DesignVision
(DVapp)
Users Guide

March 17, 2023

www.trifox.com

Trademarks

Copyright

Printed in the U.S.A.

Contents

Preface

This document is a guide to the screens, syntax, datatypes, built-in functions, and conventions that you need to know to work successfully with DesignVision.

It provides an overview of the DesignVision framework and architecture, as well as specific sections that guide you through application and report development and customization.

While this document does not assume that you are a database expert, DesignVision is typically used to develop applications that access databases and you must be familiar with any database-brand specific issues.

Organization

This document is divided into the following chapters:

- ***Chapter 1, DesignVision Overview***, provides an overview of the component suite and the commands for several utilities.
- ***Chapter 2, Installing DesignVision,*** describes how to install the software on Windows, OpenVMS, and Unix.
- ***Chapter 3, Application Development***, discusses elements of application design such as window and object relationships, actions, window elements, triggers, and how you can use these components to create effective applications.
- ***Chapter 4, Creating Forms***, is a tutorial.
- ***Chapter 5, Writing in TRIMpl***, describes the language syntax, itemizes programming rules, and provides some examples of code. This chapter is written for programmers and assumes some programming knowledge and experience.
- ***Chapter 6, Variables and Triggers,*** lists the TRIMpl variables and rules and parameters for using them.
- ***Chapter 7, Datatypes***, lists the datatypes and rules and parameters for using them.
- ***Chapter 8, Working with Lists***, discusses lists, one of the elements that makes TRIMpl such a powerful and flexible database programming tool. This chapter is written specifically for programmers.
- ***Chapter 9, Using TRIMrpc***, describes how to use the remote procedure call (RPC) function to access stand-alone `.run` files from anywhere on your network.
- ***Chapter 10, Debugging & Compiling***, gives examples and procedures, as well as tips, on debugging your TRIMpl code. This chapter is also written specifically for programmers.

Appendixes detail the database dictionary, the DesignVision file structure, and discuss issues that arise when migrating from character-based applications to windows-based ones.

Background

Trifox Inc. has been serving the relational database market since 1984 through consulting and the development of software products. In 1987, Trifox created SQL*QMX for Oracle. This easy-to-use, powerful querying and report writing tool, which is based on IBM's QMF, continues to be used at thousands of sites. In 1989, Trifox created TRIMtools, a family of application and reportwriting tools now known as DesignVision. DesignVision was developed in response to the OLTP requirements of several large application vendors.

Database Access

VORTEX is an integrated family of products that allows nearly any production application to access SQL data:

- On any or all of the major relational databases.
- Across networks.
- Across platforms.
- With a dramatic increase in the number of concurrent users.
- Without any additional hardware.

In a client/server or multi-tier configuration, VORTEX makes it possible for your SQL applications to access data on different platforms over one or more network configurations. Currently it supports only TCP/IP.

Inherent in this approach are services that allow production applications originally written for one relational database (such as Oracle) can access the same data on another database (such as Informix), even if it is spread across different databases.

VORTEX Precompilers for C and COBOL, as well as a variety of program interfaces, allow existing SQL programs to take full advantage of VORTEX services such as performance enhancement, transaction monitoring, and flat-file database access.

With VORTEXaccelerator in your configuration, you dramatically increase the number of concurrent users who can log on to a specific SQL production application. Your users experience faster performance and you won't have to change any programs or add any hardware.

Application and Report Development

DesignVision DVapp lets you design, generate, and maintain forms-based applications. You can easily port the pop-up windows, customizable menus and submenus, and custom keyboard assignments, in fact the entire application, to Windows .NET, Unix, OpenVMS, or HTML5 with no extra effort.

The reportwriter, TRIMreport, lets you create simple reports quickly, or complex reports with absolute confidence in their power.

When you want to write stand-alone applications (including triggers) without a user interface, the TRIMpl 4GL language gives you the freedom you want. The procedural language has over 100 database-specific functions that help you write powerful applications in very little time.

Reaching Legacy Data

GENESISsql is a SQL processor that accesses low-level data sources such as ISAM, SDMS, ADABAS, RMS, and MicroFocus and makes the data accessible to VORTEX clients. You can add GENESIS data sources to a VORTEX system in a matter of days, simplifying what used to be an enormous task.

Conventions

Screen shots in this manual come from the Windows version of our software.

Trifox documentation uses the following conventions for communicating information:

Example	Describes
`CHOOSE REPORT > [F3] >`	Press [F3] on the CHOOSE REPORT menu and ...
Right-click	Clicking the right mouse button.
Left-click	Clicking the left mouse button.
`connect_string`	Replace italicized text with your own variable.
`vtxnetd`	Text in bold typewriter style represents strings that you type exactly as they appear in the manual.

Support

If you have a question about a TRIFOX product that is not answered in the documentation (paper or online), contact the Customer Support Services group at:

- support@trifox.com
- Trifox Customer Support Services
  2959 Winchester Boulevard
  Campbell, CA 95008
  U.S.A.
- 408-796-1590

Revisions

September 1999

First production version of manual.

November 1999

Beginning page 82, added information for two new graphic lists: graphic_type_poly and graphic_type_bezier as well as providing examples for most other types of graphic lists.

December 1999

Updated documentation on the fast application builder, DVfast, to reflect changes in product.

Corrected errors in graphics list explanation.

Added documentation for new `list_open()` option, ***dir!***. Changed all documentation to reflect change in filenames from gui.* and gmc.* to dv.* in 5.1.0.0.1. Corrected miscellanous errors.

January 2000

Updated information about Graphic Lists, including more intuitive element names and enhanced information about graphic list ID elements.

May 2000

Added information about TRIMpl automatic datatype conversions in expressions (“***Datatype Conversions***” on page 79).

Added a general section on the various ways to specify filenames (“***Filename Specifications***” on page 58).

Modified the general section on various ways to specify filenames and URLs. Moved the information about dir! to the same section, so it now covers local, `gui!`, `vortex!`, `net!`, and `dir!`. (“***Filename Specifications***” on page 58).

November 2000

Added group syntax in TRIM / PL chapter.

August 2001

Added XML information to List chapter.

Oct ober 2009

Added local variable keyword in TRIM / PL chapter.

February 2012

Updated the predefined events.

November 2012

Added graphic_type_popup.

September 2015

Removed trimgen -t option.

January 2020

New trimmir options.

February 2021

Added trimgen -t option.

November 2022

Added glob datatype.

March 2023

Added symbolic text.

Chapter 1
DesignVision Overview

Trifox's DesignVision helps you develop applications and reports that are optimized for database access. The collection of design and development components that comprise DesignVision allow you to rapidly develop applications, customize them on the fly, and create extensions quickly and easily. You control not only the application designs themselves, but also the design process, to meet your organization's exact needs.

If you are a programmer, the data dictionary provides increased automation. You can continue to build function libraries using any editor and the built-in functions, taking advantage of DesignVision's exceptional attention to re-use.

For user interface experts, using one of the DesignVision designers lets you build and modify prototypes quickly.

When everyone is satisfied with the designs, you compile and debug application files. Trifox supplies several tools to help evaluate the application code for both the reports and applications.

The end result of the process is machine-independent binary code with data references that are offset-based and easy to relocate. You can compile the application file code on one machine and execute it on other types of machines, including to and from 32-bit /64-bit systems.

Your end users see and use the application combined with a runtime module that is specific to their machines. You develop once and can deploy on any or all of Windows, Windows .NET, or HTML5 display platforms.

Using DesignVision for applications, report writing, and stand-alone modules, lets you harness the true power of client/server application development and take advantage of the flexibility provided by database independence.

Designing Applications — DVapp

You use DVapp to create applications of one or more windows. A window is a self-contained sub-application that can include text and fields. In a conventional data-entry application, a window represents an underlying database table.

You can use DVapp with all its defaults to get a simple application up and running, or you can build your own definitions of windows, assign actions to fields in an application, and file them in the default library or data dictionary for future use.

DVapp lets you control the internal processing of the application, such as interaction with the database, which is especially important in a large user environment. These applications retrieve data from the database and put it in a list in memory where the rich set of list functions allow you to work with the data in the exact ways your users need. When all modifications are completed, other functions commit the data to the database.

Writing Reports — TRIMreport

TRIMreport allows you to create reports from data stored in your database. It consists of three main parts, all of which operate independent from the database:

- Designer
- Generator
- Runtime

The designer creates ASCII files that allow portability across platforms. The generator creates a binary file for a particular operating environment and executes the report.

TRIMpl

Both DVapp and DVreport use a language similar to C to define the actions you specify. When you create your own actions, or modify defaults, you also use this language, which is called TRIMpl. TRIMpl includes over 100 functions that handle lists in memory, display management, and interactions with relational databases.

You can create a block of TRIMpl code in a separate file as a stand-alone application, which is useful for writing utilities that load databases tables from ASCII files and manipulate data in databases. A more complicated example can be found at www.trifox.com/trimpl/index.html. Other applications can call the stand-alone piece as an external function that can receive parameters and return values.

TRIMpl functions are described at www.trifox.com/trimpl/index.html.

Final Components

The final pieces to the DesignVision environment allow you to compile, test, and run the applications and reports that you've created.

DVfast

DVfast uses the data dictionary (DVdd) to build applications. For each DVfast application name you specify, DVfast uses the information in the DVdd TDD_FAST table to build an application.

dvfast *db_login appname* [appname …] [*options*]

Options

-b	Use button groups.
-c*x,y*	Coordinates of the first window let you place your application on the user's screen. The default is 0,0.
-g	Use grid windows.
-l*language*	Specifies the application's language. The default is ENG.
-w*x,y*	Window size. The default is 24,80.

DVmenu

DVmenu helps you manage window menu definitions.

Window menu definitions are stored in the DesignVision data dictionary table called `TDD_MENU` but are rather complicated to manage. DVmenu simplifies this task by providing a graphical layout of the menu. You simply point to the menu item you want to modify or where you want to add sibling or child menus. You can also modify the trigger that is associated with the menu item.

You can run DVmenu as a stand-alone application:

DVmenu *db_login* [*options*]

You can also call it from DVapp by choosing `Menu->Edit` from the designer menu bar.

If you run DVmenu from DVapp, the optional parameters are inherited from the application.

Options

-t*set*	Specifies the trigger set to use. The default is STANDARD.
-a*application*	Specifies the application to use. The default is TDD.
-l*language*	Specifies the language to use. The default is ENG.
-p*project*	Specifies the project to use. The default is TDD.

TRIMgen

TRIMgen compiles TRIMpl and DesignVision design files. You run the resulting binary files with TRIMrun.

trimgen *design_file* [*design_file* …] [*options*]

You can compile any number of design files at a time simply by listing them all, but they all use the same set of options per command.

Options

-a	Compiles stand-alone TRIMpl file (not created with DVapp or DVreport).
-c2	Unicode (2 byte) chars.
-D*n=v*	Define n=v
-f*utrig*	Activates database function logging. *utrig* is the user trigger or routine called by the logging mechanism. The first parameter is the function name. The rest of the function parameters, if any, follow. *utrig* is always called before the function is invoked. The logged functions are exec_row(), exec_sql(), exec_proc(), commit() and rollback().
-g	Inserts DesignVision debugger code into the binary file. Defines the DEBUG conditional compile symbol.
-i	Creates a `.mir` (machine independent run) file. This ASCII file can be ported to any machine and changed to `.run` format by running TRIMmir.
-l*file*	Uses specified library file in place of `trim.fnc` or `dv.fnc`.
-m	Display and display the modified triggers warning.
-n	Disable automatic variable declaration.
-o	Enhanced optimization.
-p	Inserts DV profiling code into the binary file. Profiling results are written to *design_file*.prf.
-r	Print the filename(s) opened by TRIMgen.
-s*[file]*	Inserts the symbol table into the binary file. Required for `name_in()`. Also writes symbol table to optional file, `file`.
-t	Inserts tracking code into the run file. When an error occurs, the text "[<trigger name>, line: <line #>]" is appended to the error text.
-u	Causes TRIMgen to create a windows application, rather than the default character-based one. Defines the GUI conditional compile symbol.
-w	Displays all unreferenced variables and unreferenced user triggers.

TRIMlib

This tools speeds up the trimgen process by converting one or more function libraries into an indexed binary file.

trimlib *library_file* [*-l* | [*CAR file ...*]]

Options

-l Displays the current contents of *library_file*.

TRIMlis

Diagnostic and reporting tool for DesignVision applications and reports.

trimlis *design_file* [*list_file*] [*options*]

The default *list_file* is *design_file*.lis.

Options

-c<str> Database connection string to access the dictionary.

-f Displays only FIELD objects.

-m Displays messages in addition to writing them to *list_file*.

-n Displays trigger text without line numbers.

-o Displays objects only (WINDOW/BLOCK, FIELD, TRIGGER).

-s Adds WINDOW/BLOCK begin/end separator text so `diff` can more easily resync.

-t Displays only trigger text.

TRIMlsr

This tool enhances the performance for production systems by loading all specified run files into shared memory. Use it only when your application is complete.

trimlsr [*run_file* …] [*options*]

TRIMlsr has one option:

-l*filename* Load all `.run` files listed in *filename*.

TRIMmap

This diagnostic and reporting tool for TRIMreport shows how report blocks are connected.

trimmap *design_file* [*map_file*]

TRIMmir

Converts files from one format to another.

trimmir source destination [*option*]

If no option is specified, then source is assumed to be a mir (Machine Independent Runfile) file and destination will be a run file.

-i Convert .run file to .mir file. Same as -r2m.

-S2D Convert S(ource) to D(estination).
S and D: g (gap), j (json), m (mir), r (run), x (xml)

The only restriction is that the source and destination file types cannot be the same.

TRIMrun

TRIMrun executes compiled code, which makes the applications, reports, and stand-alone applications interactive. It is also called the "runtime."

trimrun *run_file* [[*db_login*] [*out_file*]][*options*]

The `run_file` must have a `.run` file name extension.

You must specify `db_login` only if the user is going to interact with a database. You can also specify the database connection in the application.

Specify an `out_file` with report designs to direct output to the file name given. This example command creates a report file named `myrep.out`:

```
trimrun myreport myname/mysecret myrep
```

Options

-d*n*	Displays debug error number *n*. Use **-d** with no error number to see all errors.
-f*string*	Specifies a string to use for the form feed. The default is a universal form feed command. To specify "no form feed" use the switch with no string argument.
-k*filename*	Inputs (plays back) keys from *filename*.
-m	Displays memory usage (number of bytes allocated).
-o*filename*	Outputs (records) keys to file *filename*.
-t*filename*	Shows execution trace where *filename* is output file; if no *filename* is specified, trace goes to `stdout`.
-p *parms*	This option, which must be listed last, lets you pass information to the application on the command line. You must put this option at the end of the command followed by all the parameters (everything following this option is assumed to be a parameter). This example passes the string "`myname`" into `myapp.app`: `trimrun myapp -p myname`

NOTE: *Attempting screen I/O results in errors.*

TRIMsrt

TRIMsrt is used to create standalone executable programs with the TRIMgen runtree output embeded in the executable file.

```
trimsrt <exec file> <RUN file> <new exec file> [-p <parm>...]
```

The <exec file> is one of trimrun.10k, trimrun.50k, or trimrun.100k. The values indicate the maximum size runtree that can be stored in the executable.

VORTEXsql

This utility executes SQL statements that you can either type in from the command line or have read from a file. VORTEXsql does not perform any syntax checking. The statements are assumed to be valid SQL statements for the target database. Currently this utility only performs *function remapping*.

vtxsql [*options*]

Options

/c*connect_string*	Connects to a database.
/x c | r[u]	Commits or rolls back transaction and, if you specify "u," start a read/write transaction.
/d*table*	Describes a table.
/r*filename*	Runs statements/commands in file *filename*. Files can be chained, but not nested.
/o*cmd parms*	Executes a driver command (*cmd*) using the optional parameters you specify. Look for cmd values in vortex.h
/m *mask*	Sets datetime format mask (default: MM/DD/YY).
/n*null*	Specifies what to display for NULL values.
/vy|n	Verbose messages. If you specify "n", then only errors are reported.
/b	Displays database request block.
/d	Displays database request block.
/? or **/h**	Displays help for this screen.
/q	Quits session and exits.

Chapter 2
Installing DesignVision

DesignVision for OpenVMS

The DV for OpenVMS files are distributed in `saveset` format from the Trifox ftp site, ftp.trifox.com. The following procedure describes how to install and setup DesignVision on your client machine after downloading them from the ftp site.

☞ *Installing on OpenVMS*

1. Prepare for installation.

   Create a home directory for the software. To create `trifox` in `DKA100:[USR]`, type

   ```
   create/dir dka100:[usr.trifox]
   ```

2. Install the software.

   If you received the media on a DAT tape or TK50 and the device is mka500:

   ```
   set def dka100:[usr.trifox]
   mount/for mka500:
   backup/log mka500:install.bck/save []/new
   dismount mka500:
   ```

☞ *Customizing files and environment*

`LOGIN.VTX` is file that contains a series of logical definitions and symbol assignments. To update the file for your installation, begin by executing it inside the `SYS$LOGIN:LOGIN.COM` file, as well as the `LOGIN.COM` files of your users.

1. Replace each `xxxx:[yyyy]` with the actual DV module location (the device and directory name) in the following lines

   ```
   $ ASSIGN/GROUP "xxxx:[yyyy]"      TRIM_HOME
   $ ASSIGN/GROUP "xxxx:[yyyy.obj]" TRIM_CHNL
   $ ASSIGN/GROUP "xxxx:[yyyy.bin]" TRIM_EXEC
   ```

   For example,

   ```
   $ ASSIGN/GROUP "DKA100:[usr.trifox]" TRIM_HOME
   ```

2. If you are using VORTEXaccelerator, replace `xxxx:[yyyy]` with the actual `TRIM_HOME` device and directory name in the line

   ```
   $ ASSIGN/GROUP "xxxx:[yyyy]share.file   TRIM_SHM_FILE
   ```

3. Replace XXXX with your VORTEXaccelerator Group in the line

   ```
   $ASSIGN/GROUP   "XXXXX"          TRIM_MUX_NAME
   ```

4. Replace XXXXX with the name of your system editor in

   ```
   $ ASSIGN XXXXX EDITOR
   ```

5. Activate the changes you've made by entering the command:

   ```
   $@LOGIN.VTX
   ```

6. Issue the command to create the necessary subdirectories and move all the files to the appropriate locations.

   ```
   $@INSTALL.COM
   ```

DesignVision for Unix

The DV for Unix files are distributed in compressed (gzip) `.tar` format from the Trifox FTP site, `ftp.trifox.com`. The following procedure describes how to install and set up DesignVision on your client machine.

The following instructions are for the evaluation version of DesignVision. If you are downloading and installing a licensed copy, follow the instructions you received with your license.

☞ *Installing on Unix*

1. Create a home directory for the software. For example, to create `trifox` in `/usr/local`, type

   ```
   mkdir /usr/local/trifox
   chmod 664 /usr/local/trifox
   ```

2. Transfer (ftp) the software from `ftp.trifox.com` into the directory you just created. If you are downloading evaluation software, use `anonymous/email` as the username and password to log in.

   ```
   cd /usr2/local/trifox
   ftp ftp.trifox.com/pub/products
   cd your_target_os
   bin
   get eval.tar.Z
   bye
   ```

3. Unzip the `*.tar` file and use the `tar` command to open the package of files.

   ```
   gzip -d eval.tar.Z
   tar -xf eval.tar
   ```

Customizing the files and environment

1. Specify TRIM_HOME

   All components rely on the `TRIM_HOME` environment variable to find the files they need. The customization instructions assume that you have installed the DV files in the directory `/usr/local/trifox`. Set `TRIM_HOME` in your logon script(s) as well as those of your users.

