the application.

Modular connections between the interface and application simplify maintenance and extension of the program. Planning for extensibility lets you add extra functionality to the application and mirror it easily in the interface.

Structuring the Program

The connections between the application and the interface are critical to a successful program. You can manage these connections in modules where calls to the application are mixed with DataViews calls. You can use either of the following control models or a mixture of the two in your program development.

Application-driven: The data representing events in the application controls the interface. This approach is especially useful for monitoring applications where the user should be made aware of critical data immediately.
Interface-driven: The user events control the application. This approach is especially useful for editing, defining, and browsing applications where user input determines what happens in the interface and underlying application.

Planning the User Interface

When you design the conceptual plan of the user interface, you must consider the tasks your users want to accomplish and the tools and information you can provide. You must also consider the environment where the application will run: does it have a windowing system and what kind of input does the user have? In planning the interface, you can break it down into five functional areas: data display, user input, application data, application control, and control flow. The first four areas were introduced in the Role of the Interface section. Control flow is the sequencing of these four functions. This determines what happens in the interface. Use the following questions as a starting point for your conceptual plan:

- **Application Data:** What data is available? What are its sources? How will the interface gather it? How do you describe the data and the variables it represents?
- **Data Display:** What is visible first? How do you make the data easy to understand? How do you display it? What do you display together? What contextual (static) information helps the user?
- **User Input:** What is the user trying to accomplish? How should these tasks be indicated? How should the user indicate choice? What can the user control in the interface and the application?
Once you have a conceptual plan, you can start using DV-Draw and DV-Tools to build your interface. Although DV-Draw and DV-Tools overlap, use DV-Draw primarily to define data gathering, data display, user input, and flag objects that define changes to the interface. Use DV-Tools to define application control and program control flow.

There are several steps in writing a DV-Tools program. The approach outlined below simplifies this process. At any point you can return to an earlier step and make changes.

**Implementing the User Interface**

**Implementation Path**

- **Learning** Follow the learning path described in Chapter 1.
- **Planning** Make a conceptual plan of your interface. The *Concepts* chapter contains many helpful guidelines and suggestions.
- **Preparing Views** Create your views in DV-Draw. For instructions, see the *DV-Draw User's Guide*.
- **Prototyping** Assign rules to your views to prototype view transitions. Test the transitions in DV-Draw. For instructions, see the *DV-Draw User's Guide* and the Prototyping appendix of the *DV-Draw User's Guide*.
- **Coding** Write your DV-Tools program. This manual provides many guidelines for manipulating views. For additional information see the *DV-Tools Reference Manual*.
- **Debugging** Debug your program. The *Debugging Tips* section contains trouble shooting guidelines.
- **Running** Prepare the system configuration and use the run scripts provided.

**The Role of DV-Draw in Building Your User Interface**

DV-Draw lets you lay out the graphical appearance of your interface. In DV-Draw, you can:

- Create and edit all graphical objects, including graphs and input objects.
- Describe the information to be shown in each view.
- Specify how to display the data using graphs or object dynamics.
- Define the user interactions allowed.
- Simulate the behavior of your interface by defining prototyping rules which let you test the application plan before coding.
Because DV-Draw is interactive, you can edit and see the results immediately. You can review your work in the Run or Prototype Menus, and make revisions easily at any stage of development.

To minimize coding, use DV-Draw to set up as much of the interface as possible. Each chapter in this manual includes a section explaining preparation that can be done in DV-Draw.

Use DV-Draw to create layout views for DV-Tools. Layout views pass graphical information to other parts of the interface. For example, template views pass formatting information to input objects. Layout views containing named rectangles control the position of the views. The position and names of the rectangles are extracted using DV-Tools routines.

**Using a Layout View in a Demo Program**

Naming objects in DV-Draw is an important hook to DV-Tools. Naming significant objects such as input objects, drawport rectangles, and text strings lets you manipulate these objects using DV-Tools routines. Named objects can also have an important role in user input. For example, you can use named subdrawings as pickable icons that activate changes in the current view.
The products of DV-Draw are views that contain static objects, dynamic objects (graphs, input objects, dynamic subdrawings, and dynamic primitives), and data handling structures. These views are the visual part of the interface that are manipulated by the programmatic modules of DV-Tools. DV-Draw and DV-Tools can be integrated because they use the same data structures.

DV-Tools lets you build the structure and control flow of the interface. The structure includes initialization, connection to data sources, and termination. Control flow includes data access and event handling. You can do anything in DV-Tools that you can do in DV-Draw. However, the most efficient approach is to do as much as possible in DV-Draw, then use DV-Tools to handle the remaining programmatic tasks that cannot be done in DV-Draw.

The structure of the DV-Tools portion of the program mirrors the phases of the program as a whole. The initialization section loads views, sets parameters, opens links to application data sources, rebinds to application variables, reads the first set of data, and draws the first screen. In the run phase DV-Tools updates the display to reflect changing data, captures events, and responds to these events. DV-Tools responds to events by directing changes in the interface and the application. The run phase is typically a loop. In the termination phase, DV-Tools cleans up allocated memory and closes links to data sources.

The control flow manages the four functional areas so the interface reads data and updates graphical information, gathers user input data, makes the transitions between views, and controls the application in response to user input.

A critical control function in the run phase is the handling of events. DV-Tools lets you capture a wide range of events and handle them using two models. The first model handles events on a case by case basis, calling appropriate functions based on the type of event. The second model associates events with their function calls during the initialization phase, collects events, and lets the DataViews event handler call the prearranged functions. You can combine these two models, handling certain events yourself, and letting DataViews handle events inside input objects, named objects, or special areas.
This section illustrates a typical structure and control flow of a DV-Tools program. The model is presented using a model program consisting of a description of the model program and a diagram showing the control flow. This program shows a typical application framework; it is only a subset of what you can do with DV-Tools. Some of the functionality shown may not apply to your application, but the structure serves as a useful guide. To understand the model program, you need to know these basic terms:

- **view**: contains graphical and data handling information. Typically created and edited in DV-Draw and saved as a file.
- **screen**: in windowing systems, a window on the physical screen; on a terminal, the entire physical screen.
- **drawport**: maps the graphical objects in the view to a portion of the screen.
- **graphical object**: two-dimensional form that can be drawn on a screen, manipulated in particular ways, and has defined attributes.
- **data source**: describes where data can be obtained and what variables the data represents.
- **event**: a discrete user interaction, such as a key press, a mouse pick, or the window system interpretation of a pick, including expose, resize, or quit events.
- **event request**: connects user events to user functions that modify the application or interface.
**Model Program Description**

The model program has the following structure:

**Initialization**
- Initialize the DataViews environment: environment variables and search path
- Initialize the application environment
- Set up windows and wait screens
  - Load and manipulate views and windows
  - Lay out and create drawports within windows
- Initialize and open data sources, rebind buffers as necessary
- Set up event handling
  - Set the event filters
  - Posts requests for control loop callback functions
- Draw the first displays

**Control loop**
- Handle data
  - Gather any new data from the application
  - Process the data
    - Update the application based on the data
    - Update the dynamic objects displayed in the interface
- Handle events
  - Gather any events
  - Process the events
    - Update the application based on the event
    - Update the interface based on the event

**Termination**
- Destroy any structures that have been set up
  - Destroy views and drawports
  - Close screens and data sources
- Terminate DataViews
- Clean up the application
- Exit
The following diagram condenses the model program and shows its control flow. The Actions are user-defined callback functions.

**Diagram of Model Program Control Flow**
Chapter 3
Structural Overview

Overview ................................................................. 3-1
Migrating from DV-Draw to DV-Tools ............................ 3-1
The Graphical Objects .............................................. 3-2
The Data Sources .................................................... 3-3
The Dynamics Structures .......................................... 3-4
The Display Structures ............................................ 3-7
DV-Tools Data Structure Types ................................. 3-10
The Organization of Routines .................................... 3-11
Naming Conventions for Routines .............................. 3-13
Using Routines to Access Embedded Structures .......... 3-13
A DV-Tools Program Skeleton ................................. 3-14

Table of Contents
This chapter discusses the following topics:

- Migrating from DV-Draw to DV-Tools
- Data Structures
- Organization of the Routines
- A Skeleton Program

Overview

This chapter focuses on the basic information you need for DV-Tools programming: the data structures, the organization of the routine libraries, and the minimum requirements for a DV-Tools program. Since data structures are manipulated both in DV-Draw and DV-Tools, the data structures and their relationships are introduced by showing the connections between DV-Draw terms and concepts and DV-Tools terms and concepts. These relationships are illustrated in the DataViews Data Structure Diagram. This road map to the data structures serves as a learning tool and as a reference tool. After the data structures are introduced, the organization of the routines that act on the data structures is explained. These routines are organized in a loose hierarchy called layers, where each layer of routines works on specific groups of data structures. Finally, the skeleton program shows how the most basic structures and routines fit together to make the bare framework of a working program.

Migrating from DV-Draw to DV-Tools

The simplest way to learn the data structures is to translate what you have learned from DV-Draw into the realm of DV-Tools. In DV-Draw, when you edit arcs, data sources, etc., you are acting on underlying data structures. In DV-Tools, you call routines that act directly on these same data structures. In this section, data structures are introduced in groups and correlated to their counterparts in DV-Draw.
DV-Draw can be thought of as six editors: the geometry, data source, input object, graph, dynamic feature, and rule editors. In DV-Tools, the data structures fall into four main groups which incorporate the six DV-Draw editors. The groups are display, graphical objects, data sources, and dynamics structures.

The view data structure corresponds to what you create in DV-Draw. The view contains graphical objects, data sources, and the dynamic links between them. The following diagram shows the relationships between the view and the four main groups of data structures. This chapter discusses each group in turn. As each group of data structures is discussed, the structures are shown in the corresponding block of the diagram, together with the relationships between structures within that block. The final diagram shows the complete data structures diagram and the relationships between structures in different blocks.

The following diagram shows the structures and relationships within the Graphical Objects block. The graphical objects such as arc, circle, subdrawing, and graph, are located in the Objects Menu in DV-Draw. They correspond to object data structures with abbreviations such as \textit{ar}, \textit{ci}, \textit{sd}, and \textit{dg}, but are often referred to generically using the abbreviation \textit{ob}. All graphical objects have control points and attributes. Control points are point object structures (\textit{pt}) containing world coordinate point structures (\textit{wpt}) that encode the actual position of the point in the drawing. The attributes, such as foreground color and line width, are handled by the attributes structure (\textit{at}).
Graphical objects are managed by the drawing object (\(dr\)). Within a drawing, objects have a definite order from front to back. This order is maintained by a deque object (\(dq\)) belonging to the drawing object. Although it may appear in DV-Draw as if the object name is one of its attributes, the drawing actually maintains the list of graphical object names in a name-object table. In DV-Tools you can also attach information to objects using a slotkey object (\(sk\)), which points to the information you supply.

**The Data Sources Block**

The next version of the diagram shows the structures and relationships within the Data Sources block. In DV-Draw you edit data sources starting with the Edit Data Source List Menu. Editing in this menu acts on the data source list structure (\(dl\)), which manages all the data sources used in the view. Editing in the Edit Data Source Menu acts on an individual data source structure (\(ds\)) and its data source variable structures (\(dsv\)). Data source variables have
buffers that store the data from the sources. DataViews uses these buffers when you run a view in DV-Draw or when you read data into your application.

The Data Sources Block

The next diagram illustrates the structures and relationships in the Dynamics Structures block. In DV-Draw, the following objects use dynamics structures: graphs, input objects, graphical objects with dynamics, and graphical objects with rules.

In DV-Draw, you make a graphical object dynamic by attaching a dynamic feature set, which is called a dynamic control object (dy) in DV-Tools. Dynamic features such as color and subdrawing dynamics use a threshold
table object ($tt$) to map incoming data to defined thresholds. Other dynamic features such as scale and rotation use data from a variable descriptor object ($vd$). This data structure is invisible in DV-Draw.

The variable descriptor ($vdp$) links a data source variable ($dsv$) and a dynamic graphical object. In DV-Draw you can specify which $dsv$ to get data from, define the range of this data, and describe many other attributes depending on the type of dynamics being used. Each dynamic component is controlled by a unique $vdp$. DV-Tools gives you more control over these $vdp$s. For example, you can have the $vdp$ get its data from a program buffer instead of from a $dsv$. All dynamic graphical objects are linked directly or indirectly to a $vdp$.

Input objects ($in$) are linked directly to their variable descriptors. An input object also has an attached structure called the input technique object ($it$). This object is not differentiated in DV-Draw, but you act on it whenever you change the input object type or template. The input object type is called an interaction handler ($ih$). The template used in DV-Tools is not a view, as it is in DV-Draw, but the drawing component of the view.

Graphs, which are called data group objects ($dg$), also have an attached structure, called the data group ($dgp$). Editing in the Edit Graph Menu acts on the data group. The graph is linked to its variable descriptor through the data group, and editing in the Edit Graph Variables Menu acts on the data group’s variable descriptors. The graph type is called the display formatter ($df$). The display formatter and the interaction handler ($ih$) are not structures, but groups of internal routines that handle the complex tasks of drawing and updating graphs and input objects.
In DV-Draw, rules can be attached to graphical objects or to the view’s drawing object. These rules correspond to rule objects \((ru)\). In DV-Tools, objects can share rules. Rules do not use \(vdp\)s but can reference data source variables in their condition or action.

The Dynamics Structures Block

- \(vdp\) (variable descriptor program buffer)
- \(vd\) (variable descriptor object)
- \(dy\) (dynamic control object)
- \(ru\) (rule)
- \(tt\) (threshold table)
- \(it\) (input object technique)
- \(dr\) (template)
- \(ih\) (interaction handler)
- \(dgp\) (data group)
- \(df\) (display formatter)
- \(vdp\) (variable descriptor list)
The fourth version of the diagram shows the structures and relationships in the Display Block. These structures are involved in displaying the view \((vi)\) on the screen, so they are used implicitly in DV-Draw. The screen object \((sc)\) is the DataViews representation of the window or display device. The drawport structure \((dp)\) maps a portion of the view to a portion of the screen. An example of a drawport is the DV-Draw drawing area. When you pan or zoom, the portion of the mapped view changes. The portion of the view is called the drawing viewport (abbreviated as \(wvp\) for world viewport). The part of the screen used to display the view is called the virtual viewport \((vvp)\).

**The Display Structures Block**

The following table lists the abbreviations of the structures shown in the data structures diagram. The headings “Private Types,” “Public Types,” “Graphical Objects,” and “Non-Graphical Objects” are explained in the next section.
### Private Structures | Public Types
---|---
df | display formatter | at | attribute structure

dgp | data group | wvp | world coordinate viewport

dl | data source list | vvp | virtual coordinate viewport

dp | drawport | wpt | world coordinate point

ds | data source |
dsv | data source variable |

### Objects

#### Graphical Objects
- ar | arc object
- ci | circle object
- dg | data group object
- el | ellipse object
- ic | icon object
- im | image object
- in | input object
- ln | line object
- py | polygon object
- re | rectangle object
- sd | subdrawing object
- tx | text object
- vt | vector text object

#### Non-Graphical Objects
- dq | deque object
- dr | drawing object
- dy | dynamic control object
- it | input technique object
- pm | pixmap object
- pt | point object
- ru | rule object
- sc | screen object
- sk | slotkey object
- tt | threshold table object
- vd | variable descriptor object

Data Structures Table: Key to the Symbols
The Complete Data Structures Diagram
DataViews uses many different types of data structures which are manipulated programmatically using DV-Tools routines, or interactively using DV-Draw. DV-Tools data structure types are broken into two broad categories: private types, and public types. The fields in public types are accessed directly; the private types are accessed only through DV-Tools routines. The following figure shows these categories of the DataViews data structures.

The public types are C structures that you can access directly. Typedef statements for the public types are listed in the #include files so that you can access their fields directly.

The private types are more detailed data structures that are accessed only through DV-Tools routines. In your program, you refer to them through pointers or identifiers. You must call specific DV-Tools routines to create and destroy instances of them and manipulate their internal states. Private types fall into two major classes: private structures and objects.

Private structures are C structures which are referred to using pointers. Although all of these pointers are equivalent, each private structure has a unique name. The generic typedef for a DataViews private structure is ADDRESS, which is equivalent to char *.

The private type OBJECT describes those data structures in DataViews that are self-identifying. It is a single type that is used for many different kinds of data structures. This supports the object-oriented concept called polymorphism, which allows a single pointer type to refer to many different kinds of objects. Although all objects use the typedef name OBJECT, they are
subdivided by type. Each type of object has a separate heap, and the \textit{OBJECT} data structure contains the object type and the index of the specific object within the heap. You can manipulate objects in many standard ways using routines devoted to object manipulation. Objects fall into two classes: graphical and non-graphical objects. Graphical objects share many characteristics and are manipulated in special ways due their graphical nature. Non-graphical objects are more individual in their characteristics and the ways in which you manipulate them.

DV-Tools routines are organized in several layers in a loose hierarchy. Some data structures are only manipulated on one level while others are manipulated at several levels. In general, using higher level routines simplifies coding.

\textbf{T (Tools):} The top level. Routines which manage views and high-level data handling. The \textit{T} level has modules for manipulating data source lists, drawports, drawings, data sources, data source variables, location objects, screens, variable descriptors, and views.

\textbf{VO (View Objects):} Just below the \textit{T} level. Routines which manage objects. The \textit{VO} layer has modules for manipulating specific types of objects. This layer includes routines for debugging (\textit{VOdbg}), high-level implementation of basic graphics (\textit{VOg}), and high-level implementation of utilities (\textit{VOu}). A separate layer manipulates generic objects (\textit{VOob}).

\textbf{VUer (View Utility event request):} Routines for event request handling. A request is made when an event occurs within a window or with respect to a boundary.

\textbf{VD (View Display formatter) and VN (View Interaction):} Both graphs and input objects use specialized routines tailored to handle their specific behavior. Graphs use the \textit{VD} layer, which contains the display formatters (\textit{df}). Input objects use the \textit{VN} layer, which contains the interaction handlers (\textit{ih}).

\textbf{VG/VP (View Get/View Put):} Further down are the \textit{VG} and \textit{VP} routines which handle data group (\textit{dgp}) and variable descriptor (\textit{vdp}) structures.

\textbf{VT (View Tables) and VU (View Utilities):} The \textit{VT} routines manage symbol and hash tables, and the \textit{VU} routines contain some commonly used utilities.

\textbf{GR (GRaphics):} The lowest level, device-independent graphics routines.
The chapters in this manual are organized functionally, focusing on DV-Tools concepts rather than on individual layers of routines. The most useful routines are shown in the context of manipulating the major DV-Tools data structures. To see the additional routines available in each layer and module, consult the *DV-Tools Reference Manual*.

The following figure shows which routine layers operate on the different data structures. Some data structures can be manipulated at several levels, as shown by the overlapping routine layers in the figure. Since there are low-level structures that do not appear in the data structure diagram, the layers that manipulate them are not shown.
Naming Conventions for Routines

Most DV-Tools routines follow a naming convention that provides a concise description of what the routine does and what data structure it acts upon. The convention is:

Layer | structure | action

TviLoad   A high-level T layer routine that loads a view from a viewfile.
VOciCreate An object management routine that creates a circle.
VPdcontext A mid-level routine that sets the data group’s context.
VUrghPutFields A utility routine that sets the fields of the rgb structure.

Some lower level routines in the GR and VU layers have simpler descriptive names. For example:

GRdepth Gets the number of bits per pixel.

Using Routines to Access Embedded Structures

Once you understand the relationship between the data structures shown in the data structure diagram and the routine layers, you can start using the routines to access embedded structures. The following fragment shows a typical series of calls that loads a view from a file, accesses a circle within the view, and moves its center point. The structures you must manipulate include the view, drawing object, circle object, and point object. T routines manipulate the view and drawing, and VO routines manipulate the circle and point objects.

```c
char *filename;
VIEW view;
OBJECT drawing, circle, ctrpt;

view = TviLoad (filename);
drawing = TviGetDrawing (view);
circle = TdrGetNamedObject (drawing, "circle's name");
ctrpt = VOciPtGet (circle, 1);
VOptMove (ctrpt, DV_ABSOLUTE, 20000, -10000);
```
A minimal DV-Tools application includes the three phases of the program; initialization, run, and termination. The initialization, run, and termination phases of the program handle the various functional tasks of the program. A DV-Tools skeleton program is shown below.

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

#define DVPATH NULL
#define DVCOLORTABLE NULL
#define DVVIEW NULL

main()
{
    /* Variable declarations */
    /* Initialize */
    TInit (DVPATH, dispform.file);
    screen = TscOpenSet (DVDEVICE, DVCOLORTABLE, V_END_OF_LIST);
    view = TviLoad (DVVIEW);
    drawport = TdpCreate (screen, view, NULL, NULL);
    /* Insert everything else, including run phase, between here ... */

    /* and here. */
    /* Terminate */
    TdpDestroy (drawport);
    TviDestroy (view);
    TscClose (screen);
    TTerminate(screen);
}
```

Every DV-Tools program must begin with `TInit` before any other DV-Tools routine is called and must end with `TTerminate`. `TInit` initializes the search path to be used when opening and closing files. When you use `NULL` in place of the search path parameter, `TInit` picks up the environment variable `DVPATH` as the default. If you do not specify a file of display formatters, `TInit` uses the first `dispforms.stb` that it finds in the search path, as in this example:

```
TInit (NULL, NULL);
```
Similarly, TscOpenSet picks up the device and color table from the environmental variables because the arguments are NULL. TTerminate cleans up structures allocated by TInit.

For more information about the environment variables DVPATH, DVDEVICE, DVCOLORTABLE, and DVVIEW, see Opening the Screen in the next chapter and the Setting the DataViews Environment appendix in the DV-Draw User’s Guide.
### Chapter 4  
**Display**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>4-1</td>
</tr>
<tr>
<td>Useful Data Structures</td>
<td>4-2</td>
</tr>
<tr>
<td>What to do in DV-Draw</td>
<td>4-2</td>
</tr>
<tr>
<td>Basic Function Calls</td>
<td>4-2</td>
</tr>
<tr>
<td>Coordinate Systems</td>
<td>4-3</td>
</tr>
<tr>
<td>World Coordinate System</td>
<td>4-4</td>
</tr>
<tr>
<td>Virtual Coordinate System</td>
<td>4-4</td>
</tr>
<tr>
<td>Screen Coordinate System</td>
<td>4-4</td>
</tr>
<tr>
<td>Low-Level Graphics</td>
<td>4-5</td>
</tr>
<tr>
<td>Screens</td>
<td>4-5</td>
</tr>
<tr>
<td>Opening the Screen</td>
<td>4-6</td>
</tr>
<tr>
<td>Multiple Screens</td>
<td>4-7</td>
</tr>
<tr>
<td>Screen Colors</td>
<td>4-7</td>
</tr>
<tr>
<td>Drawport List</td>
<td>4-8</td>
</tr>
<tr>
<td>Drawports</td>
<td>4-8</td>
</tr>
<tr>
<td>Creating Drawports</td>
<td>4-9</td>
</tr>
<tr>
<td>Drawing Views to the Screen</td>
<td>4-11</td>
</tr>
<tr>
<td>Drawing Individual Objects</td>
<td>4-12</td>
</tr>
<tr>
<td>Using Layout Views</td>
<td>4-13</td>
</tr>
<tr>
<td>Ordering, Clipping, and Obscuring</td>
<td>4-14</td>
</tr>
<tr>
<td>Drawport Color</td>
<td>4-16</td>
</tr>
<tr>
<td>Manipulating Drawports</td>
<td>4-16</td>
</tr>
<tr>
<td>Displaying Views</td>
<td>4-16</td>
</tr>
<tr>
<td>Dynamics in the View</td>
<td>4-17</td>
</tr>
<tr>
<td>Drawings in a View</td>
<td>4-18</td>
</tr>
<tr>
<td>Saving Views</td>
<td>4-19</td>
</tr>
<tr>
<td>GRconfig: Controlling the Number of Devices</td>
<td>4-19</td>
</tr>
<tr>
<td>Editing GRconfig.c under UNIX</td>
<td>4-19</td>
</tr>
<tr>
<td>Editing GRCONFIG.C under OpenVMS</td>
<td>4-21</td>
</tr>
<tr>
<td>Summary</td>
<td>4-21</td>
</tr>
</tbody>
</table>
Chapter 4

Display

This chapter discusses the following topics:

- Drawing and Updating Views and Objects
- Managing Multiple Views and Screens
- Using Coordinate Systems

Overview

The interface delivers output to the user in the form of graphical data display. This chapter explains how to make your graphics appear on the screen. In the simplest case, you open and set the characteristics of the screen. The screen is the programmatic representation of the window or the physical display.

Load the view information from the DV-Draw view files. Then create a drawport to map the drawing viewport to the screen viewport. The drawing viewport is a portion of the view and the screen viewport is a portion of the screen. After the first iteration of data has been gathered, the view is drawn on the screen. All of this takes place in the initialization phase. In the run phase, you must update the view’s dynamic objects according to your application data. In the termination phase, destroy the view and drawport, and close the screen.

The Relationship between the View, Screen, and Viewports
Display is more complex when you have partially obscured views, or multiple screens, views, or drawports. Programmatically creating or erasing objects, resizing drawports, changing views, and opening new screens also complicate display.

The primary data structures used for display are the screen (sc), view (vi), and drawport (dp). The view (vi) has two major components, the data sources (ds), which manage the data coming from the application, and the drawing object (dr), which manages the graphical objects. The data sources can be handled as a group using the data source list (dl).

The drawport maps the drawing viewport (wvp) to the screen viewport (vvp). When it does the mapping, it creates a transform object (xf), which handles the mathematical details of the mapping. The viewports, wvp and vvp, get their abbreviations from their different coordinate systems, the world coordinate system and the virtual coordinate system.

Most of the routines used to handle these display tasks are in the T layer, particularly the Tsc (Tscreen), Tdp (Tdrawport), and Tvi (Tview) modules.

Except for the view, most data structures used for displaying graphics are not accessible in DV-Draw. In DV-Draw, you should prepare your views as completely as possible with respect to positioning graphical objects and setting up the data handling.

You can prepare layout views containing rectangles which indicate where drawports should appear. You can also set up objects to be extracted from their view and displayed in a different view. Both of these techniques require special coding and are discussed later in this chapter.

DV-Draw lets you display the cursor coordinates by pressing <c>. Both world coordinates and screen coordinates are displayed. You can use this information to plan the location of objects or viewports.

Basic graphic display is handled by the following calls:

In the initialization phase:

Opening the screen:

```c
screen = TscOpenSet (device_name, color_table, V_END_OF_LIST);
```

Loading the view:

```c
view = TviLoad ("filename");
```
Opening the data sources: \texttt{TviOpenData (view)};

Reading the first iteration \texttt{TviReadData (view)};

Creating the drawport and specifying the viewports: \texttt{drawport = TdpCreate (screen, view, screen_vvp, drawing_wvp)};

Drawing the first time: \texttt{TdpDraw (drawport)};

\textbf{In the run phase:}

Gathering new data: \texttt{TviReadData (view)};

Displaying new data: \texttt{TdpDrawNext (drawport)};

\textbf{In the termination phase:}

Destroying the drawport: \texttt{TdpDestroy (drawport)};

Closing the data sources: \texttt{TviCloseData (view)};

Destroying the view: \texttt{TviDestroy (view)};

Closing the screen: \texttt{TscClose (screen)};

These calls are the basis of the simplest DV-Tools example program, \textit{playback.c}. Most applications require more complexity, and DV-Tools supports this by letting you manipulate its data structures in many ways. First you must understand the mechanics of the major data structures involved in display: the screen, the drawport, and the view. The following sections cover each of these data structures in detail, and contain example code fragments showing some useful manipulations of the data structures.

To display graphics on the screen, you need to understand the three coordinate systems used in DV-Tools. These coordinate systems work together to map drawing data structures to drawports and finally to the physical screen. To use the coordinate systems effectively, you must know which to use with various data structures and routines, and how to convert from one system to another. The routines \texttt{TdpScreenToWorld} and \texttt{TdpWorldToScreen} convert between the world and screen coordinate systems, and the routines \texttt{GRscs_to_vcs} and \texttt{GRvcs_to_scs} convert between the screen and virtual coordinate systems. The three coordinate systems and the structures that use them are described below.
**World Coordinate System**

The drawing viewport, the drawing object, and all graphical objects use the **world coordinate system**. Object position is specified in world coordinates.

The full extent of the world coordinate system, and therefore of the drawing, is -16K to 16K (-16384 to 16383) both horizontally and vertically. The center of the drawing is (0, 0) in world coordinates. The drawing space is square, as you can see in DV-Draw when you zoom out enough to see the whole drawing area.

**Virtual Coordinate System**

To draw a graphical object on a screen, you must convert the object’s world coordinates to physical pixel-based coordinates. This step is complicated by the fact that physical screens and windows on those screens vary in their number of pixels. DataViews provides the **virtual coordinate system** as a device independent method for specifying positions on the screen.

The full extent of the virtual coordinate system, and therefore the full extent of the screen measured in virtual coordinates, is 0 to +32K [0, 32767]. The origin is in the lower left-hand corner. Since most windows are not square, the physical length of a vertical unit is not equal to the length of the horizontal unit in virtual coordinates.

**Screen Coordinate System**

The window, which is represented in DV-Tools by the screen object, uses the **screen coordinate system** to display graphics. The screen coordinate system is a pixel-based coordinate system, with the origin (0, 0) at the lower left corner. The extent of the coordinate system depends on the number of pixels in the horizontal and vertical dimension. The screen and location objects are the major objects that use the screen coordinate system, though the location object also uses world coordinates. Many routines related to these data structures, such as the VUer event handling routines, and the lower-level utility and graphics (VU and GR) routines, also use the screen coordinate system.

In a window system, such as X Window or Microsoft Windows, you can consider each window as an independent display device with its own origin. When you open the screen object representing the window, you set the extent of the window, and therefore the extent of the screen coordinate system.
If you are working in a non-windowing system, the physical device determines the extent of the screen coordinate system.

The Three Coordinate Systems

Low-Level Graphics

You can also manipulate graphics on the screen using the lower level routines in the VU and GR layers. These routines do not act directly on the data structures discussed earlier in this chapter, but instead act on the current device, which is analogous to the screen object, and the current point. Because the model for using the low-level graphics routines is not data structure dependent, it is not discussed in this manual.

Screens

The screen object (sc) is the programmatic representation of the display device, either the entire physical screen or a window on the screen. 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 contains information about its attributes, drawports, foreground and background colors, and polling. The polling information is covered in the *Event Handling* chapter. The attributes include dimensions measured in screen coordinates. The following drawing illustrates components of the screen object.

**Opening the Screen**

The first step after `TInit` is opening a screen using `TscOpenSet`, which has the following form:

```c
screen = TscOpenSet (device_name, color_table,
                     flag, value,
                     flag, value,
                     ...,
                     V_END_OF_LIST);
```

You must specify a valid device in one of two ways. You can supply the device name as the parameter, or you can specify `NULL` instead of the device name. In the `NULL` case, the routine opens the default device specified in your environment variable `DVDEVICE`. If you use `NULL` for the color table, the routine uses the color table specified in your environment variable `DVCOLORTABLE`.

`TscOpenSet` lets you assign values for the attributes of the window. For example, you can set the window’s height, width, and offsets from the screen origin in screen coordinates using the following call:

```c
screen = TscOpenSet ("X", NULL, V_WINDOW_X, 100,
                     V_WINDOW_Y, 100, V_WINDOW_WIDTH, 800,
                     V_WINDOW_HEIGHT, 600, V_END_OF_LIST);
```
You can also open a window using window system calls and pass it to DataViews to use as a screen:

```c
screen = TscOpenSet("X", DVCOLORTABLE,
                   V_DISPLAY, (LONG) display,
                   V_WINDOW_ID, (LONG) window,
                   V_END_OF_LIST);
```

To reset the attributes of a window, call `GRset` with the attribute flags and values you want to set. The following fragments from the example `dragging.c` shows how `GRset` is called to change an attribute. In this case, the attribute is the drawing function, which is used to turn rubberbanding on and off.

```c
{  
  GRset (V_DRAW_FUNCTION, V_XOR, V_END_OF_LIST);  
  rubberbanding = YES; /* rubberbanding is now on */  
}

{  
  GRset (V_DRAW_FUNCTION, V_COPY, V_END_OF_LIST);  
  rubberbanding = NO; /* rubberbanding is now off */  
}
```

### Multiple Screens
To open multiple screens, call `TscOpenSet` repeatedly using the same generic device name, such as “X.” Note that you must be working in a windowing environment.

When you are working with multiple screens, you must be aware that many routines in the VO, VU, and GR layers operate on the current screen rather than taking a screen as a parameter. You can manage the current screen in a number of ways:

- Calling `TscSetCurrentScreen` changes the current screen to the screen you pass as a parameter.
- Calling `TdpDraw`, `TdpDrawNext`, `TdpDrawNextObject`, `TdpDrawObject`, `TdpErase`, and `TdpEraseObject` change the current screen to the screen associated with the drawport.

The types of screens, such as X or Windows, that you can open depend on the devices chosen when DataViews was installed. To change the types of screens, you must edit `GRconfig.c` located in the `src` directory. See the instructions later in this chapter.

### Screen Colors
The screen foreground and background colors serve as the inherited foreground and background color for drawings and graphics objects when you do not otherwise set these colors. To set the foreground or background...
color, call TscDefForecolor or TscDefBackcolor respectively. These routines take a color object as a parameter; color objects are covered in detail in the Objects chapter. The original foreground and background screen colors are set in the device driver, and are usually white foreground and black background, except on monochrome devices, where they are reversed.

**Drawport List**

The screen maintains an internal ordered list of the drawports it contains. The drawport list is not directly accessible, so you must perform most drawport operations on the drawport directly.

**Drawports**

A drawport (dp) defines how and where to display a view on the screen. It is a DV-Tools structure that contains a view, a screen, the portion of the view to display (in world coordinates), and the portion of the screen to display it in (in virtual coordinates). The drawport also implicitly creates transform (xf) objects, which hold the world-to-screen and screen-to-world coordinate transformation mapping. Once you define the portion of the view and the portion of the screen, the drawport handles all mappings for graphical objects so that you do not need to make any world to screen coordinate conversions.

The following figure illustrates the components of the drawport:

![Diagram](image)

A drawport can be thought of as a picture frame which contains, as its picture, a view. Like a picture frame, the drawport may mask off a portion of its view and be moved around as a unit. Drawports can be stacked on top of other drawports that they obscure.
Many drawports can belong to the same screen, so several views can be displayed on a single graphics device. Each drawport can only display a single view. When multiple drawports exist on the same screen, the screen uses its internal list to manage the order of the drawports. The order affects which drawports are in front and obscure other drawports, and how the obscured drawports are clipped. A newly created drawport appears on the bottom of the list, which correspond to the front of the screen when the drawport is redrawn.

Creating Drawports

To create a drawport, call `TdpCreate` or `TdpCreateStretch`, passing an open screen object, a view, and the two rectangles, specifying portions of each, the screen virtual viewport and the world viewport. The calls have the following form:

```c
DRAWPORT
TdpCreate (   
  OBJECT screen, 
  VIEW view, 
  RECTANGLE *vvp, 
  RECTANGLE *wvp)

DRAWPORT
TdpCreateStretch (   
  OBJECT screen, 
  VIEW view, 
  RECTANGLE *vvp, 
  RECTANGLE *wvp)
```

To decide between these calls, you must consider not only what portion of the view and what portion of the screen, but also how any mismatch in aspect ratio should be handled. You have three options: the drawport can stretch the drawing to fit the screen viewport, stretching objects in the drawing in the process; the drawport can make a “best fit” by lopping off the edges of the drawing viewport to fit the screen viewport, but maintaining the aspect ratio of the objects in the drawing; or the drawport can use a “whole world” fit, showing all of the drawing viewport, maintaining the aspect ratio, but leaving gaps where the drawing and screen viewports do not match in their aspect ratios.

You exercise the three options by using either `TdpCreateStretch` (stretching the drawing to fit) or `TdpCreate` (lopping off edges or not filling the screen viewport completely), and by passing either `NULL` or coordinates for the viewports. Passing `NULL` in place of the screen viewport makes the drawport try to use the entire screen. Passing `NULL` in place of the drawing viewport
Drawports Display

makes the drawport try to map the best fit of the drawing to the screen viewport. The following figure illustrates the result of passing these parameters:

<table>
<thead>
<tr>
<th>Whole World</th>
<th>Best Fit</th>
<th>Stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Whole World" /></td>
<td><img src="image2" alt="Best Fit" /></td>
<td><img src="image3" alt="Stretch" /></td>
</tr>
</tbody>
</table>

To use the “whole world” option, avoiding both stretching and clipping, call `TdpCreate`, specifying the `wvp`. The drawing viewport is mapped starting at the center and expanding the view equally in the x and y direction until two edges of the drawing viewport touch the boundary of the screen viewport. `TdpCreate` always maintains the aspect ratio of the objects, but in this case the unused portion of the screen may show in a different color. You can reset this color by calling `VODrOffcolor`.

To use the “best fit” approach, call `TdpCreate`, specifying `NULL` for `wvp`. The best fit is calculated starting at the center and expanding the view equally in the x and y directions until the entire screen viewport is filled. `TdpCreate` maintains the aspect ratio of the objects in the drawing, but may clip parts of the view. The fit is calculated only once, when the drawport is drawn. It is not recalculated whenever the window changes size.

To use the “stretch” approach, call `TdpCreateStretch`, specifying the `wvp`. `TdpCreateStretch` stretches the x or y dimension of the drawing viewport to match the screen viewport aspect ratio. This ensures that the entire drawing viewport is shown, but can cause distortions in the aspect ratios of the graphical objects. Choose `TdpCreateStretch` when you want the entire drawing viewport to fill the entire screen viewport. `TdpCreateStretch` may also be the best choice if application users can resize the windows where the drawports appear.
To get the best results, the drawing viewport and the screen viewport should have similar aspect ratios (as measured in screen coordinates). This minimizes clipping, stretching, and extra background. To reduce these potential problems, adjust the viewport parameters in the drawport creation call.

To get the drawing viewport, call `VOobBox` and pass in either the drawing object or the object in the view that you want to display. Then define a screen viewport with a similar aspect ratio in screen coordinates, and use `GRscs_to_vcs` to convert the screen coordinates to virtual coordinates.

In the following call, the drawport stretches the drawing to fill the screen:

```c
drawport = TdpCreateStretch (screen, view, NULL, &whole_world);
```

To draw a drawport for the first time, call `TdpDraw`. `TdpDraw` initializes the drawing information for the drawport and automatically moves the drawport to the top of the visibility list. Note that graphical objects not visible as a result of visibility dynamics are not drawn in the drawport. Objects must be visible in order to be drawn. After the drawport is drawn on the screen, the circumstances determine which routine you call to update the drawport. The following table defines the most common circumstances and which routine to call:

<table>
<thead>
<tr>
<th>Updating dynamics</th>
<th>Redrawing a drawport</th>
<th>Redrawing the screen</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TdpDrawNext</code></td>
<td><code>TdpRedraw</code></td>
<td><code>TscRedraw</code></td>
</tr>
</tbody>
</table>

Call `TdpDrawNext` to update dynamic objects after you have gathered an iteration of data. To update only a particular object, call `TdpDrawNextObject`. If an object that has visibility dynamics becomes invisible at update, it is erased using the erase method specified by its dynamic control object.

Call `TdpRedraw` to redraw the entire drawport or a portion of the drawport. This is often required after you reorder the ordered list of drawports, resize a drawport, erase a drawport that obscured other drawports, or pan or zoom the view within the drawport.

Call `TscRedraw` to redraw all the drawports on the screen or a portion of the screen. This is often required after you resize a screen, reorder the ordered list of drawports, resize a drawport, or erase a drawport that obscured other drawports. To re-establish the relative size of drawports after you resize the screen, call `TscReset` before calling `TscRedraw`. 

**Drawing Views to the Screen**

<table>
<thead>
<tr>
<th>Updating dynamics</th>
<th>Redrawing a drawport</th>
<th>Redrawing the screen</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TdpDrawNext</code></td>
<td><code>TdpRedraw</code></td>
<td><code>TscRedraw</code></td>
</tr>
</tbody>
</table>

Call `TdpDrawNext` to update dynamic objects after you have gathered an iteration of data. To update only a particular object, call `TdpDrawNextObject`. If an object that has visibility dynamics becomes invisible at update, it is erased using the erase method specified by its dynamic control object.

Call `TdpRedraw` to redraw the entire drawport or a portion of the drawport. This is often required after you reorder the ordered list of drawports, resize a drawport, erase a drawport that obscured other drawports, or pan or zoom the view within the drawport.

Call `TscRedraw` to redraw all the drawports on the screen or a portion of the screen. This is often required after you resize a screen, reorder the ordered list of drawports, resize a drawport, or erase a drawport that obscured other drawports. To re-establish the relative size of drawports after you resize the screen, call `TscReset` before calling `TscRedraw`. 

---

**Display Drawports**

**DV-Tools User’s Guide**

**FANUC CNC Parts**
Note that views containing dynamic objects can only be drawn in one drawport at a time. To draw a view with dynamics in more than one drawport at a time, clone it first and use the copy in the other drawport. To display 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.

You can draw objects, such as pop-up menus or graphs, created programmatically or extracted from a view. The major considerations are controlling when they are drawn and erased, and where they appear. Also, graphical objects that are not visible as a result of visibility dynamics are not drawn. You have two choices for displaying a special object:

Add the object to a view’s drawing object. If it is visible, it is drawn as part of the view when you call TdpDraw or TdpRedraw.

Draw an object in a drawport without its being part of a view. The drawport provides the world to screen mapping required to draw graphical objects onto the screen. To draw the object, call TdpDrawObject; the parameters are the object and the drawport in which it appears. The drawport must have been drawn with TdpDraw before calling TdpDrawObject because TdpDraw initializes necessary drawport information such as transforms. You must also be aware that an object that is not part of a drawport’s view must be managed independently because the drawport has no history or awareness of the object. For example, calls to TdpDrawNext do not update the object; instead you must use TdpDrawNextObject. TdpRedraw and TscRedraw do not redraw the object; instead you must use TdpRedrawObject. Also, an object that is not part of a view cannot be found by TloGetSelectedObject. To erase the object, call TdpEraseObject. To draw the object again after explicitly erasing it, call TdpDrawObject, which handles the required reinitialization.

The control points of the object control where it appears in the drawport. Be aware that an object may be clipped or may not appear at all if its control points are outside of the drawing viewport being displayed in the drawport.
Dynamic objects keep drawport-specific information, so they can only be drawn in one drawport at a time. To draw a dynamic object in more than one drawport at a time, clone it first and use the copy in the other drawport. To display the same dynamic object in different drawports at different times, erase the object in the previous drawport before drawing it in the new drawport. In this case, the object does not need to be cloned.

**Using Layout Views**

You can lay out the positions of DV-Tools drawports interactively in DV-Draw using named objects, and extract the position information programmatically as illustrated below. Note that you should use simple geometric objects such as rectangles to define regions in the layout view. You should not use objects with dynamics or subdrawings to define regions.

When using layout views to position a drawport, the position of the drawport is not hard-coded. To change the dp position, edit the layout view, not the source code. This example uses three existing views; `layout.v` is a layout view defining the drawport layout; `main.v` is the initial main application view; and `message.v` is the initial message view. The layout view is loaded and the named rectangles `main_dp` and `message_dp` are extracted. Their coordinates define the areas for the drawports in which `main.v` and `message.v` appear.

```c
/* XMAX = 16383, YMAX = 16383. Defined in VOstd.h. */
OBJECT screen, layout_dr, dp_rect;
VIEW layout_view, main_view, message_view;
RECTANGLE wvp, vvp, dummy;
DRAWPORT main_dp, message_dp;
RECTANGLE wholeworld = {-16384, -16384}, {16383, 16383};

/* Load the views and get the drawing from the layout view */
layout_view = TviLoad("layout.v");
layout_dr = TviGetDrawing(layout_view);
main_view = TviLoad("main.v");
message_view = TviLoad("message.v");

/* Get the main_dp rectangle */
dp_rect = TdrGetNamedObject(layout_dr, "main_dp");

/* Get the main_dp rectangle’s coordinates */
VOobBox(dp_rect, &wvp, &dummy);

/* Convert the main_dp rectangle’s coordinates to virtual screen coordinates. */
for (int i = 0; i < 4; i++) {
    vvp[i].ll.x += XMAX;
    vvp[i].ll.y += YMAX;
    vvp[i].ur.x += XMAX;
    vvp[i].ur.y += YMAX;
}
```

---

**Display Drawports**

**DV-Tools User’s Guide 4-13**

**FANUC CNC Parts**
Drawports

/* Use these coordinates to create a drawport */
main_dp = TdpCreateStretch (screen, main_view, &vvp, &wholeworld);

/* Repeat the preceding four steps to make the message drawport */
dp_rect = TdrGetNamedObject (layout_dr, *message_dp*);
VOobBox (dp_rect, &wvp, &dummy);
vvp.ll.x = wvp.ll.x + XMAX;
vvp.ll.y = wvp.ll.y + YMAX;
vvp.ur.x = wvp.ur.x + XMAX;
vvp.ur.y = wvp.ur.y + YMAX;
message_dp = TdpCreateStretch (screen, message_view, &vvp, &wholeworld);

/* Draw the created drawports */
TdpDraw (main_dp);
TdpDraw (message_dp);

/* Destroy the layout view since it is no longer needed */
TviDestroy (layout_view);

Ordering, Clipping, and Obscuring

The screen object maintains an ordered list of drawports and handles obscuring and clipping internally. You can change the order of the list to move drawports to the front or the back by calling TdpFront or TdpBack respectively.

Ordering, Clipping, and Obscuring of Multiple Drawports

The view in this drawport is clipped to the boundaries of the obscuring drawports.
Obscuring drawports must be considered when you erase, move, or resize a drawport that obscured other drawports. You may need to clean up the screen by redrawing the newly exposed areas. The simplest method is to determine what portion will need repair and call TscRedraw to redraw that portion. The following code fragments show three ways to refresh an area that was previously obscured. In each fragment, the drawport’s screen must be the current screen.

If you want to redraw the screen or a portion of the screen in its current state use TscRedraw.

```c
/* Get the virtual coordinates of the drawport, convert to screen coordinates. */
vrect = TdpGetScreenVp (dp);
GRvcs_to_scs (&vrect->ll, &srect->ll);
GRvcs_to_scs (&vrect->ur, &srect->ur);

/* Redraw the region of the screen. */
TscRedraw (screen, srect);
```

If you know that the state of the screen won’t change while it is obscured, you can take a raster image of the area before you obscure it using GRrasget and use GRrasdraw to refresh the screen. Use GRrasquery to determine if GRrasget and GRrasdraw are supported on your platform.

```c
/* Get the virtual coordinates of the drawport, convert to screen coordinates. */
vrect = TdpGetScreenVp (dp);
GRvcs_to_scs (&vrect->ll, &srect->ll);
GRvcs_to_scs (&vrect->ur, &srect->ur);

/* Take the raster image before obscuring the drawport. */
/*add 1 to make the sizes inclusive */
GRrasget (&srect->ll, srect.ur.x-srect.ll.x+1,
         srect.ur.y-srect.ll.y+1, &raster);

/* Refresh the obscured drawport using the raster image. */
GRrasdraw (&srect->ll, raster);
```

If the area being obscured is a solid color you can draw a filled rectangle in that color to refresh the screen using GRf_rectangle.

```c
/* Get the virtual coordinates of the drawport, convert to screen coordinates. */
vrect = TdpGetScreenVp (dp);
GRvcs_to_scs (&vrect->ll, &srect->ll);
GRvcs_to_scs (&vrect->ur, &srect->ur);
```
Drawport Color

Drawports can be opaque or transparent. If drawports are opaque, a filled rectangle is drawn using the drawing’s background color, obscuring any underlying drawports. If drawports are transparent, the objects in the transparent drawport appear to overlay the objects in the previous drawport. Drawports containing views from DV-Draw are opaque. Drawports containing views with the drawing’s background color set to NO_BACKGROUND are transparent. The following code fragment shows how to make the drawport transparent.

```c
view = TdpGetView (drawport);
drawing = TviGetDrawing (view);
VOdrBackcolor (drawing, NO.BACKGROUND);
TdpDraw (drawport);
```

Manipulating Drawports

You can pan, zoom, zoom to, and resize your drawports using TdpPan, TdpZoom, TdpZoomTo, and TdpResize respectively. This provides additional control over what portion of the view is displayed after the drawport has been created. After manipulating the drawport, you must redraw the contents using TdpRedraw. If you shrink the drawport, you may need to redraw underlying drawports using the code fragment above.

You may need to convert coordinates directly. To convert from screen to world coordinates, use TdpScreenToWorld; to convert from world to screen coordinates, use TdpWorldToScreen.

Some low-level routines may require you to explicitly pass the transform. To get the transform, call TdpGetXform, passing the drawport and a flag. Valid flags are DR.TO_SCREEN and SCREEN.TO.DR. The routine returns a transform object that you can use to convert coordinates for use with the drawport’s screen and view. Any resize, zoom, or pan of the drawport affects the transform, so the drawport always recalculates the transform. Even after you pass the transform to a routine it continues to reflect the current state of the drawport. The transform object is described in greater detail in the Objects chapter.
A view ($vi$) is a mechanism for the visual representation of data. It contains both graphical information and the connections to data that drive the dynamic changes in the graphics. Views are usually prepared in DV-Draw. In DV-Tools, views are drawn onto the screen using drawports. You can manipulate the entire view and access all underlying data structures. The view manages the underlying structures using two main structures: the drawing object, which manages the graphical objects used to display the data; and the data source list, which manages the data gathering. The following figure shows the components of the view and their relationship:

![Data Flow](image)

To access a view created in DV-Draw, call `TviLoad`, passing in a filename. Once a view is associated with a drawport you can access it by calling `TdpGetView`. To access the view’s drawing, call `TviGetDrawing`, passing the view as a parameter. To open all the data sources for the view and its subdrawings, call `TviOpenData`, passing the view as a parameter. To open the data sources for a view with no subdrawings, call `TviGetDataSourceList` followed by `TdlOpenData`.

A drawing object ($dr$) is a representation of the visual part of the view. It contains a list of graphical objects and a list of the names assigned to them. The graphical objects may represent geometric shapes such as lines, circles, and rectangles, or dynamic objects such as graphs, dynamic subdrawings, and input objects. The drawing portion of a view is normally created and edited in DV-Draw. In DV-Tools, you can control many aspects of the graphic display by manipulating the drawing object directly. This is discussed in detail in the *Objects* chapter.

The data source list ($dl$) is used by the view to manage data gathering from external sources. Data sources for a view are usually created and edited in DV-Draw. In DV-Tools, data is fed to the dynamic objects in the drawing for use in updating the display.
**Dynamics in the View**

View dynamics consists of gathering data and displaying data using the graphical objects. In the data gathering phase, the data sources gather the data as a stream and store it in buffers corresponding in size and type to the application variables that supply the data. The buffers are attached to data source variables (dsv), which describe the type of data, the name of the variable, and size and type of the buffer. The data source manages its linked list of data source variables. Once a data source is opened using TviOpenData or TdlOpenData, you can gather a new iteration of data and fill the buffers by calling TviReadData or TdlReadData. The data stays in the buffers until the next iteration of data overwrites it.

In the display phase of dynamics, the dynamic objects use variable descriptors to access the data buffers and interpret the data. The variable descriptor (vdp) provides information on how the dynamic object should update in response to the data. For example, graphical objects can change size, position, visibility, or attributes; dynamic drawings display different subdrawings; and graphs display new data points. To update dynamic objects, call TdpDrawNext after calling TviReadData or TdlReadData.

The following figure shows a simplified model of dynamics in a view. For more information about dynamics, see the Dynamics chapter.

**Drawings in a View**

Views can only manage one drawing at a time. You can display two drawings in the same space at the same time either by merging the views or by drawing a view as an object in another view. To merge two views, call TviMergeDrawing followed by TdpRedraw. You can subsequently remove
either the base view or the added view by calling TviExciseDrawing, passing in the merged view and the drawing to be removed. To overlay a drawing as an object, call TdpDrawObject, passing in the drawing object. In this case, the drawing object does not become part of the view, but is drawn over the existing drawing.

Saving Views

If you want to pass views between applications, or you have built editing capabilities into your application, you may need to save your view within DV-Tools. The routine TviSave opens and saves a file in binary format; TviASCIISave opens and saves in a file in ASCII format.

GRconfig: Controlling the Number of Devices

The total number of windows or other display devices that can be open at one time is limited by the GRconfig.c file. Although there are no device limits for X windows or Microsoft Windows, there are limits on hard copy output devices or windows on other windowing systems. The GRconfig.c file contains jump-table entry points that bridge the gap between the device-independent graphics routines in the GR layer and the device drivers installed. During installation, DataViews creates GRconfig.c and sets the total number of devices. This number depends on the machine and whether or not you have specified fewer devices in a tailored installation of DataViews. To increase the number of windows, you must edit the GRconfig.c file and compile the edited file using DVlink to make a GRconfig object file.

Editing GRconfig.c under UNIX

To edit GRconfig.c, use the following procedure:

Go to the dataviews/src directory.
Open GRconfig.c for editing.
Change the constant, NUM_DEV_TYPES, to the new total number of devices.
For each new device:
    Add an entry for the device name, such as X, to the GRname array.
    Add an entry corresponding to the new device to the GRmake_table array.
The following sample GRconfig.c file is set up to allow an HP7550A, an HP7475A, three metafile devices, and any number of X windows:

```c
#include "std.h"
#include "dvstd.h"
#include "dvGR.h"
#include "GRlink.h"
#define NUM_DEVTYPES 6

GLOBALDEF int GRnumdevtypes = NUM_DEVTYPES;
GLOBALDEF int GRnumdevices = 0;
GLOBALDEF DV_BOOL *GRopened = NULL;
GLOBALDEF char *GRdev_name [NUM_DEVTYPES] = {
    "X",
    "HP7550A",
    "HP7475A",
    "METAFILE",
    "METAFILE",
    "METAFILE",
};

GLOBALDEF char **GRname = NULL;
FPT_ENTRY *DDxw_make_table();
FPT_ENTRY *DDh75_make_table();
FPT_ENTRY *DDhpp_make_table();
FPT_ENTRY *DDmf_make_table();
FPT_ENTRY *DDmf_make_table();
FPT_ENTRY *DDmf_make_table();

GLOBALDEF FPT_ENTRY **GRlink_table = NULL;
GLOBALDEF FPT_ENTRY *((GRmake_table[NUM_DEVTYPES])()) =
{
    DDxw_make_table,
    DDh75_make_table,
    DDhpp_make_table,
    DDmf_make_table,
    DDmf_make_table,
    DDmf_make_table,
};

Compile a new GRconfig.o:

DVlink -c GRconfig.c
To use the new `GRconfig.o`, choose either of the following methods:

- Move `GRconfig.o` from the `dataviews/src` directory to the `dataviews/lib` directory.
- Set the `DVGRCONFIG` environment variable to the new `GRconfig.o`.

Relink your applications and DataViews.

To edit `GRCONFIG.C`, use the following procedure:

- Go to the `DV$SRC` directory.
- Open `GRCONFIG.C` for editing.
- Change the constant, `NUM_DEVTYPES`, to the new number of devices.
- For each new device:
  - Add an entry for the device name, such as `X`, to the `GRname` array.
  - Add an entry corresponding to the new device to the `GRmake_table` array.

Compile a new `GRCONFIG.OBJ`:

```
CC GRCONFIG.C
```

To use the new `GRCONFIG.OBJ`:

- Rename `DV$LIB:GRCONFIG.OBJ` to save the old version of the file.
- Move `GRCONFIG.OBJ` from the `DV$SRC` directory to the `DV$LIB` directory.
- Relink your applications and DataViews.

Another way to change the number of windows is to run `@DV$HOME:DVREINSTALL` and change the device choices interactively. This method ensures that all DataViews executables, including the shareable image, get relinked with the new version of `GRCONFIG.OBJ`. This method does not relink your applications or the demos.

This chapter has presented these basic techniques:

- How to handle multiple windows so you can display DataViews in several windows at the same time.
- How to display several views in a single window using drawports.
- How to control both windows and drawports.
How to specify the positions of drawports using DV-Draw layout views.
How to display individual objects in a drawport.
How to access the underlying structures of a view.

Using these techniques, you can implement the following features in your application:

Manage views using multiple windows, multiple drawports, or both. You can display one drawport per window in multiple windows, using the window manager to control the order of the windows, or multiple drawports in a single window, using DV-Tools routines to control the order of the drawports.

Separate different components of your application in separate views for display in different places. For example, you can separate the static components, dynamic data display, or input components into different views. The telecom demo uses one window to display the top level view and another to display output information.

Use a layout file to position information on the screen. To experiment with different designs, edit the layout file.

Use a single view that serves as a library of objects. In DV-Tools, you can extract objects from the view to display in other views as you need them.
Chapter 5
Objects

Overview ................................................................. 5-1
Useful Data Structures ............................................... 5-1
What to do in DV-Draw .............................................. 5-2
What to do in DV-Tools .............................................. 5-2
Basic Function Calls .................................................. 5-2
Manipulating Objects ............................................... 5-3
Object Hierarchy ...................................................... 5-3
Heaps ................................................................. 5-4
Control Points ......................................................... 5-4
Reference Counts .................................................... 5-5
Attributes ............................................................. 5-6
Attributes Table ....................................................... 5-8
Common Attributes .................................................. 5-8
Foreground and Background Color ......................... 5-8
Line Type and Width .............................................. 5-9
Fill Status ............................................................. 5-9
Graphical Objects ..................................................... 5-9
Line Objects .......................................................... 5-9
Circle Objects ......................................................... 5-9
Arc Objects ............................................................ 5-10
Ellipse Objects ....................................................... 5-10
Rectangle Objects .................................................... 5-10
Polygon Objects ...................................................... 5-11
Text Objects .......................................................... 5-12
Vector Text Objects ................................................ 5-12
Subdrawing Objects ................................................. 5-15
Icon, Image, and Pixmap Objects ..................... 5-16
Handling Colors for Icons, Images, and Pixmaps .......... 5-19
Setting Masks for Icons and Images ....................... 5-20
Subobjects of Graphical Objects .............................. 5-21
Point Objects ........................................................ 5-21
Absolute and Relative Points .................................. 5-22
Color Objects ......................................................... 5-23
Color Tables .......................................................... 5-25
Non-Graphical Objects .......................................... 5-26
Drawing Objects ...................................................... 5-26
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deque Objects</td>
<td>5-27</td>
</tr>
<tr>
<td>Node and Edge Objects</td>
<td>5-28</td>
</tr>
<tr>
<td>Slotkey Objects</td>
<td>5-29</td>
</tr>
<tr>
<td>Symbol Tables and Hash Tables</td>
<td>5-31</td>
</tr>
<tr>
<td>Transform Objects</td>
<td>5-34</td>
</tr>
<tr>
<td>Rule Objects</td>
<td>5-35</td>
</tr>
<tr>
<td>Summary</td>
<td>5-36</td>
</tr>
</tbody>
</table>
This chapter discusses the following topics:

- **Object Hierarchy**
- **Control Points**
- **Reference Counts**
- **Attributes**
- **Graphical Objects** - ln, ci, ar, el, re, py, vt, sd, ic, im
- **Subobjects of Graphical Objects**
- **Non-Graphical Objects** - dr, dq, pt, co, no, ed, sk, xf, pm

**Overview**

Objects are the basic building blocks of **drawings**, drawings are the visual components of views. Object types fall into two categories: **graphical objects** and **non-graphical objects**. The graphical objects have associated visual **attributes** such as color and line thickness. They correspond directly to the objects you create in DV-Draw. Non-graphical objects include data structures that manage or extend the graphical objects.

Objects can contain other objects. Most graphical objects contain one or more **control points**, which define their location on the screen. These correspond to the control points that appear when you select an object in DV-Draw.

The primary data structure for managing objects is the **drawing** (dr). Within the drawing, the graphical objects such as arcs, graphs, and input objects use **points** (pt) to control their position and the **attributes** structure (at) to control their visual appearance. Color is the only attribute common to all graphical objects. Color is specified using **color objects** (co).

Non-graphical objects extend the functionality of graphical objects. The drawing manages the front/back order of the graphical objects using a **deque** (dq), which is an ordered list of objects. **Slotkey** objects (sk) let you add arbitrary information, such as comments, to objects. **Transform** objects (xf) provide algebraic information for transforming control points when a
graphical object is rotated, scaled, or drawn on the screen. **Node** (*no*) and **edge** (*ed*) objects let you build and display abstract graphs showing the relationships between data.

In **DV-Draw**, you create the drawing object, control what objects are in the drawing, and control their order. When you create or edit a graphical object, you define the position of the control points, the values in its attributes structure, and its name. Since you can accomplish so much with **DV-Draw**, you may not need to programmatically manipulate objects in your view. However, if you need finer control over objects than **DV-Draw** provides, **DV-Tools** lets you manipulate objects directly and independently of the view.

You may be able to do some things more efficiently in **DV-Tools**. For example, to change a graphical attribute of many different objects in a view, in **DV-Tools** you can use a few lines of code, whereas in **DV-Draw** you must select each object and make the change explicitly.

**DV-Draw** can be used to save time and effort in creating your **DV-Tools** applications. You can create objects in **DV-Draw** and save them in a separate view to avoid creating them programmatically.

Although you can accomplish most of your view creation in **DV-Draw**, **DV-Tools** provides capabilities not available in **DV-Draw**. In **DV-Tools** you can set up deques to manage sets of objects. **DV-Tools** lets you create multiple attribute structures and use them to standardize the attributes of different objects. **DV-Tools** also lets you create or modify objects programmatically, required if characteristics such as size, position, or text string contents are determined at runtime.

### Basic Function Calls

The following steps create a graphical object, in this case a rectangle.

**Create the control points:**

```c
pt1 = VOptCreate (WORLD_COORDINATES, 100, 100, NULL);
pt2 = VOptCreate (WORLD_COORDINATES, 500, 500, NULL);
```

**Create the color objects:**

```c
red = VOcoCreate (COLOR_COMPONENTS, 255, 0, 0);
blue = VOcoCreate (COLOR_COMPONENTS, 0, 0, 255);
```

**Initialize the attributes:**

```c
VOuAtInit (&re_atts);
```
Set the attributes:

```c
re_atts.foreground_color = red;
re_atts.background_color = blue;
re_atts.fill_status = FILL_WITH_EDGE;
```

Create the rectangle:

```c
rect = VOreCreate (pt1, pt2, &re_atts);
```

---

**Manipulating Objects**

Objects are created and manipulated using the VO routines. The two-letter code following the VO is the abbreviation for the particular type of object on which the routine acts. For example, VOcircGet gets the attributes of a circle. There is also a group of VOob routines that works on several different types of objects, usually graphical objects. Most general actions performed on graphical objects can be done using the VOob routines. The notable exception to this is object creation, which must be performed using the VOxx routine, where xx refers to the type of object being created.

Some objects can also be manipulated by the T routines directly, using the Tsc, Tlo, Tdr and Tob routines. Actions are performed at a high level using the T routines or at a lower level using the VOob routines. We suggest using high level routines whenever possible. For example, to draw or erase an object use the TdpDrawObject or TdpEraseObject routines because an object must be drawn within the context of its drawport (dp).

All objects are declared with the same typedef name: OBJECT. This is because objects are self-identifying. Each object type has a defined constant for it such as OT_CIRCLE for a circle. To find the object type you can use the function VOobType.

```c
object = TloGetSelectedObject (location);
if (VOobType (object) == OT_CIRCLE)
  /* the selected object is a circle */
else
  /* the selected object is not a circle */
```

**Object Hierarchy**

Objects can contain other objects which may contain subobjects, and so on in a hierarchical structure. The lowest level objects are point objects which are used as control points for graphical objects. The highest level object is a drawing object, which is the graphical portion of a view and contains all the objects in the view.
Some objects must have a specific number of subobjects of a specific type. A circle object, for example, must have two point subobjects, and an arc object must have three. Other objects can contain any number of subobjects. A polygon object can have an unlimited number of point subobjects, and a deque object can contain any number of subobjects of any type.

This object hierarchy resembles a tree, with the drawing at the root of the tree and the points at the leaves. However, objects can share subobjects, for example, two rectangles can share one control point.

Heaps

When you create objects, DataViews allocates storage for the objects on a heap. Each object has its own memory block in the heap. The initial size of an object’s memory block is determined by an environment variable in your configuration file in the form:

\[ DVxxINITIALHEAPSIZE = \text{number} \]

In this example, \( xx \) is the object type, and \( \text{number} \) is the number of this kind of object for which DataViews initially allocates space.

When the number of objects in your DataViews application exceeds the current limit, DataViews allocates more memory equal to the heap size specified by the configuration variable.

Views load faster if the heap size for each object type equals the number of objects of that type in your application’s views. To determine the number of objects of a particular type in your views, use \( VObgCounts \).

The limit on the number of objects in a DataViews application is 64K for each object type except for absolute point objects. The limit on the number of absolute points is 1M.

Control Points

Control points specify the size and shape of graphical objects. To create a graphical object in DV-Tools, you must create its control point objects first, then use them to create the graphical object. This is similar to creating objects in DV-Draw. The coordinate information of the control point is part of its data structure. These control points can be used to create other graphical objects.

For example, \( TdpDrawObject \) uses control points to calculate the size and position of an object. To change the size or position of an object, change the position of its control points using \( VOptMove \). An object should be redrawn if you move its control points. To access the points of an object use \( VObPtGet \).
Objects Manipulating Objects

Objects can share control points. For example, if two rectangles are always contiguous, they can share a control point. Moving the shared control point changes the shape and position of both rectangles.

Reference Counts

When subobjects are referred to by more than one object, they are said to be multiply referenced. DV-Tools uses reference counts to keep track of objects that can be safely deallocated.

Most DataViews objects have an internal field called the reference count that tracks the number of objects that refer to that object. Whenever a DV-Tools routine attaches one object to another, the reference count of the attached object is incremented. To get the current reference count of an object, use VOobRefCount.

When an object no longer refers to a subobject, the subobject can be dereferenced by calling VOobDereference. This decrements the reference count of the subobject. If the reference count falls to zero, VOobDereference destroys or deallocates the memory used.

Two rectangles with shared control point:

If two rectangle objects share a control point, and you destroy one rectangle, the reference count of its unshared control point falls to zero and the point is destroyed.

The first rectangle is destroyed:

FANUC CNC Parts
The shared control point, however, had a reference count of two. When the first rectangle is destroyed, the reference count of the shared point falls to one and the point is not destroyed. When you destroy the second rectangle the reference count of the shared point falls to zero and the point is destroyed.

A graphical object that is part of a drawing is a subobject of the drawing, and has a reference count of one. If the object has a name, its reference count is incremented to two.

Attributes

Attributes define the visual aspects of an object such as line width, color, and fill status. Attribute information is stored in the graphical object data structure. Each type of graphical object uses a subset of the possible attributes. For example, the text_angle attribute field is used only by vector text objects while the curve type field is used only by polygons.

To access an object’s attributes you use a data structure called ATTRIBUTES. This data structure contains fields for all possible object attributes. To get an object’s attributes, declare a variable of type ATTRIBUTES and call VObAtGet. This copies the attribute information from the object data structure into the ATTRIBUTES data structure. You can then access the individual attributes. To change the attributes of an object, change the fields of the ATTRIBUTES structure and call VObAtSet. For example, to change a polygon’s fill status to EDGE_WITH_FILL use the following calls:

```c
/* declare an ATTRIBUTES structure */
ATTRIBUTES poly_attr;
```
/* get the polygon’s attribute structure */
VOobAtGet (polygon, &poly_attr);

/* set the fill attribute field */
poly_attr.fill_status = EDGE_WITH_FILL;

/* set polygon’s attribute structure to the modified attribute structure */
VOobAtSet (polygon, &poly_attr);

If you change an object’s attributes you may need to redraw the object using Tdp DrawObject. To prevent the old attributes from showing, you can erase the object before redrawing it with its new attributes. For example, if you change a vector text object’s slant, you must erase the original version, then redraw. Otherwise the old version remains visible.

The ATTRIBUTES structure is an intermediate mechanism for accessing an object’s attributes. The actual attribute values are stored in the object data structure. VOobAtGet copies these values into the ATTRIBUTES structure fields and VOobAtSet copies the ATTRIBUTES structure field values into the object data structure. Therefore, a single ATTRIBUTES structure may be used repeatedly to access the attributes of many objects without modifying internal variables.

If a field in the ATTRIBUTES structure is not used its value is EMPTY_FIELD. If the empty field’s type is float, its value is EMPTY_FLOAT_FIELD. VOobAtSet does not change the value of empty fields in the object. VOUAtInit sets all fields in the ATTRIBUTES structure variable to EMPTY_FIELD.

Another way to set an object’s attribute values is to initialize the attributes structure using VOUAttrInit, change the fields directly, and then call VOobAtSet. This changes only the specified attribute fields in the object.

VOxxCreate routines can be used to set the attribute values of an object when you create it. This specifies the initial attribute values the object has when it is created. If the attributes parameter is NULL, the object is created using default attribute values.

You can declare an ATTRIBUTES structure with predefined attribute values by using VOUAttr. This lets you set the fields of an ATTRIBUTES structure without using VOUAtInit or VOxxAtGet. VOUAttr returns a pointer to an internal ATTRIBUTES structure. The values of its fields are set using a variable-length argument list of attribute-value pairs. Because it returns a pointer to an ATTRIBUTES structure, it can be nested within calls to VOobAtSet or VOxxCreate.

Since attributes are not stored as subobjects, they cannot be shared.
The valid attributes for each object are summarized in the following table:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>ar</th>
<th>ci</th>
<th>dg</th>
<th>el</th>
<th>ic</th>
<th>im</th>
<th>in</th>
<th>ln</th>
<th>py</th>
<th>re</th>
<th>sd</th>
<th>tx</th>
<th>vt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc Direction</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Background Color</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill Status</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foreground Color</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line Type</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line Width</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Angle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Text Charspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Text Direction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Text Fontname</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Height</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Text Linespace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Position</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Text Slant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Width</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

**Common Attributes**

Some attributes are common to many types of objects. They include foreground color, background color, fill status, line type, and line width.

**Foreground and Background Color**

The foreground and background colors are defined using color objects, described later in this chapter. In the descriptions of individual objects, foreground and background color are treated as attributes, not as objects.
### Line Type and Width

The line type and width are used to draw the edge of an object. Line type and width are specified using an integer value as defined in `VOstd.h`.

### Fill Status

The fill status options are described in the *DV-Draw User's Guide*. Valid fill status flags are defined in `VOstd.h`.

---

### Graphical Objects

Graphical objects are drawn using attributes to define their appearance. All graphical objects have control points that define their size, shape, and placement within the view. The number of control points varies from one type of object to another. The following descriptions include diagrams that show the control points and attributes for each graphical object. For additional information about the graphical objects, see the *Editing Graphical Objects* chapter of the *DV-Draw User's Guide*.

#### Line Objects

To create a line object, create two point subobjects using `VOptCreate`, then pass the points and attributes to `VOnCreate`, which creates and returns a line object.

![Line Object Diagram](image)

- *begin point (1)*
- *end point (2)*
- *foreground color*
- *attributes*
- *line type*
- *line width*
- *attributes*

#### Circle Objects

To create a circle object, create its center and radius points, then pass the points and attributes to `VOciCreate`, which creates and returns a circle object.

![Circle Object Diagram](image)

- *center point (1)*
- *radius point (2)*
- *foreground color*
- *background color*
- *fill status*
Graphical Objects

Arc Objects
To create an arc object, create its start point, center point, and end point, then pass the points and attributes to `VOarCreate`, which creates and returns an arc object.

![Arc Object Diagram]

Ellipse Objects
To create an ellipse object, create the three control points. The second point is the center. Pass the points and attributes to `VOelCreate`, which creates and returns an ellipse object.

![Ellipse Object Diagram]

Rectangle Objects
Rectangle objects represent rectangles whose sides are horizontal and vertical with respect to the screen.

To create a rectangle object, create its two control points, then pass the points and attributes to `VOreCreate`, which creates and returns a rectangle object.

![Rectangle Object Diagram]
The two control points of a rectangle define its diagonally opposite corners. Since rectangles are always aligned vertically and horizontally with respect to the screen, rotating a rectangle's control points changes the shape of the rectangle: it does not rotate it. To create a rectangle that is not aligned with the screen, use a polygon object with four points.

Polygon objects represent either polygons or B-spline curves. The curve type specifies whether the object is drawn as a normal polygon, or a closed-ended or open-ended B-spline.

To create a polygon object, create point objects for each vertex. Pass the first two points and the attributes to \textit{VOpyCreate}, which creates and returns a two-point polygon resembling a straight line. Then add the remaining points one at a time using \textit{VOpyPtAdd}.

The individual point subobjects in a polygon object are accessed using \textit{VOobPtSet} and \textit{VOobPtGet}. You can determine the number of points in a polygon object by using \textit{VOobPtGet} with a parameter of zero.

If you want to add a point to the end of a polygon whose size is unknown, you can use the following call:

\begin{verbatim}
VOpyPtAdd (polygon, (int)VOobPtGet (polygon,0), point);
\end{verbatim}

You can remove points from a polygon using \textit{VOpyPtDelete}. To delete a point from the end of the polygon whose size is unknown, you can use the following call:

\begin{verbatim}
VOpyPtDelete (polygon, (int)VOobPtGet (polygon,0));
\end{verbatim}

If there is no index-th point, the routine issues an error message. A polygon object cannot have fewer than two points.
A text object represents a string of text using bit-mapped characters, specified in pixels. A text object maintains its absolute size on the screen. The size and appearance of a text object depend on the fonts available on your device.

To create a text object, create an anchor point, then pass the anchor point, attributes, and the NULL-terminated character string to `VOtxCreate`, which creates and returns a text object.

![Diagram showing text object components: anchor point, text string, foreground color, background color, text size, text direction, text justification.]

Text size is specified by an integer from 1 to 4. The number 1 usually corresponds to the smallest available font size; 4 to the largest.

Text direction specifies whether the text is drawn horizontally, or vertically from top to bottom.

The control point, called the anchor point, determines the location of the text object. Text justification specifies how the text is positioned with respect to the anchor point. The horizontal position options are left edge, centered, and right edge. The vertical position options are top edge, centered, and bottom edge. The justification attribute is determined by logically ORing the horizontal and vertical values together.

The character string associated with the text object is accessed using the `VOtxGetString` and `VOtxSetString` routines. The following code sets a text string:

```c
VOtxSetString (text_object, "new_string");
```

**Vector Text Objects**

Vector text objects are similar to text objects, but they are drawn using world coordinate vectors instead of bitmaps. World coordinate vectors are mapped to screen coordinates by the world-to-screen transform. Vector text objects maintain their relative size and position in the drawing and can be rotated, slanted, or scaled independently in the horizontal and vertical dimensions.
To create a vector text object, create an anchor point, then pass the anchor point, attributes, and the NULL-terminated character string to VOvtCreate, which creates and returns a vector text object.

Note that vector text objects do not have a background color attribute. The text direction and justification attributes are the same as those of the text object. The last six attributes use float values to allow continuous variation of the text size, shape, and style.

The font index is an index into an array of fonts, each identified by a font name string. Vector text fonts are internal DV-Tools structures that store the strokes required to draw ASCII characters in the specified style. Fonts are stored externally in font files and are loaded into memory as needed. If your DV-Tools applications uses vector text objects, the necessary font files must be located in the search path specified for TInit. Fifteen standard Hershey fonts are located in the /fonts subdirectory of the /lib directory.

Text height and width control the height and width of the vector text with respect to the text baseline. They are normally set to 1, which creates text with a default size of 1024 world coordinates high.
Text angle controls the angle of the text baseline in degrees measured counterclockwise from the horizontal. The text is rotated around the anchor point.

Text slant controls the degree to which the text is slanted or sheared to the right measured clockwise from the vertical.

Text charspace and linespace control intercharacter and interline spacing. These are normally set to zero and specify a fraction of the character height and character width to be added between characters or lines.

You can force a text object to fit inside a specific bounding rectangle by using \texttt{VOutFitRect}. This supplies the necessary height and width attribute values and anchor point position. This routine does not change the vector text object; it only obtains the information required to do so.

If vector text has been rotated, a rotated bounding box may be preferable to a bounding rectangle. \texttt{VOutGetBound} returns four world coordinate values that represent the rotated boundary of the vector text object. This boundary can be a rotated rectangle of any shape, size, and orientation. It is defined by two perpendicular vectors extending from the lower left corner of the text to the upper left corner and from the lower left corner to the lower right corner. Each vector is defined by a point relative to the anchor point; \( wx, wy \) defines the width vector, \( hx, hy \) defines the height vector. This yields a tighter boundary than the \texttt{VOobBox} routine, which only gives the minimum horizontal and vertical extent.

The vector text character string is accessed using \texttt{VOutGetString} and \texttt{VOutSetString}. The following code sets a vector text string:

\begin{verbatim}
VOutSetString (vector_text_object, "new string");
\end{verbatim}
Subdrawing objects represent whole views within drawings. When you create a subdrawing, a transform is calculated based on the scale, rotation, and position of the subdrawing object. The transform is used to draw the subdrawing’s graphical subobjects.

A subdrawing includes all the dynamics of its original view. The handling of these dynamics is discussed in the *Dynamics* chapter.

To create a subdrawing object, create an anchor point, then pass the anchor point, view or view filename, scale, and color attribute to *VOsdCreate*, which creates and returns a subdrawing object. The control point corresponds to the center of the subdrawing’s coordinate system and determines the position of the subdrawing within the drawing. The scale factor determines how the subdrawing’s coordinate system is mapped into the parent drawing’s coordinate system.

If you plan to save a subdrawing in a view, you must specify whether it is **included** or **referenced** using *VOsdViKeep*. A referenced subdrawing contains the name of the view file in which the referenced view is stored, so it requires a filename. An included subdrawing copies the referenced view into the subdrawing’s internal view structure, so a filename is optional. For more information about included and referenced subdrawings, see the *DV-Draw User’s Guide*.

DV-Tools lets you modify the structures in a subdrawing, which cannot be done in DV-Draw. You must first get the view using the following call:

```c
view = VOsdViGet (subdrawing);
```

You can extract the objects and structures from the subdrawing’s view using techniques discussed in the *Display* chapter and in the *Drawing Objects* subsection later in this chapter. The example *find_subobj* in the *fragments* subdirectory under the DV-Tools *examples* directory shows how to determine which graphical subobject of the subdrawing has been selected.
DataViews uses an internal table called the pool to make loading of referenced views more efficient. Whenever a referenced subdrawing is created or loaded, the referenced view filename is added to the pool. A view whose filename is listed in the pool is not loaded again. Instead, DataViews clones the view attached to an existing subdrawing. Because of this, you need to bypass the pool in the following situations:

If you want to programmatically modify the view belonging to a referenced subdrawing, first remove the subdrawing from the pool by calling VOsdPoolRemove. Any changes in the view are then confined to the individual subdrawing.

If you want to load the latest version of a view file that may already be in the pool, first call VOsdPoolFnmRemove. This removes the entry in the pool for the specified filename. The next time the filename is referenced, the view is loaded from the file again and the filename is added back into the pool. For example, if you call TviSave to save a new version of a referenced view, you can call VOsdPoolFnmRemove immediately afterward.

To bypass the pool for all subdrawings, set the DVSD_DEACT_POOL configuration variable to YES. This ensures that all referenced subdrawings load the current version of the view file, and you never need to call VOsdPoolRemove or VOsdPoolFnmRemove for special cases. However, the additional load time may significantly slow application startup or display.

Icon and image objects display raster images within drawings. The icon object displays a raster with a single control point, which stays a fixed size on the screen even when the icon object’s drawport is zoomed. The image object displays the raster within a rectangle defined by two control points, which changes size on the screen when the image object’s drawport is zoomed. An image object is similar to a filled rectangle, but the fill is a picture instead of a solid color. An image object can be resized by moving its control points or by resizing the drawport.

Icon objects are useful as labels or icons that mark parts of a drawing because they do not scale. Image objects are useful for pictures or backgrounds because they do scale with their surrounding objects. For example, an image object should be used for a map that serves as the background for the lines representing a network. When the drawport is zoomed to focus on a region of the network, its background maintains the correct aspect ratio with respect to the lines.
Icon and image objects refer to a third object, the pixmap object. The pixmap object is a non-graphical object that contains a pixel-based description of a source picture. Currently, the source can be a file in a compatible format or in-memory data. The in-memory data must contain a raster created using GRrraster routines or data in a compatible graphic format or pixrep structure.