   For the C shell:

   ```
   setenv TRIM_HOME /usr/local/trifox
   ```

   For all other shells:

   ```
   TRIM_HOME=/usr/local/trifox;export TRIM_HOME
   ```

2. Change your current directory to `$TRIM_HOME/obj` by typing:

   ```
   cd $TRIM_HOME/obj
   ```

3. Customize the makefile, following the directions for Unix platform in the *VORTEX Installation and Operations Manual*.

Chapter 3
Application Development

In 4GL database application development, you create applications, also called *forms*, that let your users store, change, retrieve, and work with information in a database. With DVapp, you can design forms interactively: arranging information on your screen and testing it as you go.

A form consists of windows, which contain fields, buttons, or lists. In simple applications, each window relates to a single table in the database. More complex applications may bring together data from several tables or integrate graphics.

The application *windows* are actually blocks of code that contain the information necessary to control the application:

- SELECT statements
- Text areas
- Fields, buttons, lists, checkboxes
- Triggers

Designing Your Application

You can execute a form at any time to quickly validate your design as you develop. When you are satisfied with the results, you store it in a file, which you (or anyone with permission) can move to any DVapp-supported hardware and software platform.

DVapp is tightly integrated with the DesignVision Data Dictionary (DVdd). This integration makes it simple to maintain a consistent look and feel among applications. For example, you can define the [TAB] key to work the same in all applications. Every time you create a new window, the [TAB] key code is brought in from the DVdd. If you define field behaviors, such as validations in the DVdd, every application that uses that field maintains the same rules for data input.

Data Dictionary

The Data Dictionary (DVdd) is a set of database tables that stores descriptive information about the DesignVision and database objects in your application. Storing screen field, button, and key definitions in the DVdd lets you share these objects throughout your application.

More importantly, however, the DVdd maintains relationships between objects. Foreign keys, for example, are automatically defined by the DVdd tables without you having to do more than specify them.

DesignVision provides several tools to manage the dictionary and to find and clean up any errors. These tools, which are part of the DVdd Console, include:

- Screen forms to ensure that your entries are always consistent.

- A structure checker that examines all the DVdd relationships and verifies that no orphan records exist.
- SQL scripts to move the DVdd from one system to another.

You can see from the entity relationship diagram on the following page that `TDD_CODE` relies on two tables:

- `TDD_CATEGORY`
- `TDD_LANGUAGE`

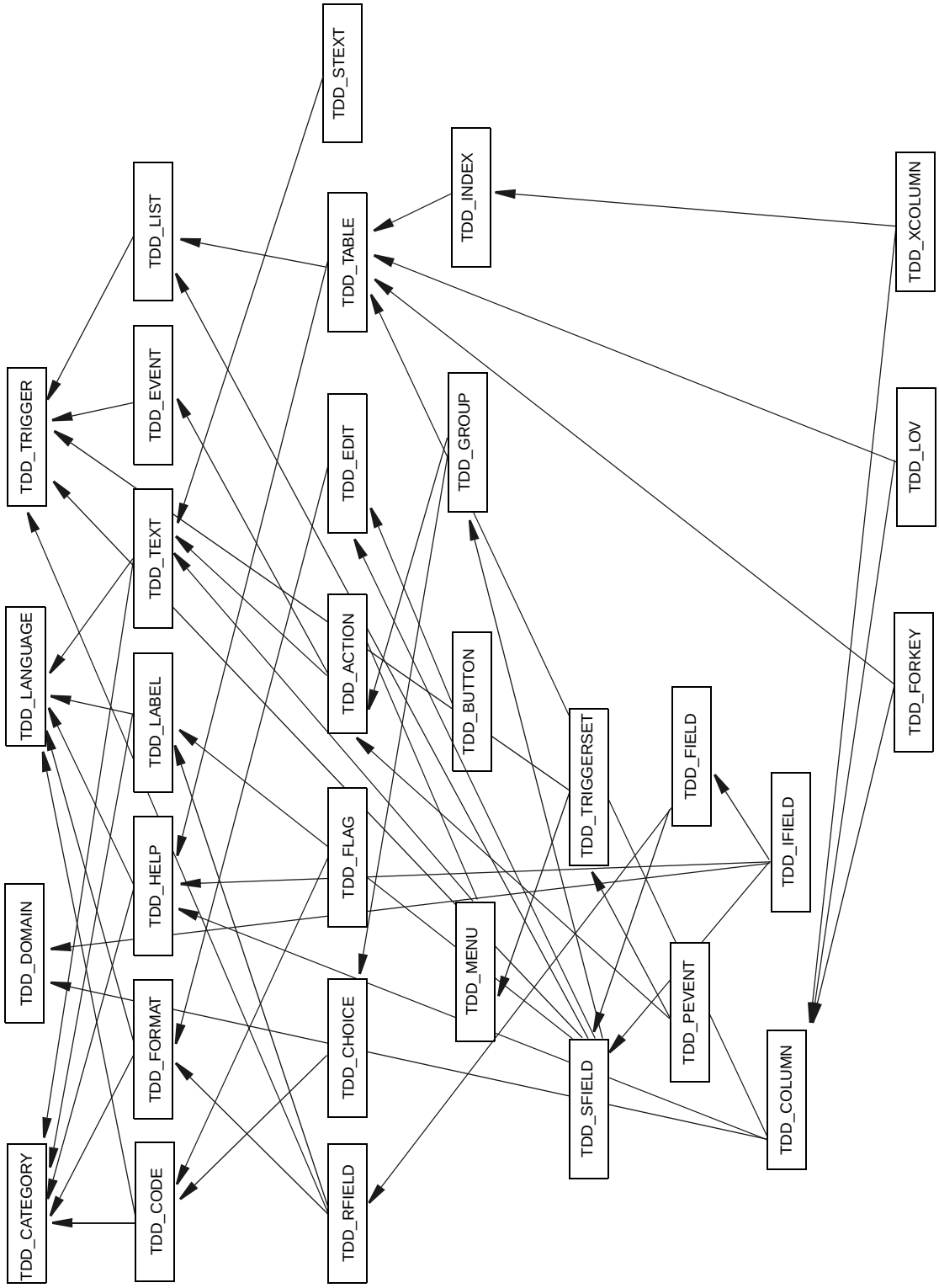
The `TDD_CODE.COD_LAN_NAME` column entry must exist in the `TDD_LANGUAGE.LAN_NAME` column and the `TDD_CODE.COD_CAT_PROJECT` column entry must exist in the `TDD_CATEGORY.CAT_NAME` column.

Using the DVdd Console to insert, update, or delete records ensures that the integrity of these relationships is maintained.

For example, if you attempt to delete a record in `TDD_LANGUAGE` and a record in `TDD_CODE` relies on it, you receive a prompt to edit the `TDD_CODE` table to delete that record. Of course, deleting the record may cause a record in `TDD_CHOICE` to require deletion and so on. In addition, the structure checker identifies unused records so you can delete them easily.

NOTE: *Do not try to use other tools to modify the DVdd. You run the risk of severely damaging the DVdd to the point that it may no longer be usable.*

Data Dictionary Relationships



Entity relationship diagram of the meta data.

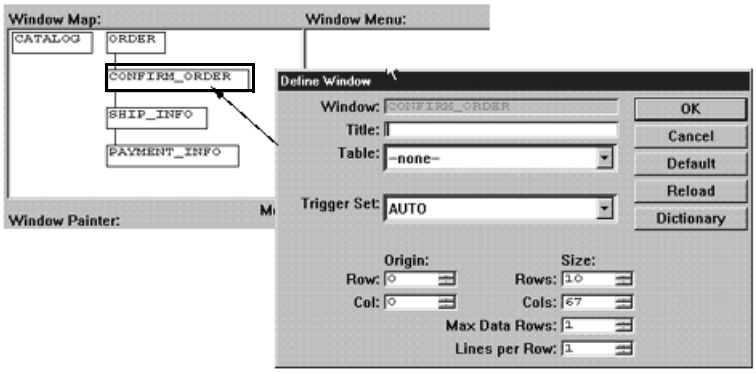
Using DVapp

The rest of this chapter documents the objects and dialog boxes you use to design forms in DVapp. It begins with a list of window attributes that govern appearance and behavior in applications. Then it describes each window element dialog box.

Every window has a comment area for your internal use. There are no restrictions on the contents of the comment area and it has no effect on the runtime behavior of the application.

Window Definition

To see this dialog box, select (with alternate button) the window from the *Window Map* and choose *Define* from the popup list.



This dialog box shows the basic attributes for each window.

Fields

Window Not editable. Name of the window.

Title Text that appears in the window border.

Table Table associated with the window.

Trigger Set The name of the trigger set associated with this window. This selection controls the contents of the Window, Update, Record, Query, and Initialization triggers that are loaded for the window.

Row/Col The row and column values indicate the upper left corner of the window position on the screen. These values are based on the font type that you choose.

Rows/Cols The height (in rows) and width (in characters) indicates the size of the window.

Max Data Rows Number of rows to display at a time. (Useful only for multi-row (record) windows.)

Lines per Row Number of lines to allow for each row. (Useful only for multi-row (record) windows.)

Action Buttons

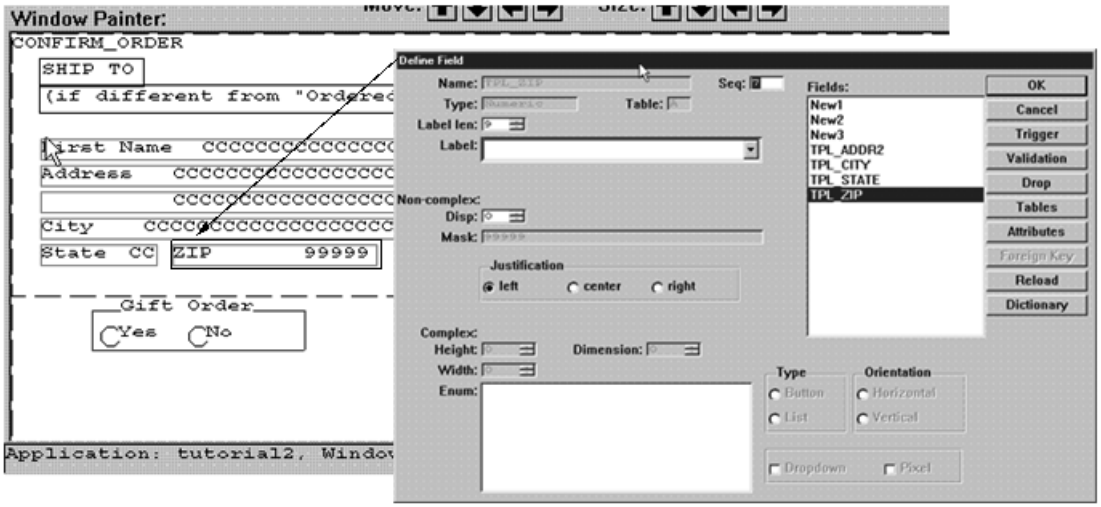
OK	Saves the settings as they appear on the screen.
Cancel	Returns to last-saved settings.
Default	Retrieves the field list for the table from the Console and creates a default field layout in the window.
Reload	Reloads the Window, Query, Initialization, Record, Table, Update triggers as well as the Menu and Select statement.
Dictionary	Activates the DVdd Console.

Window Elements

Windows contain a variety of elements such as fields and buttons, that determine the actions that users can take. In their simplest form, windows let users retrieve, modify, and update the data in one table of the database. More complex windows can handle several tables, switch connections between databases, perform text editing, and perform a variety of other tasks. In this section, we examine all of the elements of a window.

Field

To display the *Describe Field* dialog box, select the field (with alternate button) from the *Window Painter* and *Modify* from the popup menu.



This field was created by selecting all columns for the table TPL_ZIP as the default for the CONFIRM_ORDER window. Justification is typically right, unless you change it and the Label length is the same as the number of characters in the default Label.

The *Define Field* dialog box allows you to modify your application's fields. If, when you define (or create) a window, you choose a table and select the Default action, one field is created for each column in the table. The DVdd supplies all the field description information and you cannot change the information in the disabled (grey) fields.

The field information is divided into three sections: General, non-complex, and complex information. Complex fields are those that display data as radiobuttons, checkboxes, and lists. Non-complex fields are the familiar data entry fields.

Fields

GENERAL

Name Not editable. The column name associated with the field. Two TRIMpl variables are implicitly created for this field: ***Name*** and ***Name*_d**.

Seq This number represents the field's navigation position. When the end user tabs through the fields on the ORDER window, TPL_ZIP is the 7th field. This value reflects the creation order of fields, and you should edit it for greatest end user convenience.

Type Not editable. The field's datatype.

Label len The label's display length.

Label The field's label.

NONCOMPLEX

Disp The display length of the field. If the value is non-zero and less than the actual field length, the field is scrollable.

Mask Not editable. The field mask, from the DVdd.

COMPLEX

Height The height of the complex field's rectangle.

Width The width of the complex field's rectangle.

Dimensions The number of rows or columns to use to layout the field's items. If the Orientation is Horizontal, then this sets the rows; otherwise it sets the columns.

Enum The values to enumerate in the complex field.

Type Button creates a radiobutton or checkbox display. List creates a list display.

Orientation This value only applies to button type fields. Combined with Dimension, it determines the layout, horizontal or vertical.

Dropdown If checked, specifies that List fields are drop-down.

Pixel If checked, converts the height and width values to pixels.

Action Buttons

OK	Saves the settings as they appear on the screen.
Cancel	Returns to last-saved settings.
Trigger	Opens a a window in which you can view or edit the trigger text. See also "***Field Triggers***" on page 52 and "***Designer Field Variables***" on page 69.
Validation	Opens a a window in which you can view or edit the trigger text. See "***Validation Triggers***" on page 52 for more details.
Drop	Drops the field from the window.
Tables	Shows all tables and correlation variables that are associated with fields in this window.
Attributes	Detailed in the following section.
Foreign Key	Detailed in the following subsection.
Reload	Reloads the field's definitions from the DVdd.
Dictionary	Activates the DVdd Console.

There are five categories of field attributes: fixed, dynamic, presentation, font, and user attributes. Fixed attributes can only be modified by changing the DVdd and reloading the field.

Fixed Attributes

Database	Specifies the database field (loaded by the query) that is associated with the column.
Not Null	Cannot be null (used in input mode, which checks for nulls).
Unique	Creates a unique identifier substitute. Use this attribute for all fields that together could create a pointer to a specific row for a SELECT.
List	Specifies that the field is part of a window list and reserves a spot in the `window-name.WL` for data from this field. Specifying database automatically sets this attribute "on".
Protected	Sets the field to be read-only. Prevents users from typing in the field.
No Update	Specifies that value is not tested for a lockrow update, nor is the value written to a database on UPDATE.
No Regen	Protects against a global data dictionary update (regeneration).
Query	Specifies that the field cannot be null when a query is executed. (Query building actions typically use raw input mode, which does not check for nulls.)
User Attributes	Allows you to access and apply user-defined attributes, which are used by field_* operations.
default **system** **fixed** **ansi** **field** **label** **button**	Specifies the appearance of text in your application. The specifications for each font type are saved in WINDOW_DEFAULT_FILE. If this file does not exist, the values inherit the current window settings.

Dynamic Attributes

Fixed	Specifies that user entry must fill the field.
Uppercase	Automatically changes user input to upper case.
No Echo	Does not display user input.
Hidden	Does not display the field, but keeps values available for calculation.
Reset	Clears the field of existing text as soon as users click in the field.
Autoskip	Specifies that cursor moves to next field as soon as user enters data.
Raw Input	Specifies to use raw input (no validation) regardless of the page, window, or form's mode.

Presentation Attributes

No Border Specifies not to draw a box around the field. Used with *transparent* emulates dynamic labels.

Transparent Specifies the same background as the underlying window. Used with *no border* emulates dynamic labels.

Foreign Key dialog box

The only item you can change in the Foreign Key dialog box is the list of value (LOV) sequence. This determines which LOV Select, LOV Where, and LOV OrderBy is used.

Foreign Key The foreign key table and column.

Value Column Additional foreign key column.

Value Additional foreign key value.

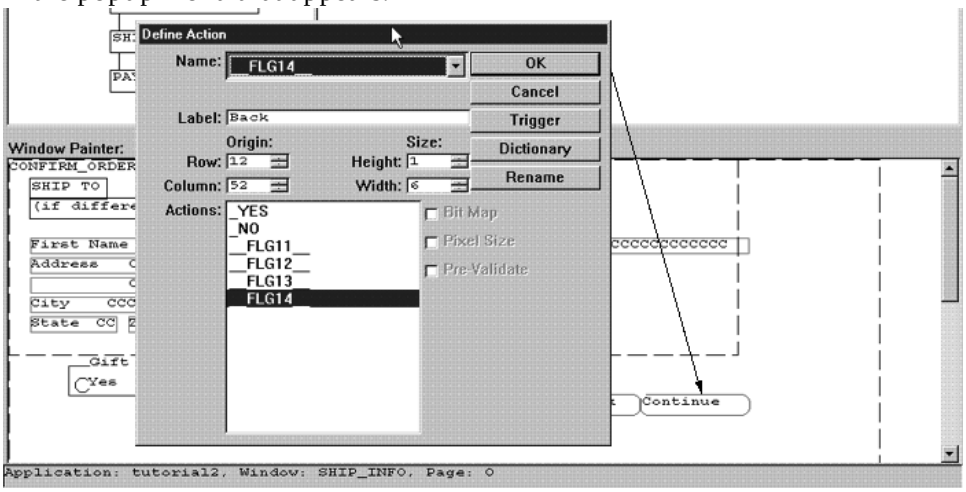
LOV Select the LOV columns.

LOV WHERE The LOV WHERE clause.

LOV OrderBy The LOV Order By clause.

Action

To view an action dialog box, select an object (with the alternate button) and click *Modify* in the popup menu that appears.



Actions can take the shape of simple action buttons, or radio buttons or check boxes.

Fields

Name Not editable.

Label The name that appears on the action button.

Row/ Column These values, which are based on the font type you choose, indicate the upper left corner of the button or checkbox relative to the upper left corner of the owning window.

Height/ Width The height (in rows) and width (in characters) indicates the size of the action item. Typically the width is a few characters wider than the Name, which appears as the item's identifier.

Actions This selection list shows all the actions for a window. You can select and modify any action without having to find, select, and edit the window object itself.

Checkboxes

Bit Map Not editable. If checked, the button displays the bitmap defined in the DVdd for this action.

Pixel Size If an action has a bitmap, converts the height and width values to pixels.

Pre-Validate Not editable. If checked, indicates a pre-validation trigger.

Action Buttons

OK Saves the settings as they appear on the screen.

Cancel Returns to last-saved settings.

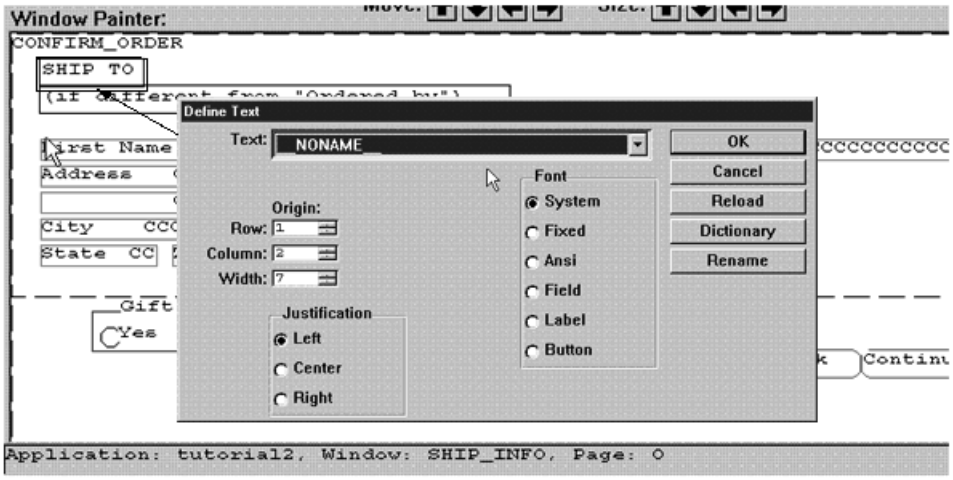
Trigger Opens a a window in which you can view or edit the trigger text. See also "***Field Triggers***" on page 52 and "***Designer Field Variables***" on page 69.