Compatible pixel formats include the GIF format of Compuserve Corporation, the PPM format of Jef Poskanzer, and the TIFF format of Aldus/Microsoft. The TIFF classes supported by DataViews are:

<table>
<thead>
<tr>
<th>TIFF Class</th>
<th>Image Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class B</td>
<td>1-bit black-and-white images</td>
</tr>
<tr>
<td>Class G</td>
<td>grayscale images</td>
</tr>
<tr>
<td>Class P</td>
<td>color images using color tables</td>
</tr>
<tr>
<td>Class R</td>
<td>color images using RGB values</td>
</tr>
</tbody>
</table>

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 or to a pixrep structure. For more information on the pixrep structure, see the VUpixrep module in the DV-Tools Reference Manual.

The pixmap is a format-independent object. The icon and image objects describe how the pixmap shows up on the screen. If you want to display a pixmap more than once you can save memory by having multiple objects refer to the same pixmap. For example, icon and image objects can process pixmap information and display it in different sizes on different parts of the screen:
To create a pixmap object, pass a filename or the address of the in-memory data to `VOpmCreate`, which creates and returns a pixmap object. A filename must refer to a file containing a picture described in a compatible format. A raster must already have been created using `GRrasget` or `GRrascreate`.

```
+----------------+................+           +----------------+................+
|   pm            |    pm             |           |   pm            |    pm             |
|                  |                  |           |                  |                  |
|  rows            |    version number |           |  columns         |    color depth    |
|                  |                  |           |  int             |    int           |
| pixel array      |                  |           |  pixel array     |    color table   |
|                   |                  |           |  R               |    R             |
| sizes            |                  |           |  G               |    G             |
|                  |                  |           |  B               |    B             |
|                  |                  |           | "file"           |                  |
```

To create an icon object, pass a pixmap object, an anchor point, attributes, and a list of settings for any other characteristics to `VOicCreate`, which creates and returns an icon object. The `TEXT_POSITION` attribute determines how the raster image is justified with respect to the anchor point. The foreground and background colors can be set, but they have no effect.

```
+----------------+                        +----------------+                        |
|    pm           |  attributes                  |    pm           |  attributes                  |
|                  |                           |                  |                           |
|                  |  foreground color          |                  |  foreground color          |
|                  |  background color          |                  |  background color          |
|                  |  text position             |                  |  text position             |
|                  |  mask info                 |                  |  mask info                 |
|                  |                           |                  |                           |
| anchor point     |  size parameters           |                  | size parameters           |
|                  |                            |                  |                            |
|                  |  height                    |                  |  height                    |
|                  |  width                     |                  |  width                     |
| color mapping    |                            |                  |                            |
| COLOR_XFORM      |                            |                  |                            |
|                  |                            |                  |                            |
|                  |                            |                  |                            |
```

To create an image object, pass a pixmap object, two control points, attributes, and a list of settings for any other characteristics to `VOimCreate`, which creates and returns an image object. The `TEXT_POSITION` attribute determines which point in the raster remains stationary when the image is scaled using `VOimScalePixmap`. The foreground and background colors can be set, but they have no effect.
The color mapping controls which colors in the screen’s color table are used to draw the icon or image. Colors are discussed in the next section, *Handling Colors For Icons, Images, and Pixmaps*. Mask information is discussed in *Setting Masks for Icons and Images*.

While the image or icon handles the raster that is displayed, the pixmap object handles pixel-based data on which the raster depends. A pixmap gets its data from files if it is referenced or from internal memory if it is included. Included pixmaps can make your view file very large since they can be substantially larger than pixel files; using referenced pixmaps can reduce the size of your view. You can manipulate pixmaps in various ways. For example, you can flip, rotate, and merge pixmaps. You can also clip a portion of a pixmap to create a new, smaller pixmap, which is more efficient to handle. If you change a pixmap, you can save the changes back to its file so that the changes are reflected in any pixmaps that reference that file the next time it is loaded.

A pixmap has its own color table, separate from the device color table, containing its original colors. The pixmap also contains its pixel array. Each pixel in the array specifies its color using an index into the original color table. However, the color table of the original file or raster referenced by the pixmap may not match the current color table associated with the screen. The actual colors used to draw the icon or image come from the screen’s color table. So, to draw the pixels using the current device color table, the pixmap color indices must be mapped to the device color indices. The color mapping information is kept by the icon or image since the mapping is specific to the graphical object and its relation to the screen in which it appears.
DV-Draw provides two ways of handling colors. You can map the original colors to the best match in the current color palette, or you can set the palette to the original colors and no mapping is required. Mapping occurs automatically when the `DVMATCH_COLORS` variable is set to `YES`. Mapping is accomplished by setting up a color transform that maps the original colors to the best matches in the color palette.

Both of these methods are available in DV-Tools. To match colors, call `VUncTransform` passing the pixmap’s color table as the source color table and the screen’s color table as the destination. `VUncTransform` returns a `COLOR_XFORM` structure that you can use for mapping the icon or image colors. To use the original colors without mapping, get the pixmap’s color table and set it as the screen’s color table using `TscOpenSet` (when you open the screen) or `GRs_color_table` (if the screen is already open).

Alternatively, you can create your own color transform. Then, once you have a color transform, attach it to the image or icon.

If you display several icons or images on the same screen, you should provide a color table suitable to all of them. After creating the pixmap objects, pass the pixmaps to `VOpnBestColors`, which weights the importance of each color based on the frequency of its appearance in the pixmaps and returns a good compromise color table for matching the colors in all of the pixmaps. Alternatively, you can create your own color table.

You can improve the colors in the display by using dithering. Dithering increases the visible range of colors by closely interspersing them. This method is used by all color monitors to make the 128 or 256 colors out of the three primary colors. Dithering is performed on pixmaps by calling `VOpnNewColorTable`. Note that `VOpnNewColorTable` permanently changes the pixmap. It can decrease detail and introduce unexpected patterns, so you may want to test it on cloned pixmaps before applying it to the pixmaps in your application.

The mask information for icons and images lets you control which portions or colors of the raster are drawn. Whatever is not drawn is transparent. Setting the mask information is optional; if the information is not set, the entire raster is drawn and no parts are transparent. DV-Tools lets you mask the icon or image using a pixel-based mask (called the mask pixmap) and an optional color transform for interpreting the mask. The mask pixmap is in addition to the main pixmap associated with the icon or image.
The mask pixmap lets you make arbitrary pixels transparent. This can only be done in DV-Tools. To do this, you need to create or clone a pixmap. The color value must be zero for the pixels corresponding to those in the main pixmap that should be transparent. You can edit individual pixels in the mask pixmap to change their value. The mask pixmap and the main pixmap do not have to be the same size since the mask pixmap is automatically extrapolated to fit the main pixmap. This mask pixmap doesn’t change or replace the main pixmap; it only signals which pixels not to draw.

The masking color transform lets you change how the mask pixmap is interpreted. Since the mask is a pixmap, it has a color index for each pixel. The masking color transform maps those color indices to new color indexes, so that color[10] or color[56] in the mask could be reinterpreted as color[0]. This lets you use the mask to make any number of colors transparent.

An example of this kind of masking is the “Mask” option in DV-Draw. Each image and icon in DV-Draw has an associated (main) pixmap which it also uses as the mask pixmap. It also has a masking color transform that maps all colors to color[1] so that all colors are initially drawn. When you select a color in the image or icon to mask, you map that color index in the mask pixmap to color[0]. Since the main pixmap and the mask pixmap are identical, the corresponding color in the icon or image becomes transparent.

Point objects are the most fundamental object in DV-Tools. Point objects represent physical points in the two-dimensional space of a drawing and are stored in world coordinates. When a point object becomes part of a drawport its world coordinates are mapped to screen coordinates by the drawport.
Point objects are used as control point subobjects for graphical objects. For example, a rectangle object contains two control points which define its position, size, and shape in the drawing. Point objects are useful for diagnostic purposes.

Point objects do not have attributes, but they can be drawn. They appear on the screen as crosses in the drawing foreground color. In DV-Draw, the control points of selected objects appear as unfilled rectangles. To draw a dot with attributes such as color, use a rectangle where the two control points coincide.

Absolute and Relative Points

Point objects can be absolute or relative. An absolute point object contains its position specified in world coordinates. In DV-Draw you can only use absolute control points.

A relative point object contains a reference point object and its position specified as an offset from the reference point. Offset values can be specified in world or screen coordinates. A relative point always maintains the same position with respect to its reference point. If you scale relative points that are specified in world coordinates, the distance between them scales proportionally. If you scale relative points that are specified in screen coordinates, the distance between them does not scale.

Relative offsets in screen coordinates are useful for creating objects that have a fixed pixel size. For example, if you want a two-pixel shadow around an object where the shadow does not expand with the object, use a relative pixel offset. If you want the shadow to scale with the object, use a world coordinate offset. To surround a vector text object with a frame that always has the same size in screen coordinates, make the frame’s second control point relative to the first by the size of the text bounding box.

When creating a point object you must indicate whether it is an absolute or relative point. If it is a relative point, you must indicate whether it is specified in world or screen coordinates.

For example, to create the center point for a circle as an absolute point use:

```c
center_pt = VOptCreate (WORLD_COORDINATE, 100, 100, NULL);
```

When creating a relative point, if `delta_x` and `delta_y` are specified in world coordinates, use:

```c
radius_pt = VOptCreate (WORLD_COORDINATE, delta_y, delta_x, center_pt);
```
When creating a relative point, if $\delta_x$ and $\delta_y$ are specified in screen coordinates, use:

$$\text{radius_pt} = \text{VOptCreate} \left( \text{SCREEN_COORDINATES}, \delta_x, \delta_y, \text{center_pt} \right);$$

The coordinates of a point object are stored in a point structure (typedef `DV_POINT`) and are accessed by calling `VOptGet`. If a relative point is specified in screen coordinates, the coordinates are returned in two parts: a world coordinate point and its offset in screen coordinates. The actual coordinates of the point are determined by converting the `wpt` point structure into screen coordinates using `TdpWorldToScreen` 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 relative point.

To move a rectangle use the following calls:

```c
DRAWPORT drawport;
OBJECT rect_obj, point;
int delta_x, delta_y, i;

TdpEraseObject (drawport, rect_obj);
for (i = 1; i <= VOobPtGet (rect_obj, 0); i++)
{
    point = VOobPtGet (object, i);
    VOptMove (point, DV_RELATIVE, delta_x, delta_y);
}
TdpDrawObject (drawport, rect_obj);
```

Point objects are static. However, point objects can be moved by using `VOptMove`. Absolute points can be moved by an absolute or relative amount. (This should not be confused with the distinction between absolute and relative points.) Relative points can be moved by changing their offsets.

The DV-Tools drawing routines transform points’ world coordinates to screen coordinates before drawing graphical objects. You can also transform coordinates directly by using `VOptXfGet`.

**Color Objects**

Color objects represent colors of graphical objects. Every graphical object contains at least one color object indicating its foreground color. Most graphical objects also contain a background color object.
There are three types of color objects. The first is an RGB color object, with
the object type id \textit{OT\_RGB}. It specifies the color using three bytes of red,
green, and blue intensity values in the range \([0,255]\).

\begin{center}
\begin{tikzpicture}
  \node (co) {co};
  \node (red) [below left of=co] {red};
  \node (green) [below right of=co] {green};
  \node (blue) [below of=green] {blue};
  \draw (co) -- (red);
  \draw (co) -- (green);
  \draw (co) -- (blue);
\end{tikzpicture}
\end{center}

The following code fragments illustrate two ways to create a red color object:

\begin{verbatim}
color_obj1 = VOcoCreate (COLOR_COMPONENTS, 255, 0, 0);
\end{verbatim}
or
\begin{verbatim}
color_obj1 = VOcoCreate (COLOR_NAME, "red");
\end{verbatim}

The second type is an index color object, with the object type id \textit{OT\_COLOR}.
It specifies the color using an index into the device’s color lookup table. The
index indicates that the following three bytes contain a 24-bit integer index in
a device’s color lookup table.

\begin{center}
\begin{tikzpicture}
  \node (co) {co};
  \node (index) [below of=co] {index};
  \node (lookup) [right of=index] {Device’s color lookup table};
  \draw (co) -- (index);
  \draw (index) -- (lookup);
\end{tikzpicture}
\end{center}

The following code creates a color object that indexes to the most saturated
red in the color palette.

\begin{verbatim}
color_obj1 = VOcoCreate (COLOR_INDEX, 8);
\end{verbatim}

The third type is a referenced color object, with the object type id \textit{OT\_REFCOLOR}.
It is a color object that points to another color object. This
is the only type of color object that maintains a reference count. This allows
Multiple references, so that when its color changes, all objects that refer to it change color. This lets you create one color object and use it in several objects.

```
 referenced color object

 Change this color object to change
 the color of the circle and rectangle
```

The following code creates a referenced color object using \textit{color\_obj1}:

```
color\_obj2 = \texttt{VOcoCreate (COLOR\_REFERENCE, color\_obj1);}
```

To change the color of a graphical object that uses a referenced color object you must change the color object that the referenced color object points to.

An object’s colors are always saved in RGB format. Colors are converted to color indices when views are loaded. Objects with \textit{INDEX} color types draw more efficiently than RGB values. RGB colors must be converted to an \textit{INDEX} before the object can be displayed.

\section*{Color Tables}

All color objects ultimately refer to the current color table, which is a structure of RGB values. Each triplet of RGB values sets the color for one index. The color table can be set up explicitly or read in from a file. For example, the following fragment from \texttt{default.clut} sets the first three colors to black, light blue, and dark blue respectively:

```
0 0 0
0 210 255
0 0 255
```

Color tables can have any number of multiples. You can create your own color tables and load them when you open a screen using \texttt{TscOpenSet}. To change the color table in an existing screen, call \texttt{GRs\_color\_table}.

Example color tables are located in the \texttt{etc} directory, including \texttt{grey128.clut}, a gray scale color table.

Colors based on color indices are closely tied to the color table. For example, if you load a color table whose first values are (255,255,255) in place of a color table whose first values are (0,0,0), all the objects that used the index 1 color object change from black to white. Colors based on RGB values do not change as much when new color tables are loaded, since DataViews maps the
RGB color objects to the closest color in the table. The determination of the closest color is based on an arithmetic algorithm that may not produce the best “by eye” matching.

Some display devices reserve colors, so a different color is substituted in the display for the specified color. If this happens, you can find out what the substitute color is by using `GRg_real_color_tab`.

---

**Non-Graphical Objects**

**Drawing Objects**

A drawing object represents the graphical portion of a view. It contains a deque of graphical objects and a deque of pointers to object-name structures. The graphical object deque maintains a priority list of all the graphical objects in the drawing. The name deque maintains a list of all the named objects and their names.

The attributes of a drawing object are foreground color, which is used to draw objects that have no foreground color of their own, and background color, which is used to erase objects in the drawing. Drawings with their background color set to `NO_BACKGROUND` act as if their background is transparent. They are erased in the current background color of the device.
A drawing can contain many kinds of graphical objects. Drawing objects belong to views, which can be contained by one or more drawports. Therefore, many operations on drawings can be performed at the T level by the Tdr, Tdp, and Tvi routines.

To extract graphical objects from a drawing object, you can use TdrGetNamedObject, passing in the name of the object, or you can traverse the drawing’s objects using VOdrTraverse, TobForEachSubobject, or TdrForEachNamedObject. You do not need to get the drawing’s deque since you can access the graphical objects and their names directly using Tob, Tdr, and VOdr routines.

**Deque Objects**

Deque objects are lists used to maintain the order of other objects. For example, drawing objects maintain their contents using deques of graphical objects, and graphical objects maintain their rules on a deque. Deque objects can be used by DV-Tools to implement other types of objects. Deque objects can be used to store priority lists of objects. Deques can also be used to manage lists of non-objects that fit into a LONG. You may find it helpful to maintain lists of the following types of objects in your application:

- objects that share functionality, such as objects that are drawn or erased together
- visible objects so you can perform intersection tests more efficiently
- dynamic objects so you can update more efficiently using TdpDrawNextObject instead of TdpDrawNext
- deleted objects so you can undelete them easily
Non-Graphical Objects

Node and Edge Objects

The deque object acts like a linked list because you can add or delete objects from anywhere in the list. Objects are deleted by their object id or by their position in the deque. You can also insert and delete deques of objects. The deque cannot have duplicate entries. Objects are 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. The bottom object of the deque has the lowest priority and the top object has the highest. In drawings, higher objects obscure the ones below them.

The objects in a deque are treated as subobjects of the deque. Therefore, `TobForEachSubobject` traverses all objects in a deque from bottom to top. As with all subobjects, the objects in the deque can be shared with other deques.

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 examples of this kind of graph are mapping the shortest routes between objects, project planning, and electrical circuit analysis. Edge and node objects are provided as application modelling tools for the DataView environment.

A node can contain multiple edge objects; an edge object can be connected to zero, one, or two node objects. Each node and edge object maintains a mark bit that indicates whether the object has been traversed, and a visit count used to analyze your abstract graph.

Each node and edge object can be graphically represented by a `geometry object` so you can also draw the structures constructed with the node and edge objects. The geometry object must be a graphical object or a deque of graphical objects. You can manipulate the control points and attributes of the
geometry object through the node or edge using routines such as \( VOnoPtSet \) or \( VOnoAtSet \). The number of points and the valid attribute fields are determined by the associated geometry object.

**Slotkey Objects**

You can attach arbitrary information to your objects using slots. A slotkey object is the means of setting up a slot and subsequently retrieving or changing the information in the slot. Slotkey objects include a name and a description of the type of information. The arbitrary information can be an integer, an array of integers, a float, an array of floats, an object, or a pointer to a NULL-terminated string.

Slots are an object level implementation of symbol tables, which are discussed in the next section. Slots are easier to use than symbol tables because they are created and maintained for you as part of the object. The information that you enter in the slots is saved with the object when you save the view and can be extracted when the view is loaded again.
The following diagram shows the structure of a slotkey object and illustrates how information is attached to an object using a slot:

```
  slotkey object
     sk
    /      \
   name    type

   ob
     sk         sk
    flag    flag
   value    value
     Data     Data
   ...     ...

slots attached to an object

  sk
   flag
  value
   Data
```

Two routine modules manipulate slots. The VOslotkey routines let you declare a slot or find a slot using a slotkey name. The VOobSlotUtil routines let you attach a specific slot to an object, place information in a slot, retrieve a slot by index, delete a slot, or determine if an object has a slot or supports adding slots. The VOslotkey routines let you declare a generic form of slot. The VOobSlotUtil routines let you create and work with specific instances of slots. Both groups of routines must be used together in order to use slots effectively.

Slotkeys are declared rather than created. To declare a slotkey, pass a unique key name and type to VOskDeclare. For array types, you must also pass a size. The following line of code declares a slotkey that contains a pointer to an object:

```
  bndsk = VOskDeclare ("BOUNDARY", VOSK_OBJECT_TYPE);
```

The following code opens a slot for boundary information in a drawing object and fills in the information:

```
  VOobSetSlot (drawing, bndsk, (LONG *)&rect_obj,
               (ULONG *) 0);
```
The last parameter in the call to \texttt{VObSetSlot} is a flags field that is used to store information such as access counts or semaphores, or to keep track of whether the slot has been accessed, changed, or initialized.

Use \texttt{VObGetSlot} to retrieve the contents of a slot. For example, the following code gets the information in the boundary slot \texttt{bndsk} from \texttt{drawing} and stores it in the value \texttt{in_obj}:

\begin{verbatim}
    VObGetSlot (drawing, bndsk, (LONG *)&in_obj, (ULONG *)0);
\end{verbatim}

You can use the same slotkey to store similar information in more than one object. For example, you can declare a single slotkey to assign revision numbers to your objects:

\begin{verbatim}
    revsk = VOsDeclare ("REVISION", VOSK_INT_TYPE);
\end{verbatim}

To open a slot and assign a revision number for an object, pass this slotkey and the object to \texttt{VObSetSlot}. Use the same slotkey to assign revision numbers to all your objects. You can subsequently change the revision number for a specific object without affecting the revision numbers of the other objects.

\section*{Symbol Tables and Hash Tables}
Symbol tables and hash tables let you handle information by building tables of the information together with keys for retrieving the information. These tables are not objects, but are structures that you can create and maintain in your application program using VT routines. For example, each drawing object uses a symbol table as the name table that associates the objects with their names. The DataViews software also uses symbol tables to maintain the addresses of its functions. You can use symbol and hash tables in similar ways.
To create symbol and hash tables, you must pass in keys and associated information. A function attached to the table assigns the information and key to a node in the table. The information is retrieved using the key or the index of the node.

**symbol table**

```
Comparison Function
   \rightarrow st \rightarrow name

Symbol \rightarrow no \rightarrow Value

Symbol \rightarrow no \rightarrow Value
```

**hash table**

```
Convert Key Function
   \rightarrow ht \rightarrow name

Comparison Function

Key \rightarrow no \rightarrow Value

KeyCode \rightarrow no \rightarrow Value

Key \rightarrow no \rightarrow Value
```

Symbol tables and hash tables differ primarily in how they assign, access, and traverse their nodes. Symbol tables usually use strings as keys, which are maintained in alphabetical order. Hash tables usually use `ULONGs` as keys, which are maintained in an arbitrary order determined by the hashing.
function. Use symbol tables if you must maintain information in some order and if you access the table more frequently than you modify it. Use hash tables if order is unimportant or if you modify the table frequently.

Both symbol and hash tables use functions to determine the mapping from the key to the node. By default, symbol tables use an alphanumeric ranking to map the key strings to nodes and hash tables use the ULONG value of the keys. However, you can use VThststrconvert to convert strings into ULONG keys for hash tables. For either type of table you must provide unique keys or a function to determine the mapping of identical keys to different nodes.

Note that pointers to nodes become invalid when you insert them into a symbol table since the table is an array and adding or deleting entries changes the position of the nodes. If you retrieve a node from the table, then add or delete entries from the table, you must retrieve the node again in order to use it.

The following code adapted from the example symload.c shows the use of a symbol table to store drawports containing preloaded views:

```c
/* Create the symbol table. */
NameTable = VTstcreate ("Object Names and Drawports",
    NULL);
VTstsniinsert (NameTable,
    VUstrclone (first_view_name),
    (int *)first_drawport);
TdrForEachNamedObject (TviGetDrawing (first_view),
    AddDrawports, (ADDRESS) NULL);

/* Add drawports to the symbol table. The object name corresponds to the view name. */
ADDRESS
AddDrawports (object, view_name, argblock)
    OBJECT object;
    char *view_name;
    ADDRESS argblock;
{
    DRAWPORT drawport;
    VIEW view;

    /* See if the view is in the symbol table. If not, load it and add it to the symbol table. */
    if (strcmp (view_name, "quit") == 0)
        return V_CONTINUE_TRAVERSAL;
    // Add drawports to the symbol table...
    // ...
Non-Graphical Objects

Transform Objects

Transform objects represent homogeneous linear transformations on two-dimensional vectors. They are used to change two-dimensional points from one coordinate system to another, to scale, rotate, stretch, or shear collections of points arbitrarily. Drawports maintain two transform objects that represent the world-to-screen and screen-to-world transformations.

The following code fragment illustrates how to scale a polygon around its center point using a transform object.

```c
/* create a transform object with no scale */
/* the transform will translate the polygon to the origin of the coordinate system */
xform1 = VOxfStCreate (1.0, -centerpt.x, -centerpt.y);

/* create a transform object that scales the polygon and translates it back to its original position */
xform2 = VOxfStCreate (scale_factor, centerpt.x, centerpt.y);
```
/* create a concatenation of xform1 and xform2 */
xform = VOxfCatCreate (xform1, xform2);

/* apply xform to each point in the polygon */
for (i = (int) VOobPtGet (poly,0); i>0; i--)
{
    pt_obj = VOobPtGet (poly, i);
    VOptGet (pt, &wcs_pt, &dummy);
    VOxfPoint (xform, &wcs_pt);
    VOptMove (pt_obj, DV_ABSOLUTE, wcs_pt.x, wcs_pt.y);
}

**Rule Objects**

To make your DV-Tools program perform a rule action, the program must run the prototype in its own drawport or must be written to interpret the components of the rule.

A rule has three components: an event, a condition, and an action. The event component specifies what type of event triggers the rule; the condition specifies the conditions under which the event triggers the action. The event is identified by a simple type flag: the condition and action descriptions are more complex and contain additional arguments. The rule object can be one of several rule objects in a table maintained by the graphical object or drawing object.

For more information on running prototypes in DV-Tools, see the `Tproto` module in the *DV-Tools Reference Manual*. 
Summary

This chapter presents these basic techniques:

How to create and manipulate graphical objects.
How to use deques, symbol tables, and hash tables to handle multiple objects or data.

Using these techniques, you can implement the following features in your application:

Create special objects as needed in your application.
Access and manipulate the objects already in the views.
Build deques of objects and act on deques as an alternative to calling the same action for many individual objects.
Overview ............................................................................................... 6-1
What to do in DV-Draw ................................................................. 6-2
Useful Data Structures ................................................................. 6-2
Data Flow ......................................................................................... 6-3
Basic Function Calls ....................................................................... 6-4
Data Sources ...................................................................................... 6-5
Data Source Lists ........................................................................... 6-5
Data Sources .................................................................................. 6-5
Data Source Variables .................................................................. 6-6
Variable Descriptors ......................................................................... 6-8
Contents of a Variable Descriptor ................................................. 6-8
Rebinding Variable Descriptors ..................................................... 6-9
Graphs ................................................................................................. 6-10
Data Group Objects ........................................................................ 6-10
Data Groups ..................................................................................... 6-11
Display Formatters .......................................................................... 6-12
Context Information ......................................................................... 6-12
Graph Feedback ................................................................................ 6-15
Graph Feedback Examples ............................................................ 6-16
Object Dynamics ................................................................................. 6-22
Dynamic Control Object .................................................................. 6-22
Sharing of Dynamic Control Objects ............................................ 6-24
Variable Descriptor Objects ............................................................ 6-25
Threshold Table Objects ................................................................. 6-25
The Subdrawing Dynamics Feature .............................................. 6-27
The Internal Dynamics of Subdrawings ............................................. 6-28
Getting Data from the Original Data Sources ............................... 6-29
Mapping to New Data Sources ....................................................... 6-29
Connecting Internal Dynamics to Application Data ....................... 6-30
Stuffing Buffers for Mapped Variables .......................................... 6-30
Rebinding for Unmapped Variables ............................................... 6-30
Rebinding for Mapped Variables ................................................... 6-32
Blinking .............................................................................................. 6-33
Overview of Blinking Behavior ...................................................... 6-33
Color Dynamics Method for Blinking ............................................ 6-34
Low-Level Blinking Method ............................................................ 6-34
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Data Source Lists</td>
<td>6-36</td>
</tr>
<tr>
<td>Dynamic Data Flow Example</td>
<td>6-38</td>
</tr>
<tr>
<td>Summary</td>
<td>6-39</td>
</tr>
</tbody>
</table>

DV-Tools User’s Guide
Overview

The application communicates with the user by generating data and sending it to the interface, where it is displayed in the form of dynamics. One of the most powerful features of DataViews is its ability to treat dynamics as part of the objects in a view, rather than as actions that must be performed by the application. The view contains information about the data that animates it: where it is, what it is, and what variables it represents. Each dynamic object contains information on how to respond to this data.

DataViews dynamics occurs in two stages: gathering data and displaying data. In the gathering stage data is gathered from outside DataViews and placed in buffers. Views manage data gathering at the highest level using data source lists. These contain data source structures which identify and describe where the data comes from. Each data source contains data source variables which maintain buffers to receive each data value. Users can gather their own data and manage their own data buffers.

In the display stage DataViews transforms the data into visible changes in the graphical images. A variable descriptor describes the data that is to be translated into dynamics.
Graphs interpret information in the data group structure to define the display of the data from the variable descriptor. The data group structure controls context information such as axes, ticks, labels, and grids, and carries the data and context information to the display formatter. The display formatter is a set of internal routines that control the drawing and updating of graphs. The data group structure is attached to a data group object which is a graphical object that has control points and attributes.

Input objects are dynamic objects. They are discussed in the Input Objects chapter.

Simple graphical objects with dynamics use dynamic control objects. A dynamic control object is an independent object that changes the attributes or control points of a graphical object according to changes in data. Dynamic control objects can also animate subdrawings, display the variable contents, and control whether or not a graphical object is visible. Dynamic control objects get their data from threshold table objects or variable descriptor objects which in turn get their data through variable descriptors. Threshold table objects take data anywhere within a range and map it to a value. A variable descriptor object (vd) contains a variable descriptor structure (vdp) and passes data without modification.

What to do in DV-Draw

To create dynamics in DV-Draw, you must set up the data sources in your view by specifying the names, types, origins, and descriptions of the data sources. You can name and describe the shape and type of data of the data source variables. The order of the variables within the data source determines the flow of data. You can associate a data source variable with a variable descriptor and describe the variable descriptor attributes used to interpret the data source variable.

In DV-Draw you can define graphs (dg) by editing their graph type (df), context (dgp), and graph variable (vdp). For each graph variable, you can edit the marker, line type, color, range, log or linear scale, and color thresholds. You can also attach a data source variable that supplies the data.

You can define dynamic feature sets (dy) and attach them to graphical objects. You can attach their variable descriptors to data source variables that supply data. You can also edit threshold table objects.

Useful Data Structures

The data structures for gathering data are the data source list (dl), the data source (ds), and the data source variable (dsv).
The data structures for displaying graph dynamics are the data group object (dg), the data group structure (dgp), the display formatter (df), and the variable descriptor (vdp).

The data structures for displaying object dynamics are the dynamic control object (dy), the threshold table object (tt), the variable descriptor object (vd), and the variable descriptor (vdp).

---

**Data Flow**

The data flows from files, processes, or functions into DataView data sources. Data then flows from the DataView data sources to the graphical objects, as shown by the large arrow in the figure above. The smaller arrows in the diagram show the relationships between the data structures.

To begin reading data you must call `TviOpenData` or `TdlOpenData` in the initialization. DataView then opens files, spawns processes, and starts functions as required. `TviReadData` or `TdlReadData` and `TdpDrawNext` control the data flow in each update cycle. In the data gathering stage, `TviReadData` or `TdlReadData` reads in enough data to fill the buffers associated with the data source variables. If `TviReadData` or `TdlReadData`
reaches the end of a file, it continues to use the last data in the buffer. In the
display stage, TdpDrawNext traverses the drawing list looking for dynamic
objects. If a dynamic object is found, its variable descriptors check for
changes in the data buffers. If there are new values, TdpDrawNext sends them
to intermediate structures (vd, tt, dy, dg) for interpretation. Then DataViews
draws the graphical changes to reflect the new data.

The Tvi*Data routines are commonly used more than the related Tdl or Tdp
routines because they act on all the data sources required by the view,
including data sources required for the internal dynamics of subdrawings. If
there are no active subdrawings in your view, you can use the Tdl*Data
routines which act only on the data sources in the view’s data source list,
which may not contain data sources referenced by subdrawings.

In addition to this standard data flow, DataViews supports writing data out
from data sources and reading data in for individual data source variables.
However, these special features are supported only for certain types of data
sources. For more information, see the descriptions of TdsWriteData,

---

**Basic Function Calls**

The following function calls gather data and display updated dynamics. The
data sources available in DV-Draw are file, process, constant, memory, and
function; gathering data from other types of program sources is discussed
later in this chapter.

**In the initialization phase:**

Open the data sources:  
TviOpenData (view);  
or  
TdlOpenData (masterdsl);

**In the run phase:**

Read in one set of data:  
TviReadData (view);  
or  
TdlReadData (masterdsl);

Display updated dynamics:  
TdpDrawNext (drawport);

**In the termination phase:**

Close the data sources:  
TviCloseData (view);  
or  
TdlCloseData (masterdsl);
Data Sources can be created in DV-Draw. You can also create and manipulate data sources in DV-Tools.

**Data Source Lists**

The **data source list** (dl) is a list of data sources. A view created in DV-Draw has a data source list. A view created with *TviCreate* contains an empty data source list.

Data source lists are extracted from the view using *TviGetDataSourceList*. You can traverse the data sources and their variables using *TdlForEachDataSource* and *TdlForEachVar* respectively.

The data source list extracted from a view does not include the data sources referenced by the internal dynamics of subdrawings in the view, so it may not be a complete list of the required data sources. To make a complete list, you must use the techniques described in the *Master Data Source Lists* section later in this chapter.

**Data Sources**

The **data source** (ds) describes a single source of data and manages a list of data source variables. The data source breaks the stream of data into discrete chunks and passes them to the data variable buffers. The data source contains flags indicating its type and format, the name of the origin of the data. After the stream has been opened, the data source also contains a pointer to the stream’s file descriptor. The following figure shows the components of a data source.

When you call *TviReadData* or *TdlReadData*, it fills all the buffers from every data source. If the data source is a process, *TviReadData* and *TdlReadData* wait until there is enough data to fill a variable buffer. To read
Data Sources

For a single data source, use `TdsReadData`. `Tv` routines act on all the data sources referenced by a view and its components; `Tdl` routines act on all data sources in a list at the same time; `Tds` routines act on individual data sources. Using `Tds` routines lets you read different data sources at different times and rates. For example, if you have two files with samples taken at different rates, you can read one file more often than the other.

The following diagram shows the relationship of the structures used to gather data:

You can create a data source by calling `TdsCreate`, then `TdsAddDsVar`. You can manage its descriptive characteristics using `TdsGetAttributes` and `TdsEditAttributes`. To change the data source’s list of variables, use `TdsAddDsVar`, `TdsDeleteDsVar`, and `TdsForEachVar`. Data sources can be extracted from the data source list using `TdlGetNamedDataSource`.

**Data Source Variables**

Each data source variable contains all the information necessary to maintain its data buffer. This includes information about the size, type and dimension of the buffer. A data source variable keeps track of variable descriptors bound to it. You cannot destroy a data source variable if it is bound to a variable descriptor. The following figure shows the components of a data source variable:
You can create a data source variable using \texttt{TdsvCreate}. You can manage its name, buffer size, and type using \texttt{TdsvGetAttributes} and \texttt{TdsvEditAttributes}. To attach or detach a variable descriptor, use \texttt{TdsvAttachVdp} and \texttt{TdsvDetachVdp} respectively. Data source variables are extracted from the data source using \texttt{TdsGetNamedDsVar}, or from the attached variable descriptor using \texttt{TvdGetDataSourceVariable}.

The values in the data source variable’s buffer can be set by calling \texttt{TviReadData}, \texttt{TdlReadData}, or \texttt{TdsReadData}. You can also set the values in the data source variable’s buffers directly. For example, \texttt{TdsvSetValue} sets a value in a position in the buffer, recasting the value to match the data type specified by the data source variable’s attribute. You can also stuff the buffer by getting the address of the data source variable’s buffer and copying your data into it. The following code fragment illustrates stuffing the buffer. For more detail, see the example program \texttt{bufstuff.c}.

```c
float *FloatBufPtr;
LOCAL ADDRESS GetDsvBuffers (ds, dsvar, argblock)
DATASOURCE ds;
DSVAR dsvar;
ADDRESS argblock;
{
    FloatBufPtr = TdsvGetBuffer (dsvar);
    return NULL;
}
LOCAL void UpdateData()
{
    float CurrentData();
```
Variable Descriptors

The variable descriptor (vdp) is the mechanism by which graphical objects are made dynamic. The variable descriptor points to a data buffer and stores information about how to interpret the contents of the buffer.

The variable descriptor passes the data on to another object or structure that converts the data into dynamic graphics. In this way, the variable descriptor provides a unique link between the data buffer and the object and therefore should not be shared by two or more objects. If several connections to the same data buffer are required, multiple variable descriptors should be created. This happens automatically in DV-Draw and should be applied in DV-Tools programming.

Simple graphical objects are linked to a variable descriptor through a dynamic control object. The dynamic control object uses either a variable descriptor object or a threshold table in conjunction with a variable descriptor object. To get the variable descriptor structure referred to by a variable descriptor object, use VOvdGetVdp.

Data group objects (dg) contain one or more variable descriptors. Data group objects access their variable descriptors using an intermediate data group structure (dgp). To get a variable descriptor from a graph, call VOdgGetDgp followed by VGdgvd.

Input objects are linked directly to a variable descriptor or list of variable descriptors. To get a variable descriptor from an input object, call VOinGetVarList.

Contents of a Variable Descriptor

The variable descriptor contains a pointer to the data buffer, information about how the data can be accessed, and details about how the data should be displayed:

- **Address of the data buffer** specifies the memory location used to store the data or the dsv used to manage the data.
- **Data type** describes how the data is stored: integer, floating point number, double precision float, etc.
Dynamics Variable Descriptors

Data range specifies the minimum and maximum valid data values.

Data dimension specifies the data shape: scalar, vector, or matrix.

Data access mode specifies whether the data address contains the actual data value (direct mode) or a pointer to the actual value (indirect mode).

Access function specifies how to access the data. For example, if the data is stored as a sparse matrix, given the indices into the matrix, the access function retrieves the data values. The access function can replace one or more of the preceding four variable descriptor elements. This function is usually NULL and a default access function is used.

Display attributes describe the variable descriptor when it is used in a graph. These include color and marker. Display attributes are described later in this chapter and in the DV-Draw User’s Guide.

Rebinding Variable Descriptors

In DV-Draw, the data source list manages data acquisition. Every variable descriptor is bound to a data source variable. In DV-Tools the application program can acquire its own data and store it either in the data source variable buffers or in application program buffers. A program buffer is attached to a variable descriptor by detaching the variable descriptor from the data source list and then re-binding it to the program buffer. Re-binding detaches the data source variable from the variable descriptor; the application program becomes responsible for supplying data to the buffer.

The following figure illustrates re-binding:

![Diagram of re-binding process]
TvdPutBuffer detaches the vdp and rebinds it to your program buffer. It also changes the access mode of the vdp to V_DIR_ACCESS. The following fragment shows rebinding using TvdPutBuffer. For a more complete example, see buf_rebind.c in the examples directory.