Dictionary Opens the DVdd Console.

Rename Allows you to rename an action button to an existing definition in the DVdd when migrating an existing V4 application to DVapp.

Text

To see this dialog box, select a text object (with alternate mouse button) and choose *Modify*.



Text allows you to create labels, instructions, and other comments to customize the application windows and dialog boxes.

Fields

Text Not editable. Shows the text that appears in the window. Text cannot wrap. You must create a new text item for each line of text in your window.

Row/ Column & Width These values, which are based on the font type you choose, indicate the upper left corner of the button or checkbox relative to the upper left corner of the owning window. Width shows the number of characters that are visible in the text item.

Justification You can justify the text left (the default), center it, or make it right justified.

Font The same fonts are available as for the field items. See "***Fixed Attributes***" on page 26.

Action Buttons

OK Saves the settings as they appear on the screen.

Cancel Returns to last-saved settings.

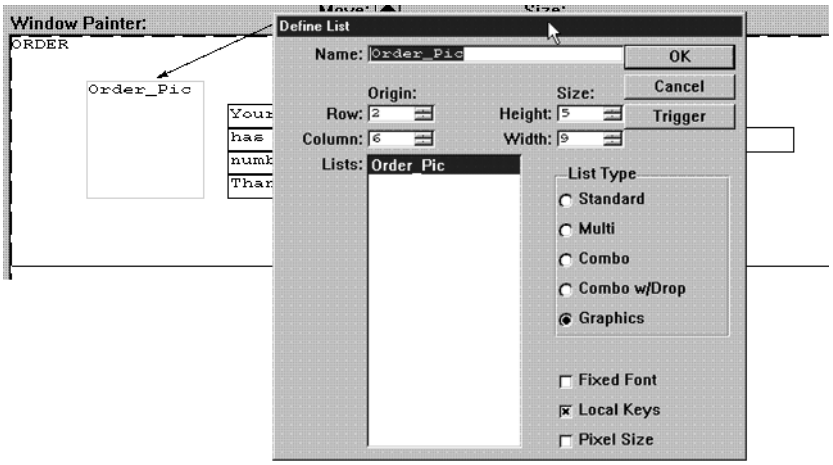
Reload Reloads the Window, Query, Initialization, Record, Table, Update triggers as well as the Menu and Select statement.

Dictionary Activates the DVdd Console.

Rename Allows you to rename an action button to an existing definition in the DVdd when migrating an existing V4 application to DVapp.

List

To see this dialog box, select a list object (with alternate button)



One unique use of a list is to display graphics, as in this example.

Fields

Name The list name.

Origin The row and column values indicate the upper left corner of the window relative to the upper left corner of the underlying window. Your choice determines these values.

Size The height (in rows) and width (in characters) indicates the size of the list item. Typically the width is a few characters wider than the longest line it displays.

Lists Shows all lists in the window.

Radiobuttons

List Type You can choose from the following:

- ***Standard*** — Single choice list. Uses system font.
- ***Multi*** — End user can select multiple items from the list.
- ***Combo*** — This list shows a single field through which end users can scroll to make their selection.
- ***Combo w/Drop*** — Like the Windows standard, this list type shows a single field with a down arrow. When end users click on the arrow, the list "drops down" in a display box.
- ***Graphics*** — DVapp's unique method for displaying bitmaps dynamically. See "***Graphic Lists***" on page 82 for a complete description.

Checkboxes

Fixed Font Specifies fixed font. Otherwise, DVapp uses the system font.

Local Keys Specifies that the direction keys --- **[Up Arrow]**, **[Down Arrow]**, **[Left]**, **[Home]**}, and so on --- are under window control. If you don't select this item, the local keys use TRIMpl key trigger settings.

Pixel Size If checked, converts the height and width values to pixels.

Action Buttons

OK Saves the settings as they appear on the screen.

Cancel Returns to the settings previously saved.

Trigger Opens an editing window that specifies the actions executed when an end user presses the button (or checks the box).

Specifying Complex Fields

DVapp supports simple edit fields — the familiar screen fields where you can type in data, either character or numeric, and where this data is returned from a database retrieval — and *complex fields*.

Complex fields' presentation is controlled through:

- The DVdd definition
- The *Define Field* dialog box

Complex fields are presented as radiobuttons, checkboxes, and lists, but the actual underlying data is an integer. Each bit position in the integer represents one of these three object elements.

Using the DVdd

You control whether the field is an edit, list, or complex field with your `TDD_SFIELD` specification and refine a complex field definition with `TDD_GROUP` as follows:

TDD_SFIELD Specification	Field
Edit name	Regular edit field
List name	List
Group name	Complex field

TDD_GROUP Specification	Complex Field
GRP_FLG_NAME	Multi-choice field (checkbox, multi-tag list)
GRP_CHO_NAME	Single-choice field (radiobutton, single-tag lists)

Regardless of single or multi-choice, you define the complex field's values in the `TDD_CODE`. These values can be as simple as Yes and No or as complex as range of shoe sizes.

Because the actual database field is always simply an integer, the application can actually query on these checkboxes or radiobuttons just as though they were regular edit fields.

Using the Define Field dialog

The *Define Field* dialog (see "***Field***" on page 23) lets you further customize the look of the complex field. In the section called *Complex*, you use the height and width parameters to define the containing area for the complex field. The dimension parameter, in conjunction with orientation, let you control the number of rows or columns the radiobuttons or checkboxes use for display. Dimension does not apply to List types.

Because the complex field's basic variable is a simple integer, you manipulate the complex field's data using regular TRIMpl code and functions. Any changes that your application makes to that variable (querying the database, for example) are automatically reflected on the screen.

Chapter 4
Creating Forms

This chapter is a tutorial that shows you how to perform the basic functions of the DVdd:

- Create and manage database tables
- Store and retrieve field definitions

You must define objects like fields and buttons in the DVdd before you can use them. As you work with the tutorial, notice that all tables eventually depend on one or more of the four root tables:

- `TDD_CATEGORY`
- `TDD_DOMAIN`
- `TDD_LANGUAGE`
- `TDD_TRIGGER`

Navigation

As you work through the tutorial you see that you can [TAB] or click your way around the fields in the DVdd table forms. If you modify a field that has a foreign key, the structure checker checks your value against the foreign key. If your value isn't found, the Console presents a List of Values prompt from which you pick an existing value.

Look at the `COL_FLD_NAME` in `TDD_COLUMN` (also noted as `TDD_COLUMN.COL_FLD_NAME`). Whatever value you put in there must exist in `TDD_FIELD.FLD_NAME`. If it does not, then the LOV for `TDD_FIELD.FLD_NAME` appears when you attempt to leave the `COL_FLD_NAME` column.

If the field is a foreign key for another table, the Console prompts you to modify that table's record(s) before continuing with your change. Using the previous example in reverse, if you modify the `TDD_FIELD.FLD_NAME` column and `TDD_COLUMN.COL_FLD_NAME` entries point to it, you are prompted to modify `TDD_COLUMN.COL_FLD_NAME`.

[Enter] in a foreign key field opens the table application for that foreign key. For example, in the `TDD_COLUMN` table, `COL_FLD_NAME` relies on `TDD_FIELD.FLD_NAME`, so pressing [Enter] in the `COL_FLD_NAME` column brings up the `TDD_FIELD` record with that value.

The Console has only one active function key, [F3], which quits the Console. The table applications have two sets of active function keys, one for non-query mode, one for query mode.

Non-Query Function Keys

Press ...	To ...
[Tab]	Move to the next field when entering definitions.
[Enter]	Open a foreign key or the parent table from a field.
[F1]	Enter query mode.
[F2]	Commit changes.
[F3]	Quit application.
[F4]	Delete current record.
[F5]	Insert a new record.
[F6]	Append a new record.
[F7]	Duplicate the current record.
Up Arrow	Scroll up one record.
Down Arrow	Scroll down one record.
[PgDn]	Scroll down one screen.
[PgUp]	Scroll up one screen.

Query Function Keys

Press ...	To ...
[Tab]	Move to the next field when entering definitions.
[Enter]	Open a foreign key or the parent table from a field.
[F1]	Execute query.
[F3]	Quit application.
[F5]	Count number of query hits.
[F8]	Open extended query editor.

Basic Console Activities

Besides creating tables and defining their elements, you can execute the following tasks from the DVdd Console.

Save Tables

`File > Save` brings up the save menu options. The console writes an ASCII file of SQL commands that describe the dictionary's contents. You can use the file with the VORTEXsql command utility (see "***TRIMrun***" on page 13) to reload the dictionary or move it to another platform or database.

Once you choose a save option, you can specify to save only the CREATE, INSERT or both statements. Your options are:

Table Selection	Saves
All tables	All table (DD and user) definitions.
All non-dictionary tables	All user definitions.
Current (filtered) tables	All tables shown in the Tables list.
Current table	Only the currently selected table.
Dictionary tables	Only the DD tables.

Refresh or Reload

When you create new table definitions in the dictionary, you need to reload (or refresh the screen) to see them in the list with **`Tools > Reload`**.

Validate the Data Dictionary

`Tools > Structure Check` validates the entire dictionary. It checks that all foreign key references are valid and displays a list of any warnings or errors. However, if you always work on the tables with the dictionary applications, your dictionary should be error free. Warnings may include the message "unused tables." Because the structure checker cannot know when your application uses a table, these messages, like the version skipped messages, should not alarm you.

Delete Versions

`Tools > Delete old versions` deletes all old versions stored in the DD tables.

Create Default Entries

`Tools > Create default TDD entries for tables` reloads the default Console entries for the dictionary tables.

Mask Database Catalog Differences

Tools > Generate lets you create tables and views that mask database catalog differences.

Create Header Files

Tools > Generate 'tdd.h' file creates a TRIMpl -and C-compatible header file that contains the definitions stored in TDD_CODE, TDD_DOMAIN, and TDD_FORMAT.

Automatic Documentation

Tools > Generate HTML pages generates HTML pages for the defined tables. You can specify from the following list:

Table Selection	Describes
All tables	All table DD table definitions.
All non-dictionary tables	All user table definitions.
Current (filtered) tables	All tables shown in the Tables list.
Current table	Only the currently chosen table.
Dictionary tables	Only the DD tables.

Creating and Managing Database Tables

The DVdd maintains your table definitions in the TDD_TABLE and TDD_COLUMN tables. It also keeps the index definitions for these tables in TDD_INDEX and TDD_XCOLUMN. The TDD console creates DDL statements to create these tables and indexes when you **File > Save**.

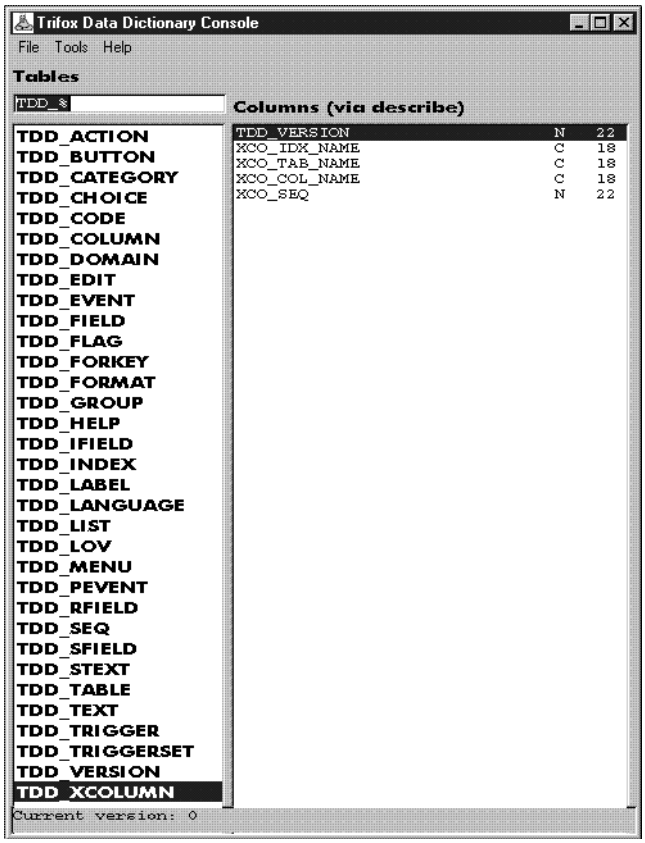
Storing and Retrieving Field Definitions

The tutorial begins by showing you how to build the foundation for a new object for applications — last name field — using the DVdd Console. After storing the information for the object, you can retrieve it and modify it at any time.

Starting the Console

The console is the main entry point to the dictionary maintenance applications. It has a tables list, a columns list, and a menu bar. The tables list shows all the tables currently defined in the dictionary. Columns shows the columns for the currently selected table. To update the columns list, click on the table name. To bring up that table's maintenance application, click a second time.

1. From DesignVision Designer, select **`File > Dictionary`**.

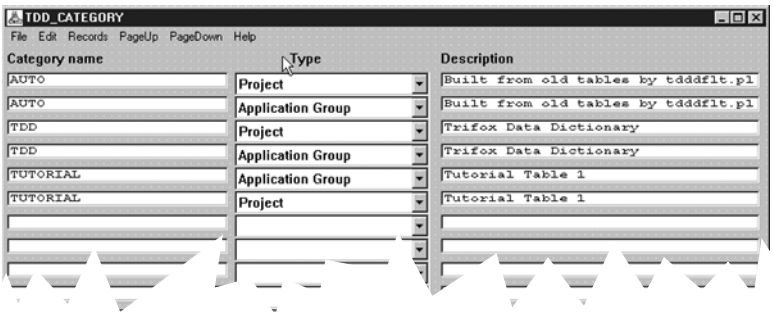


Adding a category

Begin by creating two new categories, one each Project and Application Group.

1. Click on `TDD_CATEGORY` in the tables window of the DVdd Console twice to bring up the table window.
2. Press [F1] twice to query the database and display the existing CATEGORY values.
3. **`Records > Insert`** to add a new item.
4. Define the new category with the following attributes by typing the word in bold in the field and [TAB]bing to the next field:
   - Category Name = **TUTORIAL**
   - Type = **Application Group**
   - Comment = **Trifox Tutorial**
5. **`Records > Duplicate`**.
6. In the duplicated category, change the Type to **Project**.

Your window should look like this:



7. **`Edit > Commit`** (or press [F2]) to commit the change.
   `File > Quit` to close the window.

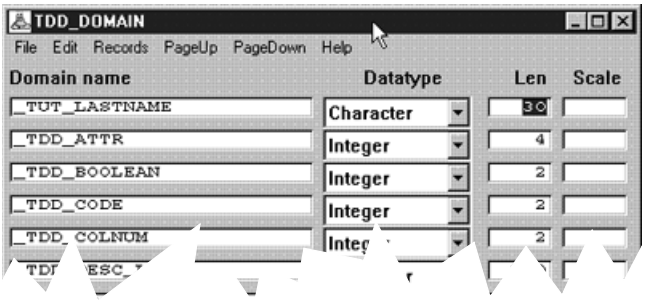
It does not matter if you add an item at the top of a list or at the end. Data is resorted after a COMMIT and when you RELOAD, items appear in alphabetical order.

Creating a domain

Now create a domain for last names.

1. Click the TDD_DOMAIN table twice to open the DOMAIN table window.
2. [F1] twice to query the database and display existing domains.
3. **`Record > Insert`** a new record with the following attributes:
   - Domain name = **_TUT_LASTNAME**

     The "_TUT_" prefix simply makes it easier to see what domains belong to an application group. If you have domains that span application groups, you may prefer not to use a prefix.
   - Datatype = **Character**
   - Len = **30**
4. Your screen should look like this:

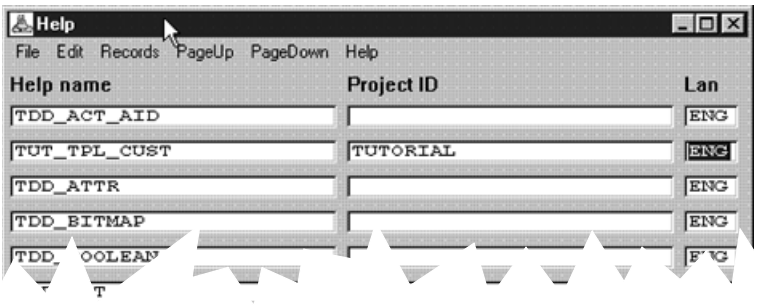


5. **`Edit > Commit.`**
   `File > Quit`.

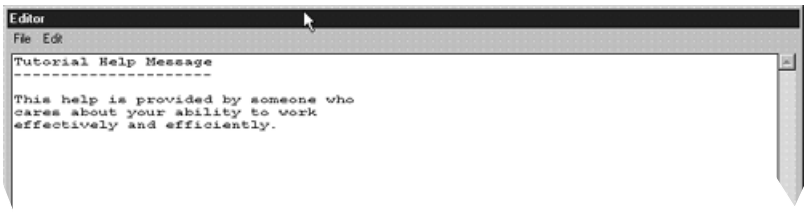
Defining the database table

Because the LASTNAME field is a column in a database table, you must define it in the TDD_TABLE. Referring to the DVdd diagram on page 21, you can see that TDD_TABLE relies on TDD_HELP, so first you must set up the TDD_HELP.

1. Open TDD_HELP and insert a record with:
   - Help name = **TUT_TPL_CUST**
   - Project ID = **TUTORIAL**
   - Lan = **ENG**
2. If you queried the database first, your screen should look like this:

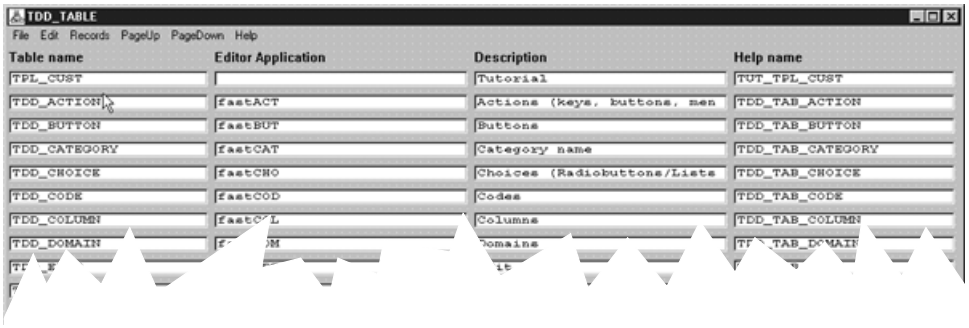


3. Put the cursor in the **Help name** field and [Enter].
4. In the Editor window that appears, type the help message as it looks in the example:



5. Click “OK”.
6. **Edit > Commit.**
   File > Quit.
7. Open TDD_TABLE and define the field with:
   - Table name = **TPL_CUST**
   - Editor name = *Leave Blank*
   - Description = **Tutorial**

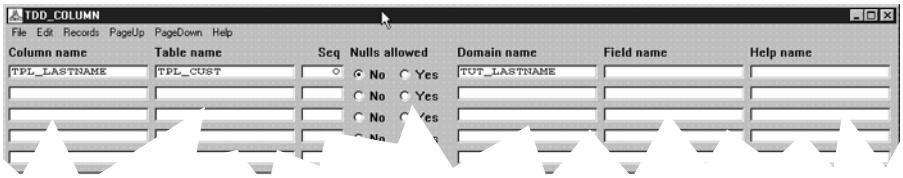
- Help name = **TUT_TPL_CUST**



8. **Edit > Commit, File > Quit.**

Creating the column

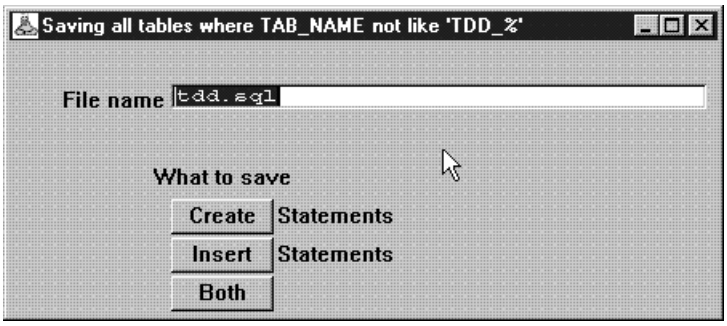
1. Open TDD_COLUMN without querying the table first and enter the following values:
   - Column name = **TPL_LASTNAME**
   - Table name = **TPL_CUST**
   - Seq = **0**
   - Nulls allowed = **No**
   - Domain name = **_TUT_LASTNAME**
   - Field name, Help name = leave empty, as shown in the following:



2. **Edit > Commit** and close the window.

Creating the create statement

1. **File > Save > All non dictionary tables.**
2. Specify a filename, if one does not appear and click **Both** in the dialog box:



3. Using any text editor, look at the file it creates and you see:

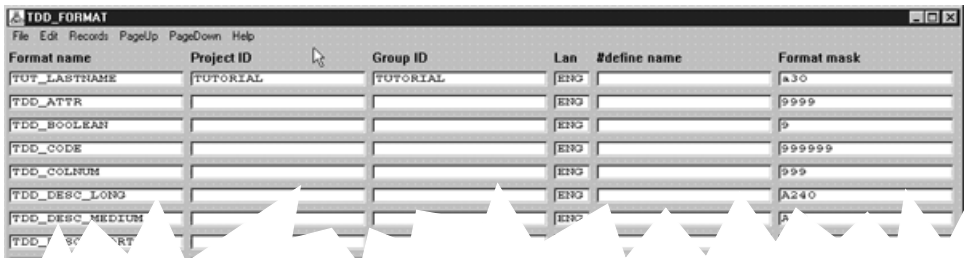
```
#
# To execute: vtxsql /c<connect_string> /rmoose.sql
#
#-----------------------------------------------------------------
# TPL_CUST
#-----------------------------------------------------------------
/xcupdate
drop table TPL_CUST;
/xcupdate
create table TPL_CUST(
  TPL_LASTNAME varchar(30) not null);
#-----------------------------------------------------------------
/xcommit
/quit
```

Determining the field's appearance

After defining the database column and table, you determine how the field appears and behaves in the application. Now you complete the definitions that you left empty in the `TDD_COLUMN` definition and connect the tables.

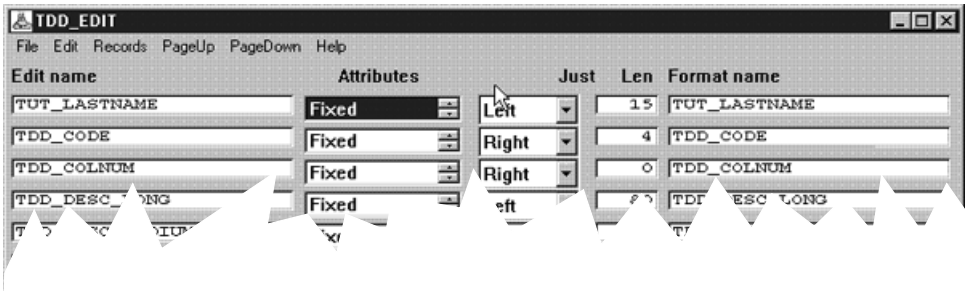
1. Open the `TDD_FORMAT` table and create an entry as follows:
   - Format name = **TUT_LASTNAME**
   - Project ID = **TUTORIAL**
   - Group ID = **TUTORIAL**
   - Lan = **ENG**
   - Format Mask = **A30**

   The window should look like this:



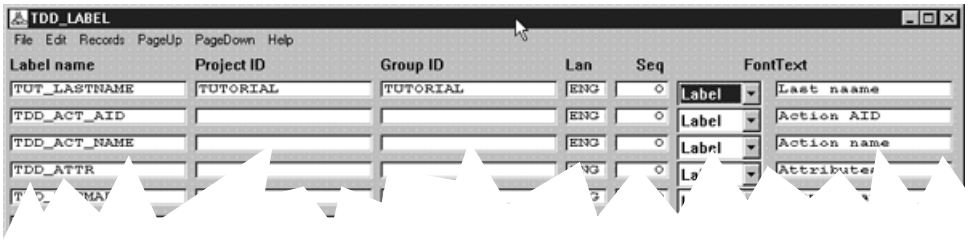
2. Commit the record and exit. (**`Edit > Commit, File > Quit`**.)
3. Open the `TDD_EDIT` table and create an entry as follows:
   - Edit name = **TUT_LASTNAME**
   - len = **15**
   - Format Name = **TUT_LASTNAME**

The window should look like this:



4. Commit the record and exit. (**`Edit > Commit, File > Quit`**.)
5. Open the `TDD_LABEL` table and create an entry as follows:
   - Label name = **TUT_LASTNAME**
   - Project ID = **TUTORIAL**
   - Group ID = **TUTORIAL**
   - Lan = **ENG**
   - Seq = **0**
   - Font = **Label**
   - Text = **Last name**

   The window should look like this:



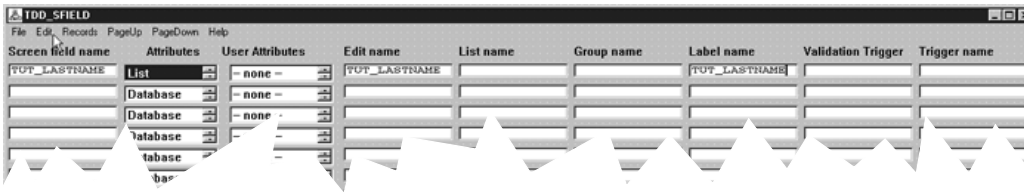
6. Commit the record and exit. (**`Edit > Commit, File > Quit`**.)

   `TDD_SFIELD` is the main DVdd table for screen field definitions. It is the first form that has multi-attributes, which work differently than you may expect. Instead of a list of checkbox items, you scroll through the list with up and down arrows, and highlight the selections that you want to use.

   Open `TDD_SFIELD` and create a new entry:
   - Screen field = **TUT_LASTNAME**
   - Edit name = **TUT_LASTNAME**
   - Label name = **TUT_LASTNAME**

To set the Attributes — Database, Not null, and List — Click in the field to highlight the [PgDn] key to “Not null.” Click in the field to highlight the selection and again, use the arrow to display “List.” Highlight this attribute, also.

The window should look like this:



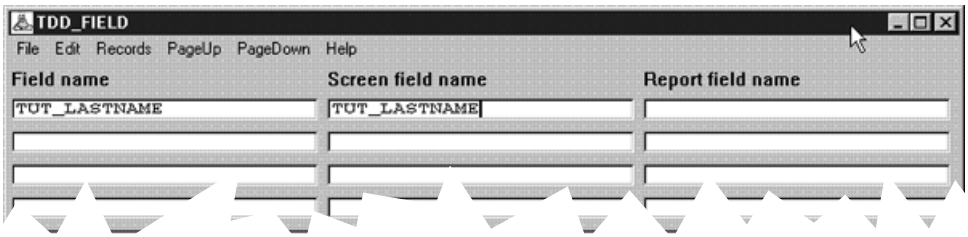
7. Commit the record and exit. (**`Edit > Commit, File > Quit`**.)

☞ *Connecting the tables*

To connect `TDD_COLUMN`, `TDD_FIELD`, and `TDD_SFIELD`, edit the `TDD_FIELD` table and create a new entry:

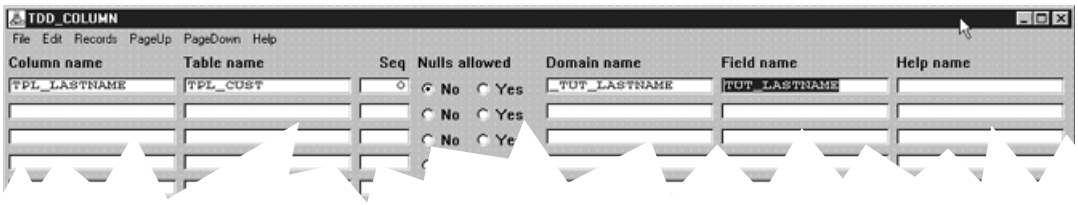
1. Open `TDD_FIELD`, **`Records > Insert`** and type:
   - Field name = **TUT_LASTNAME**
   - Screen field name = **TUT_LASTNAME**

   The window should look like this:



2. Commit the record and exit. (**`Edit > Commit, File > Quit`**.)
3. Open the `TDD_COLUMN` table.
4. **`Records > Query`** (or press [F1]).
5. Type **TPL_LASTNAME** in the *Column* name, and press [F1].
6. Tab to the Field name column, enter **TUT_LASTNAME**.
7. Commit.

You now have the `TUT_LASTNAME` field defined throughout the DVdd.



You can make other entries. For example, you accepted the default data validations by leaving the *Validation Trigger* field empty in `TDD_SFIELD` record for this field. You can see the SQL for your entries by choosing **`File > Save > All tables`** and looking in the created file for `TUT_LASTNAME`.

Completing tutorial data dictionary

Now that you have the `TPL_LASTNAME` field defined, you can use VORTEXsql to insert the rest of the tutorial data dictionary, which is called `tutorial.sql`.

This tutorial creates tables named `TPL_CUST`, `TPL_ORDERS`, `TPL_PAYMT`, `TPL_PROD`, and `TPL_SHIPSUM`. If you have any tables with those names, you must rename them or the tutorial tables will overwrite them. You may see some warning messages about tables that do not exist. This is normal for the first time you run the tutorial DD creation file because the tutorial tables do not exist yet.

Run the utility by replacing the italicized keywords with the correct value;

vtxsql /c*your_connection* **/r***tutorial.sql*

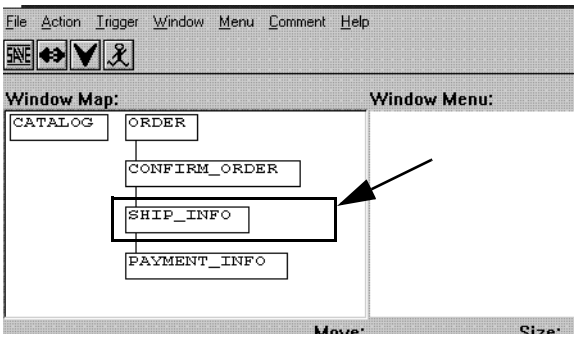
Remember that VORTEXsql commands are designated by a forward slash and then a letter. The command's parameter follows immediately without any spaces.

For a complete list of all the VORTEXsql options, refer to page 13.

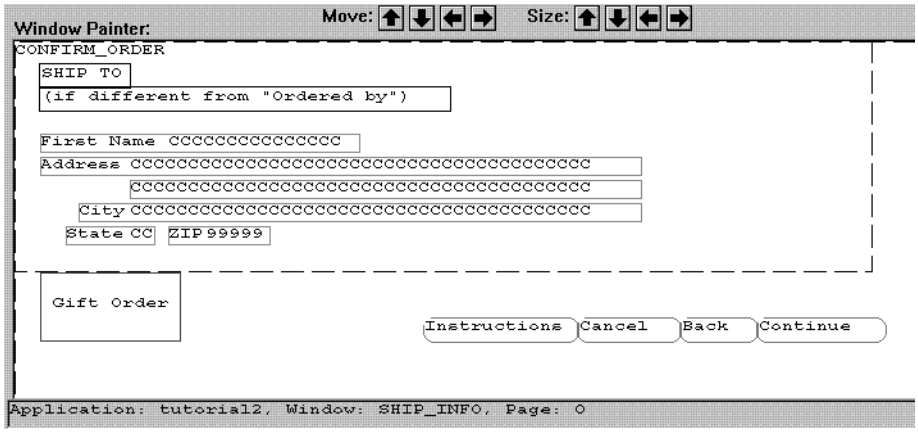
Creating fields with the data dictionary

Now it's time to see how DVapp uses the DD to create fields. Start DVapp, connect using your userid and password and open the tutorial design file.

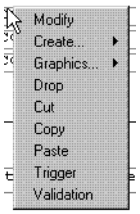
1. **File > Open** and select `tutorial.gap`.
2. Click on **SHIP_INFO** in the Window Map.



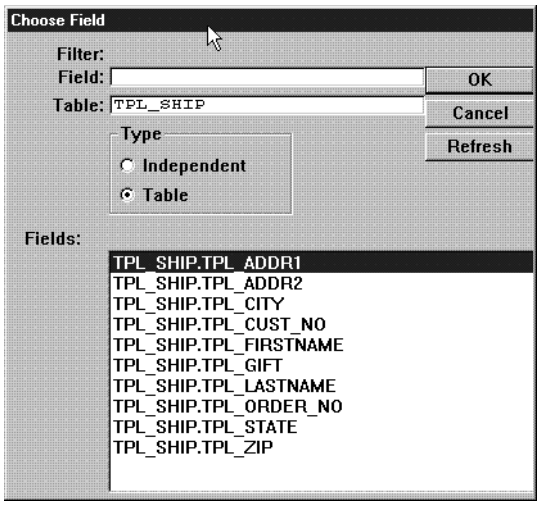
The window details appear in the Window Painter.



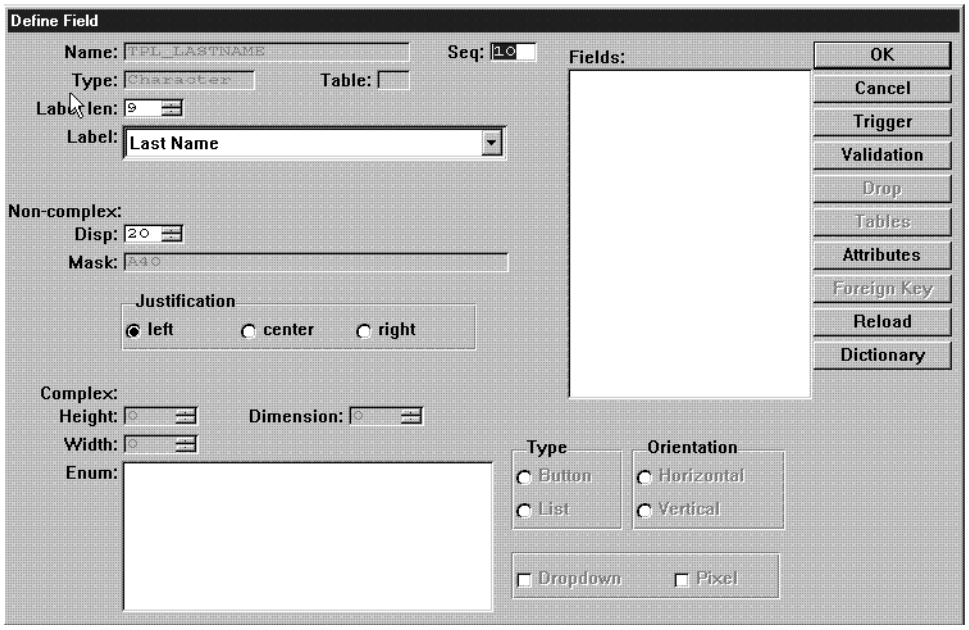
3. Alt-click your cursor about 10 characters beyond the end of the *First Name* field.
4. In the floating menu that appears, **`Create > Field`**.



5. In the *Choose Field* dialog, choose *Table as the Type* if it is not already selected and click [**Refresh]**. Scroll down the list until you see `TPL_LASTNAME`.



6. Click **TPL_LASTNAME**. The *Define Field* dialog appears with all the information you entered in the DD.



7. Choose **OK** and the field is now part of the SHIP_INFO window.

This exercise is a simple example; however, it shows how the DD manages the important aspects of field definition. All applications that use the `TPL_LASTNAME` field look and act the same. If you decide to change size of the field, you simply change it in the DD and reload your applications.

Chapter 5
Writing in TRIMpl

TRIMpl is a 4th generation language (4GL) based on the industry standard programming language C. Like with C, you can examine the code with any text editor, like vi or emacs, or even Notepad, and make your changes to the program files before they are compiled.

Triggers contain one or more TRIMpl statements. The trigger syntax is a subset of C with a few additions incorporated to facilitate operations required for the applications:

- New datatypes.
- Built-in functions.
- Control flow operators.

Object Relationships

DVapp relates windows in applications to build complete applications. The relationships can take one of two forms:

- ***Sibling*** — illustrated in left-right pairings. Sibling report blocks and windows all have the same single parent.
- ***Parent-child*** — illustrated in top-bottom layout structures.

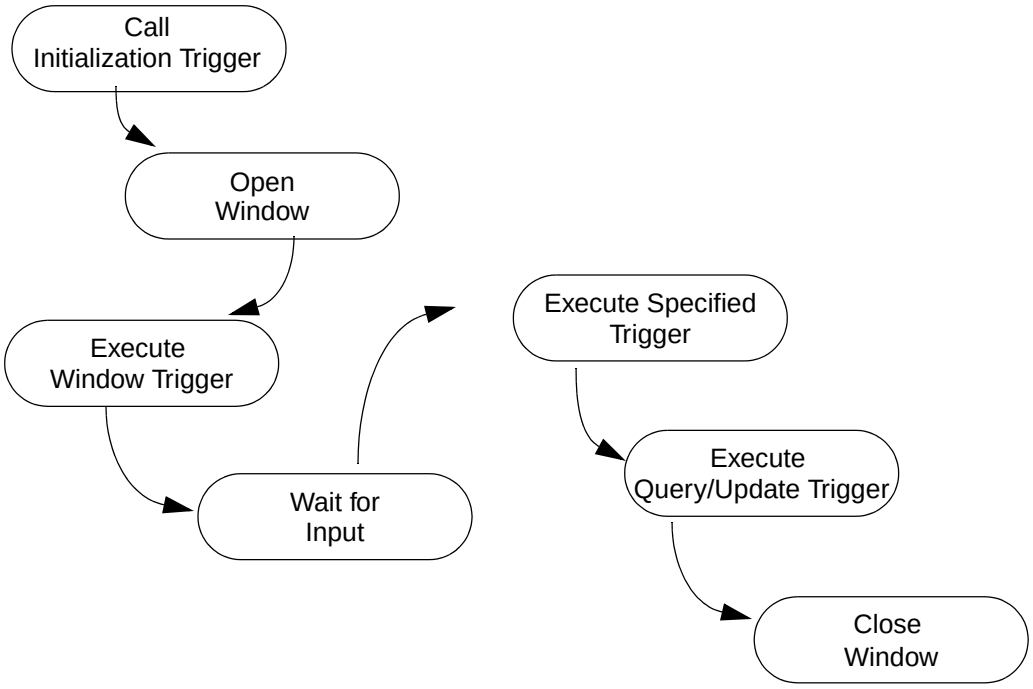
While windows are being designed, only the blocks with direct parent-child relationships appear — the Designer never displays siblings on the same screen.

At runtime, windows are organized top-down and left-right. You can modify the window sequence using the `window()` function, as described in the *TRIMpl Function Reference*.

Execution Flow

Triggers control the execution flow of an application. Windows in an application range from simple status windows to complex navigation control, data display, and data update windows. The more complex the form, the more complex the trigger code.