```c
float fnum;
LOCAL ADDRESS RebindVdps (vd_obj, vdp, argblock)
    OBJECT vd_obj;
    VARDESC vdp;
    ADDRESS argblock;
{
    /* Rebind vdp to fnum */
    TvdPutBuffer (vdp, &fnum);
    ...
}

LOCAL void UpdateData()
{
    float CurrentData();

    ...
    /* Get the data */
    fnum = CurrentData();
    ...
}
```

**Graphs**

A graph has two basic parts. The **data group object** (dg) is the object representation of the graph. The **data group** (dgp) is a lower-level data structure that contains the information needed to display a graph on the screen.

**Data Group Objects**

A data group object is the object representation of a graph. A data group object is of type OBJECT, and is managed by the VOdg routines. The data group object contains two control points, foreground and background color attributes, and a data group structure. To get the data group structure from the data group object, use VOdgGetDgp.

Data group objects that are too large for the drawport or are obscured by another drawport are clipped to display only the visible portion of the graph. However, data groups should not be obscured by other graphical objects in the same view since the updating data bleeds through.
A data group object can only be displayed in one drawport at a time. If you want to display the same data group object in several drawports at the same time, you must clone the data group object or the view it belongs to.

A data group inherits foreground color when color dynamics are applied to it, or when it is used in a subdrawing.

The following diagram shows the relationship of the graph data structures:

\[\text{variable descriptor list} \rightarrow \text{data group} \rightarrow \text{data group object} \rightarrow \text{lower right point (1)}\]

\[\text{display formatter} \rightarrow \text{df} \rightarrow \text{upper right point (2)}\]

\[\text{foreground color} \rightarrow \text{background color} \rightarrow \text{attributes}\]

**Data Groups**

Data groups are structures that control a graph display on the screen. A data group is of type `DATAGROUP` (or `ADDRESS`) and is managed by the `VPdg` and `VGdg` routines. A data group contains a **display formatter** (`df`), context information, and one or more variable descriptors. The display formatter determines the type of the graph, such as a bar graph or a strip chart. The context information specifies details of how the graph appears when displayed. The variable descriptors supply the data to be displayed in the graph.

A data group is created using `VPdgcreate`. To include a graph as part of the display, such as in a drawing, view, or drawport, create the `dgp`, populate it with its attributes, and then create the `dg` object. After creating the `dg` object, use `VODg` routines to change the color and the object’s location on the screen. All information for graphs and their variables are set using the `VPdg` and `VPvd` routines, and accessed using the `VGdg` and `VGvd` routines. For code fragments that demonstrate these routines, see the `DV-Tools Reference Manual`.

Note that all variable descriptors assigned to a graph must have the same shape and dimensions. For graphs created in DV-Draw, this is set automatically based on the graph shape. For graphs created programmatically in DV-Tools, it is the programmer’s responsibility to make the dimensions
consistent. If you want to display variables of different dimensions in the same graph, it is possible to work around this requirement. You must set all the variable descriptors to the largest size, using VPvddim, and you must supply extra data to pad the smaller variable descriptors.

**Display Formatters**

The particular method by which a data group object actually displays its data is controlled by a display formatter. Display formatters are collections of routines that draw and update data group objects.

Display formatters are implemented as internal global variables containing pointers to the display formatter routines. Display formatter routine names begin with **VD**. These have the **typedef** declaration **DISPFORM**. Every data group contains a pointer to a display formatter structure to indicate its type. You can specify the display formatter in DV-Draw, or in DV-Tools using VPdgdf. Display formatters must be externally referenced as shown below:

```
GLOBALREF DISPFORM VDline
```

The display formatters are described in the **DV-Tools Reference Manual** and in the **DV-Draw User’s Guide**.

DV-Tools programmers can create customized display formatters. For information on writing your own display formatters, see the **DataViews Graph Development Guide**.

**Context Information**

Context is any text or graphical information that helps the user interpret the data shown. Context can include a title, a legend, axis tick-marks, etc. Static information is only drawn when the graph is first drawn on the screen and is not refreshed as the data changes. Dynamic information, such as time axis labels, are drawn each time the graph is updated. Each data group can contain different context information.
Context is controlled using VP routines. VPdgcontext turns portions of the context on and off. For example, you can turn the legend, grids, and ticks on or off independently. Other VPdg and VPvd routines set the values for the context. The following figure shows some of the context elements that can be set using VPdg routines:

**Context of Data Groups**

Other context features not shown in this figure include:

- **Scroll amount** specifies the scroll method and amount for graphs with history. Set the scroll amount using VPdgscroll_amount.

- **Spatial axes** identify the data elements within a vector or matrix. Set the labels for the spatial axes using VPdgaxlabel. You can set the tick labels using VPdg ticlabfcn; otherwise default tick labeling functions are used.
Additional information is provided by the display attributes of the variable descriptor. Many attributes are used only for specific types of graphs. The following figure shows these display attributes, which can be set using VPvd routines:

**Display Attributes of Variable Descriptors**

Data groups also contain other display information of the graph. The following information is maintained by the data group and should not be modified directly by the programmer:

- **Device index** specifies the graphical device to display the data. This allows data to be routed to the proper terminal in a multi-terminal system. The default is the current open device.

The following display information is maintained by the data group object and should be modified using the VOdg routines, not the VPdg routines:

- **Viewport** indicates the data group’s position on the screen. The viewport is specified using point objects at the dg object level, and by virtual coordinates at the dgp level. The default is the full screen.

- **Color** is specified using a color object at the dg object level and a COLOR_SPEC structure at the dgp level.
Foreground color is used for tick labels, labels and title text, and viewport outline. Uses the foreground color of the data group or, if no foreground color is set, the default foreground color.

Background color is used to erase old data values. Uses the background color of the data group or, if no background color is set, the default background color.

Graph Feedback

You can query a data group for information about what it is displaying. In particular, you can get a rich set of information about the data being displayed at a specific point in the data group. For example, if the point is on a bar in a bar graph, you can get the iteration of the data being displayed, the data value, the variable descriptor supplying the data, and other information. The point that is the basis of the query can come from any source, such as a location object representing a user’s pick on a bar or line in the data viewport.

The following figure shows some of the information you can get:

You can use queries to set up feedback to a user’s pick. For example, when a user picks a point in a graph, the application can perform these actions in response:

- Get the data displayed at a point selected by the user.
- Display that information in another form, such as printf statements.
- Save the information to a database.

To set up graph feedback, use the query routine VPdgdfquery. This routine takes a data group, a flag indicating the type of information you want, a structure (indatum) containing the point that is the basis for the query, and a structure (outdatum) to hold the data that answers the query. The point must be in screen coordinates. To get a point from a location object representing a user’s pick, call VOloScpGet.
To get accurate results from the queries, you must determine the point, then make the queries without making any intervening changes to the graph. Changes such as displaying new data or resizing affect where data is displayed, so the coordinates no longer display the same data.

Using a point as the basis for queries, you can get the following information at the point location:
- the slot
- the data iteration displayed (sample)
- the interpolated iteration number (sample)
- the vdp's whose data is displayed at or near the point

The vdp's are returned in a structure that contains a count of the vdp's at the point location and an array of structures, each of which contains a vdp and its index in the data group’s list of vdp's. The array size is controlled by the define V_Q_PICKED_VDP_MAX. This define is set to 64, but you can tailor it for your application.

The vdp structures are used for other queries related to data values at the point location. For each vdp, you can set up an indatum structure that contains a point, a vdp, and the vdp’s index in the data group. For this type of indatum, you can get the following information:
- the data value displayed closest to the point
- the interpolated data value at the point based on the range of the vdp

For data groups using the stacked display formatters VDpig and VDlinefill, you can also get the floor value, which is the sum of the values underlying the datum at the point location. For data groups using the radial display formatters VDne_radial and VDradial, you can also get the sector, which is similar to the slot. These special cases are discussed in examples later in this chapter.

Currently, this feature is only supported for display formatters that display scalar data and show history by scrolling or wrapping such as VDbar and VDline. For a complete list, see the DV-Tools Reference Manual. For data groups that display history, this feature lets you recapture data that has been overwritten in the data buffers, as long as it is still displayed in a data group.

For more information on how to use the graph feedback feature, see graph_query.c in the DV-Tools examples programs directory. The following additional code fragment shows how to get all the data values displayed in a
slot at a particular location. It doesn’t query for the vdp5s at the location since
this may not provide all the vdp5s; instead it sets up indatum for the queries
using the data group’s list of vdp5s.

```c
DATAGROUP dgp;
DV_POINT pick_location; /* in screen coordinates */
int i, slot, num_vdps;
V_Q_PICK_VDP pick_vdp;
double *val_ptr;
double *values;

/* Determine if the pick is in a slot. If not, return FALSE. */
VPdgdfquery (dgp, V_Q_SLOT_AT_LOCATION,
            (ADDRESS) &pick_location, (ADDRESS) &slot);
if (slot == -1)
    return FALSE;

/* Get the number of vdp5s supplying data for the data group. */
num_vdps = (int) VGdgvd (dgp, 0);
if (num_vdps > 0)
{
    pick_vdp.vdp = (V_Q_VDP*) S_ALLOC (sizeof (V_Q_VDP));
    values = (double*) S_ALLOC (sizeof (double) *num_vdps);
    /* For each vdp, set up a query for the data values displayed in the slot. */
    for (i = 1, val_ptr = values; i <= num_vdps; i++,
        val_ptr++)
    {
        pick_vdp.location = pick_location;
        pick_vdp.vdp->vdp = (VARDESC) VGdgvd (dgp, i);
        pick_vdp.vdp->index = i;
        if (!VPdgdfquery (dgp, V_Q_DATA_VALUE,
                         (ADDRESS) &pick_vdp, (ADDRESS) val_ptr))
            return FALSE;
    }
    S_FREE (pick_vdp.vdp);
    /* Handle the values. */
    ...
    S_FREE (values);
    return TRUE;
}
else
    return FALSE;
```
Line Graph Example

The following figure shows a point on a line graph:

Queries for this point get the following information:
- data sample: 12.0
- slot: 2
- interpolated sample at location: 11.88
- vdp count: 2

The closest sample is the twelfth, which is displayed in the second slot. Two variable descriptors display data at or near the point, so the vdp array contains two filled structures. Queries using the point and these structures get the following information:

For the vdp Data 1:
- data value: 0.64
- interpolated value at location: 0.68

For the vdp Data 2:
- data value: 0.76
- interpolated value at location: 0.68

Note that the interpolated value at the location is the same for both variable descriptors since this field reflects the actual point location. The data value, however, is different since this is the value of the closest data sample along each of the two lines.
The following figure shows a point on a piggyback bar graph. This example shows how stacked graphs such as the piggyback bar graph and filled line graph determine floor value.

The graph displays three variables each with the range \([-1, 1]\). Since the values are added together, the total range for the graph is \([-3, 3]\).

Queries for this point get the following information:

- data sample: 5.0
- slot: 5
- interpolated sample at location: 5.03
- vdp count: 1

One variable descriptor displays data at the point, so the vdp array contains one filled structure. Queries using the point and this structure get the following information:

For the vdp Var:2:
- data value: 0.37
- interpolated value at location: -0.23
- floor value: -1.95

The floor value lets you take into account the height of bars stacked underneath the selected bar. It maps to the position of the bottom of this bar relative to the value axis.
The data value and the interpolated value at the location have their standard meaning in a stacked graph; they are both absolute values within the range of the variable descriptor. To calculate the value at location with respect to the value axis, use the following formula:

\[
(value_{at\ location} - low\_end\_of\_range) + floor\_value
\]

Radial Graph Example

The following figure shows a point on a radial graph. This example focuses on how the radial graph determines the slot, data sample, and sector values. The radial graph is similar to a line graph, but uses polar coordinates. A radial graph is divided into wedge-shaped slots like a pie, as illustrated by slot 1 in the following figure:

Each data point is plotted at the center of its slot and beginning of its sector. For example, “sample 7” is plotted in the center of slot 7. Slot 8 starts halfway between “sample 7” and “sample 8.” Sector 7 takes up the entire wedge between these samples.

The point shown in the figure is a little closer to sample 8 than to sample 7, so queries for this point get the following information:

- data sample: 8.0
- slot: 8
- sector: 7
- interpolated sample at location: 7.70
- \textit{vdp} count: 1
The radial graph handles queries relating to the variable descriptors in a manner similar to a line graph, so values for those queries are not shown here.

**Step Graph Example**

The following figure shows a point on a step graph. This example shows the different way the step graph determines the slot and data sample values.

Queries for this point get the following information:

- data sample: 34.0
- slot: 4
- interpolated sample at location: 34.8
- \( vdp \) count: 1

With the step graph, a point between the 34th and 35th tick marks on the time axis, but closer to the 35th tick mark, returns a data sample value of 34.0, not 35 as with other graph types. This convention is consistent with the way the step graph is drawn.

The step graph handles queries relating to the variable descriptors in a manner similar to a line graph, so values for those queries are not shown here.
A graphical object is made dynamic by attaching a dynamic control object to it. This changes the control points and attributes of the graphical object when the data changes, according to the dynamic actions specified in the dynamic control object. Note that object dynamics cannot be applied to graphs or input objects because they are inherently dynamic.

There are five basic types of object dynamics. They are **visibility**, **transformation**, **attribute**, **text**, and **subdrawing dynamics**. Visibility dynamics control whether or not the graphical object is visible. Transformation dynamics change the graphical object’s control points by rotation, translation, and scaling. Attribute dynamics change the graphical object’s attributes, such as color, fill status, and line type. Text dynamics changes the content of a text object to display the formatted variable value. Subdrawing dynamics display different subdrawings for different data values. For example, displaying a progressive series of subdrawings can be used to create an animated sequence.

The following figure shows the relationship of the structures used to display object dynamics.

**Dynamic Control Object**

A **dynamic control object** is the mechanism by which DataViews makes simple graphical objects dynamic. A dynamic control object is a set of **dynamic actions**. A dynamic action is a description of dynamic behavior to be applied to a graphical object. Each dynamic action is described by attaching a **data object** to the dynamic control object. The data object provides a source of data for the dynamic action.
The dynamic control object copies all the information in the graphical object before applying any dynamics. When dynamics are being applied to a graphical object, the dynamic control object changes the attribute values and/or the control points of the object. To reset the dynamics, the dynamic control object changes the object back to its original state using the saved information.

Note that if you want to change the graphical object explicitly, you must first reset it using \texttt{VODYReset}. This ensures that you get the object’s original values and not the changed copy.

In DV-Tools, you can add dynamics to a simple graphical object in three steps:

1. Create a dynamic control object by using \texttt{VODYCreate}.
2. Specify dynamic actions for the dynamic control object by using \texttt{VODYAttachData}.
3. Attach the dynamic control object to the graphical object by using \texttt{VOOBDYSET}.

Each dynamic action is described by attaching a data object to the dynamic control object. The data object provides a source of data for the dynamic action. A data object can be either a variable descriptor object (\texttt{vd}) or a threshold table object (\texttt{tt}).

A \texttt{vd} provides continuously varying data values normalized to the incoming data range. A \texttt{vd} is most suitable for transformation dynamics, proportional fill dynamics, and text dynamics.

A \texttt{tt} provides a fixed set of discrete data values. This discrete data value is called a \texttt{DATUM}. The type of the \texttt{DATUM} depends on the graphical attribute being controlled. For example, color dynamics require a color object as the \texttt{DATUM}; line width dynamics require an integer as the \texttt{DATUM}. A \texttt{tt} is most suitable for discrete dynamics such as visibility dynamics, attribute dynamics, or subdrawing dynamics. Threshold table objects are described later in this chapter.

Most types of dynamic actions can only be defined once for a dynamic control object, using a single data object. However, the non-absolute transformations such as relative movement, scaling, and rotation can be defined more than once using different data objects for the same dynamic control object. Note that the proportional fill options are mutually exclusive. They apply to circles,
arcs, ellipses, rectangles, and polygons and are effective only when the object has a fill status of “Fill,” “Fill with Edge,” or “Edge with Fill.” The other transformation and attribute dynamics apply to all types of graphical objects.

DV-Draw’s dynamics for simple graphical objects is a subset of the dynamics available in DV-Tools. The following table summarizes the differences between dynamics available in DV-Draw and those available in DV-Tools.

<table>
<thead>
<tr>
<th>Dynamic Characteristic</th>
<th>DV-Draw</th>
<th>DV-Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation reference point</td>
<td>Object center point</td>
<td>Any point</td>
</tr>
<tr>
<td>Number of data objects that can control a non-absolute transformation dynamic action</td>
<td>One</td>
<td>Multiple</td>
</tr>
<tr>
<td>Type of data object used by attribute dynamics</td>
<td>Threshold table only</td>
<td>vd or tt</td>
</tr>
<tr>
<td>Attributes that can become dynamic</td>
<td>Foreground/background color, line width and type, curve type, arc direction, proportional fill</td>
<td>Any attribute</td>
</tr>
</tbody>
</table>

The following code fragment demonstrates rotational dynamics using a reference point instead of the object’s center point:

```c
/* Create a dynamic control object. */
dyn_control = VOdyCreate ();
/* Set the erase method on the dynamic control object. */
VOdySetEraseMethod (dyn_control, V_DYN_ERASE_OBJECT);

/* Create a rotational point. */
rotational_point = VOptCreate (WORLD_COORDINATES, 100, 100, (OBJECT) NULL);

/* Attach the rotational data object to the dynamic control object, using rotational_point as the point to rotate around. */
VOdyAttachData (dyn_control, DYN_ROTATE, rotation_vd, &low_range, &high_range, rotational_point, (OBJECT) NULL);

/* Attach the dynamic control object to a graphical object. */
VOobDySet (graphical_object, dyn_control);
```
A dynamic control object can control one or more graphical objects. The
dynamic activity of a single graphical object can be enabled or disabled using
`VOdySetState`. If the dynamic state is enabled, dynamic updates occur;
otherwise dynamic updates are ignored.

In DV-Draw, you share a dynamic control object by naming it and then
assigning it by name to other graphical objects. To share a dynamic control
object in DV-Tools, just attach it to two or more graphical objects using
`VOobDySet`.

Note that the name of a dynamic control object behaves a special way in
DV-Draw that is not duplicated in DV-Tools. In DV-Draw, when you copy an
object with a named dynamic control object, the resulting copy of the
dynamic control object has a new name that is guaranteed to be unique. If you
close such an object using a DV-Tools routine, you do not get a uniquely
named dynamic control object. Instead, you get two separate dynamic control
objects with the same name. This situation can have unexpected results if you
save a view containing the clones and then load it into DV-Draw.

Variable descriptor objects (`vd`) work with variable descriptors (`vdp`) to
provide normalized data to dynamic control objects. The dynamic control
object either uses a `vd` directly, or uses `vd` indirectly through a threshold table.
The `vd` supplies a normalized value from the `vdp` representing the current data
value. A `vdp` maintains information about the intended incoming data range,
takes data from the data source variable or the rebound buffer, and normalizes
it to that range. Variable descriptors are described earlier in this chapter.

The threshold table (`tt`) object acts as a mapping function of data values to
objects, text strings, or values. The threshold table is a list of pairs,
comprising a threshold value and a `DATUM`, which is information in the form
of an object, integer, float, or text string. Threshold values are normalized to
the range [0,32767]. The list is sorted in increasing order of the threshold
values. The entire table must comprise only one type of datum. If the type is
`OBJECT` and the threshold table is used by a dynamic control object, the
`DATUMs` in the threshold table should all be the same object type. For
example, for subdrawing dynamics, the `DATUMs` should all be subdrawing
objects.
The threshold table takes data from the variable descriptor object and compares it against a table of threshold values. The data is mapped to the object associated with the threshold closest to, but less than, the data value. For example, if the data value falls between $T_1$ and $T_2$ it is mapped to the object associated with $T_1$, as shown in the following figure.

When a data value crosses a threshold, it is mapped to a different DATUM.

Note that the threshold values in a threshold table object are *lower limits*, not upper limits as in the color threshold tables for graphs. Data values below the lowest threshold, or values that are undefined, are mapped to the initial object. Data values above the highest threshold are mapped to the object associated with the highest threshold.

When you first create the table, it has one object and no thresholds. The table returns the object, regardless of the input value. You can then add object-threshold pairs, to build the mapping of values to objects.
Every time an update occurs, such as when `TdpDrawNext` or `V OttUpdate` is called, the table gets incoming data using a `vd`. The threshold table then compares the input datum against thresholds in the table, and generates an output datum of type `DATUM` which can be an object, float, integer, or string. This output datum is called the current output of the table, and is obtained by calling `V OttDataGet`. Before generating this output, the table saves the old current output as the last output, which is obtained by calling `V OttLastGet`. If the table has been reset using `V OttReset`, the current and last output data are set to reference the initial datum in the table.

The following code fragment shows the use of the same color threshold table for two dynamic control objects affecting two different graphical objects:

```c
/* Create two dynamic control objects */
dyn_control_A = VOdyCreate();
dyn_control_B = VOdyCreate();

/* Create a threshold table for color change and add thresholds */
color_tt = VOttCreate (color_vd,
    OBJECT_DATUM (OT_COLOR),
    green_obj);
VOttAddThresh (color_tt, (int) (32767 * 0.6666667),
    red_obj);
VOttAddThresh (color_tt, (int) (32767 * 0.3333333),
    yel_obj);

/* Attach the threshold table to both dynamic control objects */
VOdyAttachData (dyn_control_A, FOREGROUND_COLOR,
    color_tt, (float *) NULL, (float *) NULL,
    (OBJECT) NULL, (OBJECT) NULL);
VOdyAttachData (dyn_control_B, FOREGROUND_COLOR,
    color_tt, (float *) NULL, (float *) NULL,
    (OBJECT) NULL, (OBJECT) NULL);

/* Attach the dynamic control objects to the graphical objects */
VOobDySet (graphical_object_A, dyn_control_A);
VOobDySet (graphical_object_B, dyn_control_B);
```

The "Subdrawing" dynamics feature displays different subdrawings for different data values. Each subdrawing is displayed only when the data falls within a specified range. The subdrawings are usually related in some way so that their sequential display creates an animated sequence. The subdrawing dynamics feature can only be applied to subdrawings.
Subdrawing dynamics are implemented using a *tt* as the data object. The *tt* object compares the incoming data to its array of threshold value-object pairs. If the current value is between certain thresholds, the associated object is displayed.

The following figure shows a drawing that uses a threshold table object to implement subdrawing dynamics. The attached variable descriptor reads incoming data from *VAR:2*. When the value of *VAR:2* is 8.45, the displayed object is a rectangle. The illustration uses unnormalized values.

Dynamic subdrawings can be created in DV-Draw by adding subdrawing dynamics to a subdrawing. DV-Draw creates a threshold table object and attaches the original subdrawing to it as the default subdrawing. Every time you add a new threshold value, DV-Draw creates another subdrawing using the control point, scale factor, and attributes of the default subdrawing.

Creating dynamic subdrawings in DV-Tools gives you more flexibility. Each subdrawing can have a different control point, scale factor, or attributes.

A subdrawing includes all the dynamics defined in its original view. You can enable or disable the internal dynamics of a subdrawing in DV-Draw, or in DV-Tools using *VOsdSetDynamicFlag*. *VOsdGetDynamicFlag* returns a flag indicating the current status of dynamics in the referenced view: either enabled, disabled, or no dynamics. The internal dynamics of a disabled subdrawing are not active. The internal dynamics of an enabled subdrawing are active, so the subdrawing is called an *active subdrawing*.

Whether the internal dynamics are enabled or disabled does not affect your ability to attach object dynamics to the subdrawing. However, you should not attach the subdrawing dynamics feature to active subdrawings.
Getting Data from the Original Data Sources

Active subdrawings must be supplied with data. In DV-Draw, the internal dynamics of an active subdrawing can get data from either their original data source variables or from data source variables in the current view. To get data from data source variables in the current view is called **mapping**.

When variables are not mapped, the data for active subdrawings comes from the data sources specified in the subdrawing's view. In this case, you should use `TviOpenData`, `TviReadData`, and `TviCloseData`, passing in the highest level view. These routines act recursively on all the data sources in any subdrawings in the view and any subdrawings within subdrawings. The `Tvi` routines are preferred over the corresponding `Tdl` routines, which do not act recursively. If `Tdl` routines are used, the internal dynamics of subdrawings do not work.

Mapping to New Data Sources

When a variable is mapped, the initial binding between the `vdp` in the subdrawing and its data source variable (also in the subdrawing) is broken and the `vdp` is rebound to a new data source variable in the current view. The new data source variable becomes the source of data for the `vdp`.

For example, the `vdp` in a subdrawing was bound to a global data source variable named `Float:in_subdr`. If you map `Float:in_subdr` to a data source variable named `Float:in_main` in the view containing a subdrawing, the base address of the `vdp` changes to `Float:in_main`, which becomes the source of data for the dynamics. The following diagram illustrates this mapping.

![Diagram](image-url)

Global data source variables can be mapped; local data source variables cannot. Data source variables are local by default, but can be made global in DV-Draw or by calling `TdsvSetGlobalFlag` in DV-Tools.
Mappings to variables in the current view can be set up or changed in DV-Draw, or in DV-Tools using \textit{VOsdSetDvMapping}. These mappings can extend through chains of subdrawings. The process of actually rebinding the \textit{vdp}s to the new data source variables higher in the chain is called \textbf{resolving} the mappings. This usually occurs when the view is initially drawn. To resolve the mappings before drawing, you must either call \textit{VOsdPoolRemove} or set the \textit{DVSD\_DEACT\_POOL} configuration variable to \textit{YES}.

\textbf{Connecting Internal Dynamics to Application Data}

When you need to connect the dynamics in the subdrawing to data in the application, three approaches are possible:

- For mapped variables, stuffing buffers in the current view
- For unmapped variables, rebinding the variable descriptors in the subdrawing
- For mapped variables, rebinding the variable descriptors in the subdrawing

\textbf{Stuffing Buffers for Mapped Variables}

If your view contains subdrawings with mapped variables, the preferred method is stuffing data source variable buffers or a similar method that involves the data source variables. Stuffing the buffer takes advantage of any mappings that exist in the view and its subdrawings. If all variables in the active subdrawing have been mapped, you only need to stuff buffers in the current view; you do not have to access the objects within the subdrawing. A code fragment earlier in this chapter shows how to stuff buffers.

\textbf{Rebinding for Unmapped Variables}

If your subdrawing does not have mapped variables, rebinding variable descriptors in the subdrawing is a sound method. For example, you can have multiple instances of an active subdrawing where the variables are not mapped. If you want to supply these \textit{vdp}s with data from different application variables, you should use rebinding. However, you should call \textit{VOsdPoolRemove} before you rebind or make other modifications to the subdrawing’s view. This ensures that the changes are confined to the particular subdrawing and not inadvertently cloned to another subdrawing.

In the following code fragment, a view contains two subdrawings named \textit{sd1} and \textit{sd2}. Both reference the same filename and contain a circle with foreground color dynamics. In both subdrawings, the variable that controls this dynamic action is named \textit{Float:var}. The program uses the subdrawing...
name to differentiate these variables and rebind them: for \textit{sd1}, it rebinds the \textit{vdp}s attached to \textit{Float:var} to \textit{FloatBuffer1}; for \textit{sd2}, it rebinds the \textit{vdp}s attached to \textit{Float:var} to \textit{FloatBuffer2}.

```c
LOCAL ADDRESS setup_object(), check_vdp();
LOCAL float FloatBuffer1 = 0, FloatBuffer2 = 0;

/*/ In the initialization, load the view and get the drawing. */
view = TviLoad (view_name);
drawing = TviGetDrawing (view);

/*/Traverse the named objects in the drawing. */
TdrForEachNamedObject (drawing, setup_object, NULL);
...

/*/Routine checks if the named object is a subdrawing, then calls the rebind routine. */
LOCAL ADDRESS setup_object (object, name, args)
OBJECT object;
char *name;
ADDRESS args;
{
  /* If the object is a subdrawing, isolate changes by removing it from the pool. */
  if (VOobType (object) == OT_SUBDRAWING)
    VOsdPoolRemove (object);
    TobForEachVdp (object, check_vdp, name);
    return V_CONTINUE_TRAVERSAL;
}

/*/Routine rebinds the vdp(s) based on the subdrawing's name. */
LOCAL ADDRESS check_vdp (data_obj, vdp, name)
OBJECT data_obj;
VARDESC vdp;
char *name;
{
    DSVAR dsvar;
    char *dsvar_name;

    dsvar = TvdGetDataSourceVariable (vdp);
    dsvar_name = TdsvGetName (dsvar);
```
Rebinding for Mapped Variables

Mixing rebinding and mapping is not recommended because it may be confusing; the variable descriptor is bound to one data source variable before mappings are resolved and bound to a different data source variable after resolution. However, it is possible and may be the method you prefer in your application. The following code fragment shows how to rebind a mapped variable:

```c
LOCAL ADDRESS setup_object(), check_vdp();
LOCAL float FloatBuffer1 = 0;

/* Traverse the named objects in the drawing. */
TdrForEachNamedObject (drawing, setup_object, NULL);
...

/* Routine checks if the named object is a subdrawing, then calls the rebind routine. */
LOCAL ADDRESS setup_object (object, name, args)
  OBJECT object;
  char *name;
  ADDRESS args;
{
  /* If the object is a subdrawing, remove it from the pool. This resolves the mappings and isolates changes. */
  if (VOobType (object) == OT_SUBDRAWING)
    VOsdPoolRemove (object);
  TobForEachVdp (object, check_vdp, name);
  return V_CONTINUE_TRAVERSAL;
}
```
/* Routine rebinds the vdp based on the dsvar name. */
LOCAL ADDRESS check_vdp (data_obj, vdp, name)

OBJECT data_obj;
VARDESC vdp;
char *name;
{
    DSVAR dsvar;
    char *dsvar_name;

    /* Check the dsvar name before rebinding. */
    dsvar = TvdGetDataSourceVariable (vdp);
    dsvar_name = TdsvGetName (dsvar);
    if (strcmp (dsvar_name, "Float_in_main:1") == 0)
        TvdPutBuffer (vdp, (ADDRESS) &FloatBuffer1);

    ...

    return V_CONTINUE_TRAVERSAL;
}

Blinking

Many applications include the requirement that certain objects switch rapidly between colors to indicate an alarm condition. This effect is commonly called blinking, and it is particularly good for capturing a user’s attention.

DataViews supports two methods for implementing blinking, both of which require programming, so they are not available in DV-Draw. The first method is a high-level approach that uses color dynamics to make an object blink. The second method is a lower-level approach that uses changes in the platform-specific equivalent of the color table. In the second method, the color itself blinks. Both of these methods aim for the same resulting behavior, which is discussed below, but they do it in different ways.

Overview of Blinking Behavior

An object with blinking normally appears in a steady color. It changes to blinking colors only to show that there is a new alarm. The following table lists the states and colors that a blinking object might have:

<table>
<thead>
<tr>
<th>State</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Green</td>
</tr>
<tr>
<td>Unacknowledged alarm</td>
<td>Blinking red (switching between a brighter red and a dimmer red)</td>
</tr>
<tr>
<td>Acknowledged alarm</td>
<td>Steady red</td>
</tr>
</tbody>
</table>
To use color dynamics for blinking, set up a threshold table containing all the colors for the different states of the object. As with all color dynamics, the threshold table gets its data from a variable descriptor by way of a variable descriptor object. The value in the variable descriptor controls which color the object is. The following table shows how values coming into the threshold table map to the different colors in this example:

<table>
<thead>
<tr>
<th>If the value is . . .</th>
<th>The color is . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>1</td>
<td>Dim red</td>
</tr>
<tr>
<td>2 or above</td>
<td>Bright red</td>
</tr>
</tbody>
</table>

In the normal state, the value might be 0, so the color would be green. When the application detects a new alarm, it starts toggling the value between 1 and 2. This makes the object blink red. When the user acknowledges the alarm, the application changes the value to 2, so the object appears a steady, bright red. Finally, when the condition is corrected, the application changes the value back to 0, and the object returns to being green.

The advantages of this approach are that you can apply blinking on an object-by-object basis, you can use any colors in your color palette, and you can apply the blinking to existing objects easily.

The drawback of this approach is that you have to explicitly change the value in the variable descriptor. To see the color change, you have wait for the next call to `TdpDrawNext` or `TdpDrawNextObject` to update the object. Because of this, you cannot control the blinking interval precisely.

In the low-level blinking method, you create the blinking effect by acting on the platform-specific elements that control the colors for the display device. In X, these are cells in the colormap. In Windows, these are entries in the hpalette. Since you are acting on the color used within the display device (instead of on a higher-level abstraction), the color change is immediate and affects all graphics that use the color being changed.

This method uses a combination of DataViews techniques and platform-specific techniques. The DataViews techniques are used to manage the object’s state, and are discussed in this section. The platform-specific techniques are used to make the colors blink, but since they are platform-specific, they are not in the scope of this document. After reading this section, you should look at the platform-specific examples in the `blinking` subdirectory of the DV-Tools `examples` directory. Currently we provide an
When you use this method, you must be careful which colors you assign to objects. You should set up a special color table file that reserves certain colors for blinking. For example, you can use the first 120 colors as the standard palette and reserve the last eight colors for blinking. The reserved colors must not be duplicates of any other colors in the color table. If you have a bright red with the RGB values 255, 0, 0 in the standard part of the color table, you could use 254, 0, 0 as the RGB values for a blinking red in the reserved part of the color table.

Normally you want an object to be a steady color, so you should assign the object a color from the standard part of the color palette. However, you also want the object to start blinking to indicate a new alarm state. One way to do this is to give the object color dynamics and use the following color threshold table:

<table>
<thead>
<tr>
<th>State</th>
<th>Vdp value</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
<td>Green (from the standard part of the palette)</td>
</tr>
<tr>
<td>New alarm</td>
<td>1</td>
<td>Red (from the part of the palette reserved for blinking)</td>
</tr>
<tr>
<td>Acknowledged alarm</td>
<td>2</td>
<td>Bright red (from the standard part of the palette)</td>
</tr>
</tbody>
</table>

When the application detects a new alarm, it changes the value in the variable descriptor to 1, which changes the object’s color to a blinking color. When the user acts on the alarm, the application changes the value to 2, which changes the object’s color to a steady red.

Although this threshold table is similar to the one suggested in the other method, there is an important difference. In this method, you do not have to toggle between values to get the blinking. Instead, each value in the threshold table maps to a particular state, so the value actually reflects the state of the object.

You must still make the second color in the threshold table blink using platform-specific techniques, but you have more options on how to manage this. You can apply blinking to this color all the time, or only when required. If you have only a few objects with blinking, you may want to turn on
In a complex application, you may need to manage the dynamics in multiple views. The application sources for data may overlap, and you may want to handle all the data sources on a single list. You can do this by establishing a master data source list. The starting point of the master list can be the data source list from your top view, and you can add to it by merging the data source lists from your other views. When you use a master data source list, a single call to TdiReadData can gather the data for all your views, and views that share data sources can be synchronized. The following fragment shows how to build a master data source list.

```c
VIEW topview, view1, view2;
DATASOURCELIST masterdsl;

// Set up the master dsl using the dsl from the top view */
masterdsl = TviGetDataSourceList (topview);
TviMergeAddDataSources (view1, masterdsl, DS_EXACTMATCH);
TviMergeAddDataSources (view2, masterdsl, DS_EXACTMATCH);

... /* Create the drawports */ ...

/* Open and read the data using the master data source list */
TdiOpenData (masterdsl);
TdiReadData (masterdsl);
```

You can make several master data source lists. For example, you may want all the data sources involved in Process A on one list, and all the data sources for Process B on another list. Another use for master data source lists is to combine all the data sources for a view that includes enabled subdrawings.
You can merge data source lists by using the routines
`TviMergeAddDataSources` or `TviMergeDataSources`. Both routines check the
data source list to be merged against the master data source list to find
matching names. The routine then determines if the data source is a duplicate.
`TviMergeAddDataSources` adds non-duplicate data sources or data source
variables to the master list. `TviMergeDataSources` returns a list of non-
duplicate data sources, but doesn’t add any data sources to the master list. For
more information about merging data sources refer to the *DV-Draw User’s

To merge all the data source lists referenced by subdrawings in a view,
traverse the drawing belonging to the view, looking for enabled subdrawings.
You should also do this recursively for all the subdrawings within the
subdrawings. Get the view from each subdrawing by calling `VOsdViGet` and
then get the data source list from each view using `TviGetDataSourceList`.
When you have extracted all the data source lists, you can make a master data
source list that contains all the data sources.

A master data source list is not necessarily associated with a single view, but
can be shared by several views. Because of this, data source lists maintain
internal reference counts, so when views are destroyed, data source lists
belonging to other views are not destroyed. The following figure shows the
relationship of a shared data source list to neighboring data structures.
This example illustrates dynamic data flow. The data source is a file whose first three values are 12.35, 257.5, and 6.23. The data source is linked to three scalar variables, whose names are VAR:1, VAR:2, and VAR:3. The three data values are read by the data source and stored in the three variable buffers respectively.

When you call TdpDrawNext each dynamic object checks its buffer to determine if the data value has changed and if updating is necessary. Only dynamic objects that change in appearance are redrawn.
Summary

This chapter has presented these basic techniques:

- How to gather data for dynamics from standard sources such as files and processes.
- How to rebind dynamics to program variables.
- How to display data using graphs and object dynamics.
- How to manipulate the way graphs and object dynamics respond to data.
- How to build master data source lists.

Using these techniques, you can implement the following features in your application:

- Run your dynamics using any data available within your application.
- Independently control when new data is read and when the dynamics are updated.
- Let users control the type of graph and number of samples displayed, so they can see the data they need in a form they prefer.
- Set up more complex dynamics than those available in DV-Draw.
This chapter discusses the following topics:

- Input Objects and Input Technique Objects
- Interaction Handlers
- Complex Input Objects - Combiners and Multiplexors
- Programmatic Creation
- Input Handling within Input Objects

Overview

Input objects are dynamic objects that interpret input from the user by converting it into a value, record the selection in a data buffer, and change their appearance on the screen as they interpret input from the user. You can create input objects in DV-Draw and manipulate them using DV-Tools. The following types of input objects are available: buttons, checklists, menus, palette, slider and scrollbar, two-dimensional slider, single- and multi-line text entry, toggles, combiner, and multiplexor. Combiners and multiplexors can only be created in DV-Tools. If you are running on the X Window System, you can also create several OSF/Motif or OPEN LOOK widgets.

Input objects have characteristics common to all graphical objects, such as location and color. The input technique object manages most functional aspects of the interaction. The interaction template is a drawing object that controls the physical appearance and functional fine-tuning. The interaction handler determines the interaction type, such as slider, menu, or palette and controls the drawing, updating, input gathering, and erase method of the input object. The input object works with the interaction handler the way a data group works with its display formatter.

The input from input objects is the user’s form of communication with the application. Input objects provide the user with control over the application so they are critical to the control flow of the interface. Understanding how input objects work prepares you to understand event handling, the control flow mechanism of DataViews.
The primary data structures used for input objects are the input object \((in)\) and the input technique object \((it)\). The input object is a dynamic graphical object with control points, an attributes structure, and one or more variable descriptors. The input technique object uses the interaction handler \((ih)\) and the template, which is a drawing object \((dr)\).

In DV-Draw, you can create input objects, position them within the drawing, assign the foreground and background colors, attach the variable, set the range, and set the flags. When you create an input object, DV-Draw creates the input technique object for you, and lets you set the interaction handler, template, list of labels, key origin, and local key bindings. You can also create and edit the template in DV-Draw.

Input objects are active in the Run and Prototype Menus. Since you can accomplish so much of the setup in DV-Draw, you may only need DV-Tools to rebind their variables and implement the control flow of the application.

You must use DV-Tools to accomplish the following:

- create a combiner or multiplexor input object
- change the list of labels in checklists, menus, and toggles at runtime
- attach an echo function
- write your own interaction handlers
The following figure shows how some of the DV-Draw options in the Edit Input Object Menu correspond to DV-Tools calls:

![Diagram showing DV-Draw options and their corresponding DV-Tools calls]

The following function calls get the variable descriptor, data source variable, and data buffer from an input object created in DV-Draw. You can manipulate these data structures as described in the *Dynamics* chapter.

Get the input object:

```c
menu = TdrGetNamedObject (drawing, "menu.input");
```

Get the variable list:

```c
VOinGetVarList (menu, &vdplist, &numvars);
```

Get the attached dsv:

```c
dsv = TvdGetDataSourceVariable (vdplist[0]);
```

Get the dsv's buffer:

```c
info.data_ptr = (float*)TdsvGetBuffer (dsv);
```

After you have created the input object in DV-Draw, you need to access its data to act on user input. You can do this using the following call:

```c
VUerServiceResultPost (1, HandleInput, info.data_ptr, 0, menu, INPUT_DONE, 0);
```
The following function calls create, run, and terminate a text menu input object. Most of these calls apply to all input objects. Many of the calls in the initialization phase occur when you create an object in DV-Draw; these calls are shown to illustrate programmatic creation of an input object.

The following listing assumes that the template drawing already exists as a view file.

**Before the main:**

Globally reference the handler:
```
GLOBALREF INHANDLER VNmenu;
```

Set up the menu labels:
```
char *Labels[] = {"Choice 1", "Choice 2", "Choice 3"};
```

Set up the value list:
```
float Values[] = {0.5, 1.0, 1.5};
```

**In the initialization phase:**

Create the input object:
```
MenuIn = VOinCreate (ll_pt, ur_pt, &Attributes);
```

Create the vdp:
```
Vdp = VPvdcreate(
    (ADDRESS) &MenuValue, V_F_TYPE);
```

Attach the vdp:
```
VOinPutVarList (MenuIn, &Vdp, 1);
```

Set the flags:
```
VOinPutFlag (MenuIn, SAVE_RASTER, NO);
VOinPutFlag (MenuIn, ERASE_METHOD, NO_ERASE);
```

Get the template:
```
TemplateView = TviLoad ("menu.lay");
TemplateDr = TviGetDrawing (TemplateView);
```

Create the technique object:
```
MenuIt = VOitCreate (VNmenu, TemplateDr);
```

Destroy the used view:
```
TviDestroy (TemplateView);
```

Attach the labels:
```
VOitPutList (MenuIt, TEXT_LIST,
    (ADDRESS)Labels, NUM_LABELS);
```

Attach the values:
```
VOitPutListValues (MenuIt,
    (ADDRESS)Values, NUM_LABELS);
```

Set the keys:
```
VOitPutKeys (MenuIt, DONE_KEYS, "\004");
VOitKeyOrigin (MenuIt, DONE_KEYS, LOCAL_KEYS);
```

Join input object and technique:
```
VOinTechnique (MenuIn, MenuIt);
```

Draw the input object:
```
TdpDrawObject (drawport, MenuIn);
```
In the run phase:
Gather input: \[\text{location} = \text{VOloWinEventPoll (mode)};\]
Process input: \[\text{VUerHandleLocEvent (location)};\]

In the termination phase:
Erase the input object: \[\text{TdpEraseObject (drawport, MenuIn)};\]
Dereference the components: \[\text{VOinDereference (MenuIn)};\]

Input Objects

An input object \((in)\) is a graphical object that controls the user interaction and updates its variables in response to user input. An input object contains two control points defining its area. The attributes of an input object are foreground and background color.

It also contains an input technique object, draw and erase flags, and an array of variable descriptors which store input values either in \(dsv\) buffers or user buffers.

To create an input object create two points, then pass the points and attributes to \(\text{VOinCreate}\). Input objects can inherit foreground color if the foreground color is set to the default color. The background color attribute of input objects cannot be inherited. If the color attributes are \text{NULL} the input object uses the default foreground and background colors. Once an input object is created, a variable descriptor array is attached using \(\text{VOinPutVarList}\). The draw and erase flags are set by \(\text{VOinPutFlag}\). These flags indicate how to erase the input object and whether or not to draw the input object’s outline. Use \(\text{VOinTechnique}\) to attach the input technique object.
**Drawing and Erasing**

### Input Objects

You can draw an input object as part of a view using `TdpDraw`, or as an individual object using `TdpDrawObject`. Input objects become active when drawn and remain active until erased or deactivated by calling `TdpEraseObject` or `VOinState`. If an input object is covered by another object in the same drawport, it remains active and continues to take input even though it is not visible.

You can set the draw and erase flags by calling `VOInputFlag`. These flags are not valid for widgets. Draw and erase methods are explained in the *DV-Draw User’s Guide*.

---

**Input Technique Object**

The input technique object (`it`) maintains information that helps the input object perform its tasks. It contains an interaction handler and a template drawing object. It can also contain a list of action key origins, key bindings and an echo function. Depending on the type of interaction handler, it can also contain a list of text labels and a list of values for setting the input object’s variable.

To create an input technique object, pass a template drawing and an interaction handler to `VOitCreate`. The interaction handler and template are described later in this chapter.
An echo function is a programmer-defined routine that is called whenever an input object is drawn, erased, or updated, or whenever it accepts input. Echo functions let you customize the behavior of an input object when it receives an event and are attached using `VOitPutEchoFunction`. Echo functions are described in the Event Handling chapter.

### Handling Lists

Some interaction types require lists of values, text strings, and objects. The list of values is used to associate values with the items in menus, toggles, checklists, multiplexors and combiners. The list of text labels is used to supply text strings for the labels in checklists, toggles, menus, and multiplexors that use text or button items. The list of text labels can also be used for button input objects, but only the first label is displayed. The list of objects is used to embed input objects in combiners or multiplexors. Most value and text string lists can be set in DV-Draw; object lists and all lists for combiners and multiplexors must be set in DV-Tools. To associate a list with an input object, create the list then call `VOitPutList`, passing in the input technique object, the type of list (`TEXT_LIST`), the list, and the number of items in the list.

### Action Keys

Action-key bindings specify what key presses within an input object result in certain predefined actions. Actions include Select, Done, Restore, and Cancel, as described in the DV-Draw User’s Guide. An input object can use the global action key bindings or its own local key bindings. Global action-key bindings can be set in DV-Draw, in the configuration file, or by using `VUerPutKeys`. Local action-key bindings can be set in DV-Draw, or by using `VOitPutKeys` then calling `VOitKeyOrigin` to change the precedence to the local key bindings.

A key press in an input object results in an event, which is handled by the event handler. For information about the event handler, see the Event Handling chapter.

### Interaction Handlers

The interaction handler (`ih`) is a collection of routines that determine how the input object interacts with the user. These routines are broken down into functional tasks: the initial draw, accepting input, drawing update, redrawing, and erasing. Drawing and erasing are explicitly accessed by the user. Accepting input is implicitly performed by the DataViews event handler. Updating the input object can be done by either the user or the internal routines.
The interaction handler specifies the input object type, and interprets the information in the template, input technique object, and input object. This information determines the physical appearance of the input object and the contents of the variable descriptor during the course of the interaction. Different interaction handlers require different template settings, action key bindings, number and type of variable descriptors, and different function calls to set up the input object. For the specific function calls that can be used to set up each type of input object, see the Interaction Summary table at the end of this chapter.

The interaction handler is an entry point to a set of internal routines. These routines begin with the prefix VN. They must be externally referenced using a GLOBALREF declaration.

The following list shows which interaction handlers are used by which input object types.

<table>
<thead>
<tr>
<th>Interaction Handler</th>
<th>Input Object Type in DV-Draw</th>
</tr>
</thead>
<tbody>
<tr>
<td>VNbutton</td>
<td>Button</td>
</tr>
<tr>
<td>VNchecklist</td>
<td>Object Checklist, Text Checklist</td>
</tr>
<tr>
<td>VNcombiner</td>
<td>Not available in DV-Draw</td>
</tr>
<tr>
<td>VNmenu</td>
<td>Object Menu, Text Menu</td>
</tr>
<tr>
<td>VNmultiplexor</td>
<td>Not available in DV-Draw</td>
</tr>
<tr>
<td>VNpalette</td>
<td>Palette</td>
</tr>
<tr>
<td>VNsider</td>
<td>Slider, Scrollbar</td>
</tr>
<tr>
<td>VNsider2D</td>
<td>Slider2D</td>
</tr>
<tr>
<td>VNtext</td>
<td>Text Entry</td>
</tr>
<tr>
<td>VNtextedit</td>
<td>Text Editor</td>
</tr>
<tr>
<td>VNsoggle</td>
<td>Object Toggle, Text Toggle</td>
</tr>
<tr>
<td>VNwcheck</td>
<td>Motif Checklist, OPEN LOOK Checklist</td>
</tr>
<tr>
<td>VNwmenu</td>
<td>Motif Menu, OPEN LOOK Menu</td>
</tr>
<tr>
<td>VNwradio</td>
<td>Motif Radio, OPEN LOOK Radio</td>
</tr>
<tr>
<td>VNwsider</td>
<td>Motif Slider, OPEN LOOK Slider</td>
</tr>
<tr>
<td>VNwtext</td>
<td>Motif Text Entry, OPEN LOOK Text Entry</td>
</tr>
<tr>
<td>VNwtoggle</td>
<td>Motif Text Toggle, OPEN LOOK Text Toggle</td>
</tr>
</tbody>
</table>

The template is a drawing object that defines the physical layout and functional elements of the interaction. It is a graphical description of the user interaction that is parsed by the interaction handler. Templates can be created...
in DV-Draw and stored as view files. Editing the template lets you change the appearance and details of the input object’s behavior without relinking the application.

A template can be used by only one type interaction handler but can be used by several input objects of the same type. The text object VType.flag identifies the interaction handler the template will be used with. Objects in the template must be named according to the naming convention described in the Editing Input Objects chapter in the DV-Draw User’s Guide. Motif and OPEN LOOK interaction handlers are described in the Using DataViews with X manual.

You can edit a template attached to an input object at any time. However, you must redraw the input object to see the effect of your edits. You can access the template using the following calls:

```c
/* Get the input technique from the input object and then the template */
MenuIt = VOinTechnique (MenuIn, DONT_SET_THE_VALUE);
MenuTemplate = VOitGetTemplate (MenuIt);

/* Make modifications to the template ... */
/* then redraw the template */
TdpDrawObject (drawport, MenuIn);
```

To attach a different template to the input technique object use the following calls:

```c
/* Get the input technique object attached to the input object */
MenuIt = VOinTechnique (MenuIn, DONT_SET_THE_VALUE);
VOitPutTemplate (MenuIt, NewMenuTemplate);
TdpDrawObject (drawport, MenuIn);
```

Another way to change the appearance of an input object is by using VOinGetInternal, which is described in the DV-Tools Reference Manual.

To handle input, an input object accepts input, stores the input in its variable, and is redrawn to echo the current input. To accomplish these tasks, the input object interacts with the event handler. When an input object is drawn, its interaction handler posts event requests internally. These requests check for specified key and mouse presses and cursor motion in specified areas within the input object. For example, a slider input object posts a request for motion
Making Input Objects Do Useful Work

The internal input handling loop sets a value in a variable descriptor, and updates the input object to reflect the current value. To make something happen as the result of the input, such as controlling the interface, application, or an output object, you must interpret the variable’s value and perform the appropriate action. To know when a user input has changed the variable, you can either continually monitor that variable or post an event request with the event handler so it notifies you when the user interacts with the input object.

If input needs to be checked only after a user event, you can post an event request with the event handler to call your function to handle the menu selection when it occurs. Your function checks the variable value and performs the appropriate action, such as switching screens or setting application states.

If the input value needs to be monitored regardless of user input, you can constantly poll the value and perform any associated actions. This is not as useful as posting requests with the event handler.

If an input object and an output object share a data buffer, user input affecting the buffer is reflected when the output object is updated.

Complex Interaction Handlers

The combiner, VNcombiner, and multiplexor, VNmultiplexor, handlers are complex input objects composed of multiple simple input objects. A combiner input object combines two or more input objects into a single input object. You can interact with any input object in a combiner at any time. A
multiplexor input object is a menu in which the choices are embedded input objects. You can only interact with an input object in a multiplexor after selecting it in the menu. The following figures show a combiner and a multiplexor each with two embedded sliders:

Combiners and multiplexors can only be created using DV-Tools. When combiners and multiplexors are drawn, they clone their embedded input objects, so changing the simple input objects after they are embedded does not change them in the combiner or multiplexor. If you need to access an embedded input object, give it an echo function. Echo functions are described in the next chapter.

The following function calls create, run, and terminate a combiner input object. For a complete example of creating and using a combiner, see in_create.c in the examples directory.

**In the initialization phase:**

Create the input object:

```c
CombinerIn = VOinCreate (ll_pt, ur_pt, &Attributes);
```

Set up the vdp:

```c
VOinPutVarList (CombinerIn, Vdps, NUM_OF_INVDPS);
```

Set the flags:

```c
VOinPutFlag (CombinerIn, SAVE_RASTER, NO, ERASE_METHOD, NO_ERASE);
```

Get the template:

```c
TemplateView = TviLoad ("combiner.lay");
TemplateDr = TviGetDrawing (TemplateView);
```

Create the technique object:

```c
CombinerIt = VOitCreate (VNcombiner, TemplateDr);
```

Destroy the used view:

```c
TviDestroy (TemplateView);
```

Embed the input objects:

```c
VOitPutList (CombinerIt, OBJECT_LIST, (ADDRESS)InObs, NUM_OF_INOBS);
```
### Interaction Summary

This table summarizes most of the differences in the function calls required when using different kinds of interaction handlers.

<table>
<thead>
<tr>
<th>VOinPutVarList</th>
<th>Template</th>
<th>VOitPutList</th>
<th>VOitPutListValues</th>
<th>VOitPutKeys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>1 non-text</td>
<td>req</td>
<td>1 string</td>
<td>--</td>
</tr>
<tr>
<td>Object Menu</td>
<td>1 non-text</td>
<td>req</td>
<td>--</td>
<td>0 or n</td>
</tr>
<tr>
<td>Text Menu</td>
<td>1 non-text</td>
<td>opt</td>
<td>n strings</td>
<td>0 or n</td>
</tr>
<tr>
<td>Slider</td>
<td>1 non-text</td>
<td>opt</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>1 non-text</td>
<td>req</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Text Entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Line</td>
<td>1 text</td>
<td>opt</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Text Entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-Line</td>
<td>1 text</td>
<td>opt</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
This chapter has presented these basic techniques:

- How to create and manipulate input objects.
- How to create multiplexor and combiner input objects, which are not available in DV-Draw.
- How to signal special actions using action keys.

Using these techniques, you can implement the following features in your application:

- Change the range of a slider or the choices in a text menu at run-time.
- Use combiners to handle the input from a group of input objects.

Since input objects are closely tied to event handling, more features using input objects are described in the *Event Handling* chapter.
## Chapter 8

### Event Handling

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>8-1</td>
</tr>
<tr>
<td>Useful Data Structures</td>
<td>8-3</td>
</tr>
<tr>
<td>What To Do in DV-Draw</td>
<td>8-3</td>
</tr>
<tr>
<td>What To Do in DV-Tools</td>
<td>8-3</td>
</tr>
<tr>
<td>How This Chapter Is Organized</td>
<td>8-4</td>
</tr>
<tr>
<td>Basic Function Calls</td>
<td>8-4</td>
</tr>
<tr>
<td>Polling for Events</td>
<td>8-5</td>
</tr>
<tr>
<td>Wait or No Wait</td>
<td>8-5</td>
</tr>
<tr>
<td>Polling Multiple Windows</td>
<td>8-6</td>
</tr>
<tr>
<td>Location Objects</td>
<td>8-7</td>
</tr>
<tr>
<td>Masking Events</td>
<td>8-7</td>
</tr>
<tr>
<td>Types of Events</td>
<td>8-8</td>
</tr>
<tr>
<td>The Event Handling Loop</td>
<td>8-9</td>
</tr>
<tr>
<td>The Primary Loop</td>
<td>8-9</td>
</tr>
<tr>
<td>The Secondary Loop</td>
<td>8-10</td>
</tr>
<tr>
<td>Posting Event Requests</td>
<td>8-11</td>
</tr>
<tr>
<td>The Client ID</td>
<td>8-12</td>
</tr>
<tr>
<td>The Function Pointer</td>
<td>8-13</td>
</tr>
<tr>
<td>Boundary Event Types</td>
<td>8-13</td>
</tr>
<tr>
<td>Object and Screen Region Information</td>
<td>8-14</td>
</tr>
<tr>
<td>Key and Mouse Information</td>
<td>8-15</td>
</tr>
<tr>
<td>Window Event Types</td>
<td>8-16</td>
</tr>
<tr>
<td>Label, Args, and Argsize</td>
<td>8-17</td>
</tr>
<tr>
<td>Priority of Requests</td>
<td>8-18</td>
</tr>
<tr>
<td>Managing Requests</td>
<td>8-19</td>
</tr>
<tr>
<td>Service Routines</td>
<td>8-20</td>
</tr>
<tr>
<td>Parameters for the Service Routine</td>
<td>8-20</td>
</tr>
<tr>
<td>EventRequest</td>
<td>8-20</td>
</tr>
<tr>
<td>Location Object</td>
<td>8-21</td>
</tr>
<tr>
<td>Example: Changing the Interface</td>
<td>8-22</td>
</tr>
<tr>
<td>Example: Changing the Control Flow</td>
<td>8-24</td>
</tr>
<tr>
<td>How Input Objects Use the Event Handler</td>
<td>8-26</td>
</tr>
<tr>
<td>Posting Service Result Requests</td>
<td>8-28</td>
</tr>
<tr>
<td>Examples of Service Result Requests</td>
<td>8-29</td>
</tr>
<tr>
<td>Example of a Service Result Routine</td>
<td>8-30</td>
</tr>
<tr>
<td>Event Handling</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Echo Functions</td>
<td>8-31</td>
</tr>
<tr>
<td>When to Use Echo Functions</td>
<td>8-33</td>
</tr>
<tr>
<td>Handling Window Expose and Resize</td>
<td>8-34</td>
</tr>
<tr>
<td>Explicit Event Handling</td>
<td>8-35</td>
</tr>
<tr>
<td>Handling Device-Specific Window Events</td>
<td>8-37</td>
</tr>
<tr>
<td>Setting Up Location Objects</td>
<td>8-38</td>
</tr>
<tr>
<td>Building the Control Loop</td>
<td>8-40</td>
</tr>
<tr>
<td>Handling User Input in the Control Loop</td>
<td>8-40</td>
</tr>
<tr>
<td>Handling Application Data in the Control Loop</td>
<td>8-41</td>
</tr>
<tr>
<td>Summary</td>
<td>8-42</td>
</tr>
</tbody>
</table>
This chapter discusses the following topics:

- Polling for Events
- The Event Handling Loop
- Posting Event Requests
- Service Routines
- Posting Service Result Requests
- Echo Functions
- Handling Window Expose and Resize Events
- Explicit Event Handling
- Building the Control Loop

Overview

Besides letting you display information to the user in a variety of ways, DV-Tools also lets you get information from the user. This is a critical part of program control flow, since the interface or the application generally changes as a result of user commands. User commands are processed in two stages: **event gathering** and **event handling**.

Event gathering, also called **polling**, determines what action, if any, the user has taken. In DV-Tools, these user actions are called **events**.

Events include:

- mouse button presses or releases
- key presses or releases
- mouse movements
- window events such as expose, resize, or window quit

Event handling is your program’s response to events. To specify these responses, you post **event requests**. Each event request defines a triggering event and specifies a **service routine** to be called whenever the triggering event occurs. The service routine is written by you. This concept is similar to a callback in the X Window System or a window procedure in Windows.
An event request can be associated with an object, a region of the screen, or a particular window. For example, an event request can be triggered by a mouse pick anywhere on the screen, or can require a pick inside or outside a certain object. This feature lets you designate any object or rectangular screen region as a pickable hot spot.

When an event is received, you pass it to the DV-Tools event handler, a routine that checks the event against the list of posted requests. If the event satisfies one of the posted requests, the event handler calls the associated service routine.

The typical event handling model falls into three phases which correspond to the three main phases of any DV-Tools program: initialization, control, and termination.

In the initialization phase:
- Post event requests responding to specific events.
- Set an optional mask to filter out events you do not want to process.

In the control loop:
- Poll for events (requires one function call).
- Handle events (requires one function call).

In the termination phase:
- Terminate all posted requests.

This code model has the advantage of a tight control loop that is easy to debug and maintain. An event request can be added or removed by writing or deleting a line of code. Another advantage is flexibility. You can activate or deactivate event requests at run time, thus changing the behavior of your program to suit the context without having to add branches to your control loop.

You do not have to handle all events through the event handler. Events are returned in a DataViews device-independent structure called a location object. You can get information about the event from this structure and process events explicitly if you prefer this method.

Service routines can return results, called service results, and you can post requests, called service result requests, for specific results of a service routine. When the event handler detects that a service routine has returned a
specify a result, it calls an additional routine called a **service result routine**. By using a series of service result requests, you can set up a chain of routines to be called in response to a single event.

Service result requests are most frequently used on input objects. Unlike other objects, input objects already have internal event requests and service routines which redraw them and update their associated variables in response to user actions. To add application-specific functionality to an input object, post a service result request on the input object’s internal service routine.

### Useful Data Structures

The following data structures are used for event handling: the location object \(lo\), the event request \(er\), the service routine \(sr\), and the **echo function** \(ef\). Echo functions are discussed later in this chapter. The screen \(sc\), the input object \(in\), and other graphical objects are used to determine the context in which events are accepted.

Some of the data structures described in this chapter do not appear in the structural diagram shown in previous chapters. They are illustrated in separate functional diagrams in this chapter.

### What To Do in DV-Draw

To get certain types of user input, create input objects in your view. In both DV-Draw and DV-Tools, input objects have internal event requests that update their appearance and the variables associated with them.

Choose objects in the view that you want to be selectable buttons or pickable hot spots. Name these objects to make it easier to post event requests on them. For easy selection, make these objects either filled or transparent.

You can prototype event handling behavior in a limited way using the rules feature in DV-Draw. Note, however, that rules do not transfer directly to DV-Tools when you play back your views, nor do they cover the full range of functionality available through the DV-Tools event handler.

### What To Do in DV-Tools

Post event requests on specific objects, on rectangular screen regions, on specified windows, or without restriction.

Write service routines to act on events. If you used rules to prototype behavior in DV-Draw, write routines to produce the behavior emulated by those rules. Service routines let you go beyond the limited functionality of the Rules Menu and can perform any task needed by your application.
Input objects already have event requests that take care of updating the input objects and the variables associated with them. To extend this minimal functionality, post service result requests on the input objects.

You can replace the default service routine for an input object by supplying an echo function.

The most complex task for the DV-Tools programmer is posting requests for the event handler, usually in the initialization phase. Once requests are posted, the event handler takes care of processing them appropriately for each event received. Therefore, this chapter begins with a discussion of the simpler topics of the polling procedure and the event handling loop, even though these occur later in your program code. It then goes on to discuss the more complex subject of posting your requests to get the results you want.

The following table lists the basic calls for gathering and handling events in three main phases of a DV-Tools program. These routines are all discussed in this chapter.

**In the initialization phase:**

- Set an event mask:
  ```c
  VOoscWinEventMask ((ULONG) V_KEYPRESS | V_BUTTONPRESS | V_WINDOW_QUIT, (ULONG) 0);
  ```

- Post a boundary event request:
  ```c
  er = VUerBoundaryEventPost (Obj1, YourFcn1, YourStruct, sizeof (YourStruct),
  YourLabel, VUER_DOE_EVENT, V_BUTTONPRESS, button[1], Obj1, Scr1Xform, V_INSIDE);
  ```

- Post a window event request:
  ```c
  er = VUerWinEventPost (Scr1, YourQuitFcn, YourStruct, sizeof (YourStruct),
  YourLabel, V_WINDOW_QUIT);
  ```

- Post a service result request:
  ```c
  er = VUerServiceResultPost (Scr2, YourFcn2, YourStruct, sizeof (YourStruct),
  Slider1, INPUT_DONE, YourLabel);
  ```

- Attach an echo function:
  ```c
  VOitPutEchoFunction (MenuIt, EchoMenu, NULL, 0);
  ```

**In the run phase:**

- Poll for an event:
  ```c
  location = VOloWinEventPoll (V_NO_WAIT);
  ```
When the user takes a discrete action with a device such as a keyboard or mouse, the operating system packages the action in a system-dependent event structure, which is placed in the system event queue. The DataViews polling routine, \texttt{VOloWinEventPoll}, takes the oldest event off the system event queue and converts it to a DataViews location object. The location object can be passed directly to the event handler for processing of all posted requests.

DataViews uses an event mask to determine what types of events to return to your program. You can set or reset the mask at any time, but you usually set it on each screen as you open it.

The following figure illustrates the event gathering sequence:

When polling for events, you can either update at regular intervals or wait until one of the specified masked events occurs. The following function call is used to poll for events:

\[
\text{location} = \text{VOloWinEventPoll} \text{ (mode)};
\]
The *mode* flag can be:

- **V_WAIT** Stops execution until a masked event occurs.
- **V_NO_WAIT** Returns immediately, with or without a masked event to process. Returns *NULL* if no event was received.

Use **V_WAIT** if you want program execution to wait for user input before proceeding.

Use **V_NO_WAIT** when you need continuous dynamics in your views while waiting for a user command. For example, you can poll periodically with **V_NO_WAIT**, then update dynamics whether or not a command was received.

The following code fragment shows how **V_NO_WAIT** might be used in a simple control loop. **VUerHandleLocEvent** is the event handler, discussed later. Note that if you use **V_NO_WAIT**, you must verify that a valid (non-*NULL*) event was received before you call the event handler.

```c
while (!done)
{
    location = VOloWinEventPoll (V_NO_WAIT);
    if (location)
    {
        /* Call event handler only if event was received. */
        VUerHandleLocEvent (location);
    } /* Update dynamics. */
    TdpDrawNext ( . . .);
}
```

### Polling Multiple Windows

In a multiple screen environment, you can either poll all the screens or poll a particular screen. The usual method is to poll all windows simultaneously using **VOloWinEventPoll**. This routine returns a location object for events that occur in any screen. Events are returned in the order in which they occur. **VOloWinEventPoll** only returns events from screens that were created using the same device name as the current screen, so all your screens should be the same kind of device, such as an X window.

To poll just one screen, set the screen to be polled to the current screen and poll using **VOscWinEventPoll**. This routine is similar to **VOloWinEventPoll** except that it only polls the current screen. Events on other screens are ignored and remain on their event queues.
**Location Objects**

The location object stores detailed information about the event such as the type of event, the key or button that was pressed, and the screen and coordinates where the event occurred. The following illustration shows the components of the location object:

![Location Object Diagram]

In many cases, you can simply verify that VOloWinEventPoll returned a valid (non-NULL) location object, then pass the location object directly to the event handler for internal processing without further examination. However, VOlo routines are available to get information from the location object. These routines are discussed in the Service Routines section later in this chapter.

Each polling call returns new information to the same location object. To store the information in the location object for later use, you must clone the location object or extract the information you need and store it in your own data structure before your next polling call.

**Masking Events**

DataViews uses an event mask to control what types of events are returned by the polling routine. Events whose types are not in the mask are ignored. The default mask on each screen includes:

- key presses
- button presses
- key releases
- button releases
- expose and resize events
- enter and leave events
- pointer motion events
- window iconify events
- window quit events
You can set a different mask on any screen using `VOscWinEventMask`. For example, to filter out all events except key presses and button presses, use the following call:

```c
VOscWinEventMask ((ULONG) V_KEYPRESS | V_BUTTONPRESS,
                  (ULONG) 0);
```

In this call, the first parameter is the standard event mask, composed of `ULONG` flags bitwise ORed together. Each flag represents a type of event to be returned. When you set this mask, you must make sure it will pass all the event types that the application needs. For example, if you have button input objects on a screen, you may also need to poll for key and button releases and pointer motion in addition to key and button presses. Otherwise, the button input objects may not behave as expected.

The second parameter in this call is a system-specific event mask which lets you capture special system events. For example, if you pass the `XColormapChangeMask` as the second parameter, you can capture events relating to colormaps in addition to the standard events of the standard event mask.

If you use the system-specific mask, you must OR an additional flag into the standard event mask to tell DataViews how to interpret the second mask. In this example, the first parameter should include `V_XWINDOW_MASK` in its list of flags. For a complete list of standard event types and flags indicating mask extensions, see the description of `VOscWinEventMask` in the DV-Tools Reference Manual.

Each call to `VOscWinEventMask` applies only to the current screen. The simplest way to set masks on all your screens is to call `VOscWinEventMask` immediately after you open each screen. To set or change the mask for a window at other times, make sure the window you want is current before calling `VOscWinEventMask`. Calling `TscSetCurrentScreen` first is one way to do this.

---

**Types of Events**

There are two basic kinds of events: **boundary events** and **window events**. Boundary events combine a keyboard or mouse event with the boundary of some region of the DataViews screen. The region can be a particular drawport, any rectangle, any graphical object, or the whole screen. You can
specify boundary events inside or outside the region. This capability lets you turn any graphical object or region of the screen into a pickable hot spot. The designated region can be:

- inside a specified object
- outside a specified object
- inside a rectangular region of the screen
- outside a rectangular region of the screen
- anywhere on the screen

Boundary events can specify:

- key presses
- key releases
- mouse button presses
- mouse button releases
- mouse motion

Window events apply only to windowing systems. They are returned by your system’s window manager and include such events as window expose and window quit. The handling of window events by DataViews supersedes the handling of these events by the window manager. In order for your views to redraw and resize properly when windows are exposed and resized, your program must include code for handling these types of events. The procedure for doing this is discussed in the Handling Window Expose and Resize section.

---

**The Event Handling Loop**

The event handler is invoked when a valid location object is passed to `VuerHandleLocEvent`. The event handler may do nothing in response to the event, or it may call one or more service routines. You specify how the event handler responds to events by posting requests beforehand. This section discusses the general behavior of the event handler. The following sections describe in detail the types of event requests that are available and how to post them.

**The Primary Loop**

The primary loop of the event handler checks the location object against the list of posted event requests. Event requests are kept in a list whose order is determined by the type of event and the order in which events were posted. It is possible for an event to satisfy more than one event request, but in most
cases the event handler services only the first matching request that it finds. The event handler’s rules for which events are serviced, and in what order, are discussed below in the Priority of Requests section.

If the event is found to match a posted request, VUerHandleLocEvent calls the service routine designated by that request. The service routine may return a result, which is then passed to the secondary loop for checking. The internally posted event requests of input objects are also serviced in the primary loop.

The following figure illustrates the primary loop of the event handler:

**The Secondary Loop**

If an event does not match any event requests in the primary loop, the event handler returns and never progresses to the secondary loop. If an event satisfies an event request, the primary loop passes it to the corresponding service routine, which returns a service result. This service result is then passed to the secondary loop for processing.

In the secondary loop, the event handler checks the service result against the list of service result requests, and calls any service result routines that apply. If more than one service result request is satisfied, all of their service result routines are executed.

Service result requests are processed by a recursive procedure. Each service result routine can return a result, which in turn is checked against the list of service result requests. The service result is of type int. A zero result, or a result that does not satisfy any request, breaks out of the recursive loop and is
returned by \textit{VUserHandleLocEvent}. It is good practice, however, to return a result from all service routines. Be careful not to return a result that causes the same routine to be called again, since this situation causes an infinite recursion.

The following figure illustrates both loops of the event handler:

\begin{itemize}
  \item Event requests and service result requests must be posted before \textit{VUserHandleLocEvent} can process them. Every event request includes a pointer to a service routine, which is called when the event request is satisfied. The service routine returns a service result, which can in turn satisfy a service result request. This section discusses event requests, which are processed in the primary loop of the event handler. Service result requests are discussed later in this chapter.
\end{itemize}
DV-Tools has three major routines for posting event requests. Each of these routines posts requests for a particular kind of event:

- **VUerBoundaryEventPost**: Key, button, or motion events with respect to rectangular regions of the screen or graphical objects.
- **VUerWinEventPost**: Window events.
- **VUerCatchAllEventPost**: Any event.

All event request posting routines return the private type `EVENT_REQUEST`, which can be passed to the routines `VUerDeactivate` and `VUerActivate` to deactivate or reactivate the request at run time. The `EVENT_REQUEST` is also passed internally to the service routine. This can be used later to identify which event request resulted in the service routine being called.

Each posting routine takes different parameters, defining which service routine is called, what information is passed to it, and exactly which event triggers the request. Possible parameters include:

- **Client**: the client id that identifies the client making the request
- **Fcn**: a pointer to the service routine to be called
- **ErInterpretation**: the type of boundary event (for `VUerBoundaryEventPost`)
- **BndingRect, EdgeObj, InOut**: object and screen region information for boundary events (for `VUerBoundaryEventPost`)
- **PickEventType** and **PickSyms**: key and mouse information (for `VUerBoundaryEventPost`)
- **WinEventType**: the type of window event (for `VUerWinEventPost`)
- **Args**: an argument structure that provides information
- **ArgSize**: the size of the argument structure
- **Label**: a label that identifies this event request
- **XformObj**: the transform object required for converting the graphical object’s world coordinates to screen coordinates

These parameters are discussed in the following sections. For complete details about these and other parameters, see the `VUerPost` module in the **DV-Tools Reference Manual**.

### The Client ID

The client ID parameter identifies the client making the request. The client ID can be an `OBJECT`, or an integer cast to type `OBJECT`. This section describes some additional uses for the client ID.
An important use of the client ID is grouping event requests. You can deactivate, reactive, or delete a group of event requests that share a client. For example, to group all requests posted on a certain screen, you can pass that screen as the client ID for all requests and deactivate all of them with one call to `VUerDeactivateClient`. The routines that do this are discussed later in the *Managing Requests* section.

The client ID can also be used to supply information for the service routine. For example, if you have a service routine that erases an object, you can pass the object to be erased as the client ID for the request.

Input objects should not be used as clients posting event requests because it would interfere with requests that the input objects post internally. For details on how to respond to events on input objects, see the *Posting Service Result Requests* section below.

**The Function Pointer**

The `Fcn` parameter simply specifies a pointer to the service routine to be called when the event request is satisfied.

**Boundary Event Types**

The `ErInterpretation` parameter identifies the type of boundary event being posted for by the generic routine `VUerBoundaryEventPost`. There are five types of boundary events. `VUerBoundaryEventPost` can post event requests for all five types. `VUerBoundaryEventPost` determines the type of event based on the value of the `ErInterpretation` flag, and takes additional parameters depending on the type of event specified. The boundary event types and their corresponding flags are:

- **Simple Edge**  
  `VUER_SE_EVENT`: A key or button event anywhere on the screen.

- **Boundary Edge**  
  `VUER_BRE_EVENT`: A key or button event inside or outside a rectangular region specified in screen coordinates.

- **Object Edge**  
  `VUER_DOE_EVENT`: A key or button event inside or outside a visible graphical object.

- **Position**  
  `VUER_POS_EVENT`: A motion or position event inside or outside a rectangular region specified in screen coordinates.

- **Object Position**  
  `VUER_OPOS_EVENT`: A motion or position event inside or outside a visible graphical object.
Other routines also let you post event requests for object edge and object position events. `VUerObjectEdgePost` posts requests for object edge and object position events. `VUerRectEdgePost` posts requests for simple edge, boundary edge, and position events. These routines and their parameters are described in the *DV-Tools Reference Manual*.

Requests for position or object position events require `V_MOTIONNOTIFY` in the event mask. For all the interpretation options except simple edge, you must specify whether you are looking for an event inside or outside the region or object by using the `InOut` flag of the posting routines. The `InOut` flag is described later in this section.

The `BndingRect`, `EdgeObj`, and `InOut` parameters provide additional information about boundary events when you use `VUerBoundaryEventPost`. This lets you monitor for events in specific locations, such as a button press inside a rectangular region. The `ErInterpretation` flag determines what object or screen information you must supply.

The `BndingRect` parameter is 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. For example, you may want to specify the background of a drawport when the cursor moves into it.

The `EdgeObj` parameter specifies 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. The object must be visible for the request to be serviced. For objects with a fill status of `EDGE`, the location object is outside the object unless it directly intersects the edge of the object. If you want picks inside the object’s area but not on the edge to trigger the request, make the object `DV_TRANSPARENT` or filled.

The `InOut` parameter is a flag that specifies 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`.

The `XformObj` parameter is required for converting the graphical object’s world coordinates to screen coordinates when you post for object edge and object position events.

The following code fragment sets up requests for button presses inside and outside the bounding box of an object. For the complete example, see the example program `event_post.c`. 

**Object and Screen Region Information**

---

| 8 - 14 | DV-Tools User’s Guide |
Event Handling

Posting Event Requests

/* Get the bounding box of the object to use as the region in the event request.*/
VOObBox (info->poly[i], &wvp, &svp_delta);

/* Transform the bounding box in screen coordinates. */
TdpWorldToScreen (info->drawport, &wvp.ur, &svp.ur);
TdpWorldToScreen (info->drawport, &wvp.ll, &svp.ll);

/* Sort to ensure that the coordinates are in the correct order. */
VOuVpSort (&svp);

/* Post a request for picks inside the bounding box. Call DrawMark to mark the
doctor when this happens.*/
VUerBoundaryEventPost ((OBJECT) 2, DrawMark,
(Address) info, sizeof (INFO), i,
VUER_BRE_EVENT, V_BUTTONPRESS,
buttonsyms, &svp, V_INSIDE);

/* Post a request for picks outside the bounding box. This means that the object
is deselected. Call ClearMark to erase the mark when this happens.*/
VUerBoundaryEventPost ((OBJECT) 2, ClearMark,
(Address) info, sizeof (INFO), i,
VUER_BRE_EVENT, V_BUTTONPRESS,
buttonsyms, &svp, V_OUTSIDE);

Note that when you resize a drawport or screen, you must repost event
requests for objects or regions within it, supplying new screen coordinates for
boundary edge and position event requests, and new transforms for object
ege and object position event requests. This does not apply to input objects,
since they repost their own event requests.

Key and Mouse
Information

The PickEventType and PickSyms parameters lets you specify mouse or
keyboard events when you post for simple edge (VUER_SE_EVENT),
boundary edge (VUER_BRE_EVENT), or object edge
(VUER_DOE_EVENT) using VUerBoundaryEventPost. You can designate
these events to accept any keystroke or mouse button, or a limited set of keys
or buttons.

To indicate a mouse or keypress event, set PickEventType to
V_BUTTONPRESS, V_BUTTONRELEASE, V_KEYPRESS, or
V_KEYRELEASE. Each event request can handle events of only one type.
Therefore, to call the same service routine for both a mouse and keyboard
event, such as calling user_quit in response to either the <Q> key or the right
mouse button, you must post two separate event requests.

To indicate specific keystrokes or buttons, set up PickSyms, which is a zero-
terminated array. Valid values for PickSyms are defined in GRkeysymdef.h.
GRkeysymdef.h maps different sets of defines to the mouse buttons and
keyboard keys, including keypad and function keys. To specify a single set of defines, edit GRkeysym.h so only one set is defined. Buttons are usually numbered with integer values from left to right starting with 1.

The following example sets up PickSyms to look for the function key <F1> using the X11 Latin 1 set of defines.