The typical control flow in a window is:



Typical flow of control begins with calling the Initializing Trigger and ends with Closing the Window.

The `window()` TRIMpl function executes initialization, window, query, and update triggers. `[raw_]input()` is usually responsible for calling key triggers when the end user strikes the key. You can also call them directly using `exec_key()`. You may want to do this if you have both a menu item and a key that perform the same action. The menu item could call the key trigger directly, thereby simplifying application design and maintenance.

NOTE: *While you can call triggers recursively, all variables not explicitely defined as local are static so exercise caution!*

For more details about triggers, code modules that contain one or more TRIMpl statements, and how write and use them, read "***Trigger Operations***" on page 66.

Changing Objects' Status

Each menu item and button has an 8-bit flag that keeps track of the object's state. Typically, when a user selects a button or menu item, the selection invokes the appropriate trigger. However, functions that work with radio buttons and checkboxes need to check the object's state to know what action to take.

DesignVision has seven preset flags that are defined in the header file `trim.h` for character-based applications and `dv.h` for Windows applications:

File	Symbol	Value	Description
`trim.h`	flg_active	1	Field active (echos to screen).
	flg_modified	2	Field's variable modified.
	flg_input	4	Input has been performed from field.
	flg_output	8	Output has been performed on field.
`dv.h`	flag_out	4	Field modified for output. Must be set when setting 16, 32, or 64.
	flag_in	8	Field modified by user input.
	flag_disable	16	Item is disabled.
	flag_return	32	Return this action (when you want radio button or checkbox action).
	flag_tagged	64	Item is tagged (radio button/checkbox is on).

Variables

You can put a predefined or user variable in a report or window text and give it a value at runtime. In text the variable is preceded by an ampersand (&). For example, in a report page footer, a line of text can include the current date and page number:

```
Report generated on &G.TIME   Page &G.PAGENO
```

Inline

You can insert actual code into text by placing an ampersand (&) followed by the code enclosed in delimiters. The delimited code can be a single statement or multiple statements. The delimiter can be any character. The following example uses a forward slash (/).

If a single statement returns a value the value is inserted in the text; otherwise, the code is replaced by a zero (0) length string:

```
&/to_char(CODE, "0999")/
```

Without the formatting the value is placed in a default format. In this example, the code does not have a terminating semicolon (;) so the inline function must return a value. If the line is terminated with a semicolon, the code returns a void value that results in a zero-length string substitution.

Triggers and Functions

A trigger is a special module of TRIMpl code that is used to control an application's behavior, including:

- Alter default execution flow.
- Send control codes to the output device(s).
- Create and manipulate user variables.

DVapp has two types of triggers each with its own area of influence:

- ***Field*** — Local to field, accessed in designer with alternate button (typically the left mouse button for most right-handed users).
- ***User*** — Local to the window, selected from pulldown menu.

If a trigger contains multiple statements, you must enclose each block of statements in braces ({ }).

Triggers are considered window elements for the purposes of designing applications and reports. However, they are logical, not visual. Triggers are the elements that contain the instructions for each action in an application or report generation. Triggers can either be local to the *field*, and accessed in the Designer with the alternate mouse button, or local to the entire application. These triggers, called *user* triggers are available from the pulldown menu in the designer.

User-Defined Functions and Triggers

You can define functions interactively using the DVapp (or TRIMreport) Editor and the USERTRIG action.

If your user-defined triggers need to be available to other applications or reports, however, you must write and save them in the `dv.fnc` file.

You can execute a block of code that has been assigned to a trigger variable at any time. Especially when parameters are passed to the trigger, assigned blocks behave like function calls.

Stand-alone

TRIMpl stand-alone code, a complete report or application design, can behave as a function. In general, using stand-alone code as a function is not good practice because of the overhead of reading in another file. Arguments can be passed to a run file within the `call()` function. For example:

```
call(``mytest.run'',``arg1'',``arg2'');
```

`mytest.run` can reference each of the passed arguments as `parm[0]` through `parm[n]` in the main trigger if the file is an application or report design, or stand alone design. Whatever the called function returns (up to 400 bytes, unless it is returned in a list) is passed back by the `call()` function.

Field Triggers

Field triggers do not have any parameters passed in. However, the active field is restored at exit from a field trigger. This means that if the field trigger modifies the active field, either using a `window-name.AF` variable or the `active_field()` function, on exit, the original value is restored.

Validation Triggers

Field validation triggers automatically receive two parameters: the character string representation of the value in the field and the format mask for the field. These parameters are referenced as `parm[0]` and `parm[1]`.

Calling Conventions and Return Values

You can reference values passed in parameters as `parm[0]` through `parm[n]` where the parameter variable's datatype is inherited from the argument. Because an argument is passed by value, not reference, a structure such as an array cannot be passed, although a list can.

You can pass an unlimited number of arguments to a function and read the number with the `count(parm)` function. Errors occur if you refer to a `parm[n]` that has not been passed results in a runtime error or if a function returns a value and no variable receives.

You can also pass the original function parameters to another function without having to explicitely enumerate them by using the parm() function. The parm() function takes up to two optional parameters, a starting zero-based index and a count. parm() without parameters places all of the incoming parameters into the called function's parameter list. parm(m) places the parameters beginning with index m and parm(m,n) places the parameters beginning with index m and for count n. For example, function/trigger oldfnc is called as oldfnc(1,'There",56):

```
newfnc("hello",parm());       /* newfnc("hello",1,"There",56) */
newfnc(parm(2,1),parm(0,2)); /* newfnc(56,1,"There")         */
newfnc(parm(),parm());   /* newfnc(1,"There",56,1,"There",56) */
```

NOTE: *The* `return()` *keyword is not allowed in certain code blocks, such as windows or field triggers.*

TRIMpl Language Syntax

TRIMpl has a number of similarities to the C language:

- A pre-processor phase that lets you replace variables and keywords with other strings. It processes the `trim.h/dv.h` first and handles triggers in the order they appear.
- The flexibility to put the commands anywhere in the TRIMpl code.
- Both `/* */` and `//` style comments are supported.
- Requirement that code declarations are enclosed in braces ({ }). While C allows a set of braces to contain no code, TRIMpl expects at least a semicolon (;). For example, C allows

```
main()
{
}
```

While TRIMpl requires

```
{
;
}
```

However, the TRIMpl code example has no function declaration; in this example, the name of the file is used for the name of the function or stand-alone program.

TRIMpl supports the following commands (which always begin with a #):

Command	Use
#include	Specify the file name without ' ''..'' ' or '<..>'. If you do not specify the search path in `trim/dv.ini` or with the `include_path` parameter, the preprocessor only searches the current working directory. For example: `#include std.trg` (“***Naming Conventions***” on page 55 for more information about `trim/dv.ini` settings.)

#define, #undef	TRIMpl supports both simple defines such as
	`#define mydef 1`
	and macros. The definition as well as the invocation of the macro ***must*** be on a single line. For example,
	#define hello(A,B) printf(A^^” and “^^B)
	Specifying TRIMgen `-u` automatically creates
	`#define GUI "1"`
	Specifying TRIMgen -g automatically creates
	`#define DEBUG “1”`

#if[*n*]def, #else, #endif	If you are running in window display mode, the following example places the application name in the window title.
	`#ifdef GUI`
	`window_name(design_name());`
	`#end`

#trigger [name]	Specify a trigger within a standalone trigger. The trigger becomes a function ‘name’ that can be called from within the standalone trigger. The standalone trigger must be delineated by #trigger with no name.

TRIMpl does not support certain standard C functions and control operators. If it encounters the unsupported constructs, TRIMpl returns syntax errors.

The following constructs are not currently available in TRIMpl; however, stand-alone functions to address these needs are relatively simple to develop.

- do … while()
- Pre-increment / decrement
- Auto-operators
- Bit shifting

Naming Conventions

Functions, blocks, windows, and variables have *identifiers*, names that you use in an application to specify one of the items. Block names, or block identifiers are specific to code blocks, so that you can reference them outside of the function to which they belong.

Like SQL, TRIMpl treats all identifiers as uppercase text by default. By convention, data types, built-in functions, and other keywords are all lowercase. You can change the case by altering the `uppercase` and `uppercase_sql` options in `trim/dv.ini`.

All identifiers have the following format:

```
[window-name | block-name].identname}
```

If you don't specify a block name, TRIMpl assumes the current window or block name.

The reserved blockname **G** represents:

- Predefined variables.
- User variables defined in the main trigger.
- Variables in the global window trigger in DVapp.

Syntax Extensions

TRIMpl has extensions to the standard C syntax that simplify certain database and application operations:

Concatenate

The double caret (^^) concatenates two strings.

```
field = "Salary in dollars is " ^^  sal;
```

field

Indicates the area on the application window or report page where the attributes from the DEFINE FIELD dialog in DVapp should be placed.

```
field = salary * 1.1;
```

causes

```
salary * 1.1
```

to appear on the page or screen as specified by the field definition.

This keyword (in lowercase text) indicates the actual character area on the application window that corresponds to the definition in the *Define Field* dialog in DVapp.

field_d

Usually `field_d` and `field` go hand in hand; that is, if a value is written to `field`, then `field_d` always contains the character version of that value. On the other hand, if a character value is written to `field_d`, then `field` is not modified.

synonyms

`PARENT` or `P`. In a DVapp window trigger, `P` or `PARENT` refers to the Global window. In the key, update, validate, and field triggers, `P` or `PARENT` refers to the current window.

parm.n and parm[n]

To reference a parameter in a user trigger or function, TRIMpl automatically numbers parameters by using either the variable `parm.n` or `parm[n]`, where *n* is zero-based. (TRIMpl automatically uses the style with square brackets and we recommend that you follow this convention.) Since users know neither the number nor the type of parameter until runtime, this convention allows dynamic type conversions when you need them.

When you pass parameters to TRIMrun with the `-p` option or pass them through the call function, they are accessible in the main triggers as `parm.n` or `parm[n]`. Because only one parameter area is available, parameters must be stored in local variables before make other function calls.

Use `count(parm)` to get the number of parameters at runtime.

IN predicate

The IN predicate, identical to the IN predicate in SQL, appears in conditional statements. It is functionally equivalent to a series of OR clauses in an if statement.

LIKE predicate

The LIKE predicate, used in conditional statements, is identical to the LIKE predicate in SQL. Two special characters are used as wildcard characters: "_" — any single character and "%'' — zero or more of any character.

@

Putting an "at sign" (@) in front of a field variable with a list attribute causes the variable to be replaced at runtime by a constant integer that represents the field's list column number.

For example,

```
printf(list_curr(p.wl,@p.salary));
```

prints the column corresponding to the field `salary` from the current item in the window list.

export/import

Use these keywords to make variables accessible across applications without having to pass them as parameters or return them as return values.

This example, which can be added anywhere in the code, exports a variable `VAR`:

```
export VAR;
```

Any application that needs to access `VAR` must add the following

```
import VAR as MYVAR;
```

From then on `MYVAR` can be used like any other variable.

local

The `local` keyword defines the variable as local to this invocation of the trigger. Local variables can only be defined in user triggers. Local variables are useful in writing

recursive functions and if they are defined with an initialization value, they will be initialized on every entry to the trigger.

For example,

```
local int VAR;
local int VAR = 42;
```

Local variables are stored in a special variable stack. The stack size is controlled by the stack_size keyword in the `dv.ini/trim.ini` file; the default size is 8000 bytes. A runtime error will be thrown if the variable stack space is exceeded.

master

The `master` keyword defines the variable as the master variable instance for all variables of the same name. Any assignment to this variable will propogate to all other variables in the application with the same name. This can be used to initialize variables in multiple windows with one assignment. For example,

```
#trigger one
{int VAR; printf("one"); if (VAR) printf(VAR);}
#trigger two
{int VAR; printf("two"); if (!VAR) printf(VAR);}
#trigger
{
master int VAR;
one(); two();
VAR = 42;
one(); two();
}
```

This will display

```
one
two
0
one
42
two
```

Groups

TRIM/PL has the concept of groups of variables or expressions. It enhances the assignment and return statements. There are two ways to use groups. The first is in simple assignments. For example,

```
[var1,var2, ... ,varN] = [expr1,expr2, ... ,exprN];
```

Notice the "[]" brackets around the variables and expressions. These identify the group members. This syntax is equivalent to

```
var1 = expr1;
var2 = expr2;
...
varN = exprN;
```

If there are more expressions than variables the additional expressions are ignored. If there are more variables than expressions then the last expression is repeated for the additional variables. For example,

```
[a,b,c] = [88];
```

results in all three variables being set to 88.

The second use of groups is with return() statements. For example,

```
[var1,var2, ... ,varN] = X();
```

where X() does a return([expr1,expr2, ... ,exprN]);. The same assignment rules described above apply. X can be a trigger(), execute(), or call().

Filename Specifications

Each of the functions that operate on files needs a specification to identify the correct file (or URL). The functions are:

- `append()`
- `delete()`
- `dump_scr()`
- `file_copy()`
- `list_file()`
- `list_open()`
- `log()`
- `open()`

You can specify a "filename" as:

- ***Local*** (current working directory) — specified simply with the filename.
- ***Display*** — `gui!`*`filename`*
- ***Database*** — `vortex!`*`filename`*
- ***Internet*** — `net!`*`type`*`:`*`url`* (read only)
- ***Directory*** — `dir!`*`options pathname`* (read only)

Local

The following example instructs the program to open a file on the same machine as the application server in the current working directory for a file called `myfile.txt`.

```
ll = list_open("myfile.txt",100);
```

To open a file on the same machine, but different directory, simply specify the full directory path.

Display

In some cases you want to open or modify a file on the machine that is displaying the application, which can be running either fat (where the server machine is already acting as "display") or thin client. In the situation where the file is on the display machine, you prefix the filename with the specifier `gui!`:

```
ll = list_open("gui!myfile.txt",100);
```

If the filename contains wildcard characters, then the Display's File Open/Save dialog will open:

```
ll = list_open("gui!\tmp\*.lst",100);
```

`gui!` has five subspecifiers:

- ***Clipboard*** — to read or write to the display client's clipboard, specify "`gui!clipboard:`".
- ***Dir --*** to bring up a Windows file dialog box. The returned list has the name of the chosen file.
- ***Edit*** — to start up an editor on the display client to edit the list, use "`gui!edit:`*`editor`*" where *editor* is the name of an editor program on the client that accepts a filename on the command line. For example,

  ```
  list_file(ll,"gui!edit:notepad","a");
  ll = list_open("gui!edit:notepad",100000)
  ```

  sends the contents of the *ll* list to the notepad editor and then reads it back again when notepad exits.
- ***Image*** — to add images to the display client that can then be displayed on the screen, use "`gui!image:`*`imagename.type`*" *Imagename* specifies the name of the image and *type* specifies the type of image (.gif, .bmp, or .jpg) to display. (For more information about displaying images, see "***Graphic Lists" on page 82***.)
- ***Peer*** — not applicable to file I/O and not discussed here.

Database

If the file to manipulate is located on a machine other than the local, or display, machine use `vortex!` to specify the file, which can be located where a VORTEXserver database

connection has been made or simply where the VORTEXserver file copy program runs. In this example, `list_open()` uses the Oracle database connection to fetch the file located on the machine specified as *dbmachine*.

```
{
connect(0,"net:scott/tiger@dbmachine!VTX0");
ll = list_open("vortex!myfile.txt",100);
.
.
.
}
```

Retrieving a file from the machine that runs the VORTEXserver file copy program might look like this:

```
{
connect(0,"net:whatever@filemachine!vtx99");
ll = list_open("vortex!myfile.txt",100);
.
.
.
}
```

Again, you can specify as much directory path information as necessary to find the file.

Internet

To access files through HTTP or FTP, you use the `net!` prefix and specify *type* as either `http` or `ftp`. Note that you can't update any files with this prefix. It is read only. The following example accesses the file "myfile.txt" located on an FTP server:

```
ll = list_open("net!ftp://ftp.trifox.com/pub/outgoing/myfile.txt",100);
```

You can also access HTTP servers with

```
ll = list_open("net!http://www.trifox.com/myfile.txt",100);
```

Directory

The read-only directory specification lets you specify a directory and filename mask and get all the qualifying entries. Use the syntax

`dir![-rfd]` *`pathname`*

where

r scans directories recursively. Used alone it assumes rf.

f returns only filenames (default, if you don't specify another option)

d returns only directory names.

When you specify any option, be sure to put a blank space before *`pathname`*.

The following example returns all files with an `.a` extension that are located in the directory `/usr` or any of its subdirectories.

```
LL = list_open("dir!-rf /usr/*.a",1000);
```

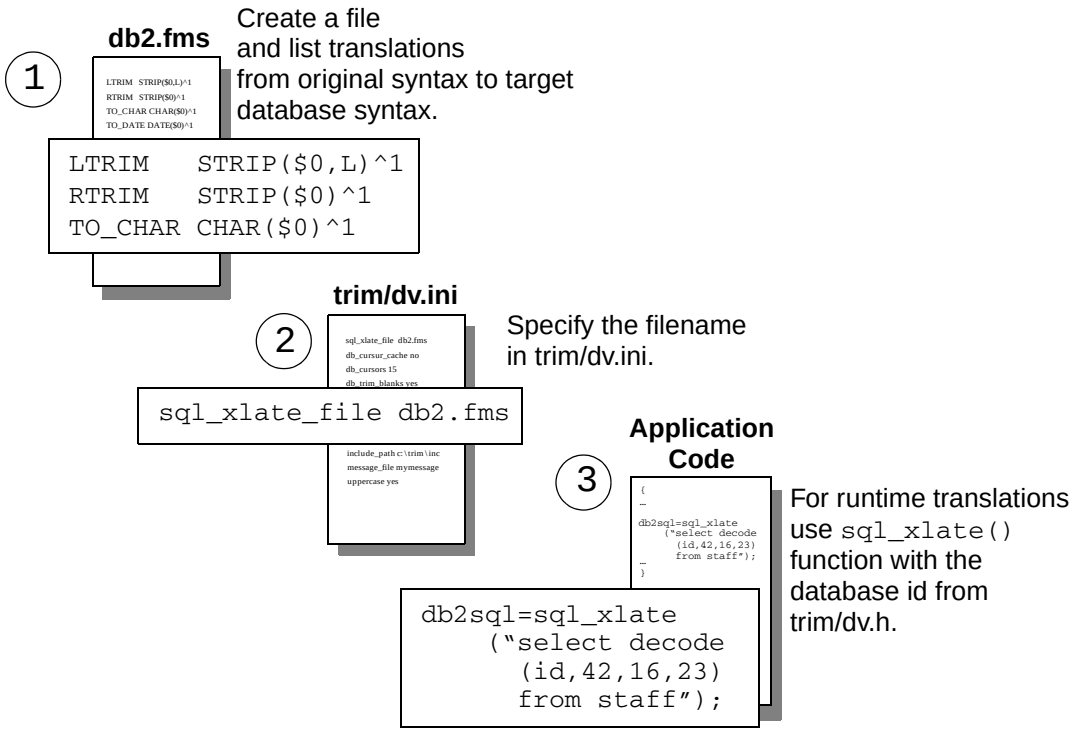
SQL Syntax Translation

While all SQL database vendors support ANSI-92 SQL syntax, many have additional functions to support their tools. These additions present a problem if you want to build truly database-independent applications. You perform SQL translations with `sql_xlate()` runtime.

For example, Oracle supports a DECODE function whereas DB2 has a CASE statement. The following specification in a function mapping file enables the function `sql_xlate()` to translate the Oracle function to run correctly with a DB2 database.

```
DECODE  CASE WHEN $0=$1 THEN $2 #2[WHEN $0=$3 THEN $4 ]ELSE ~0 END
```

If you want to use dynamic translation, the function mapping file ***must*** be available to the runtime application.



Using `sql_xlate()` *requires several steps, including creating the function mapping file.*

The function mapping file contains lines of paired of functions, like the one above, and comments, which are marked with a "#" at the beginning of the line. The first part of the function mapping definition is the name of the function to replace (or map). The second part, separated by one or more blanks from the first part, is the string to use instead.

This mapping string can contain certain control characters that reference the parameters from the function. If any of the control characters are necessary in the actual output, you must escape them with a '\'.

Control Character	Definition
*	All parameters. The parameters in the function are simply copied as they appear including the commas (',').
$*n*	Parameter *n* where 0 is the first parameter.
~*n*	Parameter *n* from the end where 0 is the last parameter.
?*n*[*s*]	Optional parameter *n*. If parameter *n* is given, then *s* is processed.
#*r*[*s*]	Repeat for the next *r* parameters. For each set of parameters found in *s*, *r* is added to *n*. *s* is repeated as long as there are at least *r* parameters left in the parameter sequence.
^*n*	Error out if parameter *n* is found.

Here is an example for mapping Oracle to DB2:

```
DECODE  CASE WHEN $0=$1 THEN $2 #2[WHEN $0=$3 THEN $4 ]ELSE ~0 END
SUBSTR  SUBSTR($0,$1?2[,$2])
LTRIM   STRIP($0,L)^1
RTRIM   STRIP($0)^1
TO_CHAR CHAR($0)^1
TO_DATE DATE($0)^1
NVL     CASE WHEN $0 IS NULL THEN $1 ELSE $0 END
MAX     MAX(*)
MIN     MIN(*)
AVG     AVG(*)
SUM     SUM(*)
COUNT   COUNT(*)
```

Chapter 6
Variables and Triggers

Trigger Types

The TRIMpl code contained in the window's triggers controls application window behavior. Triggers are not visual elements, but logical ones. They contain the instructions for each visual element. Triggers can have user-initated actions:

- ***Action*** — Controls action button behavior.
- ***Key*** — Controls action when key is pressed.
- ***Menu*** — Creates a list of items with associated items.
- ***Predefined*** — Determines pre-defined window events, such as focus change.

Or they can be controlled by the TRIMpl code:

- ***Initialization*** — Initializes the window.
- ***Query*** — Manages database queries.
- ***Radio button*** — Controls radio button behavior.
- ***Update*** — Manages database updates, inserts, and deletes.
- ***Window*** — Controls the overall window behavior.

Windows Event Triggers

TRIMpl includes 16 pre-defined event triggers (their definitions are in the file `dv.pev`).

`G.AUX` is a predefined integer variable that is used to return values from certain events. (See "***Predefined Variables***" on page 67 for a complete list.)

Event name	Action taken
Single click right mouse button	User clicked right mouse button.
Single click left mouse button	User clicked left mouse button.
Double click right mouse button	User double-clicked right mouse button.
Double click left mouse button	User double clicked left mouse button.
Focus change	User clicked in different field.
Window	A window event occurred. AID contains the window number where the event occured and AUX contains the sub-event. There are currently two sub-events defined in dv.h: wev_expand wev_collapse

Event name	Action taken
IPC event occurred	An IPC message was received.
Window resize	A window resize message was received.
Communication event notification	A window communication event was received.
Close event notification	A window close message was received.
Window change	User switched to a different window
Popup menu	User picked a popup menu item (`g.aux` is the index.
Client killed	The client was closed by a user action.
Forwarded actions	A child window forwarded an action (`G.aux` is the index.)
WM_USER + 998 message received	`G.aux` contains wParam.
Timer	`input_timer` popped.

Variable Names

All variable identifiers must begin with an alphabetic character between the values of A through Z and be no more than 30 characters long. Variables are case-insensitive and are converted, by TRIMpl, into all uppercase unless the `trim/dv.ini uppercase` option is set to `false`. ("***Naming Conventions***" on page 55 for more information.)

After the first alphabetic character, a variable identifier can include alphabetic characters, digits (0-9), and / or an underscore (_), pound sign (#), or dollar sign ($).

The following are examples of valid names for variables:

```
int      A00001;
trigger  A$$$$$;
list     B_$$01;
char     foo_zz$#[50];
char(50) foo_zz$#;
```

TRIMpl has a number of variable datatypes to meet individual user needs. However, it does not allow for structure declarations or for arrays exceeding two-dimensions. To implement these features, TRIMpl uses lists. All variables must be declared before an expression in a block of code.

When a design is compiled, all of the variables in the design are allocated memory and made globally available, unless declared with the `local` keyword, given their unique extension and variable / field name. If the `trim/dv.ini` option `uppercase` is not specified, TRIMpl ignores case; otherwise, it is case-sensitive if `uppercase` is set to `false`. Also, the variables are initialized to NULL unless explicitely initialized.

Scope

For local scope, you do not need to fully qualify the variable name. A variable's value is preserved across invocation of triggers. You can access block or window variables from

anywhere if you provide a fully-qualified name (the locally declared name with appropriate prefix). TRIMgen automatically adds a prefix to a variable as explained below.

DVapp Trigger Types

In DVapp, variables declared in the main trigger are assigned to the Global window. Declaring a variable called *trig01* in both the main trigger and in the Global window trigger generates a syntax error.

Similarly, a variable space overlap exists between a field's main trigger and its validation trigger. The field *TPL_ZIP* cannot have a variable called `var01` in both its field and validation trigger. In addition, even though variables that overlap can reference each other as local variables without having to fully qualify the reference, full qualification is a wiser choice.

Trigger	Variable	Description	Example	Note
Key	w.K.n.v	w - current window name n - key number v - variable name	W1.K.0.len	You can only reference these variables from inside the key trigger.
Update	w.U.U.v	w - current window name v - variable name	W1.U.U.cnt	You can only reference these variables from inside the update trigger.
Window	w.v	w - current window name v - variable name	W1.mode	
Field & Validation	w.f.v	w - current window name f - current field name v - variable name	W1.id.i	You can only reference these variables from inside the field and field validation trigger.
User	u.F.F.v	u - user trigger name v - variable name	upper.F.F.i	These variables can only be referenced from in the user trigger.

DVreport Trigger Types

Trigger	Variable	Description	Example	Note
All	b.v	b - current bock name v - variable name	B1.count	
User	uF.F.v	u - user trigger name v -variable name	wait.F.F.j	These variables can only be referenced from inside the user trigger.

Stand-Alone Trigger Types

Trigger	Variable	Description	Example	Note
All	b.v	b - current bock name v - variable name	G.len	
User	uF.F.v	u - user trigger name v -variable name	days.F.F.cnt	These variables can only be referenced from inside the user trigger.

Trigger Operations

Trigger variables are defined as offsets into a code execution block. The only operations allowed are assignments and executions.

Examples

```
/*
** Dynamic trigger assignment
*/
{
trigger oper;
char    resp[1];
numeric arg1,arg2,result;

arg1 = prompt("Type in 1st argument->");
arg2 = prompt("Type in 2nd argument->");
resp = prompt(Type in function code (A = add, S = sub) ==> ");

if (resp == "A") oper = { return(parm.0 + parm.1); };
else             oper = { return(parm.0 - parm.1); };

result = execute(oper,arg1,arg2);

printf("Result of operation is " ^^ result);
}
```

Predefined Variables

TRIMpl provides 16 global predefined variables.

Variable	Represents
G.PAGELENGTH	Length of report page.
G.PAGEWIDTH	Width of the report page.
G.PAGHEADERLINES	Current number of page header lines.
G.PAGEFOOTERLINES	Current number of page footer lines.
G.LINENUMBER	Current line number.
G.PAGEOFFSET	Column offset within page.
G.PAGENUMBER	Current page number.
G.SCREENROWS	Screen height in number of rows and lines.
G.SCREENCOLS	Screen width in number of columns and characters.
G.INPUT_DONE	Forces field re-entry when set to zero.
G.KEY	Value of last key pressed.
G.ERROR_CODE	Last error encountered.
G.TIME	Starting time of execution.
G.MODIFIED	Returns true when field has been changed.
G.INPUT_DATA	Returns true when anything has been entered in field.
G.AID	Set to a value (from the data dictionary) after a `[raw_]input()` call. If the value is -1, the variable is set to `G.KEY`. On a FORWARD EVENT with the value -1, `G.AID` is set to `G.AUX`.
G.AUX	Auxiliary variable.

Do not set any predefined variable to NULL. The results are unpredictable.

All predefined variables are ***int*** except `G.TIME` which is ***datetime*** and `G.MSG` which is a ***string***.

Predefined variables, which are used internally by TRIMrun, are available to you for modification; however, be conservative, unexpected results can occur because of incorrectly redefined variables.

NOTE: *Not all of the predefined variables are used for both applications and reports, as the names indicate.*

Example

This example changes the predefined variables to create a multi-column report.

The default paginate trigger contains a single statement:

```
paginate(footer|break|header);
```

Change the paginate trigger to produce a two-column report:

```
if (G.PAGEOFFSET == 0) {/* 1st pass thru the page ? */
   G.PAGEOFFSET = G.PAGEOFFSET + 40; /* shift by 40 columns      */
   G.LINENUMBER = G.PAGEHEADERLINES; /* reset line number count  */
   }
else {                               /* 2nd pass thru the page   */
   G.PAGEOFFSET = 0;                  /* reset back to column 0   */
   paginate(footer|break|header);     /* do the actual page break */
```

DVapp Predefined Window Variables

DVapp automatically allocates the following predefined variables for each defined window.

- ***window-name.WL*** — Window list.
- ***window-name.AF*** — Active field in window.
- ***window-name.AR*** — Active row in a window.

In addition, each *field* in a window has two implicitly defined variables:

- ***field*** — Data (in correct type).
- ***field_d*** — Character string representation of data.

TRIMreport Predefined Block Variables

The TRIMreport predefined variables are automatically allocated for each report block that has a SELECT statement defined. In addition, each column in the SELECT list has an implicitly defined variable.

- ***block.CURRENT*** — Sequential number of current fetch.
- ***block.EOS*** — End-of-scan flag.

Miscellaneous Predefined Variables/Symbols

NULL — Equal to nonexistent value.

SYSDATE — Current date and time value.

field — Currently active field.

field_d — Currently active field's raw data.

Variable Array Declarations

TRIMpl allows for variable array declarations. An array defined for *n* elements has elements var0 through var(*n*-1). For example:

```
{
int   i, x[2];
numeric  n[4];
datetime d[3];
trigger  t[3];
list     z[5];
char     s[2][10]; /* array of 2 strings of max length 10 chars */

x[0] = 1;
x[1] = 5;
n[2] = 1.5;
d[1] = "05-MAR-89";
s[0] = "Hello";
s[1] = "World";

for (i=0;i<2;i++) printf(s[i]);
}
```

NOTE: *Only char/string variables are multidimensional (although a list containing lists is a multidimensional construct). If a program exceeds the bounds of an array, a runtime error is generated and program execution stops.*

Designer Field Variables

Although TRIMpl supports six variable datatypes, only three are supported for window or report fields:

- char
- datetime
- numeric

In DVapp, a field defined in the window is a variable. For example, if an application has a window field called *balance* that is defined as a numeric, then the window trigger can reference the field as though it were any other variable.

If the window has *n* rows, then each row of a field in the window can be referenced as `fieldname[0]` through `fieldname[n-1]`. For example, the third row of the balance variable can be referenced as `balance[2]` the first row as `balance[0]` or `balance`.

Conversion

Data is converted at runtime to the appropriate datatype, if discrepancies appear to exist. For example, in the statement `total = sal + x;`, if `sal` is numeric and `x` is char, x is converted to numeric if it contains a valid number (for instance, 23.2).

Variables are automatically converted to an appropriate datatype in all cases except `int`. Automatically converting to `int` can lose data, you must use the `to_int()` function when converting any other datatype to `int`.

Trigger and list variables cannot be converted to other types because no other datatype makes sense.

The table below summarizes the results of straight assignments between data types.

to **from**	**int**	**char**	**numeric**	**datetime**
int	x	String of digits	Number	Converts number of days since 01/01/0000
char	Using to_int(), converts digits until non-digit char	x	Converts digits until non-digit char. Accepts "."	Default format: DD-MON-YY
numeric	Requires to_int()	String of digits	x	Number of days since 01/01/0000 and hours, minutes, seconds of fractional days
datetime	number of days since 01/01/0000	Default format: DD-MON-YY	Number of days since 01/01/0000 and fraction of day.	x

Without the formatting, the value is placed in a default format. In the example above, the code does not have a terminating semicolon (;) and so the inline function must return a value. If there is a semicolon, the code returns a void value that results in zero-length string substitution.

Chapter 7
Datatypes

TRIMpl offers eight (char and string are considered the same datatype) datatypes for variables:

Type	Description	Initial Value	Code
char / string	Alphanumeric string	NULL	1
datetime	Internal date / time	NULL	12
int	Integer	0 (zero)	0
glob	Large object char / binary	NULL	17
list	Dynamic multi-dimensional structure	NULL	16
numeric	Fixed or floating point	0 (zero)	2
rowid	Binary	NULL	98
trigger	Block of code	NULL	32

TRIMpl also supports arrays of any of these datatypes.

char/string

TRIMpl treats character variables as strings of characters. *char* and *string* keywords are interchangeable. The string size is the length of the longest string that can be accommodated by the variable. You cannot directly address individual characters in a string; instead, you must use string functions.

```
{
char x[10],y[10];
x = "Hello, X";
y = x;
printf(y);
}
```

datetime

The *datetime* variable holds year, month, day, and time data. For example:

```
{
datetime x,y;
x = "10-JUL-90";
y = x;
printf(y);
x = to_date("10-JUL-90 12:43","DD-MON-YY HH:MI");
}
```

glob

The *glob* variable holds character or binary data up to 2^^31-1 bytes. It can be loaded from a database or clipboard and used to insert lob data into a database. For example:

```
{
glob glb;
list ll;
char buf[10];
db_command(db_cmd_desc_lobs,"YES");
ll = list_open("select col_int, col_clob from test",100);
glb = list_curr(ll,1);
if (glob_util(glob_char,blobfile)) buf = "Clob";
                                   else buf = "Blob";
printf("Type/len: "^^buf^^"/"^^glob_util(glob_size,glb));
exec_sql("insert into newtest(col_int,col_clob)
                      values(:1,:2)",42,glb);
}
```

Note that the db_command() must direct the database to return lob descriptors. The default is char/binary limited to 65k.

int

TRIMpl's *int* behaves the same as C's except that it accepts NULLs. The variable's byte size is machine-dependent. To define an *int*, you can declare:

```
{
int x = 1;
int y;
y = x;
printf(y);
}
```

list

The *list* variable is a dynamic *mxn* matrix of data. Each cell of the matrix can contain any variable datatype including other lists. You manipulate columns in each row with (`list_*()`) functions. For example:

```
{
list lvar;
```

```
lvar = list_open("1 1",5);              /* 2 columns */
list_mod(lvar,0,"Hello","World");
printf(list_curr(lvar,0),list_curr(lvar,1));
}
```

numeric

The *numeric* variable declares a floating point datatype. The variable uses 22 bytes and has a precision of 38 digits with a maximum exponent of +125. To define a *numeric*, you can say:

```
{
numeric x,y;
x = 1.25678;
y = x;
printf(y);
}
```

rowid

rowid's datatype is opaque, its contents depend on the database you query. You can move character data into and out of a rowid variable. However, use caution; if you change the value of rowid, it no longer references the same database row.

This example shows how to use a rowid value retrieved from a select statement in a following update statement.

```
{
list LL;
rowid myrowid;
LL = list_open("select rowid, name from staff", 100);
myrowid = list_curr(LL,0);
exec_sql("update staff set name = 'oldname'
          where rowid =:1",myrowid);
}
```

trigger

The *trigger* datatype, or variable, is a pointer to another block of code. For more information about writing and using triggers, read "***Variables and Triggers***" on page 63.

After you assign a code block to a trigger variable, you can pass the block to other trigger functions as a parameter. You execute a trigger variable by passing it to the `execute()` function. For example:

```
{
trigger tvar;
tvar = {printf("World");};
printf("Hello");
execute(tvar);
}
```

Number Operations

Use the *int* and *numeric* datatypes for numerical computation. *int* is a whole number and *numeric* can include floating point numbers. For example, `int x` can have values NULL or `+/-maxint` where `maxint` is machine-dependent; `numeric x` can have a value of NULL or a positive or negative real number.

Unlike variables in C, NULL is another state for a variable in TRIMpl. NULL is not zero or any other digit; however, for certain operations, TRIMpl treats a variable as such.

TRIMpl supports arithmetic, bit-wise, and boolean operations.

Arithmetic

Operation	int	numeric	Example
Auto-increment	yes	yes	x++;
Auto-decrement	yes	yes	x- -;
Unary negation	yes	yes	x = -x;
Addition	yes	yes	x = x + 9;
Subtraction	yes	yes	x = x-6;
Multiplication	yes	yes	x = x * 14;
Division	yes	yes	x = x / 2;
Modula	yes	no	x = x % 2;

Bitwise

Operation	int	numeric	Example
AND	yes	no	x = x& 4;
OR	yes	no	x = x | 255;
XOR	yes	no	x = x ^15;
One's complement	yes	no	x = ~n;

Boolean

Operation	int	numeric	Example
Logical AND	yes	yes	x = x && y;
Logical OR	yes	yes	x = x | | y;

Operations Involving Nulls

A NULL is treated as a 0 (zero) in all operations except for multiplication and division where NULL is a NULL. A non-NULL value divided by a NULL results in a division-by-zero error.

Datetime Operations

You can perform the following arithmetic operations on **datetime** variables.

Operation	Example
Auto-increment	d++;
Auto-decrement	d - - ;
Addition	d = d + 7;
Subtraction	D = d-365;

Although a datetime variable also includes time information, these operations alter only the day values.

Valid Dates

Valid datetime values are from 12:00 a.m., January 1, 0000 to 12:59:00 p.m. December 31, 9999, inclusive. Leap year days are automatically resolved. If you try to set a datetime variable with an invalid date (for instance, February 30, 1990) you receive an illegal data error.

Datetime Manipulation

You can convert *datetime* variables to *int* or *numeric* datatypes. The result of the conversion is the number of days since January 1, 0000. The fractional day is also returned for a *numeric*. However, you cannot directly convert ***ints*** and ***numerics*** back into datetime.

To convert an *int* or *numeric* to *datetime*, simply add the data to a datetime variable that contains JAN 01, 0000:

```
dt = to_date("JAN-01-0000","MON-DD-YYYY") + 728932;
```

Examples

```
/*
** Calculation of Day of the Week
*/
{
datetime dd;
char     day[9];
int      x;
dd     = prompt("Type in a date ==> ") /* Get a date from user  */
x      = to_int(dd) % 7;/* Any date MOD 7 returns 0-6; 0=Sunday */
day    = decode(x,0,"SUNDAY",   1,"MONDAY",  2,"TUESDAY",
                3,"WEDNESDAY",4,"THURSDAY",5,"FRIDAY",6,"SATURDAY");
printf(dd ^^ "is a " ^^ day);
```

```
}

/*
** Calculate Age in Days
*/
{
datetime dd;

dd = prompt("Please type in your birthdate (DD-MON-YY) ==> ");
printf("You are " ^^ to_int(SYSDATE-dd) ^^ " days old.");
}

/*
** Calculate Number of Days in Month
*/
{
datetime dd;
int      days,year,month,leap;

dd    = prompt("Type in a date (DD-MON-YY) ==> ");
year  = to_int(to_char(dd,"YYYY"));
month = to_int(to_char(dd,"MM"));

if ((!(year % 4) && (year % 100)) || !(year % 400)) leap = 1;
if (month in (4,6,9,11))   days = 30;
else if (month != 2)       days = 31;
else                       days = 28 + leap;

printf("That month contains " ^^ days ^^ " days");
}
```

Storing and Retrieving Datetime Data

Even though the default format mask only shows date, DV and TRIM store the entire datetime value it receives (which can include the time value). This feature can be misleading when for example, the application uses SYSDATE in calculations and does not account for timestamp data storage.