```c
PickSyms[0] = XK_F1;
PICKSyms[1] = 0;
```

You can look for more than one key. For example, the following example sets up PickSyms to look for the picks <q>, <Q>, or <F10>, which signal a quit:

```c
PickSyms[0] = XK_q;
PICKSyms[1] = XK_Q;
PICKSyms[2] = XK_F10;
PICKSyms[3] = 0;
```

You can make your request unrestricted by setting the first element of PickSyms to 0. This lets any key or button trigger the event request:

```c
PickSyms[0] = 0;
```

The event posting routines copy PickSyms, so you do not have to preserve the original array after you post the request.

**Window Event Types**

The WinEventType parameter identifies the type of window event being posted for by VUerWinEventPost. VUerWinEventPost uses the same flags used by the event mask to identify types of window events. The window event types and their corresponding flags are:

- **Resize** (VUER_RESIZE_EVENT) A resize event.
- **Window Quit** (VUER_WINQUIT_EVENT) A window quit event.
- **Iconify** (VUER_ICONIFY Event) An iconify event.
- **Expose** (VUER_EXPOSE_EVENT) A expose event.
- **Enter** (VUER_WIN_ENTER_EVENT) A window enter event.
- **Leave** (VUER_WIN_LEAVE_EVENT) A window leave event.
Note that \texttt{V\_WINDOW\_QUIT} should be included in your event mask to make sure events of that type are not filtered out.

The following function call posts an event request for a window quit event:

\begin{verbatim}
er = VUerWinEventPost (Screen1, TerminateScreen, NULL, 0, Label, VUER\_WINQUIT\_EVENT);
\end{verbatim}

Window event requests apply only to the screen that was current at the time of posting.

Window expose and resize events must be handled explicitly in DataView. The standard procedure for doing this is discussed later in this chapter, in the \textit{Handling Window Expose and Resize} section.

\textbf{Label, Args, and Argsize}

The \texttt{Label}, \texttt{Args}, and \texttt{Argsize} parameters let you pass information to the service function. This can be any information you want the service routine to have, such as information about the event that is not provided by the location object and other parameters, or information that is not about the event at all. These parameters are used by both \texttt{VUerBoundaryEventPost} and \texttt{VUerWinEventPost}, as well as several other routines for posting event requests.

The \texttt{Label} parameter is an integer that identifies this event request. For example, you can use \texttt{Label} to identify which request called a service routine that is called by several event requests. You can then \texttt{switch} on the value of \texttt{Label} in your service routine.

The \texttt{Args} parameter can be the address of any variable or structure, and \texttt{Argsize} indicates the size of the structure. Pass \texttt{NULL} for both parameters if you do not use \texttt{Args}.

Both \texttt{Label} and \texttt{Args} are copied at the time you post the event request, and the copy is used each time the event request is serviced. In other words, even if you use variables to post the event request, \texttt{Label} and \texttt{Args} behave like constants. To provide access to changing data in the service result routine, set up an argument structure containing pointers to the data that interests you. The fields of the structure must be preset with the correct addresses for the data before you post the event request.
The priority of event requests is an important consideration because it is possible for regions to overlap, so that a single event satisfies more than one boundary event request. For example, a display might have requests posted for mouse button 1 inside an object, inside a region containing the object, and anywhere on the screen, as shown in the following figure:

When the user clicks inside the object, DataTypes processes only one of these requests. Which one is processed depends on the order in which events are posted. In the case of other types of events such as outside events, more than one event request can be processed.

DataTypes maintains four lists of event requests, in the following order:

- window event requests
- inside and simple edge event requests
- outside event requests
- all other requests (the catch-all list)

On all lists, requests are checked in reverse order of posting, with the last request posted being the first one checked.

If the location object satisfies a window event request, VUserHandleLocEvent processes that request and exits. Only one window event is processed.

If there was no window event request, the event handler proceeds to the list of inside and simple edge events. If a match is found, the appropriate service routine is called. Only one event from this list is processed.

If there is no match, the event handler proceeds to the list of outside event requests, and if there is still no match, to the catch-all list. All requests on these lists that match the event are processed, even if there is more than one. However, if a match is found on the list of outside events, the handler does not check the catch-all list.
Note that, in the case of boundary events (but not window events), a service result of `INPUT_UNUSED` is treated as no match, and the event handler proceeds to the next list of requests.

---

### Managing Requests

After posting event or service result requests, you can manage them in the following ways:

- **Deactivating a request**: To deactivate a request, call `VUerDeactivate (er)`.
- **Activating a request**: To activate a request, call `VUerActivate (er)`.
- **Deactivating a group of requests**: To deactivate a group of requests that share the same client, call `VUerDeactivateClient (client)`.
- **Activating a group of requests**: To activate a group of requests that share the same client, call `VUerActivateClient (client)`.
- **Destroying a group of requests**: To destroy, or clear, a group of requests that share the same client, call `VUerClearAll (client)`.

Requests become active as soon as they are posted. They remain active until you deactivate or destroy them, even if the monitored objects or regions of the screens have been erased.

Events that are deactivated, whether individually or in a group, retain their positions in the event request list. The event handler simply passes over them when it processes events. When deactivated events are reactivated, their position in the list is unchanged. By contrast, if you clear requests and repost them, they are moved to the top of the event list.

You can activate and deactivate requests either individually or in a group with a common client. However, you can only clear requests in groups. Keep this in mind when you designate client IDs for your requests.

All requests must be cleared in the termination phase of your program.
Every event request contains a pointer to a service routine that is called when the request is satisfied. Since you define the service routines, the resulting actions are only limited by the form of your routine and the information available to it. Service routines can:

- change the interface
- control the application
- affect control flow by changing the event request list

A single service routine can be called by multiple event requests. In this case, the routine is called when any one of the requests is satisfied. Service routines can also be called in an unpredictable order depending on the sequence of user commands.

This section discusses the service routine parameters, which provide the service routine with information about the context in which it was called. An example demonstrates how a service routine can change the interface, and another example shows how a service routine can affect the control flow by changing the event request list.

Service routines have the following form:

```c
int function (  
    OBJECT Client,  
    EVENT_REQUEST EventRequest,  
    int Label,  
    OBJECT Location,  
    ADDRESS Args);
```

### Parameters for the Service Routine

The `Client`, `Label`, and `Args` parameters are specified when you post the event request. Their roles were discussed earlier in the *Posting Event Requests* section. `EventRequest` and `Location` are supplied by `VUerHandleLocEvent`. They represent the event request and the location object that triggered the function call.

### EventRequest

You can check the `EventRequest` parameter to determine which event request resulted in the call to the service routine.
**Location Object**

The *location* parameter passed the location object to the service routine. DV-Tools provides *VOlo* and *Tlo* routines for extracting information about the triggering event directly from location objects.

The following routines extract information from a location object returned by *VOloWinEventPoll* or *VOscWinEventPoll*:

- **VOloButton**: The mouse button associated with the event.
- **VOloKeyString**: The key string associated with the event. Normally one character long, although you can bind function keys to longer key strings in the `#include` file `GRkeysymdef.h`.
- **VOloKeySym**: The key symbol value associated with the event. This is a device-independent interpretation set. `GRkeysymdef.h` defines several sets of virtual key symbols, and `GRkeysym.h` determines which set is used.
- **VOloMaxPoint**: The screen coordinates of the point with the largest *x* and *y* values. Used to define the new window size.
- **VOloRegion**: The rectangle in screen coordinates that defines the exposed region, or the union of the exposed regions, of the window.
- **VOloState**: A mask indicating any modifier buttons or keys that were pressed during the event. For example, it records if a shift or control key was held down when the event occurred.
- **VOloType**: The type of event, such as key press, key release, or resize. Uses the same flags used to set the event mask.
- **VOloWinEventGet**: Copies the *WINEVENT* structure from the location object so that you can access any of the other *WINEVENT* fields. Note that this structure is overwritten along with the other information in the location object when you call the polling routine again.

The following routines extract information from any location object:

- **VOloKey**: The key or button pressed, if any.
- **VOloScpGet**: The event position in screen coordinates.
- **VOloWcpGet**: The event position in world coordinates.
- **VOloScreen**: The screen where the event occurred.
- **TloGetSelectedDrawport**: The drawport where the event occurred.
- **TloGetSelectedObject**: The visible object at the position where the event occurred.
Note that routines such as `TloGetSelectedObject`, `TloGetSelectedObjectName`, `TloGetSelectedSubObject`, and `TloGetSelectedSubObjectName` can only extract information about graphical objects that are visible.

The service routine can use the information it gets from the location object in a variety of ways. For example, a service routine can call `VOloScreen` to determine the screen on which the event occurred, and then make that screen current, as shown below:

```c
screen = VOloScreen (location);
(void) TscSetCurrentScreen (screen);
```

For another example of how a service routine can extract and use information from the location object, see the Handling Window Expose and Resize section.

You can also extract information to set up your own location object, as discussed in the Setting Up Location Objects section.

### The WINEVENT Structure

The location object includes a public structure called the WINEVENT that stores detailed information about the event. This information includes the event type, the time the event occurred, and the eventdata structure, which is a copy of the data structure that comes directly from the window system.

The WINEVENT structure is described in the Public Types section of DV-Tools Reference Manual.

### Example: Changing the Interface

The following code fragment, from the example program `event_post.c`, shows a service routine that changes the appearance of the interface. It can be called for any one of three polygon objects. This service routine draws the bounding box around the selected object and fills a mark to indicate that the object has been selected. The mark is a hollow circle that is already drawn in the view near its corresponding polygon.
To determine the context of the event, this routine uses:

Various fields of the argument structure, info. The array info->mark identifies the mark object for each polygon.

A global array, bound_box, containing the bounding boxes for each polygon.

The parameter label, a number between 0 and 2 that is used as a subscript in the two arrays. This parameter identifies which mark to fill and which bounding box to draw.

```c
LOCAL int DrawMark (OBJECT client, EVENT_REQUEST er, int label, OBJECT loc, ADDRESS args);
{
    ATTRIBUTES attr;
    INFO *info = (INFO *)args;

    /* Draw the bounding box and fill the mark object for the selected polygon. */
    TdpDrawObject (info->drawport, bound_box[label]);

    VOobAtGet (info->mark[label], &attr);
    if (attr.fill_status != mark_state[label])
    {
        TdpEraseObject (info->drawport, info->mark[label]);
        attr.fill_status = mark_state[label];

        VOobAtSet (info->mark[label], &attr);
        TdpDrawObject (info->drawport, info->mark[label]);
    }

    return INPUT_USED;
}
```
Service routines can also change the control flow by manipulating the event request list. The service routine can do this by activating or deactivating requests. The following code fragment shows a service routine that deactivates its own event request when it detects a stop flag in its argument structure:

```c
int HandleInput (  
    OBJECT client,  
    EVENT_REQUEST er,  
    int label,  
    OBJECT loc,  
    ADDRESS args);  
{  
    INFO *info = (INFO *) args;  
    ...  
    if (info->stop)  
        VUerDeactivate (er);  
    ...  
}
```

Your service routines can also post new requests, clear event requests, or change the location object and pass it back to the event handler. These practices should be used with care. Invoking the event handler from within a service routine causes an infinite loop if you do not change the location object, or if you change it so that it still triggers the same service routine. Posting and clearing requests from within service routines can also affect the priority of the request list.

The following code fragment shows another way that a service routine can manipulate the event request list. In this example, the service routine deactivates all posted boundary requests from the main program. Pointers to the main program’s requests are stored in a global array so that the service routine can access them. The service routine then pops up a new view, posts its own requests for that view, and goes into its own polling loop. When its polling loop is complete, it clears its own requests, removes the popup view from the screen, and reactivates the requests for the main program before exiting. This behavior is similar to that of a modal dialog in X.
LOCAL EVENT_REQUEST *boundary_events [20];

{
    int i, done;
    /* Deactivate all boundary events posted in main program */
    i = 0;
    while (boundary_events [i] != NULL)
    
        VUerDeactivate (boundary_events [i++]);

    /* Pop up new view */
    .
    /* Post requests for new view, using any of the posting routines */
    er = VUerBoundaryEventPost (Client . . .);
    . . .
    /* Poll and handle events */
    done = 0;
    while !done
    
        { 
            location = VOlloWinEventPoll (V_WAIT);
            VUerHandleLocEvent (location);
        }

    /* Clear requests for all clients in this service routine. */
    VUerClearAll (Client);
    . . .
    /* Remove the view from the screen */
    . . .

    /* Reactivate all boundary events for main program */
    i = 0;
    while (boundary_events [i] != NULL)
    
        VUerActivate (boundary_events [i++]);

    return INPUT_USED;
}
When an input object is drawn, its interaction handler posts inside and simple edge event requests on appropriate areas of its template. For example, in the slider template shown below, the areas enclosed in dashed rectangles have event requests posted:

The requests are posted internally when the input object is drawn, and are cleared when it is erased. If the input object is partially covered by another drawport but not erased, requests on the uncovered parts remain active.

If other event request regions overlap those defined for an input object, you must specify which event requests take precedence, since only one inside or simple edge request can be processed for the event. If you draw the input object first and then post the request on the overlapping region, the overlapping region’s request takes precedence. If you draw the input object last, the input object’s internally posted requests take precedence.

Interaction handlers post several requests for different types of events, usually using \texttt{VUerRectEdgeDpPost}. (If the input object’s template includes the \texttt{PostType:OBJECT} flag, the interaction handler uses \texttt{VUerObjectEdgeDpPost} instead.) For example, in order to respond to the cursor position in the slider area, the interaction handler posts the following event request:

\begin{verbatim}
VUerRectEdgeDpPost (Input, NULL, NULL, 0, &ClipEchoVP, V_INSIDE, NULL, CURSOR_IN_SLIDER, gdp, &ClipEchoVP);
\end{verbatim}
This call supplies the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td><em>Input</em>: the input object whose interaction handler is posting the request.</td>
</tr>
<tr>
<td><strong>FcnPtr</strong></td>
<td><em>NULL</em>: the NULL function pointer indicates that the event handler calls internal service routines.</td>
</tr>
<tr>
<td><strong>Args</strong></td>
<td><em>NULL</em>: no arguments are passed.</td>
</tr>
<tr>
<td><strong>ArgSize</strong></td>
<td>0: the size is zero because no arguments are passed.</td>
</tr>
<tr>
<td><strong>BndingRect</strong></td>
<td><em>&amp;ClipEchoVP</em>: the area for which the event request is being posted, which is clipped in this case.</td>
</tr>
<tr>
<td><strong>InOut</strong></td>
<td><em>V_INSIDE</em>: a flag indicating that the request is for events that intersect the area.</td>
</tr>
<tr>
<td><strong>Keystr</strong></td>
<td><em>NULL</em>: causes this request to be interpreted as a position event request, with no key press required. If the <em>Poll</em>:No flag were set in the template, a string equivalent to the select keys would appear instead.</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td><em>CURSOR_IN_SLIDER</em>: the label passed to the internal service routine. The service routine uses the label to determine what action to take.</td>
</tr>
<tr>
<td><strong>drawport</strong></td>
<td><em>gdp</em>: the drawport containing the input object.</td>
</tr>
<tr>
<td><strong>cliprect</strong></td>
<td><em>&amp;ClipEchoVp</em>: the clipping rectangle containing the area for which the event request is being posted.</td>
</tr>
</tbody>
</table>

When you move your cursor in the slider, the event handler matches that event to the slider’s request for motion events inside its slider area. This triggers a service routine within the interaction handler. In this case, the internal service routine determines a new value corresponding to the cursor position, sets the new value in the slider’s variable descriptor, redraws the slider to reflect the new value, and returns *INPUT_ACCEPT*.

The internal service routines in interaction handlers return one of the following flags, depending on the event:

- **INPUT_UNUSED**: No event request was satisfied.
- **INPUT_DONE**: The input sequence was completed.
- **INPUT_ACCEPT**: The input was used by an input handler.
- **INPUT_CANCEL**: The input activity was canceled.
For example, when you select “Cancel,” the service routine returns
INPUTCANCEL. When you press a designated SELECT_KEY with the
cursor in the Done area, the service routine returns INPUT_DONE. For more
information about actions and their service results, see the VN section of the

Posting Service Result Requests

You can post a service result request to call an additional routine if a service
routine returns a specific result. This practice is called monitoring, and the
client object whose service results are monitored is called a monitored client.
This practice is most commonly used on input objects or the results of your
own service routines.

To monitor the activity of input objects, you monitor for one of the four
possible service results listed above. For example, you can call TakeAction on
receiving a result of INPUT_ACCEPT, and DisplayHelpMessage on
INPUT_UNUSED.

The internal event requests of input objects, and your own event requests, are
serviced in the primary loop of the event handler. Service result requests are
serviced in the secondary loop. Echo functions, an alternative to service result
requests for input objects, are also serviced in the secondary loop. Echo
functions are discussed later in this chapter.

To post a service result request, call VUerServiceResultPost:

EVENT_REQUEST VUerServiceResultPost (OBJECT Client,
int (*FcnPtr),
ADDRESS Args,
int ArgSize,
OBJECT MonitoredClient,
int ResultMask, Label);

Client, FcnPtr, Args, ArgSize, and Label have the same role in service result
requests as in event requests. The other parameters are:

MonitoredClient The Client of the event request whose result is being
monitored. To monitor an input object, pass the input object
as the MonitoredClient. Client and MonitoredClient should
not be identical; if they are, an infinite loop can result.
The internal service routines of input objects have four possible service results, listed in the previous section. If you monitor your own service routines, the result can be any integer value. However, because a bitwise operation is used to check ResultMask against the service result, monitored service results should ordinarily be powers of 2 to prevent unintended function calls. You cannot monitor for a service result of 0.

Examples of Service Result Requests

The following example posts two service result requests. For a complete example, see the example program *popup.c*.

```c
VUerServiceResultPost ((OBJECT) CLIENT_LINE_MENU, HandleInput, (ADDRESS) &info, sizeof (info), info.input[0], INPUT_DONE, INPUT_DONE);

VUerServiceResultPost ((OBJECT) CLIENT_LINE_MENU, HandleInput, (ADDRESS) &info, sizeof (info), info.input[0], INPUT_CANCEL, INPUT_CANCEL);
```

Both postings request calls to the service routine HandleInput in response to events on the monitored client, which is the input object info.input[0]. The service result flags INPUT_DONE and INPUT_CANCEL, which trigger the respective requests, are borrowed for the Label parameter. Label is passed to HandleInput, which uses it in a switch statement to determine what action to take.

The following example shows how to use the same posting for multiple events. It posts a request to call ActOnInput when either a Done or Accept result is returned on the monitored input object Slider. In this case the same call is made for both types of event.

```c
er = VUerServiceResultPost ((OBJECT) Client1, ActOnInput, (ADDRESS) &info, sizeof (info), Slider, INPUT_DONE | INPUT_ACCEPT, SOME_LABEL);
```
When a service result matches a request, the service routine called can also return a service result. You can post a service result request for this result that is serviced in another cycle of the secondary event handling loop. The Client of each request becomes the MonitoredClient of the following request. In this way, you can build multiple layers of requests. The following code fragment shows how you might post requests for a chain of callbacks:

```c
/* Post request for picks inside the object HotSpot */
er1 = VUerObjectEdgePost (HotSpot, Func1,
                         (ADDRESS) &info, sizeof (INFO), HotSpot,
                         Xform, V_INSIDE, PickSyms, Label);

/* Monitor HotSpot; call Func2 if function returned MENU_1 */
er2 = VUerServiceResultPost ((OBJECT) Client1, Func2,
                         (ADDRESS) &info, sizeof (info),
                         HotSpot, MENU_1, Label);

/* Monitor Client1; call Func3 if function returned MENU_2 */
er3 = VUerServiceResultPost ((OBJECT) Client2, Func3,
                         (ADDRESS) &info, sizeof (info),
                         Client1, MENU_2, Label);
```

A pick inside the object HotSpot might result in one, two, or three function calls, depending on the service results of each function. The result of the last function called is returned to VUerHandleLocEvent.

Example of a Service Result Routine

The following example of a service result routine is taken from the example program inputs.c. The input object being monitored is a menu, and info->data_ptr, a field in the argument structure, is preset to the address of the menu’s data buffer. By using the contents of this buffer in a switch statement, the service result routine can take an appropriate action based on the user’s menu selection.
Input objects can use echo functions as an alternative to service result requests. Either type of function is called in the secondary loop of the event handler, after the input object’s internal routine is finished. There are three major differences between the two types of function:

A service request is satisfied only when the input object takes input. An echo function is called when the input object takes input, and also whenever it is drawn, erased, updated, or redrawn.

A service request is satisfied only when the input object’s internal service routine returns a specified result. An echo function is called regardless of the service result.
A service request is lost when an input object is cloned, or when the input object is saved and restored. An echo function becomes an intrinsic part of the input object, and is cloned, saved, and restored with the input object.

To use an echo function with an input object, attach the echo function to the input technique object using the following call:

```c
VOitPutEchoFunction (It, EchoFcn, Args, ArgSize);
```

*It* is the input technique object, and *EchoFcn* is your echo function. *Args* and *ArgSize* represent a user-defined argument block similar to that used in posting event requests.

The general form for the echo function is shown below. All parameters except *Args* are passed to the echo function by the interaction handler.

```c
void
 echo_fcn (  
   OBJECT Input,  
   int Origin,  
   int State,  
   double *Value,  
   VARDESC Vdp,  
   RECTANGLE *EchoVP,  
   ADDRESS args)
```

Some types of interaction handlers require a slightly different form, depending on the type of data provided by the input object. For details, see the Interaction Handlers chapter of the DV-Tools Reference Manual.

An echo function has certain advantages over service routines because of the parameter information passed directly to it by the input object. This information includes:

- **Origin**: The action, such as drawing or taking user input, that is taking place in the input object.
- **Vdp**: Variable descriptor attached to the input object.
- **Value**: Value in the buffer of the input object’s variable descriptor.
- **EchoVP**: The echo area position in world coordinates.
- **State**: The service result flag, equivalent to *ResultMask* in *VUserServiceResultPost*. 

---

**Echo Functions**

---

**Event Handling**
**Origin** indicates which internal action is calling the echo function. Valid origins are:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL_DRAW</td>
<td>Occurs when you call <code>TdpDrawObject</code> or <code>TdpDraw</code> to draw the input object or when you call <code>TscRedraw</code> or <code>TdpRedraw</code> to redraw after resizing.</td>
</tr>
<tr>
<td>TAKE_INPUT</td>
<td>Occurs when an event in the input object is handled by <code>VUerHandleLocEvent</code>.</td>
</tr>
<tr>
<td>UPDATE_DRAW</td>
<td>Occurs when <code>TdpDrawNext</code> or <code>TdpDrawNextObject</code> is called to update the input object explicitly. Since the interaction handler echoes new input implicitly, you only need to update explicitly if you have manipulated the data or the input object programmatically.</td>
</tr>
<tr>
<td>CONTEXT_REDRAW</td>
<td>Occurs when <code>TscRedraw</code> or <code>TdpRedraw</code> is called to redraw the input object.</td>
</tr>
<tr>
<td>ERASE</td>
<td>Occurs when you call <code>TdpEraseObject</code> or <code>TdpErase</code> to erase the input object.</td>
</tr>
<tr>
<td>SETUP_FOR_DRAW</td>
<td>Occurs when the internal drawing information is set up at initial draw and after resizing.</td>
</tr>
<tr>
<td>CONTEXT_DRAW</td>
<td>Occurs when the static portion of the input object is drawn or redrawn.</td>
</tr>
<tr>
<td>CLEANUP_DATA</td>
<td>Occurs when data is being cleared prior to destroying the input object.</td>
</tr>
<tr>
<td>DATA_RESET</td>
<td>Occurs when data is set initially or reset.</td>
</tr>
</tbody>
</table>

The following additional origins represent sub-actions in drawing, taking input, updating, redrawing, and erasing. They are also valid points when the echo function is called, but are used only in special circumstances.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP_FOR_DRAW</td>
<td>Occurs when the internal drawing information is set up at initial draw and after resizing.</td>
</tr>
<tr>
<td>CONTEXT_DRAW</td>
<td>Occurs when the static portion of the input object is drawn or redrawn.</td>
</tr>
<tr>
<td>CLEANUP_DATA</td>
<td>Occurs when data is being cleared prior to destroying the input object.</td>
</tr>
<tr>
<td>DATA_RESET</td>
<td>Occurs when data is set initially or reset.</td>
</tr>
</tbody>
</table>

You can write an echo function with `switch` statements based on the values of **Origin** and **State**. In this way, a single echo function can handle more cases than a service routine. Since the echo function is called in many different circumstances, you must set it up to handle the different possible cases. These cases include the different values for **Origin** and **State**, and different numerical values in the input object’s variable descriptor.
When to Use Echo Functions

Because an echo function receives information directly from its input object, particularly the origin, variable descriptor, and value, and because it is an integral part of the input object, echo functions are useful in the following circumstances:

- When you want to act on the input object’s variable descriptor or its value, which are passed to the echo function by the input object. If you use a service routine, you are responsible for passing this information in the argument block.
- When you clone an input object, or save and restore an input object as part of a view. Because echo functions are an intrinsic part of their input objects, they are cloned, saved, and restored with their input objects. Service result requests are not part of the input object, so if you clone an input object, or save and restore it, you must post its requests again. Note that input objects embedded in multiplexors and combiners are cloned. Using echo functions with embedded input objects means you do not have to post service result requests more than once.
- When you want an action to take place at a time other than input. You can supply an echo function that makes different actions occur depending on whether the input object is being initially drawn, updated, or whether input is accepted, done, or canceled.
- When you want to add your own functions to the input object’s internal functions for drawing, accepting input, updating, echoing, and erasing.

Handling Window Expose and Resize

Your code should handle window expose and resize events so that your application will behave properly when these events occur. The code fragments below show a standard way of doing this.

```c
VUerWinEventPost (some_client, ExposeResizeHandler, (ADDRESS) NULL, 0, EXPOSE_LABEL, (ULONG) VUER_EXPOSE_EVENT);
VUerWinEventPost (some_client, ExposeResizeHandler, (ADDRESS) NULL, 0, RESIZE_LABEL, (ULONG) VUER_RESIZE_EVENT);
```

The client you specify can be anything, but should not be the same as any monitored client you have in your application. The label parameter is used to flag the type of event in the service routine.
The following code fragment shows a service routine for handling expose and resize events. On expose events, you must redraw the screen. On resize events, you must reset and then redraw. However, `TscReset` generates an expose event automatically, which results in the screen being redrawn. Therefore, you do not have to redraw explicitly on resize events.

```c
int ExposeResizeHandler (OBJECT scr_client, EVENT_REQUEST request, int label, OBJECT loc, ADDRESS args);
{
  switch (label)
  {
    case EXPOSE_LABEL:
      TscRedraw (scr_client, VOloRegion (loc));
      break;
    case RESIZE_LABEL:
      TscReset (scr_client);
      break;
  }
  return INPUT_USED;
}
```

Events can be handled without posting requests, by extracting information directly from the location object and taking the appropriate action. This is similar to extracting information from the location object in a service routine, a technique discussed previously in the Service Routines section. The only difference is that events are processed directly in the control loop or your own subroutine without going through the event handler.

The following example shows a control loop that handles certain types of events explicitly. Picking a named object with mouse button 1 switches to a corresponding view. Mouse button 3, “Q,” “q,” or a window quit event sets the `done` flag. A window resize or expose results in a screen reset or redraw.
The example uses `VoloWinEventPoll` to poll for location objects, then uses
`VoloScreen`, `VoloType`, `VoloKeySym`, `VoloRegion`, `VoloButton`, and
`TloGetSelectedObjectName` to get information from the location object.

```c
/* Poll. Skip event handling code if no event occurred. */
location = VoloWinEventPoll (V_NO_WAIT);
if (location)
{
    /* Change to the screen where the event occurred. */
    current_screen = VoloScreen (location);
    TscSetCurrentScreen (current_screen);

    /* Switch on type of event returned. */
    switch (VoloType (location))
    {
        /* If it's an expose event, redraw. */
        case V_EXPOSE:
            TscRedraw (current_screen, VoloRegion (location));
            break;

        /* If it's a resize event, reset the screen. An expose event automatically
        follows and causes a redraw. */
        case V_RESIZE:
            TscReset (current_screen);
            break;

        /* If it's a key press, q or Q, quit. */
        case V_KEYPRESS:
            keysym = VoloKeySym (location);
            if (keysym == 'q' || keysym == 'Q')
                done = YES;
            break;

        /* If it's a button press, get which one. */
        case V_BUTTONPRESS:
            switch (VoloButton (location))
            {
```
Event Handling

Handling Device-Specific Window Events

/* Button 1: See if we switch views based on object’s name. */
case 1:
    obj_name = TloGetSelectedObjectName (location);
    if (obj_name)
    {
        new_drawport = CreateNewDrawport (obj_name);
        if (new_drawport)
        {
            current_drawport = new_drawport;
            TdpDraw (current_drawport);
        }
    }
    break;
break;
break;
break;

Note that input objects are not used in this example. If you have input objects in your interface, you must call the event handler to update them.

DV-Tools lets you handle device-specific events such as X window events. When a device-specific event occurs, DataViews converts the event into a location object, using either the V_NON_DV_WINDOW_EVENT or V_NON_STANDARD_EVENT flag as the event type. It also copies the system’s event data structure into the eventdata field of the WINEVENT structure.

The V_NON_DV_WINDOW_EVENT event type is used for events from windows that have not been explicitly opened as screens. These windows may be other windows in your application or may be windows belonging to widgets created in DataViews. Location objects representing events from widgets in DataViews should be sent to the event handler.

The V_NON_STANDARD_EVENT event type is used for masked non-standard events. For more information on creating event masks for non-standard events, see VOscWinEventMask in the DV-Tools Reference Manual.
You can handle the event explicitly or by using the event handler. To handle the event explicitly, extract the system’s event from the *eventdata* field and handle it using C or system calls. The following fragment shows some calls that handle device-specific events:

```c
/* if it's a nonstandard event */
case V_NON_STANDARD_EVENT:
    /* Get the winevent structure and handle the event*/
    nst_we = VOloWinEventGet (location);
    HandleNstEvent (nst_we);
    break;
```

To handle the event using the event handler, post an event request using *VUerCatchAllEventPost*:

```c
EVENT_REQUEST
VUerCatchAllEventPost (  
    OBJECT Client,  
    VUERFCNFUNPTR user_fcn,  
    ADDRESS Args,  
    int ArgSize,  
    int Label)

int
user_fcn (  
    OBJECT Client,  
    EVENT_REQUEST Request,  
    int Label,  
    OBJECT Loc,  
    ADDRESS Args)
```

All nonstandard events, or events that do not match any other event request, match this event request and trigger its service routine. Therefore, the service routine must sort the events into different kinds and handle the interesting kinds. The service function should call *VOloWinEventGet* as a first step in the sorting process.
You may choose to set up your own location object. For example, you can simulate a series of external events such as playing back pre-recorded key presses. You can also take information from an event and pass it to another location object in which you control other information, then pass that location object to the event handler.

To set up a location object, you can either modify the location object returned by the polling routine or create and manage a location object yourself. Note that if you use the location object returned by the polling routine, your next call to the polling routine will overwrite it. If you use your own location object, you can keep or destroy it by manipulating its reference counts.

To create a location object, use VOloCreate. To set the location object’s internal values, use TloSetup or TloWinEventSetup. These routines set up location objects analogous to those returned by VOloWinEventPoll.

TloSetup requires the following parameters:
- a location object
- a screen object
- a drawport
- a position in screen coordinates
- a key or button

TloWinEventSetup requires the following parameters:
- a location object
- a screen object
- a drawport
- a WINEVENT structure

The following information must be supplied in the WINEVENT structure:
- an event type
- a position
- a button or first character

A location object set up using TloWinEventSetup can only be a key press/release or button press/release event. It cannot be a window resize event or other window related event.
The following code fragment uses TloSetup to modify a location object. It changes the coordinates of a location object to a position inside a text entry input object. This lets the input object process the keystroke as if the cursor were positioned inside the text entry box, regardless of its actual position.

```c
void
Text_Focus (  
    OBJECT Location);  
{
    DV_POINT scs;  
    OBJECT transform;  
    RECTANGLE worldVP, srect;  

    VOobBox (TextEntryIn, &worldVP, &srect);  
    scs.x = (worldVP.ll.x + worldVP.ur.x) / 2;  
    scs.y = (worldVP.ll.y + worldVP.ur.y) / 2;  
    transform = TdpGetXform (drawport, DR_TO_SCREEN);  
    VOxfPoint (transform, &scs);  

    /* To keep keystroke information constant, pass the result of VOloKey back  
    to TloSetup. */  
    TloSetup (Location, VOloKey (Location), &scs, NULL, NULL);  
}
```

After modifying the location object, pass it to VUserHandleLocEvent.

---

The control loop has three major functions:

- To act on user input.
- To act on application data.
- To update the application and the display.

You determine the order in which these functions are handled, so long as they are all handled in the loop. This section shows you how to build a control loop using examples from telecomdemo, a simulation of a communications network.
Handling User Input in the Control Loop

So far this chapter has discussed acting on input from users, including gathering input, handling input explicitly, and working with the event handler to act on input. The following fragment shows how this portion of the control loop is implemented in the `main.c` module:

```c
while (*Control.quit_flag == FALSE) {
  OBJECT LocationScreen;
  while (location = VOloWinEventPoll (V_NO_WAIT)) {
    LocationScreen = VOloScreen (location);
    if (LocationScreen == ControlScreen) {
      TscSetCurrentScreen (ControlScreen);
      HandleControlInput (location);
    } else if (LocationScreen == StatusScreen) {
      TscSetCurrentScreen (StatusScreen);
      HandleStatusInput (location);
    } else if (LocationScreen == LogScreen) {
      TscSetCurrentScreen (LogScreen);
      HandleLogInput (location);
    }
  }
}
```

In `telecomdemo`, three screens are polled for events. Whenever an event occurs in a screen, it is handled by a subroutine that includes both explicit event handling and event request handling. In handling the event, the interface may be changed. For example, if the user requests a detailed map of local network, a drawport is drawn displaying a view of the network. The application can also be changed. For example, if the user requests a restart, the simulation starts over at the beginning.

Handling Application Data in the Control Loop

In your control loop, you must also handle the data from the application and use it to update the dynamics being displayed. You can also check critical information in the application and act on it, giving the application more direct control over the interface and the control flow of the program.
To implement this control, set up variables in the program to hold critical information about the state of the application. In the control loop, this state information can be updated when other data is updated using \texttt{TviReadData} or \texttt{TdlReadData}, or it can be updated directly by the application. The state information can then be checked and actions called.

The following fragment from the \texttt{main.c} module shows the portion of the control loop that updates the data and checks the state information for the next step:

\begin{verbatim}
if (TimeForNextStep())
  if (*Control.pause_flag == 0)
    {
      UpdateStatusScreen(); /* Calls \texttt{TdpDrawNext} to update the status view. */
      UpdateControlScreen(); /* Updates the iteration count in the control view. */
      NextSimulationStep(); /* Checks state information, acts on it, and updates the status of the components. */
    }
/* End of main while loop. */
\end{verbatim}

In \texttt{telecomdemo}, the data comes from a simulation module. The data is stuffed into data source variables or variable descriptors buffers directly instead of being read using \texttt{TviReadData} or \texttt{TdlReadData}. All the data updating, checking on critical state information, and acting on it occurs by calling \texttt{NextSimulationStep}. Updating the interface to display the most recent data is handled by \texttt{UpdateControlScreen} and \texttt{UpdateStatusScreen}.

These two fragments from \texttt{telecomdemo} show the entire control loop from \texttt{main.c}. The function calls to other modules handle the details of each part of the control loop, such as handling the location objects, updating screens, and reading the next set of simulated data.
Summary

This chapter has presented these basic techniques:

- How to poll for different kinds of events.
- How to use the event handler to call service routines when certain kinds of events occur.
- How to post service result requests to monitor events in input objects or to monitor the results of your own service routines.
- How to handle events on a case-by-case basis.
- How to define echo functions and attach them to input objects.

Using these techniques, you can:

- Set up the meaningful events for the application. Handle them when they occur and ignore any others.
- Change the cursor to show whether or not input is required.
- Let users control aspects of the application.
- Let users control aspects of the interface, such as what data is displayed and what data to display next.
# Chapter 9

## Coding Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>9-1</td>
</tr>
<tr>
<td>Include Files</td>
<td>9-1</td>
</tr>
<tr>
<td>Function Prototypes and Declarations</td>
<td>9-1</td>
</tr>
<tr>
<td>Compiling and Linking under UNIX</td>
<td>9-2</td>
</tr>
<tr>
<td>Required DataViews Libraries</td>
<td>9-3</td>
</tr>
<tr>
<td>Library Order for Tproto</td>
<td>9-3</td>
</tr>
<tr>
<td>When to Use DVlink</td>
<td>9-4</td>
</tr>
<tr>
<td>Compiling and Linking under OpenVMS</td>
<td>9-4</td>
</tr>
<tr>
<td>Using the Options File</td>
<td>9-5</td>
</tr>
<tr>
<td>Linking without DVLINK</td>
<td>9-5</td>
</tr>
<tr>
<td>Linking with the DataViews Shareable Image for OpenVMS</td>
<td>9-6</td>
</tr>
<tr>
<td>The Options File for the Shareable Image</td>
<td>9-6</td>
</tr>
<tr>
<td>Linking Errors Encountered with the Shareable Image</td>
<td>9-6</td>
</tr>
<tr>
<td>Making a New Version of the Shareable Image</td>
<td>9-7</td>
</tr>
<tr>
<td>Required DataViews Object Files</td>
<td>9-7</td>
</tr>
<tr>
<td>Coding Tips</td>
<td>9-8</td>
</tr>
<tr>
<td>Avoiding Hard Coding</td>
<td>9-8</td>
</tr>
<tr>
<td>Modifying Private Data Structures</td>
<td>9-8</td>
</tr>
<tr>
<td>Using Traversal Routines</td>
<td>9-8</td>
</tr>
<tr>
<td>Using Enums</td>
<td>9-8</td>
</tr>
<tr>
<td>Using Object Utility Routines</td>
<td>9-8</td>
</tr>
<tr>
<td>Debugging Tips</td>
<td>9-9</td>
</tr>
<tr>
<td>Common Errors</td>
<td>9-9</td>
</tr>
<tr>
<td>Debugging Routines</td>
<td>9-9</td>
</tr>
<tr>
<td>Linking with the Optimized Library</td>
<td>9-11</td>
</tr>
<tr>
<td>Under UNIX</td>
<td>9-12</td>
</tr>
<tr>
<td>Under OpenVMS</td>
<td>9-12</td>
</tr>
</tbody>
</table>
This chapter discusses the following topics:

- **Include Files**
- **Compiling and Linking**
- **Coding and Debugging Tips**
- **Linking with the Optimized Library**

### Overview

Once you understand the relationship of the data structures and routines, you can begin programming. This chapter provides instructions for compiling and linking as well as tips on coding and debugging.

### Include Files

Required header files are listed in the *DV-Tools Reference Manual* in the introduction to each routine layer. If additional header files are required for a particular module, they are listed in the introduction to the module. Routine descriptions may also list additional headers that are required. All header files are listed in the *Include Files* chapter in the *DV-Tools Reference Manual*.

Header files should be included in order from the most general to the most specific, proceeding from the highest layer function declarations to the lowest. For example, a program that uses only $T$ layer functions requires the following header files:

```
std.h
dvstd.h
dvtools.h
dvGR.h
Tfundecl.h
```

### Function Prototypes and Declarations

The include files containing function declarations and prototypes can be used with standard C, ANSI-C, and C++ compilers. A macro automatically creates either function declarations or prototypes as required by the compiler.
All data types are standardized to work equivalently in standard C, ANSI-C, and C++, so you can follow the data types as listed in the synopses in the DV-Tools Reference Manual and Quick Reference Manual without worrying about type mismatches due to type promotion in your compiler.

### Compiling and Linking under UNIX

During the development phase, you can use the following procedure to compile and link your program.

**Set Environment**
Prepare your environment by setting up your configuration file and using the `DVinit` script from the `dataviews` directory.

**Compile and Link**
Compile and link your executable and object files. You must include all required object files, all of the DataViews libraries, and any flags and libraries needed by your system. The object files, libraries, and flags must be in the following order:

```
cc  <cpp flags>  <c flags>  <object files>  <ld flags>
    <DataViews libraries>  <graphics libraries>
    <system libraries>  [ <libDVs> ]  <math library>
```

If you are using a compiler other than `cc`, replace `cc` with the command for your compiler. As a starting point for writing your linking command, run `DVlink` with the `-preview` option, as shown on the following line:

```
DVlink -preview myfile.c
```

It echoes the flags, object files, and the order for the libraries that are required for your system. The required DataViews object files are listed after the OpenVMS linking instructions. The required DataViews libraries are discussed in the next section. Note that the final library to link is the standard math library, which may be linked automatically depending on your compiler.
**Required DataViews Libraries**

When linking your DataViews application, you must also include the DataViews libraries in the following order:

- `libDVvw`: View Widget (include if using the View Widget)
- `libDVfmt`: Display formatters and interaction handlers
- `libDVmenu`: DataViews menu code
- `libDVmtv`: M, T, and, VO routines
- `libDVtiff`: tiff routines
- `libDVutil`: Utility routines
- `libDVgr`: Graphics device drivers

If you are linking on a Sun, you must include the following DataViews library after all of your system-specific libraries:

- `libDVs`: Definitions to satisfy unused external references for use with shareable versions of the libraries for SunOS and Solaris. This library should always be linked after all graphics libraries such as libX11 and libsunwindow.

An optimized library without debug information is also available. See Linking with the Optimized Library later in this chapter.

**Library Order for Tproto**

If your application uses Tproto routines and you are not using the `DVlink` script, you must change the order of your link libraries. The modified order must be:

- `libDVvw`: include if using the View Widget
- `libDVfmt`
- `libDVmenu`
- `libDVmtv`
- `libDVmenu`
- `libDVtiff`
- `libDVutil`
- `libDVgr`
  ...
- `libDVs`
When to Use DVlink

Under UNIX, you can use the DVlink script to link DataViews demos, examples, and simple applications. DVlink is not suitable for linking complex applications requiring special libraries since DVlink may not be able to determine the correct order for linking in these circumstances. The following command line options let you see what DVlink does:

- **-verbose**  echoes commands before DVlink executes them
- **-preview**  echoes commands without DVlink executing them

When used, the -verbose or -preview option must be the first option on the command line after DVlink. If you use any additional options, such as -g, you should place them later on the command line. DVlink uses only these two options, and passes all other options to the compiler. The order on the command line is important because DVlink does not look ahead, but passes the first unrecognized option and all subsequent options to the compiler.

Compiling and Linking under OpenVMS

During the development phase, you can use the following procedure to compile and link your program.

**Set Environment**

Prepare your environment by setting up your configuration file and using the @DVINIT command file from the dataviews directory. If you redefine C$INCLUDE after running DVINIT, be sure to add the logical DV$INCLUDE in the list of search directories.

**Compile**

Compile your program by entering

```
CC your_filename
```

**Link**

Link your program by entering

```
DVLINK <obj file> [, <obj file>,....<obj file>] [,<libname>/LIB,...<libname>/LIB]
```

The square brackets indicate optional elements in the command.

DVLINK is a symbol defined by DVINIT.COM to run the file DVSBIN:DVLINK.COM. DVLINK automatically links with all DataViews libraries and all shareable images required by your system. You need to supply only the names of your application’s object files and any additional object libraries. Additional libraries should have the qualifier /LIB after each
library filename. The default name of the executable file is the name of the first object file, with the extension .EXE. To specify a particular name for the executable, use the /EXE qualifier after the first object file.