You can adjust the value to DATE, removing TIME with the following procedure:

```
mydate = to_date(to_char(SYSDATE, “DD-MON-YY”));
```

If your datetime mask uses only "YY" for the year, then the century value is read from the operating system. For example,

```
mydate = to_date("09-JUL-00","DD-MON-YY");
```

If your machine year is 1999, then mydate is 1900; if it is 2000, then mydate is 2000. If you must use two character year masks and there is a possibility of an ambiguous century, use the "RR" mask.

char/string Operations

C lets you manipulate a string one character at a time. TRIMpl treats a string as an entity. Constant strings are designated by double quotes (‘‘abcd...“ABC...“). To include a double quote in a string, simply place two double quotes together.

To copy a string from a source to a destination:

```
dest_buffer = source_buffer;
```

TRIMpl provides direct assignment for strings. Note, however, that assignment is not simply a pointer to the buffer but an actual copy of the contents of `source_buffer` to `dest_buffer`.

Concatenation (^^) is the only operation specifically for `char/string` variable datatypes:

```
{
char s1[10],s2[10],s3[20];
s1 = "Trifox"; s2 = "Inc";
s3 = s1 ^^ ", " ^^ s2;
printf(s3);                    /* result: "Trifox, Inc" */
}
```

NULLs

In a **char/string** a NULL value is a zero-length string. Thus, you can set a **char/string** variable to NULL in either of the following ways:

```
str1 = "";
```

or

```
str1 = NULL;
```

Auto-Truncation

TRIMpl automatically truncates the source string to fit the destination:

```
{
char src[10], dst[4];
src = "TRIMtools"; dst = src; printf(dst); * result: "TRIM" */
```

Token Pasting

To make long string constants easier to use, TRIMpl supports token pasting, the merging of two or more string constants into one string at generate time.

```
{
char dst[80];
dst = "TRIMtools "
      "are "
      "nice!";
printf(dst); /* result: "TRIMtools are nice!" */
}
```

Examples

```
/*
** Case Conversion
*/
{
char text_string[80];
text_string = "Trifox Tools";
text_string = translate(test_string,"abcdefghijklmnopqrstuvwxyz",
                                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ");

printf(test_string);            /* result: "TRIFOX TOOLS" */
}

/*
** String Token Parser
*/
{
int  i,j;
char src[80];

i = 1;         /* initialize index to 1st chararacter            */
src = prompt("Type in a string->");/* get string into SRC variable     */
while (i <= length(src)){          /* while index is within the string */
   if(substr(src,i,1)! = " "){    /* if index is not pointing to space*/
      for(j=0;(j+i) <= length(src);j++)/*then get length of the token */
         if(substr(src,(i+j),1)==" ") break; /* if space then break  */
      dst = substr(src,i,j); /* copy substring to destination        */
      printf("Token->" ^^ dst);                  /* print the token */
      i = i + j;                       /* adjust the index           */
      }
   i++;
   }
}
```

Symbolic Text

Text in dvApp applications such as column labels or text objects can be dynamically replaced at runtime by using symbolic text with the gui_config function. Prefix the text with “$” and use gui_config() to set the translation table. For example if a field label is set to “$MyLabel” and gui_config() is used *before* the window is opened

```
list ll;
list_mod(ll,1,”MyLabel”,”Compensation”);
gui_config(gui_config_textlist,ll);
```

then “$MyLabel” will be replaced with “Compensation”.

A new call to gui_config() with a different translation table will not take effect until the window is reopened. Also if a matching symbolic text is not found, then the original symbolic text, e.g. “$MyLabel”, will be displayed.

Glob Operations

While glob variables appear simple, there are limitations to their use because of their potential size. For example

```
{
glob glb1,glb2;
glob_util(glob_file,glb1,”SomeFile”);
glb2 = glb1;
glob_util(glob_file,glb1,”ADifferentFile”);
}
```

The assignment “glb2 = glb1;” does not make a copy of glb1 but rather glb2 is pointing to the same data as glb1. This means that when glb1 is loaded from a new file, glb2 is also pointing to that new data.

Also it is very important to understand that

```
glob_util(glob_free);
```

will free all globs in the entire session, not only variables but also any globs that are in lists.

Datatype Conversions

TRIMpl automatically converts datatypes in expressions where necessary. All conversions are performed "left-to-right." That is, all data in an expression is converted to the type of the first (leftmost) data. If you don’t want all the datatypes to be converted to the first type that appears, be sure to convert all datatypes in an expression.

For example, consider the following code:

```
{
char    buf[32];
numeric n;

n   =  .6020599913279624;
buf = "0.3010299956639812";

/*********************************************************************/
/* Example one                                                       */
/*********************************************************************/
if (buf > n) printf("Wrong");
else printf("Correct");
if (n < buf) printf("Wrong");
else printf("Correct");

/*********************************************************************/
/* Example two                                                       */
/*********************************************************************/
n   =  .6020599913279624;
buf = "0.6020599913279624";
if (n != buf) printf("Wrong");
else printf("Correct");
if (buf != n) printf("Wrong");
else printf("Correct");
}
```

In Example one, variable *n* converts to a char string when the

```
if (buf > n)
```

expression is evaluated. *n* converts to ".6020599913279624," and the comparison is performed on the character strings "0.3010299956639812" and ".6020599913279624". In this case, the automatic conversion assumes an inappropriate datatype: because the character "0" collates higher than the character "." (in ascii), "0.3010299956639812" is greater than ".6020599913279624".

In the second `if` statement

```
if (n < buf)
```

the character string in `buf` converts to a numeric and the result is the "correct" numeric answer.

Example two has the same problems. To avoid inappropriate datatype conversions, explicitly convert the data. For example,

```
{
char    buf[32];
numeric n;

n   =  .6020599913279624;
buf = "0.3010299956639812";

/*********************************************************************/
/* Example one                                                       */
/*********************************************************************/
if (to_number(buf) > n) printf("Wrong");
else printf("Correct");
if (n < to_number(buf)) printf("Wrong");
else printf("Correct");

/*********************************************************************/
/* Example two                                                       */
/*********************************************************************/
n   =  .6020599913279624;
buf = "0.6020599913279624";
if (to_number(buf) != n) printf("Wrong");
else printf("Correct");
if (n != to_number(buf)) printf("Wrong");
else printf("Correct");
}
```

Now you can clearly see the datatypes that are being compared.

Chapter 8
Working with Lists

TRIMpl lists are largely responsible for much of the language's power and flexibility. Very few operations exist that cannot be performed on items in TRIMpl lists and by placing data items in a list, you use local memory instead of accessing database resources every time the items are needed.

List Components

A list is a unique datatype that is very well suited for the dynamic nature of accessing relational data. A list variable is a dynamic *mxn* array of data. The data can be of any type, including triggers and lists. Internally the list has two parts:

- The list header.
- Dynamically-allocated data body.

The list header is a pointer to a data block and the current row of data and serves as the entry point to the data. The header is the object that is defined by a list variable. The data block contains the title of the list, the basic list structure, and the data within the data body.

As an example, a list variable, LL1, is assigned to a list. LL1 points to the data body. Also, LL1 has its current-row pointer pointing to data row 3. As long as only one list header (variable) points to the data body, any operation can be performed on the data to which LL1 points. However, when a second list variable, LL2, is assigned to LL1, the following occurs:

1. `LL2` points to the same data body as `LL1`.
2. Although `LL1` has current row set to row 3, `LL2` points to row 0.
3. The reference count for the data body is incremented by 1.
4. Because the reference count for the data body is greater than one, the data body becomes a read-only list. Any list modification operations, such as adding, deleting, or modifying records returns an error.

Aside from the number of columns, the data in the list is free-form. For example, data in column 0, row 0, of the list could be an integer value while data in column 0, row 1, of the list could be a datetime value. In fact, lists can be stored within lists.

No matter how many list variables are pointing to the list data body, only one copy of the data exists in memory. Having a single copy improves application performance by reducing redundant data.

Lists can also be shared across applications and users by storing them in shared memory. Lists in shared memory are inherently read-only and are usually used for storing static lookup tables such as zip codes.

Graphic Lists

Use a graphic list to display graphics in a window. Because it is implicitly declared when you create a graphic list object in your window, you do not declare a list variable for it. You can put as many graphic list commands as required into the list.

You can easily create dynamic graphs with text captions and so on just by combining these simple elements.

The format of the graphic list record is a single column with the following elements:

1. *id* — the object's id, usually 0.

   The object id is used to identify which graphic object has been clicked. The list trigger is called when the graphic object is clicked and the id is returned in the `g.AUX` predefined variable. If you do not need this feature, simply set the id(s) to -1.

2. *type* — the graphic object type.
3. *r* — the beginning row coordinate of the list object.
4. *c* — the beginning column coordinate of the list object.
5. *h* — see ***Type*** below.
6. *w* — see ***Type*** below.
7. *name* — the name of the bitmap that has been loaded into the image area by using `file_copy()`.
8. *width color fillcolor* — see specifics in each type description.

Type Definitions

The following graphic types are defined in `dv.h`. Beginning coordinates for all graphics list items are represented as `r,c` (0,0) in the upper left corner and are always expressed in pixels. The graphic types are:

- graphic_type_line
- ***graphic_type_rectangle***
- ***graphic_type_roundrect***
- ***graphic_type_arrow***
- ***graphic_type_ellipse***
- ***graphic_type_text***
- ***graphic_type_bitmap***
- ***graphic_type_poly***
- ***graphic_type_bezier***
- ***graphic_type_area***
- ***graphic_type_pie***
- ***graphic_type_file***

- ***graphic_type_popup***

All types except bitmap, text, file, popup, and area can have an optional line width and color. You specify the line attributes as

```
( width red green blue
```

`width` can be values from 0 to 5 and the the colors are identified with the standard RGB values from 0-255. Additionally rectangle, roundrect, ellipse, and pie can have an optional fill color

```
( width red green blue red green blue
```

graphic_type_line

To draw a line from (10,10) to (35,40), add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_line^^" 10 10 35 40");
```

To add some color, append the width and color option. The following presents a thin red line.

```
list_mod(structure,1,"0 "^^graphic_type_line^^"
          10 10 35 40 (1 255 0 0");
```

graphic_type_rectangle

To draw a rectangle with corners at (10,10) and (35,40), add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_rectangle^^" 10 10 35 40");
```

To add some color, append the width and color option. The following presents a thin green line:

```
list_mod(structure,1,"0 "^^graphic_type_rectangle^^"
                 10 10 35 40 (1 0 255 0");
```

To fill the rectangle with red, add a fill color:

```
list_mod(structure,1,"0 "^^graphic_type_rectangle^^"
          10 10 35 40 (1 0 255 0 255 0 0");
```

graphic_type_ellipse

The ellipse is bounded by an imaginary rectangle defined by the points `r,c,h,w`. To draw an ellipse that fits inside the above rectangle, add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_ellipse^^" 10 10 35 40");
```

To make the line blue and the inside of the ellipse purple, add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_ellipse^^" 10 10 35 40
          (1 0 0 255 255 0 255");
```

graphic_type_text

This type does not use `w,h;` the actual text string follows `c`.

graphic_type_roundrect

This command draws a rectangle similar to *graphic_type_rectangle* but with rounded corners. To draw a blue rectangle with rounded corners at (10,10) and (35,40), add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_roundrect^^"
          10 10 35 40 (1 0 0 255");
```

graphic_type_bitmap

To present an image, first load the image into the internal graphic memory and then reference that name in the graphic list. If you do not define `h,w` for a this type, then the graphic fills the entire graphic list area.

For example, if the window list object is called `structure`, the following code loads it with a bitmap stored in file "`structure1.bmp`".

```
file_copy("structure1.bmp","gui!image:structure1.bmp");
list_mod(structure,1,"0 8 0 0 0 0 structure1");
list_view2(structure,-1,-1,-1,0);
```

graphic_type_arrow

This is the same as a *graphic_type_line* except that it has an arrow at the second point (w,h).

graphic_type_poly

This type draws a line that connects all the points given in the list record. To draw a line through (10,10), (20,20), (20,40), and (55,66), add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_poly^^"
          10 10 20 20 20 40 55 66");
```

To make it a green line, add the width and color:

```
list_mod(structure,1,"0 "^^graphic_type_poly^^"
   10 10 20 20 20 40 55 66 (1 0 255 0");
```

graphic_type_bezier

This option draws a Bezier curve. You need to specify a minimum of four points and then you can specify sets of three points. To draw a Bezier curve through the points (10,10), (20,20), (20,40), (44,55), (75,71), (99,101), and (111,123), add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_bezier^^"
          10 10 20 20 20 40 44 55 75 71 99 101 111 123");
```

graphic_type_pie

This option draws an arc. You need to specify a starting point, radius, start angle, and span. The start angle is the number of degrees counter-clockwise from the x-axis to begin

the pie. The span is the degrees of the arc. To draw a pie starting 30 degrees from the x-axis and spanning 45 degrees, add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_pie^^"
          100 100 20 30 45");
```

To outline the pie in black and fill it with red, add the following line to the graphic list:

```
list_mod(structure,1,"0 "^^graphic_type_pie^^"
          100 100 20 30 45 (1 0 0 0 255 0 0");
```

graphic_type_area

This type creates an active area which returns its id when the user left-clicks anywhere within the defined area, unless another graphic object is under the cursor. The graphic list's trigger executes and the area's id is returned in g.AUX. The boundaries of the area are defined the same as for graphic_type_rectangle.

graphic_type_file

This type saves the graphic objects defined in the preceding list rows to a file. For example,

```
list_mod(structure,1,"0 "^^graphic_type_file^^
       " gui!c:\tmp\myfile.bmp");
```

writes the bitmap image described in the structure list to c:\tmp\myfile.bmp.

graphic_type_popup

This type creates a popup menu attached to the graphic list. It is unique among graphic list items in that it does not have beginning coordinates. It appears based on where the cursor is when the right mouse button is pressed. The format is

```
id graphic_type_popup  level  text
```

level is an integer used to group submenu items where 0 is the top level. For example, to create a popup menu with items

```
File...
   Open
   Save
```

the list entries are

```
list_mod(structure,1,"1 "^^graphic_type_popup^^" 1 Open");
list_mod(structure,1,"2 "^^graphic_type_popup^^" 1 Save");
list_mod(structure,1,"1 "^^graphic_type_popup^^" 0 File");
```

The submenu items are presented first and then the containing item. There can be multiple submenu levels. The id value for the containing item does not have to be unique as it will never be returned.

Creating Lists

You can create a list using one of three basic functions:

- `query()`
- `list_mod()`
- `list_open()`

Query()

You can only use `query()` in window applications (DVapp designs). When an application calls `query()` it executes the current window's SELECT statement and assigns all the returned information to that window's preallocated *`window-name`*`.WL` list variable. If there is an old list data body, it is replaced by the new one.

list_mod()

Using `list_mod()` on a NULL list creates a list data body based on the number of columns in the `list_mod()` call.

`list_mod()` is typically used to add new records to an existing list. For example, if list `LL1` points to a two column list, then

```
list_mod(LL1,1,"Hello","Dolly");
```

adds a new record to the list after the current record, if any, with the values "Hello" and "Dolly" in the first and second columns, respectively.

If `LL1` is NULL, meaning that it does not point to an existing list data body, then the above example first creates the list data body and then inserts a new record.

list_open()

`list_open()` lets you define an empty list or load a list from a file, directory, database or shared memory. You can specify any of the following:

- Column definition
- SQL SELECT statement
- File
- Directory and filename mask (see "***Directory***" on page 60).
- Control list
- Shared memory

The `list_open()` function definition is

```
list list_open(spec,limit[,title[,...]])
expr        spec
int         limit
string      title
```

The `spec` parameter is where you specify the different methods.

Column Definition

This approach creates a list data body using a series of blank separated numbers. The list data body has one column for each blank separated number. The values are only used

during display operations, `list_view(), list_view2(), list_view3()`, to determine the display column widths. The list data body is empty.

```
LL1 = list_open("1 4 8 2 6",100);
```

creates a list data body with five columns.

SQL SELECT

You can use a SQL SELECT statement to create a list data body with as many columns as there are select-list items. The list data body has the minimum of the select result table or `limit` number of rows. For example, if the STAFF table has seven columns and 35 rows,

```
LL1 = list_open("select * from staff",100);
```

creates a list data body with 7 columns and 35 rows. You can also use the *SQL* keyword to specify a resultset if your SQL syntax does not begin with the *SELECT* keyword. For example,

```
LL1 = list_open("sql select * from staff",100);
```

File

You can also create a list data body from a file, either ASCII or a previously stored binary list. `spec` contains the filename to open. If the file is ASCII, then a one-column list data body is created with a row for each line of the file. If the file is a previously stored binary or XML list, then an exact replica of that list is created.

Control list

If the ASCII file is delimited, then `spec` can contain a control list that describes the columns in the file. You can describe both variable and fixed column files as well as numeric and datetime formats.

Shared memory

Finally, a list header can point to a shared memory list. The `spec` for this approach is

"**S** *listname*"

The “S” tells `list_open()` to look in the shared memory control file for a list called *`listname`*. Shared memory lists are read-only and are typically used for static data.

Partial List Loading

Specifying a limit on the `list_open()` or `query()` function that is less than the amount of data available in the database or file creates an incomplete load. When a list variable is assigned to a list and the last data a row is copied to the list variable’s data body, an internal flag is set. This flag indicates that the information in the list is complete.

The `list_eos()` function returns `1` when the maximum or fewer rows have been loaded from a source or `0` when the loading function reaches its limit before all the data can be read in.

You can use `list_more()` to continue loading a list from the source used to create the list.

```
{
list ll;                        /* Assume staff has 30 rows of data */
                                /* Select 4 rows into list          */
ll = list_open("SELECT * FROM staff", 4);
printf("1. Number of data rows = "^^list_rows(ll));
printf("2. End of load status = "^^list_eos(ll));
list_more(ll,100,false);        /* Load up to 100 more rows of data */
printf("3. Number of data rows = "^^list_rows(ll));
printf("4. End of load status ="^^list_eos(ll));
}
```

When you open a list, the load limit begins at 4. However, `list_more()` allocates additional space as needed. Also, if you have set the clear flag in `list_more()` to true instead of false, then the line 3 prints that the list contains 26 rows. `list_more()` also works when the data source is a file. If the source is a file, `list_more(11,100,false);` reads the next 100 rows. It then resets the current row pointer to the first new row.

Status Information for a List Row

Each data row entry in a list has an integer variable indicating its list status. The status is used in a variety of ways by different functions. For example, if a SELECT in a `list_open()` returns a number of data rows, the rows all have their status set to `1`. If `list_mod()` modifies a row, the status is set to `2`. You access a row's list status through `list_stat()`.

The list status can be used in a variety of ways, but you must be careful since certain TRIMpl functions affect the value. Using `list_stat()` on a NULL list variable generates an error. If you use it on an empty list, `-1` is returned, otherwise the correct status is returned. Also, `list_vis()` returns a count of non-zero list status data rows.

Saving a List

Lists can be saved to files, either ASCII, binary, or XML, or to shared memory using `list_file()`. The entire list can be saved or specific columns can be omitted.

Saving a list as ASCII creates a text file with all data items lined up. The list's structure is not preserved in the text file. This means that when the file is used to create a new list using `list_open()`, the new list has only one column. If you specify the binary option in `list_file()`, the list's structure is preserved as is the datatype of each item. The binary file is in a machine-independent format. Finally if you specify the XML option in `list_file()`, the list is stored in XML format. If you subsequently use `list_open()` on the file, the new list will preserve the original list's format.