```
DVLINK <obj file> [/EXE = <name>,<obj file>.....<obj file>] [.,<libname>/LIB,...,<libname>/LIB]
```

The following sections discuss these other linking possibilities:

Using the Options File

A special options file lets you specify the shareable images that should be linked with when using DVLINK. The options file is referred to by the logical name DV$LINK_OPTIONS. To find its actual name on your system, type:

```
SHOW LOGICAL DV$LINK_OPTIONS
```

If you need to link with additional shareable images, make a private copy of the options file. For each additional shareable image, add the following line to your copy of the file:

```
SYS$SHARE:<shareable image file>/SHARE
```

You must then redefine DV$LINK_OPTIONS to make DVLINK.COM use your customized options file. Use the following command before your DVLINK command:

```
DEFINE DV$LINK_OPTIONS <options file>
```

You can also put references to object libraries into the options file.

Linking without DVLINK

If you are not using DVLINK or the DataViews shareable image, you must link with LIBDVT.OLB (the DataViews library) and the DataViews object files, which are listed later in this chapter.

If you are not using DVLINK, but are linking with the DataViews shareable image, you must include USYSDATA.OBJ, which is not included in the shareable image, and link with LIBDVTSHR.EXE. Both are located in DVSLIB.
Linking with the DataViews Shareable Image for OpenVMS

The DataViews release for OpenVMS includes a shareable image, `LIBDVTSHR.EXE`, which you may link with to save disk space. To link the DataViews shareable image, use `DV$:[ETC.SHARE]LINKSHARE.COM`. The `DV$LIB:LINK_LIBDVTSHR_* .OPT` files contain the necessary PSECT and UNIVERSAL symbol declarations. For more information on shareable images, see OpenVMS Guide to Creating Modular Procedures in your OpenVMS documentation set.

You can link with the shareable image as the default for your system or on a case-by-case basis for particular applications:

- To link with the shareable image by default, specify it during the `VMSINSTAL` procedure.
- To change the default linking to use the shareable image without re-installing, edit `DV$ETC:SITE_SPECIFIC.DAT` to change the symbol `DV__SHARE` to `YES`, then re-execute `DV$HOME:DVINIT.COM`.
- To link with the shareable image for a particular application, set `DVSHARE` to `YES` and link using `DVLINK`.

The Options File for the Shareable Image

The options file specifically for linking the DataViews shareable image is `DV$LIB:LINK_LIBDVTSHR_BASE.OPT`. This file contains the windowing-system independent options, including PSECT attributes and UNIVERSAL symbol declarations.

Linking Errors Encountered with the Shareable Image

When you link your application with the DataViews shareable image, you may get the following types of errors that did not occur when you linked with `LIBDVT.OLB`:

```
%LINK-W-NUDFSYMS
%LINK-I-UDFSYM
%LINK-W-USEUNDEF
```

If this happens, add the symbols mentioned in the error messages to the list of UNIVERSAL symbols in `DV$LIB:LINK_LIBDVTSHR_BASE.OPT`, as shown in the following line:

```
UNIVERSAL=<new symbol>
```

You should also inform DataViews Corporation Technical Support so those declarations can be included in future releases. After adding the declarations, you must make a new version of the shareable image following the directions in the next section.
If you get a \texttt{LINK-W-MULDEF} error message, you have redefined a DV-Tools routine. You should contact Technical Support for assistance in resolving the problem.

You should make a new shareable image when you change \texttt{DV$LIB:LINK_LIBDVTSHR\_BASE.OPT}, add a user-defined graph or function descriptor set, or change the device drivers. For more information on adding user-defined graphs or function descriptor sets, see \textit{Technical Note #26}. For more information on changing device drivers, see the \textit{DataViews Installation and System Administration Manual}.

To create a new version of \texttt{LIBDVTSHR.EXE}, type the command:

\begin{verbatim}
@DV$:[ETC.SHARE]LINKSHARE
\end{verbatim}

If you are not using \textit{DVlink}, you must include the object files required by DataViews. Object files are located in the \texttt{$DVHOME/lib} (UNIX) or \texttt{DV$LIB} (OpenVMS). The required object files are:

<table>
<thead>
<tr>
<th>UNIX Object</th>
<th>OpenVMS Object</th>
<th>When Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawNames.o</td>
<td>\texttt{DRAWNAMES.OBJ}</td>
<td>if relinking DV-Draw</td>
</tr>
<tr>
<td>ToolNames.o</td>
<td>\texttt{TOOLNAMES.OBJ}</td>
<td>if linking a DV-Tools application</td>
</tr>
<tr>
<td>GRconfig.o</td>
<td>\texttt{GRCONFIG.OBJ}</td>
<td>always</td>
</tr>
<tr>
<td>Usysdata.o</td>
<td>\texttt{USYSDATA.OBJ}</td>
<td>always</td>
</tr>
<tr>
<td>VNwglobals.o</td>
<td>\texttt{VNWGLOCALS.OBJ}</td>
<td>if using any widget-based input objects</td>
</tr>
<tr>
<td>VNXmotif.o</td>
<td>\texttt{VNXMOTIF.OBJ}</td>
<td>if using the Motif-based input objects</td>
</tr>
<tr>
<td>XWmotif.o</td>
<td>\texttt{XWMOTIF.OBJ}</td>
<td></td>
</tr>
<tr>
<td>VNXolit.o</td>
<td>not applicable</td>
<td>if using the OPEN LOOK-based input objects</td>
</tr>
<tr>
<td>XWolit.o</td>
<td>not applicable</td>
<td></td>
</tr>
</tbody>
</table>
**Coding Tips**

The demo and example programs serve as coding samples. You can examine them to learn how DV-Tools programs are structured, how data structures are used, how views are integrated with the code, and what routines are used in various situations. You can adapt views and modules from these programs to suit your application.

**Avoiding Hard Coding**

Using the configuration file variables `DVPATH`, `DVDEVICE`, `DVVIEW`, and `DVCOLOORTABLE`, you can set up your environment without hard coding. Using `DVPATH` to control the precedence of search directories eliminates the need to hard code the pathnames.

Creative use of DV-Draw can also help you avoid hard coding. Using views to pass information to your DV-Tools program lets you change the interface by editing a view instead of by editing code. For example, you can use named rectangles in a view to control the position of drawports instead of supplying hard-coded positions. To move or resize the drawports, edit the rectangles in DV-Draw and rerun the program without recompiling. For more information about this technique, including a code fragment, see the Display chapter.

**Modifying Private Data Structures**

Some DV-Tools routines return pointers to internal fields of private types. One such routine is `VOdrGetName`, which returns a pointer to an internal character string. When you use this kind of routine, do not modify the part directly, but make a copy and modify the copy.

**Using Traversal Routines**

DV-Tools includes many routines for traversing the lower-level components within a higher-level structure. Most of these routines contain the phrase “ForEach” or “Traverse” in their name. These routines require a programmer-supplied function that takes as its arguments the object being acted on and the argument block.

Some traversal routines include `TdpForEachDrawport`, `VOobTraverse`, and `VOciTraverse`. For information about specific traversal routines, see their synopses in the DV-Tools Reference Manual.

**Using Enums**

You can declare any enum as type `int` rather than using the full type name.

**Using Object Utility Routines**

DV-Tools includes utilities in the `VOutil` module to manipulate objects and determine information about them. For example, `VOuIsDynamic` determines whether or not an object has dynamics, and `VOuVpVisible` determines whether or not a viewport is visible. For information about these utility routines, see the DV-Tools Reference Manual.
**Debugging Tips**

**Common Errors**

If your DV-Tools program fails to compile, check to make sure you have used the right number and types of arguments. In particular, make sure your point data structures, point objects, rectangle data structures, and rectangle objects are declared correctly. An example of potentially confusing types is `VOptGet` which takes one point `OBJECT` and two `DV_POINT` data structure pointers as seen below:

```c
void VOptGet (OBJECT point, DV_POINT *wpt, DV_POINT *spt_offset)
```

`DVlint` is a script that checks your code against DataViews lint libraries. Using `DVlint` can reveal inconsistencies in argument number and type. Note that `DVlint` is much more stringent than a compiler.

Another common error is the incorrect usage of pointers. Make sure you have used `*` and `&` correctly. For example, `TdpResize` takes the address of a `RECTANGLE` structure but `TdpGetScreenVp` returns a pointer to a `RECTANGLE` structure as seen below:

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

**Debugging Routines**

DV-Tools includes three modules of debugging routines. These routines are available as both object files and source code. The object files are located in the `lib` directory and the source code is located in the `tooldebug` subdirectory of the `src` directory. The three modules are:

- **VOdebug** Prints information about objects, including attributes, the number of objects, the objects in a deque, and the control points.
- **VOdbgCounts** Prints the number of each type of object.
- **VOdbgObPts** Prints control points (x,y) of an object.
- **VOdbgOb** Prints the object description.
### Debugging Tips

**Coding Environment**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOdbgDqList</td>
<td>Prints the contents of a deque object.</td>
</tr>
<tr>
<td>VOdbgAttr</td>
<td>Prints the attributes of an object.</td>
</tr>
<tr>
<td>VUdebug</td>
<td>Prints information about graphs, including context control, color, and variable descriptors.</td>
</tr>
<tr>
<td>VUdbgCcf</td>
<td>Prints the contexts control flags of a dgp.</td>
</tr>
<tr>
<td>VUdbgColor</td>
<td>Prints a COLOR_SPEC structure.</td>
</tr>
<tr>
<td>VUdbgCtt</td>
<td>Prints a COLOR_THRESHOLD structure.</td>
</tr>
<tr>
<td>VUdbgDgp</td>
<td>Prints a data group pointer.</td>
</tr>
<tr>
<td>VUdbgVdp</td>
<td>Prints a variable descriptor pointer.</td>
</tr>
<tr>
<td>VUdbgalloc</td>
<td>Provides information about memory allocation. These routines are intended for use by sophisticated users and are not documented in the DV-Tools Reference Manual. See the source code for more information about these routines.</td>
</tr>
<tr>
<td>VUmalloc</td>
<td>Allocates memory, adding status information.</td>
</tr>
<tr>
<td>VUfree</td>
<td>Frees memory after performing validity checks.</td>
</tr>
<tr>
<td>VUrealloc</td>
<td>Reallocates memory after updating status info.</td>
</tr>
<tr>
<td>VUdbgCheckAlloc</td>
<td>Checks the validity of the allocated block.</td>
</tr>
<tr>
<td>VUdbgFindBad</td>
<td>Prints corrupted blocks.</td>
</tr>
<tr>
<td>VUdbgReportAllocs</td>
<td>Prints maximum and current allocations.</td>
</tr>
<tr>
<td>VUdbgMemStats</td>
<td>Prints size distribution of allocated blocks.</td>
</tr>
<tr>
<td>VUdbgMemFill</td>
<td>Determines the initialization of allocated blocks.</td>
</tr>
<tr>
<td>VUdbgMark</td>
<td>Sets a reference mark to check against later.</td>
</tr>
<tr>
<td>VUdbgReportSince</td>
<td>Prints status of all blocks allocated since the mark.</td>
</tr>
<tr>
<td>VUdbgReportLeaks</td>
<td>Prints leaks since the mark.</td>
</tr>
<tr>
<td>VUdbgMarkChanges</td>
<td>Tags all blocks in order to trace changes.</td>
</tr>
<tr>
<td>VUdbgFindChanges</td>
<td>Prints blocks that have changed since tagging.</td>
</tr>
</tbody>
</table>
You can use VOdebug and VUdebug either by calling them from your application or by invoking them from the debugger. In either case the routines must be linked with your executable. The scripts supplied for linking examples and demos use the Makefile in the etc directory. This makefile is set up to link the VOdebug and VUdebug routines with applications by default.

To use the VUdbgalloc module, you must explicitly link it with your executable. These routines redefine the DataViews default memory allocation routines VUmalloc, VUfree, and VUrealloc so that they keep additional status information on memory allocation. This is useful for finding memory leaks in your program. You can use the DVlink script to link the VUdbgalloc module with your executable:

```
DVlink source_program.c -g dataviews/lib/VUdbgalloc.o -o executable.x
```

If you are using a UNIX debugger, such as dbx, you may be able to set up aliases to call routines in the VOdebug and VUdebug modules. For example, you can add the following line to your dbx initialization file, .dbxinit.

```
alias obdescribe "call VOdbgOb(!*)"
```

The debugger also lets you access information about objects. For example, to get information about a circle object, you can enter the following line in the debugger:

```
obdescribe circle
```

If you are using dbxtool, you can define a button in the dbxtool menu by adding the following line to .dbxinit:

```
button expand obdescribe
```

After completing development, you can increase the speed of your executable by linking with the optimized library libDVvo.a (UNIX) or LIBDVTO/LIB (OpenVMS). The library contains faster, less verbose versions of the T and VO routines contained in the non-optimized library libDVmtv (UNIX) or LIBDV/LIB (OpenVMS). The non-optimized library provides validity checking and debug information that is useful during development, but slows down the finished version of the application. For other tips on optimizing DataViews performance, see the Tips for Improving DataViews Performance chapter.
Linking with the Optimized Library

Under UNIX

To use the optimized library in UNIX, modify DVlink to include a reference to libDVtvo.a before libDVmtv.a in the variable DVLOADLIBS. If you are using your own link script, you need to add a reference to libDVtvo.a before libDVmtv.a.

Under OpenVMS

In OpenVMS, if you are linking with the object libraries, set DV$SHARE to NO and put LIBDVTVO/LIB before LIBDVT/LIB in your DVLINK.COM file.

If you are using the shareable image, you can modify it to use the optimized library, then link with the modified shareable image. Use the following steps:

Step 1. Create a command file to extract the DataView routine modules from the optimized library:

LIBRARY/LIST=EXTRACT.COM LIBDVTVO
CREATE/DIRECTORY [.TEMP]
EDIT EXTRACT.COM

In the editor, delete the header lines. Modify each remaining line to be a LIBRARY/EXTRACT command. Use the following model, where module is the module written out by the LIBRARY/LIST command above. Note that each module name appears twice in the extraction command.

LIBRARY/EXTRACT=module/OUTPUT=[.TEMP]module LIBDVTVO

For example, the first three lines of EXTRACT.COM should read:

LIBRARY/EXTRACT=T/OUTPUT=[.TEMP]T LIBDVTVO
LIBRARY/EXTRACT=TDP/OUTPUT=[.TEMP]TDP LIBDVTVO
LIBRARY/EXTRACT=TDR/OUTPUT=[.TEMP]TDR LIBDVTVO

Step 2. Run the file:

@EXTRACT.COM

Step 3. Save a copy of the original LIBDVT.OLB:

COPY LIBDVT.OLB LIBDVT_SAVE.OLB

Step 4. Put the optimized modules into LIBDVT.OLB:

LIBRARY/REPLACE LIBDVT.OLB [.TEMP]*.OBJ

Step 5. Generate a new version of the shareable image:

@DV$:[ETC.SHARE]LINKSHARE
Step 6.  Restore *LIBDVT.OLB* to its original state unless you want *LIBDVT.OLB* to contain the optimized modules also:

```
RENAME LIBDVT.OLB LIBDVT_OPTIMIZED.OLB
RENAME LIBDVT_SAVE.OLB LIBDVT.OLB
```
Chapter 10
Tips for Improving DataViews Performance

Introduction ................................................................. 10-1
Configuration Variables Affecting Speed ....................... 10-2
General Optimizations .................................................. 10-2
  Set Aside a Large Chunk of Memory ......................... 10-2
  Link with the Optimized Libraries ......................... 10-3
  Use Deques to Consolidate Function Calls ............ 10-3
Optimizing Load Speed .................................................. 10-4
  Streamline the DataViews Search Path .................. 10-4
  Use Binary View Files ........................................... 10-4
Optimizing Draw Speed .................................................. 10-4
  Make the Initial Drawing of Views Faster ............ 10-4
  Make Drawing Look Smoother ............................ 10-5
  Prevent Screen Lockup ....................................... 10-5
  Redraw the Entire Screen Rather than Repairing .... 10-6
  Limit Updating to Particular Objects ................. 10-6
  Speed Up Your Polygons ................................... 10-6
  Erase Vector Text by Drawing Over It ................ 10-6
  Merge Subdrawings into Views ............................ 10-7
Optimizing Event Handling Speed .................................... 10-7
  Use the Event Mask to Filter Events ................. 10-7
  Combine Event Requests .................................... 10-8
Reducing the Size of Your Executable ............................... 10-8
  Eliminate Unused Device Drivers .................... 10-8
  Link with the DataViews Shared Libraries ........ 10-8
  Eliminate Graph Types .................................... 10-8
  Link with the Optimized Libraries .................. 10-9
  Strip the Symbol Tables .................................... 10-9
Suggestions for OpenVMS ............................................... 10-9
  Monitor System Workloads ............................... 10-9
  Tune User Authorization Parameters ............. 10-10
  Tune System Parameters ................................. 10-10
  Hardware Recommendations ............................. 10-11
Suggestions for OPEN LOOK ............................................ 10-11
  Work Around Slow Raster Operations ............. 10-11
Chapter 10  
Tips for Improving DataViews Performance

This chapter discusses the following topics:

- Configuration Variables Affecting Speed
- General Optimizations
- Optimizing Load Speed
- Optimizing Draw Speed
- Optimizing Event Handling Speed
- Reducing the Size of Your Executable
- OpenVMS-Specific Suggestions
- Open Look-Specific Suggestions

Introduction

The tips discussed in this chapter can help you make significant performance improvements in your application. Most of these tips are useful when you are designing the code modules. Others are useful only at the final stages of development. To get the full benefit from these tips, you should review all of them early in the development process, as soon as you are familiar with DV-Tools.

It is also a good idea to review the tips in the Creating Views for Optimal Performance chapter of the DV-Draw User’s Guide. Many of the tips there can be implemented at any time during the development. You can automate some of the changes by writing and using utility programs that make the changes for you.

Certain performance improvements may include trade-offs in the look of the graphics or in convenience, or may require changes in the code to support both speed and an acceptable look. To determine the best outcome for your application, experiment with the options that seem promising.
### Configuration Variables Affecting Speed

The following configuration variables provide the simplest approach to performance tuning since you can set them easily at any point during development, then run trials without editing views or recompiling your code. These variables are documented in the *Setting the DataView Environment* appendix of the *DV-Draw User’s Guide* and in the *Installation and System Administration Manual*.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DVAVOID_INV_UPDATE</strong></td>
<td>Specifies whether to skip updates on invisible graphs. For better performance, set to <em>yes</em>.</td>
</tr>
<tr>
<td><strong>DVDEVICE_WIDE_LINES</strong></td>
<td>Specifies whether to use the driver to draw wide lines under X. For better performance, set to <em>no</em>.</td>
</tr>
<tr>
<td><strong>DVFDS_CATCH_SIGNAL</strong></td>
<td>Specifies whether to catch error signals from functions in function descriptor sets. For better performance, set to <em>no</em>.</td>
</tr>
<tr>
<td><strong>DVPRE80_COLORDYN</strong></td>
<td>Specifies whether to activate pre-8.0 color dynamics. For better performance, set to <em>no</em>.</td>
</tr>
<tr>
<td><strong>DVSD_DEACT_POOL</strong></td>
<td>Specifies whether a view is loaded each time it is referenced by a subdrawing. For better performance, set to <em>no</em>.</td>
</tr>
<tr>
<td><strong>DVPATH</strong></td>
<td>Specifies the DataView search path. For better performance, trim this path as described in the <em>Streamline the DataView Search Path</em> section later in this chapter.</td>
</tr>
</tbody>
</table>

### General Optimizations

#### Set Aside a Large Chunk of Memory

Allocate and free 2-3Mb of memory at the beginning of the *main* before calling *TInit*. This brings the memory for the application into the process space, eliminating many of the calls to get memory from the operating system.
**Link with the Optimized Libraries**

DataViews provides an optimized library which makes your DV-Tools code significantly faster. This library, `libDVtvo.a` (UNIX) or `LIBDVTVO/LIB` (OpenVMS), contains streamlined versions of the `T` and `VO` routines contained in `libDVmtv` (UNIX) or `LIBDVT/LIB` (OpenVMS). During development, you can use the non-optimized library to get validity checking and additional debugging information. When you are fine-tuning your application performance, you should link with the optimized library. This is recommended for all applications as part of the final preparations for delivery. For linking instructions, see the *Coding Environment* chapter.

**Use Deques to Consolidate Function Calls**

You can save multiple calls to the same routine by building a deque (list) of objects, then calling the routine for the deque. You should create the deque large enough to contain all the objects that you will be adding. It is better to make the deque too big than to run out of space. For example, if you want to add several objects to a view, you should follow these steps:

1. Create a deque large enough to contain the objects using `VOdqCreate`.
2. Create the objects and add them to the deque using a `VOdqAdd*` routine.
3. Add the deque to the view’s drawing object using `VOdrObAdd`.

Note that if you use this technique, you cannot call `TloGetSelectedObjectName` to get the name of a selected object, since names for the objects were never added to the drawing.

An additional benefit of deques is that they help consolidate Xlib graphics calls. For example, every time you call `TdpDrawObject`, `TdpDrawNextObject`, or `TdpEraseObject`, the X client buffer flushes its graphics to the screen. If you call the routine for several objects, the number of flushes grows. By consolidating the DV-Tools calls using a deque, you reduce the number of flushes to one. The performance gains are most noticeable for X server connections to other platforms where network lag is significant.

You can create deques for any data structure that fits into a `LONG`, so you may find deques useful for handling more than just objects. For more information about using deques, see the *DV-Tools Reference Manual*. 
Optimizing Load Speed

Streamline the DataView Search Path

To gain the greatest increase in load performance, make the following changes to the DataView search path (the DPATH configuration variable):

- Consolidate directories as much as possible and eliminate extra directories from the path.
- Move the most commonly used directories to the beginning of the path; move directories containing less frequently used files, such as dispforms.stb, to the end of the path.
- Put subdrawings in a directory with a short path name, then list this directory near the beginning of the path.

You can implement this tip any time during development, but it is particularly useful at the final stages.

Use Binary View Files

Binary files load faster, so you should use binary files for all performance tests and for the completed application. ASCII files are transferrable between systems and somewhat human-readable, so they are useful during development and for archives. When you have completed development, you can run the ViewConvert utility to convert ASCII views to binary. This utility is available as source code in the utilities subdirectory of the DV-Tools examples directory. Note, however, that network file systems used by multiple hardware platforms may require files in ASCII to run the completed application.

Optimizing Draw Speed

Make the Initial Drawing of Views Faster

The initial drawing of a view seems faster when the view is preloaded, since you do not have to wait for both the loading and the drawing. Preloading during initialization slows startup and takes up more memory, but makes execution faster. To cover the slower startup, you can load an initial view called a wait view, display it immediately, then preload the other views. The demos and the example program symload.c show how to use symbol tables to manage preloaded views.
If you can’t preload all the views because of memory constraints, you can still make substantial performance improvements by preloading some views. You may get the most improvement by preloading the largest views and views that are used by referenced subdrawings.

Once you load a view, keep it in memory rather than destroying it if there is a possibility that you will use it later in the application.

When you can predict which views may be needed next in the application, load them before they are actually required. For example, when you display a view that contains “hot-spots” that cause jumps to other views, load those other views in advance. This improves view transition time without requiring preloading of all views.

**Make Drawing Look Smoother**

If you notice an annoying flicker during drawing or updating operations, you can correct this problem by using **double-buffering**. When you use double-buffering, all graphics changes are first written to off-screen memory, then the screen is updated in one draw that encompasses all the changes. Double-buffering provides cleaner drawing because the intermediate drawing steps do not show on the screen. This sometimes gives the impression of faster drawing even though double buffering is actually slower.

To set up double-buffering, use one of the following methods:

- Use the :o option on the device name passed to TscOpenSet. For example, ”W:o” for Windows or ”X:o” for X.
- Pass a flag to TscOpenSet or GRset. The flag for Windows is V_WIN32_DOUBLE_BUFFER. The flag for X is V_X_DOUBLE_BUFFER.
- Set the DataViews*doubleBuffer resource to yes (X only).

**Prevent Screen Lockup**

When running on X, the window for displaying DataViews can sometimes remain blank for an unusually long time before the first draw. This is called **screen lockup**. To prevent this, wait for an expose event on the window before calling any DataViews drawing routines such as TdpDraw. The example programs win_X11.c and dv_w_motif.c show how to do this when you create your own window and pass it to TscOpenSet. When you let DataViews create the window internally in the TscOpenSet call, you should set the V_X_EXPOSURE BLOCK window attribute to YES.
### Redraw the Entire Screen Rather than Repairing

When many objects in a view have dynamics, updating using `TdpRedrawNext` may be faster than updating using `TdpDrawNext`. The standard update routine, `TdpDrawNext`, erases the current state of the dynamic objects using their specified erase methods, then draws them in their new state. `TdpRedrawNext` bypasses the erase step and just redraws the entire view with dynamic objects in their new state each time. It is safe to use in views with graphs or input objects because it does not reset them. However, `TdpRedrawNext` is faster in only a few situations, such as when you have a high ratio of dynamic to static objects and most of the dynamic objects change on the update. To determine if `TdpRedrawNext` is faster for your application, run test cases.

### Limit Updating to Particular Objects

It is more efficient to identify particular objects that need updating and call `TdpDrawNextObject` only for those objects than to call `TdpDrawNext`. If several objects need updating, create a deque containing those objects and pass the deque to `TdpDrawNextObject`. The performance improvement shrinks as the number of objects in the deque approaches the total number of dynamic objects.

To use this tip, you must write code that determines which objects are affected when particular program variables change. It may be easier to add this code if you are already making connections between the program variables and the objects during rebinding.

### Speed Up Your Polygons

You can speed up polygon drawing by using the `V_X_POLY_HINT` window attribute flag. This flag lets you indicate to the X driver how complex the polygons are, so the X driver can optimize its performance in drawing them. For more information, see `TscOpenSet` in the DV-Tools Reference Manual.

### Erase Vector Text by Drawing Over It

When vector text has been drawn over a solid background, it may be faster to erase it by drawing a rectangle over it than by using `TdpEraseObject`. Before drawing over the vector text, you should get its screen bounding box using `VOvtXfBox`. You can choose from several routines for drawing the rectangle:

- `TscDrawBackground` and `GRdraw_background` draw in the background color of the screen.
- `GRf_rectangle` draws in the current color, which you can set using `GRcolor`.

**Merge Subdrawings into Views**

When a subdrawing does not have dynamics applied to it, you can reduce its drawing time by merging the objects in it directly into the view where they are displayed. When you do this, you must take the scale and rotation of the subdrawing into account. Use the following procedure to merge the objects:

1. Get the transform for the subdrawing using `VOsdGetXform`.
2. Get the view using `VOsdViGet`.
3. Traverse the drawing in the view to get each graphical object.
4. Using `VOxfPoint`, apply the transform to all the control points in the graphical objects.

You can merge subdrawings that contain only one object, even if dynamics are applied to the subdrawing. When you do this, you must transfer the dynamics from the subdrawing to the merged object, and you must also take the scale and rotation of the subdrawing into account.

You can implement these tips as a utility you run once to change the views permanently, or as part of view loading that is repeated each time you run the application. If you change your views permanently, you will also see improvements in load time.

Before merging your subdrawings, consider whether you want to lose the flexibility that the subdrawing model provides. When you merge the objects from a subdrawing into a view, they can no longer reflect changes you might make to the subdrawing view.

---

**Optimizing Event Handling Speed**

**Use the Event Mask to Filter Events**

During the coding phase, set up the event mask to filter only for events of interest; eliminate events that the window does not use, particularly events that occur frequently such as motion, key release, enter notify, and leave notify events. If you occasionally need to gather additional events, change the event mask at these times, handle the additional events, then restore the more limited mask. For example, you can determine when the user needs a drag, change the mask to pass motion events, then change the mask back when the drag is completed. For more information on setting the event mask, see `VOscWinEventMask` in the *DV-Tools Reference Manual*. 
Combine Event Requests

If you use many event requests, you can make `VUserHandleLocEvent` considerably more efficient by combining similar event requests. This reduces the size of the event request list that the event handler must traverse. For example, instead of posting 100 event requests for picks on objects, post a single event request for simple edge events. In the service routine, determine which event you have. In the same vein, you can post a single event request for picks in a particular region of the screen where a group of related pickable objects are drawn.

When testing for the event type, put the most common cases early in the set of `case`, `if`, or `elseif` statements. Less common events, like quit events, should be handled later in the sorting.

Reducing the Size of Your Executable

Eliminate Unused Device Drivers

To eliminate device drivers you no longer need, re-install DataViews, choose only the device drivers you still need, and relink your application. For more information on installation, see the DataViews Installation and System Administration Manual. Implement this tip when you are preparing the application for delivery.

Link with the DataViews Shared Libraries

Link with the shared libraries (for Sun) or the shareable image (for OpenVMS) following the directions in the Coding Environment chapter. Linking with the shared libraries for Sun is the default when you use DVlink. Implement this tip during the coding phase or in the final preparations for delivering the application.

Eliminate Graph Types

You can eliminate graph types (display formatters) not used in your application. The process is similar to graph integration, which is described step by step in Technical Note #26: Integration Instructions for Graphs and Function Descriptor Sets. However, instead of adding a new graph, you remove the names of graphs. You must edit `dispforms.stb` in the `etc` directory and use the edited version in your call to `TInit`. You must also edit `MusrGraphs` to remove the unused display formatters. `MusrGraphs` is located in `src/names/tables` (UNIX) or `DVS:[SRC.NAMES.TABLES]` (OpenVMS).
### Tips for Improving DataViews Performance

Also follow the directions in the step for making new versions of the name tables, then relink your executables. Note that you must not eliminate `VDbar` or `VDcolorbar` since DataViews requires them internally.

You can implement this tip at any point during development. If you don’t want to limit your graph options during view creation, wait until the final preparations for delivery. Since this tip requires several steps, including the modification of some DataViews source code files, you may want to review the entire procedure before starting.

### Link with the Optimized Libraries

As mentioned earlier, you can link with an optimized library that contains streamlined versions of the `T` and `VO` routines. This makes the application faster and smaller.

### Strip the Symbol Tables

Strip the symbol tables from your application using the UNIX `strip` command. See your UNIX documentation for more information. Stripping the symbol tables eliminates debugging information, so you may want to implement this tip only at the end of development.

### Suggestions for OpenVMS

To improve the performance of DataViews on your system, you should develop a knowledge of your system workload and tune your system using the OpenVMS tuning tools. If you cannot achieve adequate performance through tuning, you may need to upgrade the memory or CPU of the system.

The first step in determining the proper tuning for your system is using the OpenVMS monitoring tools to develop knowledge about your system loads and tuning needs. The OpenVMS monitoring tools discussed in this section are documented in your OpenVMS reference manuals, particularly the *Guide to OpenVMS Performance Management* and *Guide to Setting up a OpenVMS system*.

When you monitor your system, you may find that DataViews applications can be both I/O and CPU intensive, depending on the design of the particular application. Watch for page faults and swaps using monitoring tools such as `MONITOR`. A DataViews application should not page frequently after it has loaded the executable image and all views, and should never swap. Swapping and frequent paging indicate the need for tuning or more memory.
Suggestions for OpenVMS Tips for Improving DataViews Performance

**Tune User Authorization Parameters**

Use `AUTOGEN` regularly to profile system use and tune system parameters. After any major system reconfiguration, such as installing a new version of OpenVMS or DataViews, run `AUTOGEN` once a week for several weeks. To investigate system virtual memory usage, use utilities such as `SHOW MEMORY`, `MONITOR`, and `SHOW PROCESS` while running the DataViews application.

Tune the User Authorization Record of each user of DataViews. For a single user VAXstation 4000 VLC with 24 Mb, we use the following parameter settings for the authorization record, which can be a starting point for your parameter tuning:

- Maxjobs: 0 Fillm: 100 Bytlm: 65536
- Maxacctjobs: 0 Shrfillm: 0 Pythlm: 0
- Maxdetach: 0 BIOLm: 200 JTquota: 1024
- Prclm: 10 DIOlm: 200 WSdef: 600
- Prio: 4 ATLlm: 200 WSS quo: 600
- Queprio: 0 TQE lm: 20 WSextent: 8200
- CPU: (none) Enqlm: 200 Pgflquo: 20000

Adjust `WSextent` and `Pgflquo` based on the number of users on the system and the applications they run. For a single user, make `WSextent` and `Pgflquo` as high as you can. For multi-user systems, `WSextent` should be a minimum of 4000. If the application loads a lot of views, `WSextent` should be 8000 or more. For instructions on setting `WSextent`, see your OpenVMS documentation on the `AUTHORIZE` utility. If the virtual memory requirements of the application are high, you must use a higher setting of `Pgflquo`. To support higher settings of `WSextent` and `Pgflquo` in the User Authorization Record, you may also have to change system parameters.

**Tune System Parameters**

Using the information gained from monitoring system loads, tune `SYSGEN` parameters such as `WSMAX`, `WSDEC`, and `PFRATL`.

Make `WSMAX` is at least as high as your highest `WSextent` value in the User Authorization Records. For instructions on setting `WSMAX`, see your OpenVMS documentation on `SYSGEN` and `AUTOGEN`.

Note that all settings of `Pgflquo` in User Authorization Records must be smaller than the size of the `PAGEFILE.SYS` file. If you increase `Pgflquo`, `PAGEFILE.SYS` should also be larger. To find out the size of `PAGEFILE.SYS`, use the `SHOW MEMORY` command and look in the `Total` column of the last line displayed. If `PAGEFILE.SYS` is large, you can split it into smaller files.
Tips for Improving DataViews Performance

and spread it across more than one disk to balance the I/O load. If your application loads many views, you can improve loading time by moving the views to a disk that does not contain the system pagefile.

One simple method of alleviating paging problems is to enable voluntary working set decrementing, which lets the system trim memory from processes with low page fault rates. This can be done by setting WSDEC and PFRATL to non-zero values appropriate to your system. This can generate improvements on systems with a number of mostly dormant processes that cannot simply be dispensed with. For information on setting these parameters, see the Guide to OpenVMS Performance Management.

To improve overall performance, eliminate idle processes, use a minimal number of DECterm windows, and don’t run unnecessary utilities. They all use memory, which limits the memory available to the key applications.

Hardware Recommendations

Having enough memory is important for acceptable performance. To run OpenVMS, DataViews, and DECwindows/Motif on a single user VAXstation 3100 or 4000, you must have a minimum of 16 Mb, which still may not provide acceptable performance. For better performance, you need 24 or 32 Mb of memory. The system administrator should tune the system and determine whether there is enough memory and CPU available.

Disk I/O speed is a significant factor in view loading performance. Some disks are slow; for example, SCSI disks limit the speed of machines that normally run quite fast. When loading large numbers of views, competition for the disk by other processes can degrade DataViews performance sharply. Limiting disk access by other processes can improve performance.

Suggestions for OPEN LOOK

Work Around Slow Raster Operations

Rasters are slow on the Open Look server because of a problem in the server. It can affect speed on raster erase, images, icons, raster versions of the strip charts, smoothed spectro graphs, and smoothed stacked spectro graphs. There is no known work-around for this problem, so you may want to avoid the features that depend on rasters.
**add** - to create a new element to or include an existing element in a structure or object, e.g. add a variable to a data source or add a graph to a display.

**ADDRESS** - typedef of generic pointer. Used for buffers and argument blocks as well as the base type for all DataView private structures.

**ar** - arc object.

**arc object** - *(ar)* DataView graphical object.

**arg** - argument block.


**aspect ratio** - the ratio between the height and width of a device screen, window, or any viewport rectangle on the screen.

**at** - attribute structure.

**attach** - synonym for add.

**attribute** - *(at)* DataView public structure.

**ATTRIBUTES** - attribute *(at)* typedef.

**AXISDESC** - axis descriptor DataView private type.

**axil labels** - see label.

**background color** - the color on which the context is placed. When DataView erases a graph, it redraws the graphics using the background color.

**bounding box** - an invisible box used in defining a specific drawing area in DV-Draw, such as the isolation of a group of objects.

**ci** - circle object.

**circle object** - *(ci)* DataView graphical object.
**clone** - generic routine name verb. Make an exact copy of a data structure. This is usually a deep copy, meaning all its sub-objects are copied as well.

**close** - generic routine name verb. Deallocate use of a limited resource, such as a screen object or file, freeing it for use by subsequent open calls.

**co** - color object.

**color object** - DataViews non-graphical object.

**color palette** - see palette.

**COLOR_SPEC** - color specification typedef.

**color specification** - DataViews public structure.

**color table** - DataViews public structure.

**COLOR_TABLE** - color table typedef.

**color threshold** - DataViews public structure. A particular value within the range of a variable. Color thresholds can be assigned a unique color so that the color of a variable changes as its value crosses the threshold (e.g., when the pressure in a valve crosses into the danger zone).

**COLOR_THRESHOLD** - color threshold typedef.

**constant** - a data source variable that does not change in value while a view is running. One use for a constant in a graph is as a reference point with which to compare other changing values. Another use is in graph types that require more variables than you want to display.

**context** - all the parts of a graph that help a user interpret the data being represented. The context includes: title, legend, axis labels, axis ticks, axis tick labels, viewport outline, and grid lines. The context is drawn using the graph’s foreground color after erasing to the background color. Each part of the context may be modified by the user.

**control points** - point objects that appear as small squares when creating an object or when an object is selected. Control points are used to change the size, shape, and location of an object. See also the **move point** entry.

**coordinate system** - manner of measuring horizontal and vertical space on the screen.
copy - to duplicate a single object or group of objects. Also, the result of such a duplication. A copy of an object contains the same attributes as the original. A copy of a group contains identical attributes of all of the objects contained in the group.

create - generic routine name verb. Allocates memory for a private structure or an object and returns a pointer to it. Usually fills the data structure with specified values or default values.

cs - color specification.

color specification.

ct - color threshold.

color threshold.

ctb - color table.

color table.

ctt - color threshold table.

color threshold table.

cursor - a small shape, usually an arrow, that indicates your current position on the screen. The cursor can be moved by specific cursor control keys or by a locator such as a mouse.

cursor control keys - keys on the keyboard that move the cursor. The keys that are used to move the cursor are system-dependent. See DataViews Platform-Specific Notes for a list of the cursor control key definitions for your system and device that runs DataViews.

data group - (dgp) DataViews private structure. Underlying data structure of a data group object. Contains all the information needed to display a graph on the screen, such as variable descriptors and a display formatter to determine the graph’s type.

DATAGROUP - data group (dgp) typedef.

data group object - (dg) DataViews graphical object. Object representation of a graph. Contains a data group structure and control points determining where it is to be displayed on the screen. Data group objects come in 38 predefined types of graphs, such as bar graph, pie chart, line graph. Also referred to as display formatters and graphs.

data source - (ds) DataViews private structure. Contains information on where and how to obtain data for use by the dynamic objects in drawings. The source of data can be a file, a process, or a constant. A data source may contain one or more data source variables.
**DATASOURCE** - data source \((ds)\) typedef.

**data source list** - \((dl)\) DataViews private structure. Contains a list of all the data sources which supply data to dynamic objects in drawings. Data source lists may be shared by several views.

**DATASOURCENAME** - data source \((dl)\) list typedef.

**data source type** - file, process, or constant.

**data source variable** - \((dsv)\) DataViews private structure. Contains a variable buffer and information about how to interpret its contents. Data is moved from a data source into its data source variables from which it can be interpreted by a variable descriptor.

**data structure** - any of the DataViews public or private types.

**data type** - the kind of binary number that a particular variable is stored as. DataViews can use data stored as double precision floating point, single precision floating point, long integer, unsigned long integer, short integer, unsigned short integer, byte, and unsigned byte. The actual length of these numbers is system-dependent.

**DataViews external global** - global variable in the DV-Tools link library which is directly accessible to the applications programmer using the `GLOBALREF` \((extern)\) declaration. Examples are display formatters and input handlers.

**DataViews internal global** - global variable in the DV-Tools link library not directly accessible to the applications programmer. Can only be accessed through DV-Tools routines, for example, current screen.

**DataViews type** - predefined data structures used by DV-Tools and applications programs calling DV-Tools routines. The DataViews types are classified as either public (see DataViews public type) or private (see DataViews private type).

**DataViews private type** - DataViews type which can only be accessed through DV-Tools routines, not directly by the applications programmer. These types include private structures, graphical and non-graphical objects.
DataViews public type - DataViews type which can be accessed directly by
the application programmer. Most of these are C structures located in the
include file dvstd.h. A few others can be declared directly by the application
programmer such as argument blocks and data buffers.

DATUM - datum (long) typedef.

DATUM_TYPE - data type typedef.

default - a value automatically supplied by DataViews or an action
automatically taken unless another is specified by the user.

delete - to remove an element from the structure to which it is attached. When
an item is deleted, the Undelete option appears on the menu. When you delete
a display variable from a graph, the variable is still attached to its data source.
Otherwise, when you delete an element, it is permanently removed.

deqe object - (dq) DataViews non-graphical object.

dereference - generic routine name verb. Decrement the reference count of
an object and, if the reference count becomes zero, destroy it. Used in VO
routines.

destroy - generic routine name verb. De-allocate memory for a private
structure or an object, destroying any sub-structures that are not referenced by
other data structures.

device - 1) graphics terminal or workstation window on which the DataViews
graphical output appears. 2) device name.

device name - string used to refer to a specific device available to
DataViews.

device-specific - features or operations that are only pertinent to certain
devices that run DataViews. Device-specific implementations of DataViews
exist because DataViews can run on systems with different display and cursor
control capabilities.

df - display formatter.

dgp - data group (pointer).

directory - an area within an operating system that contains locally
accessible files and other directories.
**DISPFORM** - display formatter (df) typedef.

**display area** - the portion of the screen on which graphs are drawn. The display area is analogous to a piece of paper on which graphs are drawn: the “paper” can be moved closer or further away and repositioned.

**display formatter** - (df) 1) DataViews private structure. 2) a VD module, containing a set of functions that implement the drawing, updating, and erasing, of a datagroup (graph).

**display formatter slot** - a vertical area unit of a display formatter that accommodates a unit of data, a time slice.

**display formatter slot count** - the number of slots a display formatter can accommodate.

**dl** - data source list.

**double precision floating point** - a floating point number that has twice the number of decimal places of a normal floating point number. Usually shortened to “double.” DataViews converts double precision numbers to single precision floating point numbers prior to display.

**dp** - drawport.

**dq** - deque object.

**dr** - drawing object.

**drawing** - actual graphical representation on the screen. A drawing can be as simple as a line, or complex with objects and graphs.

**drawing area** - the part of the screen in DV-Draw in which the drawing appears and in which it may be edited.

**drawing coordinate** - world coordinate.

**drawing object** - (dr) DataViews graphical object.

**drawing viewport** - rectangular portion of a view, or drawing, specified in world coordinates.
**drawport** - *(dp)* DataViews private structure. Contains all information needed to display a view on a screen. This includes pointers to the view structure and screen object, the drawing and screen viewport rectangles, world-to-screen coordinate transform, and information for clipping to obscuring drawports.

**DRAWPORT** - drawport *(dp)* typedef.

**ds** - data source.

**dsl** - data source list.

**dsv** - data source variable.

**DSVAR** - data source variable *(dsv)* typedef.

**DV_COORD** - coordinates typedef.

**DV_POINT** - point typedef.

**DVLink** - DataViews script that links the application program with modules from the DataViews library.

**dynamic control object** - *(dy)* a mechanism by which DataViews makes simple graphical objects dynamic. A dynamic control object can give a graphical object transformational, attribute, and subdrawing dynamics. Transformation dynamics change the graphical object’s control points by rotation, translation, and scaling. Attribute dynamics change the graphical object’s attributes, such as color, fill status, line type, etc. Subdrawing dynamics substitute a different subdrawing.

**dynamic object** - graphical object that has a dynamic control object attached to it.

**dynamics** - changes in data and its representation over time.

**ed** - edge object

**edge object** - *(ed)* DataViews non-graphical object. An edge object may contain a graphical object as its representation on the screen.

**el** - ellipse object
Glossary

**element** - a display or a discrete part of a display. In this manual, the following portions of a display are referred to as elements: data source, input variable, graph, display variable. Each of these can become the selected element. Each number within a vector or matrix is also called an element.

**ellipse object** - *(el)* DataViews graphical object

**er** - event request.

**event** - when interacting with a program via some physical device such as a mouse or keyboard, the operating system packages this interaction in a system-dependent event structure which DataViews converts to a device-independent event structure, the location object.

**event request** - *(er)* DataViews private structure.

**EVENT_REQUEST** - event request *(er)* typedef.

**feedback** - the capability of seeing what the graph will look like when it is run. DataViews uses dummy data during a preview. Graph Edit Menus also provide feedback in displaying the title, graph type and axes labels of the current graph. The display area also provides feedback as to the relative size and placement of all the graphs in the display.

**foreground color** - the color of the title, axes labels, axes ticks, legend, text, and viewport outline of a graph.

**format** - the way in which data is represented in a file or process. Data can either be stored as ASCII representations of numbers or as binary numbers.

**fp** - float point.

**general object** - *(ob)* DataViews graphical object.

**get** - generic routine name verb. Return an attribute or pointer to a sub-structure of a data structure.

**graph** - graphic representation of data that appears on the monitor screen when a display is run. Each graph is composed of: a viewport, a graph type, one or more variables, and context. A graph can be any size that is large enough to accommodate the graph type and context (if context is going to be shown) that the user chooses. The number of graphs that can be used in a single display is limited only by the size of each of the graphs and of the screen.
**graph type** - the display formatter that translates a data group into a visual format.

**grid** - 1) the reference grid in a graph 2) in DV-Draw a constrained set of drawing positions that a user may pick.

**group** - one or more graphs that have been selected by the Select Group option. Groups can be moved, copied, deleted, and edited as a single element, although individual graphs in the group can be edited separately as well.

**highlighting** - special manner of displaying some graphical feature to make it stand out, e.g. reversing color, high contrast, or outlining.

**horizontal axis** - the axis which shows the column or element number of the variables in a matrix format. When a vector variable is used in a graph, the horizontal axis shows its length. The horizontal axis is not used with scalar variables.

**horizontal axis label** - a user-supplied name that identifies the column axis of a matrix graph. The default label is blank.

**horizontal axis tick labels** - numbers identifying the scale of the horizontal axis.

**icon** - a pictorial representation of an object, such as a graph type.

**ih** - interaction handler.

**in** - input object.

**include files** - files containing information used by DV-Tools when running an application.

**INHANDLER** - interaction handler (ih) typedef.

**input object** - (in) DataTypes graphical object.

**input technique object** - (it) DataTypes non-graphical object.

**interaction handler** - (ih) DataTypes private structure. One of the DV-Tools data structures that facilitates the input and display of drawings and information. There are seven types provided with DV-Tools: Text Input, Menu, Slider, Checklist, Toggle Item, Combined Form, and Multiplexor Form.
it - input technique object.

iteration - the display of one value for every variable in every graph in a display. Users can set the number of iterations independently from the number of samples a particular graph can show. The start and end iteration number is user-selectable. See also the sample entry.

keypad - the rectangular array of numeric keys on a computer keyboard. The keypad is generally located on the right side of the keyboard. On some terminals, keypad numbers double as cursor control keys.

label - a name which identifies an axis to a viewer (axis label) or the numbers on axes ticks which show the scale of the graph (axis tick label).

legend - a table in the upper right-hand corner of a graph which shows the name of each variable attached to the graph and the colors assigned to each variable. The legend graph type displays only the legend of each variable attached to it.

line object - (ln) DataViews graphical object.

line type - line pattern (solid, dotted, dashed, etc.) that is used when drawing a line.

ln - line object.

lo - location object.

load - generic routine name verb. Retrieve a data structure from its file representation into memory, returning a pointer to it.

LOCAL - defines a variable for local use, that is, it can be accessed only by code in that module.

location object - (lo) DataViews non-graphical object. References the location of the cursor through the positioning of the mouse.

locator - a graphics control device, such as a mouse or a pen and tablet, that moves the cursor on the screen and may also contain selection buttons.

master dsl - combined list of each drawing’s data sources. They are grouped into the master data source list (dsl) to simplify the information call when the program is running.
<table>
<thead>
<tr>
<th>Glossary Entry</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>matrix</strong></td>
<td>an array of numbers containing one or more rows of one or more values. Also called a table. DataViews reads matrices in column-row order, that is column first, then row. A matrix of one row is called a vector.</td>
</tr>
<tr>
<td><strong>menu</strong></td>
<td>a set of options shown on the screen, either as text or icons. When you select an option, DataViews executes the operation associated with it.</td>
</tr>
<tr>
<td><strong>message box</strong></td>
<td>the rectangular area on the top of the Main Menu which displays prompts, error messages, and help text.</td>
</tr>
<tr>
<td><strong>monitor</strong></td>
<td>another term for the CRT or screen portion of a computer terminal. DataViews can run on color or monochrome monitors, depending on the version used.</td>
</tr>
<tr>
<td><strong>mouse</strong></td>
<td>a locator device attached to a computer terminal to control the cursor. A user moves the mouse across a surface, which in turn moves the cursor on the screen. A mouse can have one to three buttons, each of which performs specific operations, such as selecting an option or returning to a previous menu.</td>
</tr>
<tr>
<td><strong>move point</strong></td>
<td>a small white square that appears in the center of the selected graph or group. The move point is used to move the entire graph or group without changing their size or shape. See also Control Point.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>the title of a variable, constant, or data source which is used to identify it within DataViews. The name of a data source is derived from the origin of the file or process used in the data source. The default name of a constant data source is “constant,” but users can assign a different name, if desired. Variables are assigned default names, Var:1, Var:2, Var:3, etc., in sequence. Constants are assigned the default name of zero. Users can assign new names.</td>
</tr>
<tr>
<td><strong>name-value pair</strong></td>
<td>(nv) DataViews public structure.</td>
</tr>
<tr>
<td><strong>NAME_VALUE_PAIR</strong></td>
<td>name-value pair (nv) typedef.</td>
</tr>
<tr>
<td><strong>no</strong></td>
<td>node object.</td>
</tr>
<tr>
<td><strong>node object</strong></td>
<td>(no) a DataViews non-graphical object. A node may contain a graphical object as its representation on the screen.</td>
</tr>
<tr>
<td><strong>normalized device coordinate</strong></td>
<td>coordinate in the range [0.0,1.0].</td>
</tr>
</tbody>
</table>
**NULL** - argument that usually sets an attribute to default, that is, sets it to some pre-specified value or data list.

**nv** - name-value pair (changed from DF_ARG in 5.0)

**ob** - general object.

**object** - DataViews private type. Objects fall into two categories: Graphical objects: lines, polygons, etc. and Non-graphical objects: color, threshold tables, etc. Graphical objects are the basic building blocks of drawings. Every entity drawn on the screen in a view is represented in memory with a graphical object, so the images themselves can be thought of as objects.

**OBJECT** - typedef for all DataViews objects.

**object descriptor** - synonym for object id.

**object id** - integer (typedef OBJECT) which uniquely identifies each created object. It is not strictly a pointer, but can be thought of as such when being passed to and from DV-Tools routines.

**object slot** - a means of attaching arbitrary information to graphical objects.

**open** - generic routine name verb. Allocate use of a (theoretically) limited resource such as a screen object or file.

**origin** - the specification of where in the system a data source is located. In the case of a file data source, the origin is the full pathname of the file containing the data. In a process data source, the origin is the command or the pathname plus any parameters of the script that generates the data. Constant data sources originate within the system and have no origins.

**outline** - a box drawn around a graph’s area.

**palette** - a visual display of all colors available on a particular system or terminal that users can choose from when assigning new colors to a variable, or the foreground or background.

**pan** - to move the display area so that a different point is in the center of the screen.
pathname - sequence of directory names, which provide the operating system with all the information it needs to uniquely identify a file, device, or directory. Pathnames have the following format in UNIX “/dir1/dir2/.../file” or “[dir1.dir2]file” in OpenVMS.

PLR_POINT - polar point (pp) typedef.

point - point [structure]. DataViews public structure.

point object - (pt) DataViews graphical object. Coordinate established on the display device, indicating the point from where an object is drawn.

pointer - address of a data structure in memory. Also used as a synonym for object id.

polar point - (pp) DataViews public structure.

polygon object - (py) DataViews graphical object.

pp - polar point.

preview - sample run of a display, using dummy values created by DataViews.

private structure - DataViews private type which is not an object and therefore can not have multiple references. The T, VP, VG, VU, and VT routines all manipulate private structures. Private structures all have a base type as ADDRESS, but they are given descriptive type names to distinguish them.

process - with a process data source, a program that generates data for display by DataViews. Process data source specifications may contain any parameters or arguments needed to run the program or script, but must not require user input from the terminal.

pt - point object.

py - polygon object.

quit - to leave the current menu and return to the previous menu. Also, to leave the DataViews system entirely.
range - the minimum and maximum values a variable descriptor can reach. Users can specify a different range for any variable. Any values outside the range are truncated to the end of the range. A variable can have different ranges in different graphs in the same display. Variables in the same graph should have the same range for the graph to be meaningful. The scale and range of a graph are taken from the first variable attached to the graph. Ranges are shown in brackets in DV-Draw.

re - rectangle object.

rect - rectangle (structure).

rectangle - (rect) DataViews public structure.

RECTANGLE - rectangle (rect) typedef.

rectangle object - (re) DataViews graphical object.

reference - 1) generic routine name verb: increment reference count of an object. Used in VO routines. 2) to contain a pointer to a data structure.

reference count - the number of times an object is referred to by other objects.

rgb - r-g-b specification.

RGB_SPEC - rgb specification (rgb) typedef.

rgb specification - (rgb) DataViews public structure. The letters represent the red, green, and blue values.

RS232 - a commonly used electrical standard for connecting terminals, printers, and other devices to computers. An RS232 connection is also called a serial port.

ru - rule object.

rule object - (ru) DataViews non-graphical object. A rule object connects a graphical object to a description of an action that depends on a specified event and condition.

sample - one value of a variable. The number of samples in a graph determines how many times a graph is updated before previous samples are overwritten.
save - generic routine name verb. Store a representation of a data structure in a file so that it may be preserved for future use.

sc - screen object.

scalar - a single datum or number.

screen - the TV-tube-like portion of a computer terminal which displays data.

screen coordinate - actual coordinate system of the display device, usually in pixels, where the lower left corner is (0,0) and the upper right corner depends on the resolution of the device. In the case of window systems, the display “device” is the window being used to display the view, not the physical device. Used by VO and GR routines. Synonym: pixel coordinate.

screen object - (sc) DataViews non-graphical object.

screen viewport - rectangular portion of a screen, specified in screen coordinates.

script - a set of UNIX shell commands that are executed together.

sd - subdrawing object.

search path - the path to a specific directory is substituted here.

select - to pick an element to work with. All options apply to the currently selected element until a new element is selected.

shape - the number of rows and columns of data that make up the variable.

shell - the UNIX operating system environment which provides access to files, allows users to run programs, etc. There are a number of shells commonly used in UNIX systems, the two most common being the Bourne shell and the C shell.

simple polling - uses TloPoll to poll the cursor to capture key or mouse button presses. The keys that can be captured are limited to the ASCII character set and the mouse button presses are identified as <ctrl A> <ctrl B> <ctrl C>.

sk - slotkey object.
slot - internal field that can be used to attach arbitrary information to objects. A slot can contain any one of the following data types: integer, integer array, float, float array, object, pointer to a NULL-terminated string. You can think of an object’s slots as being arranged in a table that can be accessed using either a slotkey object or an index.

slotkey object - \((sk)\) provides a name, a description of the type of information, and a means of accessing the information in the object’s slot.

sn - symbol table node.

snp - symbol table node.

spt - screen (coordinate) point.

sre - screen (coordinate) rectangle.

st - symbol table.

standard in - \((\text{stdin})\) in UNIX, standard in refers to the default source of input data for a program or a shell command. Standard in is normally assigned to the keyboard, but may be redirected so that input may be taken from a file or from the output of another program.

standard out - \((\text{stdout})\) in UNIX, the output of programs and shell commands are sent to standard out by default. Standard out is normally assigned to the terminal, but may be redirected into a file or used as input for another program.

stp - symbol table.

str - string.

structure - a part of DataViews that is composed of one or more elements. The following parts of DataViews are referred to as structures within this manual: display, data source, graph.

subdrawing object - \((sd)\) DataViews graphical object.

svp - screen (coordinate) viewport rectangle.

symbol table - \((st)\) DataViews private structure.

symbol table node - \((sn)\) DataViews private structure.
SYMNODE - symbol table node (sn) typedef.

SYMTABLE - symbol table (st) typedef.

template - drawing which defines the appearance and certain aspects of functioning of an input object.

text dynamics - dynamics that displays the formatted variable value of a text object.

text object - (tx) DataView's graphical object.

threshold - color threshold or value threshold.

threshold table object - (tt) DataView's non-graphical object.

tick labels - strings (usually numeric) adjacent to ticks on the axes of a graph.

ticks - small lines on the axes of a graph that indicate the scale of the range used in the graph.

time axis - the axis that represents the number of samples that a particular graph has shown. In a scalar graph, the time axis is the horizontal axis. In a matrix graph, the time axis is shown as a digital number below the horizontal axis.

time axis label - a user-supplied name that identifies the time axis of a graph. The default time axis label is blank.

time axis tick labels - numbers identifying the scale of the time axis.

time slice - a unit of data that fills a slot in a display formatter. Data handled as a matrix or vector are displayed one time slice at a time.

transform object - (xf) DataView's non-graphical object.

traverse - successively visit every element of a structure. DataView's traversal routines allow the application to search for a given element of a structure, as well as to perform operations on elements of a given structure.

tt - threshold table object.

tx - text object.
valuator - a graphical representation of option selecting device. Also referred to as a slider.

valuator technique - the method by which sliders are read for input. Provides a means for the user to select options or commands from the slider.

value axis - the axis that represents the numerical value of variables in a graph. The value axis is the vertical axis of a scalar graph. In a matrix graph, the value axis is shown as a digital readout on or below each element. The scale of the value axis is set by the range of the first variable used in the graph.

value axis label - a user-supplied name that identifies the vertical axis of a scalar graph showing the values of the variables in the graph. The default label is blank.

value axis tick labels - numbers identifying the scale of the vertical axis.

VARDESC - variable descriptor (vdp) typedef.

variable - a single piece of data whose value can change over time. Variables can have both input and display characteristics. The input characteristic is shape. The display characteristics are range, line type, color, color thresholds, and shape. Variable characteristics are initially assigned defaults by DataViews.

variable descriptor - (vdp) DataViews private structure. Gives complete information on a specific variable: Memory location, how to access, type, range, etc. All dynamic objects and graphs contain one or more variable descriptors. DO NOT confuse with variable descriptor object (vd). For historical reasons, the VP and VG routine names use the abbreviation vd to refer to variable descriptors, while in the VO routines, vd refers to a variable descriptor object. Variable descriptors are also abbreviated as vdp.

variable descriptor object - (vd) DataViews non-graphical object.

variable shape - number of rows and columns of a variable.

vd - variable descriptor object.

vdp - variable descriptor (pointer).

vector - a matrix containing one row of values.

vector text object - (vt) DataViews graphical object.
vertical axis - axis that shows the number of rows in a variable. When a vector variable is used in a graph, the vertical axis only shows one row. The vertical axis is not used with scalar variables.

vertical axis label - a user-supplied name that identifies the row axis of a matrix graph. The default label is blank.

vertical axis tick labels - numbers identifying the scale of the vertical axis of a matrix graph.

vi - view.

view - (vi) DataViews private structure. A 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 data sources which supply the data needed to make the drawing dynamic.

VIEW - view (vi) typedef.

view file - the file representation of a view (vi). View files can be created and read in by DV-Draw and DV-Tools using the TviLoad and TviSave routines. They are the only means of communicating graphical information between DV-Draw and DV-Tools. There is a sub-directory with this name that holds the pre-drawn graphs, the menu and slider templates, and other system drawings.

viewport - a rectangle structure specifying a rectangular portion of the screen in which a graph, a drawport, or any other graphical entity is shown. A viewport may be the full size of the screen or smaller. The viewport of a graph may be outlined in its foreground color. Viewports may be specified in world (wvp), virtual (vvp), or screen coordinates (svp), and are required as parameters in TdpCreate, VOobBox and several other routines.

virtual coordinate - describes a “virtual” screen which is mapped by DV-Tools to the actual screen dimensions. Virtual coordinates fall in the range of [0.0] (lower left corner) to [32767, 32767] (upper right corner), and represent a proportional fraction of the actual screen height and width, so dimensions specified in virtual coordinates vary in their appearance depending on the aspect ratio of the display screen. Used by TdpCreate and some VG and VP routines.
visibility dynamics - object dynamics that determines whether or not a graphical object is visible in the view depending on the value of its data source variable.

visibility list - ordered list of drawports determining which drawports shall appear in front of which on the screen. Drawports in front obscure those in back.

vpt - virtual (coordinate) point.

vre - virtual (coordinate) rectangle.

vt - vector text object.

vvp - virtual (coordinate) viewport rectangle.

window - region on a workstation screen within which a DataViews application can be run. Windows are usually maintained by the local workstation’s “window system” and appear in DV-Tools as devices to which screen objects are attached.

window event polling - uses VOloWinEventPoll or VOscWinEventPoll to poll the cursor to capture not only key and mouse button presses and releases but also window expose, resize, and quit events.

world coordinate - used to define the position of DV-Tools graphical objects. World coordinates are defined so that 0,0 is the center of the drawing, and fall in the range of [-16383,-16383] to [16383,16383]. Used by T and VO routines.

world viewport - defines how much of a drawing is displayed, and tailors the drawing to a specific terminal type. Used when setting up drawports.

wpt - world (coordinate) point.

wrap-around - the action of overwriting previously displayed samples when there is more data to be shown than the number of samples called for in the graph. If a graph can show 10 samples, for example, the 11th sample wraps around and overwrite the 1st sample, the 12th sample overwrites the 2nd, and so forth.

wre - world (coordinate) rectangle.

wvp - world (coordinate) viewport rectangle.
xf - transform object.

**zoom factor** - the amount by which the display area is magnified or shrunk during a zoom operation. The zoom factor can be any number between 1 (magnify to same size, i.e.: no change) and 4 (quadruple the size of the viewport).

**zoom in** - to magnify the display area by the current zoom factor. The display runs at normal size.

**zoom out** - to reduce the display area by the current zoom factor. The display runs at normal size.
A
active subdrawing 6-28
application
  control 2-5, 8-40
data rebinding 6-9
dialog 2-2
display 4-21
  framework 3-14
separation 2-2
arc object 5-10
aspect ratio 4-9
attribute
  dynamics 6-22
  object 5-1, 5-6
  table 5-8
screen 4-6

B
blinking 6-33
block diagram 3-3
block structures diagram 3-2
B-spline curves 5-11
buffer 6-6
  access 6-8
  rebinding 6-9
  stuffing 6-7
button 8-21
  event request 8-15

C
callback routine see service routine
checklist 7-8
circle object 5-9
client 8-12, 8-28
  monitored client 8-28
clipping 4-8, 4-14
coding tips 9-8
color
  attribute 5-8
drawport 4-16
  graph 6-14

icon 5-19
image 5-19
input 7-8
object 5-1, 5-23
  pixmap 5-19
RGB 5-24
screen 4-7
  table 1-5, 5-25
threshold
  object dynamics 6-27
transform 5-20
combiner 7-8, 7-10
compiling 9-2
configuration file 1-3, 1-5, 9-8
control flow 2-7, 2-10
  callback routines 8-20, 8-31
  diagram 2-10
  planning 2-5
control loop 2-9, 8-40
circle object 5-1, 5-4
conventions 1-7
coordinates
  conversion 4-16
drawport mapping 4-8
  event 8-21
layout view 4-13
point object 5-22
screen 4-4
transform 5-34
vector text 5-14
virtual 4-4
window system 4-4
world 4-4

D
data
  access 6-8
  buffer 6-6
display 6-1
  dynamic object 6-22
graph 6-11
Index

flow 4-18, 6-1, 6-3, 6-38
object dynamics 6-22
rebinding 6-9
gathering 6-1, 6-5
input object 7-1
input objects 7-9
planning 2-5
range 6-9
data group 6-1, 6-11
object 6-2, 6-10
data source 3-3, 6-1, 6-5
data source list 4-17, 6-1, 6-5
master 6-36
merging 6-37
data source variable 6-1, 6-6
data structure
diagram 3-7, 3-9
private type 3-10
public type 3-10
DATUM 6-23, 6-25
dbx 9-11
debugging 9-9
routines 9-9
deq object 5-1, 5-27
dereference 5-5
device see also screen, window
monochrome 4-7
dialog see also input
application 2-2
directory structure 1-3
display
block diagram 3-7
objects 4-12
views 4-11
display formatter 6-2, 6-12
drawing see also display, drawing object
drawing object 4-17, 5-1, 5-26
deq 5-27
merging 4-18
subdrawing 5-15
drawport 4-1, 4-8
aspect ratio 4-9
clipping 4-8, 4-14
color 4-16
selected 8-21
transform 4-8
DV-Draw 2-5, 3-1
DVINIT 9-4
DVinit 9-2
DVLINK 9-12
DVlink 1-5, 9-11
DVLint 9-9
DVSD_DEACT_POOL 5-16, 6-30
dynamic control object 6-2, 6-22
dynamics 4-17
block diagram 3-4
graph 6-10
object 6-22
subdrawing 6-22, 6-27, 6-28
E
echo function 8-3, 8-31
device-specific events 8-37
device-specific 8-37
edge object 5-2, 5-28
eclipse object 5-10
event
device-specific 8-37
location 8-14
mask 8-7
model 2-7
setting up 8-38
event handling
device-specific events 8-37
event request handler 8-9
explicit 8-35
event request
button 8-15
client 8-12
key 8-15
label 8-17
loop 8-9
management 8-19
object 8-14
posting 8-11
service result request 8-28
region 8-14
F
feedback in graphs 6-15
fill status 5-9
floor value 6-19
function prototypes 9-1
Index

G
geometry object 5-28
graph 6-10 see also data group, display formatter
color 6-14
context 6-12
feedback 6-15
graphical object see object
graphics
  low level 4-5
GRcolor 4-15
GRf_rectangle 4-15
GRg_real_color_tab 5-26
GRrascreate 5-18
GRrasdraw 4-15
GRrasget 4-15, 5-18
GRraster 5-17
GRs_color_table 5-20, 5-25
GRscs_to_vcs 4-3, 4-11
GRset 4-7
GRvcs_to_scs 4-3, 4-15

H
heap 5-4

I
icon 1-5
icon object 5-16
image object 5-16
include files 9-1
input planning 2-5
input object 7-5 see also input object technique,
  interaction handler
    action keys 7-7
    echo function 8-31
    erase method 7-6
    event request handler 8-26
    monitoring 8-28
    template 1-5, 7-8
input technique object 7-1, 7-6
interaction handler 7-1, 7-7
  complex 7-10

K
key
  event request 8-15

L
input 8-5, 8-21
input object action 7-7

L
layout view 4-2, 4-13, 9-8
  input object template 7-8
line
  attribute 5-9
  object 5-9
linking 9-2
  object files 9-7
  Tproto 9-3
  UNIX 9-2
  VMS 9-4
lint 9-9
location object 8-2, 8-7, 8-21
  button 8-21
  coordinates 8-21
  creation 8-38
  graph feedback 6-15
  key 8-21
  polling 8-5

M
Makefile 9-11
mapping variables 6-29
mask
  event 8-7
  image and icon 5-20
menu 7-8
monochrome 4-7
Motif widgets 7-8
motion 8-14
mouse 8-5 see also location object
  button 8-21
  event request 8-15
  multiplexor 7-8, 7-10

N
naming
  objects 2-6
  naming convention
    routines 3-13
  node object 5-2, 5-28
Index

O
object 5-1
  color 5-23
  data group 6-2, 6-10
  data structure 3-10
  deque 5-27
  display 4-12
  drawing 5-1, 5-26
  dynamic control 6-2, 6-22
  dynamics 6-22
  edge 5-28
  graphical 5-1, 5-9
    arc 5-10
    attribute 5-6
    block diagram 3-2
    circle 5-9
    control point 5-4
    ellipse 5-10
    icon 5-16
    image 5-16
    input object 7-5
    line 5-9
    polygon 5-11
    rectangle 5-10
    subdrawing 5-15
    text 5-12
    vector text 5-12
  heap 5-4
  input technique 7-1, 7-6
  list 5-27
  location 8-7
  node 5-28
  non-graphical 5-1, 5-26
  pixmap 5-16
  point 5-21
  reference count 5-5
  rule 5-35
  screen 4-5
  selected 8-21
  slotkey 5-29
  subobject 5-3
  threshold table 6-23, 6-25
  transform 5-34
  type 5-3
  variable descriptor 6-23, 6-25
  object files 9-7

obscurring 4-8, 4-14
OPEN LOOK widgets 7-8

P
palette 7-8
  pixmap object 5-16
  planning 2-5
  point object 5-1, 5-21
  absolute 5-22
  control point 5-4
  relative 5-22
  polling
    location 8-14
    motion 8-14
  polygon object 5-11
  private type 3-10
  program
    buffer 6-10
    coding 9-8
    compiling 9-2
    control flow 2-10
    linking 9-2
    optimizing 9-11
    skeleton 3-14
    structure 2-9
  prototyping
    rule object 5-35
    public type 3-10

R
range 6-9
rebinding 6-9
rectangle object 5-10
reference count
  data source list 6-37
  object 5-5
 referenced
  color 5-25
  resize event 8-21
  resolving mappings 6-30
  RGB 5-24
routines 3-11
  drawing 5-27
  naming convention 3-13
  object manipulation 5-3
  traversal 9-8
rubberbanding 4-7
rule object 5-35

S
screen 4-1, 4-5
attribute 4-6
color 4-7, 5-25
coordinates 4-4
event mask 8-8
multiple 4-7
  event gathering 8-6
  selected 8-21
viewport 4-1
scripts 1-5
scrollbar 7-8
service routine 8-20
  result 8-28
slider 7-8
slotkey object 5-1, 5-29
structure
  block diagram 3-2
  data structures diagram 3-7, 3-9
stuffing the buffer 6-7
subdrawing 5-15
  dynamics 6-22, 6-27
  internal dynamics 6-28
subobject 5-3
  of graphical objects 5-21
  referenced 5-5
  selected 8-22
summary
  by chapter 1-6

T
TdlCloseData 6-4
TdlForEachDataSource 6-5
TdlForEachVar 6-5
TdlGetNamedDataSource 6-6
TdlOpenData 4-17, 4-18, 6-3, 6-4, 6-36
TdlReadData 4-18, 6-3, 6-4, 6-5, 6-7, 6-36, 8-41, 8-42
TdlReadData 6-36
TdpBack 4-14
TdpCreate 4-3, 4-9, 4-10
TdpCreate 4-10
TdpCreateStretch 4-9, 4-10, 4-11, 4-13
TdpCreateStretch 4-10
TdpDestroy 3-14, 4-3
TdpDraw 4-3, 4-7, 4-11, 4-12, 4-16, 7-6, 8-32, 8-36
TdpDrawNext 4-3, 4-7, 4-11, 4-12, 4-18, 6-3, 6-4, 6-27, 8-32, 8-41
TdpDrawNext 5-27, 6-38
TdpDrawNext 6-3
TdpDrawNextObject 4-7, 4-12, 5-27, 8-32
TdpDrawNextObject 4-11
TdpDrawObject 4-7, 4-12, 4-18, 5-3, 5-4, 5-7, 5-23, 7-4, 7-6, 7-9, 7-12, 8-32
TdpDrawObject 4-12
TdpErase 4-7, 4-11, 8-33
TdpEraseObject 4-7, 5-3, 5-23, 7-5, 7-12, 8-33
TdpEraseObject 7-6
TdpForEachDrawport 9-8
TdpFront 4-14
TdpGetScreenVp 4-15
TdpGetScreenVp 9-9
TdpGetView 4-16, 4-17
TdpGetXform 8-39
TdpGetXform 8-39
TdpMoveTo 4-16
TdpMoveTo 4-16
TdpRedraw 4-11, 4-12, 4-16, 8-32, 8-33, 8-36
TdpRedraw 4-18
TdpRedrawObject 4-12
TdpResize 4-16, 9-9
TdpScreenToWorld 4-3, 4-16, 5-23
TdpScreenToWorld 5-23
TdpWorldToObject 4-3
TdpWorldToObject 4-16
TdpZoom 4-16
TdpZoomTo 4-16
TdrForEachNamedObject 5-27, 6-31, 6-32
TdrGetNamedObject 4-13, 5-27, 7-3
TdsAddDsVar 6-6
TdsAddDsVar 6-6
TdsCreate 6-6
TdsDeleteDsVar 6-6
TdsEditAttributes 6-6
TdsForEachVar 6-6
TdsGetAttributes 6-6
TdsGetNamedDsVar 6-7
TdsReadData 6-6, 6-7
TdsAttachVdp 6-7
TdsCreate 6-7
Tds_detachVdp 6-7
Index

TdvEditAttributes 6-7
TdvGetAttributes 6-7
TdvGetBuffer 6-7, 7-3
TdvGetName 6-31, 6-33
TdvSetGlobalFlag 6-29
TdvSetValue 6-7
text
  entry 7-8
  hardware 5-12
  list 7-7
  vector 5-12
text dynamics 6-22
threshold table object 6-2, 6-23, 6-25
TInit 3-14, 4-6, 5-13
TInit 3-15
TloGetSelectedDrawport 8-21
TloGetSelectedObject 4-12, 5-3, 8-21, 8-22
TloGetSelectedObjectName 8-22, 8-35, 8-36
TloGetSelectedSubObject 8-22
TloGetSelectedSubObjectName 8-22
TloSetup 8-38, 8-39, 8-40
TloWinEventSetup 8-38, 8-39
TobForEachSubobject 5-27
TobForEachVdp 6-31, 6-32
toggle 7-8
Tproto 9-3
transform 5-1, 5-34
  color 5-20
  drawport 4-2, 4-8, 4-16
transformation dynamics 6-22
traversal routines 9-8
TscClose 4-3
TscCloseCurrentScreen 3-14
TscDefBackcolor 4-7
TscDefForecolor 4-7
TscErase 4-11
TscOpenSet 3-15, 4-2, 4-6, 4-7, 5-20, 5-25
TscRedraw 4-11, 4-12, 4-14, 4-15, 8-32, 8-33
TscReset 4-11, 8-36
TscSetCurrentScreen 4-7, 8-22, 8-35
TTerminate 3-14
TTerminate 3-15
TvdGetDataSourceVariable 6-7, 6-31, 6-33, 7-3
TvdPutBuffer 6-10, 6-32, 6-33
TviASCIISave 4-19
TviCloseData 6-4, 6-29
TviCreate 6-5
TviDestroy 4-3, 7-4, 7-11
TviExciseDrawing 4-18
TviGetDataSourceList 4-17, 6-5, 6-36, 6-37
TviGetDrawing 4-13, 4-16, 4-17, 6-31, 7-4, 7-11
TviLoad 3-14, 4-2, 4-13, 4-17, 6-31, 7-4, 7-11
TviMergeAddDataSources 6-36, 6-37
TviMergeDataSources 6-37
TviMergeDataSources 6-37
TviMergeDrawing 4-18
TviOpenData 4-3, 4-17, 6-3, 6-4, 6-29
TviReadData 4-3, 4-18, 6-3, 6-4, 6-5, 6-7, 6-29, 8-41, 8-42
TviSave 4-19, 5-16
U
  updating
dynamics 6-38
V
  validation 1-5
value see also buffer, data
  list 7-7
variable descriptor 6-1, 6-8
  bound to data source variable 6-6
  object 6-2, 6-23, 6-25
rebinding 6-9
VDbar 6-16
VDline 6-16, 6-18
VDlinefill 6-16
VDn_radial 6-16
VDpig 6-16, 6-19
VDradial 6-16, 6-20
VDstep 6-21
vector text object 5-12
  fit 5-14
  fonts 1-5
VGDgvd 6-8, 6-17
view 4-1, 4-16
display 4-11
drawing viewport 4-1
dynamics 4-17
  saving 4-19
virtual coordinates 4-4

Index - 6  DV-Tools User's Guide  FANUC CNC Parts
virtual coordinates 4-2
visibility 4-8, 4-14
visibility dynamics 6-22
VNcombiner 7-10
VNmultiplexor 7-10
VOciAtGet 5-3
VOciCreate 5-9
VOciTraverse 9-8
VOcoCreate 5-2, 5-24, 5-25
VOdbgCounts 5-4
VOdbgOb 9-11
VOdebug 9-9, 9-11
VOdgGetDgp 6-8, 6-10
VOdrBackcolor 4-16
VOdrGetName 9-8
VOdrOffcolor 4-10
VOdrTraverse 5-27
VOdyAttachData 6-23, 6-24, 6-27
VOdyCreate 6-23, 6-24, 6-27
VOdySetState 6-24
VOelCreate 5-10
VOicCreate 5-18
VOimCreate 5-18
VOimScalePixmap 5-18
VOinCreate 7-4, 7-5, 7-11
VOinDereference 7-5, 7-12
VOinGetInternal 7-9
VOinGetVarList 6-8, 7-3
VOinPutFlag 7-4, 7-5, 7-6, 7-11
VOinPutVarList 7-4, 7-5, 7-11, 7-12
VOinState 7-6
VOinTechnique 7-4, 7-5, 7-9, 7-12
VOitCreate 7-4, 7-11
VOitCreate 7-6
VOitGetTemplate 7-9
VOitKeyOrigin 7-4, 7-12
VOitKeyOrigin 7-7
VOitPutEchoFunction 8-4, 8-31
VOitPutEchoFunction 7-7
VOitPutKeys 7-4, 7-7, 7-12
VOitPutList 7-4, 7-7, 7-11, 7-12
VOitPutListValues 7-4, 7-12
VOitPutTemplate 7-9
VOLnCreate 5-9
VOloButton 8-21, 8-35, 8-36
VOloCreate 8-38
VOloKey 8-21, 8-40
VOloKeyString 8-21
VOloKeySym 8-21, 8-35, 8-36
VOloMaxPoint 8-21
VOloRegion 8-21, 8-35
VOloScpGet 6-15, 8-21
VOloScreen 8-21, 8-22, 8-35
VOloState 8-21
VOloType 8-21, 8-35, 8-36
VOloWcpGet 8-21
VOloWinEventGet 8-21, 8-37, 8-38
VOloWinEventPoll 7-5, 7-12, 8-4, 8-5, 8-6, 8-21, 8-35, 8-38
VOloAtSet 5-29
VOloPtSet 5-29
VOobAtGet 5-6, 5-7
VOobAtSet 5-6, 5-7
VOobBox 4-11, 4-13, 5-14, 8-39
VOobDereference 5-5
VOobDySet 6-24, 6-27
VOobDySet 6-23, 6-25
VOobGetSlot 5-31
VOobPtGet 5-4, 5-11, 5-35
VOobPtSet 5-11
VOobRefCount 5-5
VOobSetSlot 5-30, 5-31
VOobSlotUtil 5-30
VOobTraverse 9-8
VOobType 5-3, 6-31
VOpmBestColors 5-20
VOpmCreate 5-18
VOpmNewColorTable 5-20
VOptCreate 5-2, 5-9, 5-22, 5-23, 6-24
VOptGet 5-23, 5-35, 9-9
VOptMove 5-4, 5-23, 5-35
VOptXfGet 5-23
VOpyCreate 5-11
VOpyPtAdd 5-11
VOpyPtDelete 5-11
VOreCreate 5-3, 5-10
VoscWinEventMask 8-4, 8-8, 8-37
VoscWinEventPoll 8-6, 8-21
VosdDrGet 5-15
VosdGetDynamicFlag 6-28
VosdPoolFnmRemove 5-16
VosdPoolRemove 5-16, 6-30, 6-31, 6-32
Index

VOsdSetDsvMapping 6-30
VOsdSetDynamicFlag 6-28
VOsdViGet 6-37
VOsdViKeep 5-15
VOskDeclare 5-30, 5-31
VOTAddThresh 6-27
VOTTCreate 6-27
VOTTDataGet 6-27
VOTTLastGet 6-27
VOTTReset 6-27
VOTTUpdate 6-27
VOTTCreate 5-12
VOTTGetString 5-12
VOTTSetString 5-12
VOuAtInit 5-2, 5-7
VOuAttr 5-7
VOvdGetVdp 6-8
VOptCreate 5-13, 5-15
VOPTFitRect 5-14
VOPTGetBound 5-14
VOPTGetString 5-14
VOPTSetString 5-14
VOxFCreate 5-35
VOxFPoint 5-35, 8-39
VOxFSCreate 5-34
VPdgaxlabel 6-13
VPdgcontext 6-13
VPdgcreate 6-11
VPdgdf 6-12
VPdgdfquery 6-15, 6-17
VPdgsscroll_amount 6-13
VPdgticlabfcn 6-13
VPvdcreate 7-4
VUdbgalloc 9-10, 9-11
VUdbgalloc 9-11
VUdebug 9-10, 9-11
VUrActivate 8-19
VUrActivateClient 8-19
VUrBoundaryEventPost 8-4, 8-12, 8-13, 8-14, 8-15, 8-17
VUrCatchAllEventPost 8-12, 8-38
VUrClearAll 8-5, 8-19
VUrDeactivate 8-19
VUrDeactivateClient 8-19
VUrHandleLocEvent 7-5, 7-10, 7-12, 8-5, 8-9, 8-18, 8-32
VUrObjectEdgePost 8-26
VUrPutKeys 7-7
VUrRectEdgePost 8-26
VUrServiceResultPost 7-3, 8-4, 8-28, 8-29, 8-32
VUrWinEventPost 8-4, 8-12, 8-16, 8-17

W

widget 7-1
event 8-37
window see also screen, WINEVENT
WINEVENT 8-22
event request posting 8-16
world coordinates 4-2, 4-4