If the `file` specification is in the form "**S** *listname*", then the list is filed in shared memory as *listname*. A subsequent `list_open()` using "**S** *listname*" points to this list data body.

The following example creates a separate copy of the subset of the data in a list.

```
{
list ll; list gg;
ll = list_open("SELECT * FROM staff",100); /* Retrieve data into a
                                            list */
printf("number of columns in LL = "^^list_cols(ll));
list_file(ll, "temp.out","b",0,3,4);        /* Save a subset of the
                                             list to temp.out */
gg = list_open{"temp.out",100);          /* Load and assign gg from */
the printf("number of columns in GG = "^^list_cols(gg)); /* file */
```

```
}
```

When you use `list_file()`, the column options (0,3,4) only indicate which columns are to be used in creating the sublist, and not the order in which they appear in the list. The order is always low to high. Thus, the following statements are equivalent:

```
list_file(ll, "temp.fil", "b", 1,2,3);
list_file(ll, "temp.fil", "b", 3,1,2);
```

Remember:

- To preserve the columns when saving a list, save it as a binary file: `list_file(ll, "tt.file", "b");`
- To merge all columns into one string for every data row in the list, save the list as an ASCII file: `list_file(ll, "tt.file", "a");`

List Reference

A reference count in the list data body keeps track of how many list variables are referencing it. If the count is greater than one, then the list cannot be modified. You can reference lists in a number of subtle ways. For example, if you pass a list as a parameter to a function and assign it to a local list variable, the reference count is incremented. Even when control leaves that function, the reference count is not decremented because all TRIMpl variables are static. You must explicitly close the list, either by assigning the variable to NULL or by calling `list_close()`.

If you need to create a list in a function and pass it back to the caller, use `list_close()` in the `return` statement. For example,

```
{
list LL1; LL1 = list_open("select * from staff",100);
return(list_close(LL1));
}
```

The `LL1` list variable is closed but the list data body is not destroyed. Of course, the calling TRIMpl code must assign the returned list to a list variable.

Referencing a list passed to a function as its parameter marker, that is, *parm[0]*, for example, causes no new list reference to be made. This method lets you manipulate the current row index information in the original list.

Getting Information on Lists

You can get information about the list, not just its contents. These functions operate on elements of a list:

Function	Returns
`list_colix()`	Column position given name.
`list_colname()`	Name of a column in a list.
`list_cols()O`	Number of columns in a list.
`list_eos()`	Load status of file.

Function	Returns
`list_pos()`	Current row index value.
`list_rows()`	Number of rows in a list.

Navigating Through Lists

To reference current row pointer information, use `list_pos()`. If two list variables are assigned to the same data body, each has its own independent current row pointer.

When a list variable is first assigned, the list variable's current row pointer is set to the first row in the data body (row 0) if it exists, or `-1` if the list is empty (list outline).

Most `list_*` operations are performed on the current row or a column within the current row. If you want to implement vector operations you must create a user function.

A number of functions change a list variable current row index. However, the following functions are primarily geared to performing just those functions:

Function	Moves
`list_seek()`	To an absolute data row.
`list_next()`	To the next data row. You can also use the ++ operator on the list variable.
`list_previous()`	To the previous data row. You can also use the -- operator on the list variable.

Example

If the current row index is set to data row 0 and `list_previous()` is called, no operation is performed. Also, if the current row index is set to the last data row in a list, no operation is performed when `list_next()` is called. However, using `list_seek()` to find a row greater than the number of rows in the list, generates a "list row out of range" error.

```
{
list ll; /* Assume staff has 30 rows */
printf("The Current Row Index = "^^list_pos(11));
ll = list_open("SELECT * FROM staff", 100);/* Select all columns */
                                           /* from staff table */
printf("There are now "^^list_rows(ll)^^ "rows loaded");
printf("The Current Row Index (with data) = "^^list_pos(ll));
list_next(ll);                             /* Move current row to */
                                           /* next */
printf("The Current Row Index (after list_next()) = "^^list_pos(ll));
list_previous(ll);                         /* Move current row back*/
                                           /* one */
printf("The Current Row Index (after list_previous()) =
    "^^list_pos(ll)); list_seek(11,9);     /* Move to the 10th data*/
                                           /* row*/
printf("The Current Row Index (after list_seek()) = "^^list_pos(ll));

}
```

Displaying (Viewing) Lists

You can use one of three `list_view()` functions to display lists in their basic format.

These functions draw a box on the screen and display a list of information. End users can scroll the window using directional keys ([Up], [Down], etc.) to see all the list data and select rows from the list.

The functions not only return a selected row's column, but also set the current row index to that row and may set the list status to be a value of `item_select()` or `item_tagged()`, `list_view2()`).

Terminal Manager

Calling a `list_view()` function invokes the built-in terminal manager. (The terminal manager is always called when a DVapp application starts running, but is only called when needed by a stand-alone TRIMpl design.) The terminal manager causes TRIMrun to search for a file called *`filename`*`.KEY` which contains a terminal initialization string. The correct key file must be available when using these functions in stand-alone applications.

Note that the current row index is set to the last data row indicated by the cursor when the user exits a function.

Rules for list_view()

When you work with `list_view()` functions, remember:

If …	Then …
Data in a list row / column is not viewable (for example, a list variable, trigger variable, or row id value)	That column in the row appears as blank spaces.
A row in a list has its status set to `item_delete()` and is not displayed	It is still included in calculations of the current row position.
The list is empty or all its rows have their list status set to `item_delete()`	The function returns a null value.

All the `list_view()` functions draw boxes to display data following these rules:

If …	Then …
You give coordinate or size values that exceed the screen dimensions	The function to use internal maximum values.
You use "0,0" as the "column,row" origins	The top and left sides of the box are suppressed.
You specify one or more optional `view_cols()`	All columns are displayed.
You don't specify view column options.	All columns are displayed.
Rows of data in a list can't all fit into the box or window	A plus sign (+) appears at the top or bottom left corner of the box to indicate that more data is above or below the window contents.

Choosing the Correct list_view

TRIMpl has three `list_view*` functions that have different strengths. The following table provides direction for you.

Use …	To …
`list_view()`	Find which column's data in the selected row is to be returned when the user presses [Enter].
`list_view2()`	Tag multiple items.
`list_view()`, `list_view3()`	Tag single rows.
`list_view3()` with browse option	Display a popup informational window (no input).
`list_view2()`	Return a value between 0 and 31 when the predefined exit key is pressed. This function allows a user to toggle a row's list status between 4 and 1 by pressing [Enter].
`list_stat()`	Pre-tag rows (sets the row's list status value to 4).
	Place and size the box/window.
	NOTE: If the exit key is set to [Enter], then `list_view2()` behaves just like `list_view()` but does not return a value. It does update the current row index.
`list_view3()`	To let users select only one item and return the `rtrn_col` value and update the current row index. If `abt_key` is pressed, NULL is returned and the current row value remains the same. This function also supports options, which are currently defined in `trim/dv.h` as "***List_view3 options***".

Reading Data from a List

You can read data from a list in several ways. The previous section on `list_view()` describe how to select data from a visible list and return it via a function. The following functions let you extract information from a list. Note that all operations affect only the data row specified by the current row index.

Use …	To …
`list_curr()`	Read a column from a list.
`list_find()`	Search column for a key.
`list_get()`	Get multiple columns from a list.
`list_ixed()`	Read list at an absolute position.

Use …	To …
`list_read()`	Read columns from a list.

Neither `list_find()` nor `list_ixed()` operate on the current row index. `list_find()` moves through all the columns searching for the key string and returns that value when it finds a match. `list_ixed()` also circumvents the current row index and reads a column at a given row index.

`list_curr()` and `list_read()` differ in that the latter advances the current row index.

`list_get()` allows for data within the columns of the current row to be moved out into appropriate variables. The number of variables must be equal or less than the number of data columns in a list row.

Deleting, Inserting, and Updating Rows

You have three basic functions for manipulating data rows in a list: `list_mod()`, `list_modcol()`, and `move_f2l()`.

list_mod()

Use this function to insert, update, and delete records in the list. When you INSERT, the current row index is set to the newly inserted item. When you DELETE, the current row index is moved forward, if possible. When an item is deleted, it is physically removed from the list. (This action is different than setting the list status to `item_delete`.) By DELETING all items, you can empty a list without changing its structure. When you UPDATE, the current row index stays on the updated record.

list_modcol()

To modify a single column in a current data row, use `list_modcol()`. This function alters the list status for a row entry to `item_update`.

move_f2l()

For window applications, `move_f2l()` provides a simple method of moving the window field data to the window list. It keeps track of which fields are list fields, whether there is a unique row identifier in column 0 of the window list and so on.

Summary

Remember the following when using list variables:

- A list variable is a pointer to a data body.
- Data rows begin with 0 and end with *n*-1.
- Data columns begin with 0 and end with *n*-1.
- DVapp window lists (window list variables) use column 0 as the row identifier. (You can turn this off by `Atl-click on the window > Attributes> Window Atr Dft Sel`.)

- When references exceed 1, lists become read-only.
- A list can contain any variable type in any cell.

Chapter 9
Using TRIMrpc

If you have been struggling with how best to make use of TRIMpl portability, you'll welcome the Remote TRIMpl enhancement to DesignVision functionality, called TRIMrpc.

Now you can access TRIMpl application code (the RPC— remote procedure call) from any machine on the network. Any programs you created with TRIMpl, including calculations, text manipulation, and access programs, for example, become transparently available to any other TRIMpl "client" on the same network.

Maintaining multiple versions for platform compatibility, moving and updating files are activities of the past. Simply connect through the new driver and call the code. Reuse gets a new meaning for TRIMpl programmers and the systems on which they work.

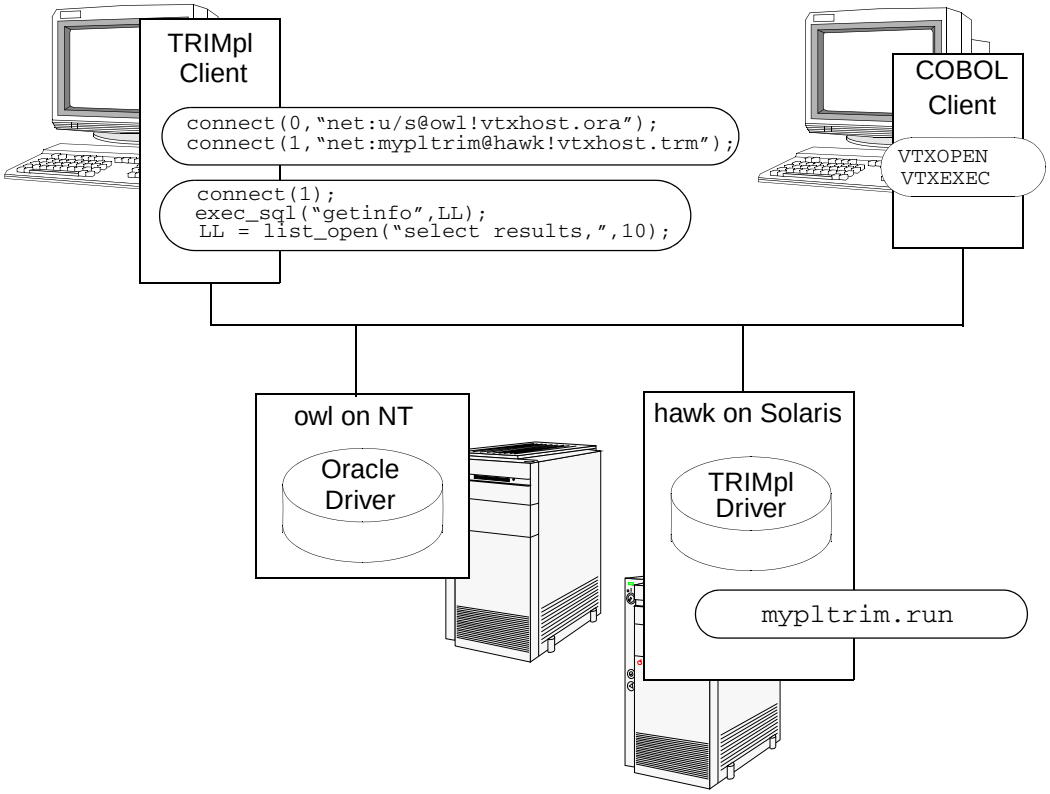
`vtxhost.trm` is a complete TRIM runtime. Because the variables are kept in a static variable space, they remain the same for all calls. You can use persistent stored procedure calls and be confident that they will run the same on all platforms with all development languages.

Creating the TRIMpl Application

To access your useful TRIMpl applications that do not require screen input and output (reading values from a screen or writing to a screen is not currently available through TRIMrpc), you must include the following new `#defines` in the `trim/dv.h` file when you create the executable runtime:

```
#define  db_connect                    1
/* connect to data base*/
#define  db_release                    2
/* release database*/
#define  db_commit                     3
/* commit work*/
#define  db_rollback                   4
/* rollback work*/
#define  db_open                       6
/* open cursor &describe*/
#define  db_exec                       11
/* execute SQL statement */
```

How TRIMrpc Works



TRIMrpc works like VORTEX drivers in your enterprise. Your client issues a connect, this time to a TRIMpl application on a networked machine, and calls the RPC when it needs the stand-alone application's functionality.

Client Functions

The client can issue one of five TRIMpl functions (which are completely documented in the ***TRIMpl Reference Guide***) which are translated to one of six new `#defines` for the RPC. The RPC performs its tasks and returns requested information to the client. In this process, either the client or the RPC application can use any database on the network for which it has access and authority.

Releasing the database is implicit when a client exits or when it reuses an existing connect ID for a new connection. You can pass up to two parameters (depending on the function) in your call.

- `connect()` — Connect to a database using a standard connect string.

  ```
  connect(0, "net:mypltrim@hawk!/usr2/bin/vtxhost.trm");
  ```

  The protocol must be "net" since the RPC is on a remote machine. The second element in the connect string is the RPC application name without the `.run` extension. Following the exclamation mark, put the fully-qualified path to the executable.

- `exec_sql()` — Issue a command to the remote procedure code (RPC).

  ```
  exec_sql("getdata", LL);
  ```

- `list_open()` — Get results from the RPC.

  ```
  ll = list_open("select results", 10)
  ```

- `commit()` — Commit the work.
- `rollback()` — Roll back the work.

Parameters

When you write the TRIMpl stand-alone application, use the `#define`s according to their purpose. The RPC receives the calls as a series of four parameters.

list_open()

This function translates into the following four parameters:

parm[0] — db_open

parm[1] — A command string that must begin with SELECT.

parm[2] — This (optional) parameter contains a list of parameters.

parm[3] — If the function doesn't return an error, the RPC returns a list to the client. The list must have at least one row.

The driver automatically executes a `list_more(list,n,true)` where *n* is twice the number of records that could fit in the fetch buffer.

If the columns in the list have names these are used in the describe. Otherwise, the RPC uses "COL*n*" to describe the columns.

exec_sql()

parm[0] — db_exec

parm[1] — Command string.

parm[2] — This (optional) parameter contains a list (which could be multi-row) of parameters passed from the client's `exec_sql()` call.

parm[3] — If the function doesn't return an error, the RPC returns an integer. You control the integer's meaning in the RPC.

commit()

parm[0] — db_commit

parm[1] — An integer that indicates if a write transaction should follow a commit/rollback.

rollback()

parm[0] — db_rollback

parm[1] — An integer that indicates if a write transaction should follow a commit/rollback.

For example for the following TRIMpl client call

```
exec_sql("cmd",1958)
```

The RPC receives:

parm[0] is db_exec
parm[1] is “cmd”
parm[2] is a list with one row and one column with a value of 1958

Error Handling

If the client’s TRIMpl generates an error, the TRIMrpc driver traps it and returns a database error. Use `error()` to explicitly pass back an error message.

Example

TRIMpl client:

```
{list LL;
         LL = list_open(prompt("Enter list_open parm ==>
"),10000);
            list_view(LL,0);
}
```

Driver:

```
{list LL;
if (parm[0] == db_connect) connect(0,"net:niklas/back");
else if (parm[0] == db_open)    LL = list_open(parm.1,1000);
else                             LL = NULL;
return(LL);
}
```

Chapter 10
Debugging & Compiling

Because both the character-based and window-based run times are identical, the debugger functionality is the same between the character-based and window version. However, presentation differences between the character-based and the window versions do exist.

The character-based debugger is a full-screen application with both a menu-driven and a command-line interface. All commands are available via either method.

Using Debugger from DVapp

Running the debugger from Designer

You can run the debugger from your interactive *Designer* session or compile the `.gap` file with debugging information. If you compile the application, the debugger window appears automatically.

To run the debugger from your interactive *Designer* session:

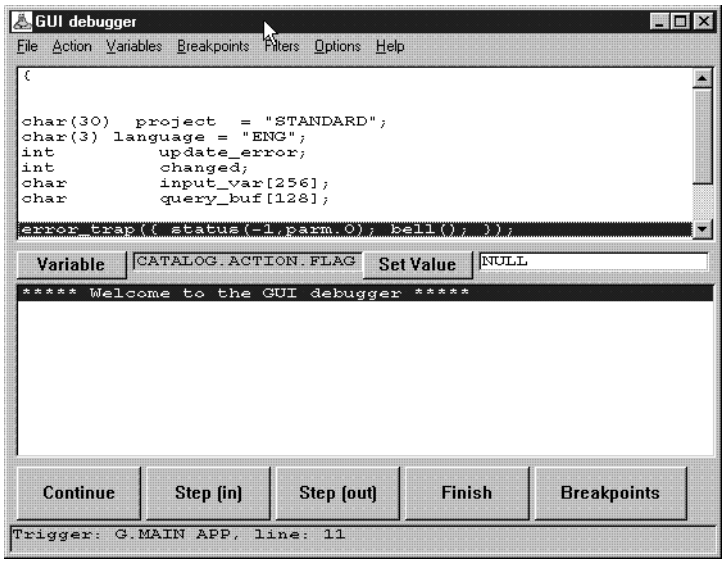
1. Start DVapp and open an application (`.gap` file).

   File > **Open**, choose *filename.*

2. **File** > **Run Options**, check **Debug**, click **OK**.

3. Run the application by clicking the run icon.

   And step through your application to debug errors.



File

Restart Restarts the application.

Quit Quits the application.

Action

Errors Displays the last errors, both database and non-database.

Traceback Displays the stack trace from the current trigger to the beginning of the application.

Variables

Tag Presents a list of all the application's variables. You can tag multiple variables to be displayed using the *Display tagged* menu item. See also `Options > Auto display tagged variables`.

Display tagged Displays the value(s) of the tagged variable(s).

Watchpoints Presents a list of all the application's variables. You can tag multiple variables to be watched. Execution stops before any of these variables are modified by TRIMpl code.

Display watchpoints Displays the value(s) of the watchpoint variable(s).

Parameters Displays the value(s) of the parameter(s) to the current trigger.

Breakpoints

Lines Presents a list of the current trigger's code lines. You can tag multiple lines where execution stops. See also the ***Breakpoints*** action button.

Triggers Presents a list of the applications triggers. You can tag multiple triggers where execution stops.

Reset all Resets all breakpoints.

Filters

Variables Presents a dialog where you can provide a filter to use when displaying the list of variables.

Triggers Presents a dialog where you can provide a filter to be used when displaying the list of triggers.

Watchpoints Presents a dialog where you can provide a filter to use when displaying the list of variables.

Options

Auto display tagged variable(s)	If set, the values of all tagged variables are displayed whenever control returns to the debugger. See also `Variable > Tag` menu item.
Display lists	If set, list variables are presented in a `list_view()` window.
Stop on non DB error	If set, execution stops on any non-database error.
Stop on DB error	If set, execution stops on any database error.

Buttons

Continue	Releases the debugger until the next breakpoint, watchpoint, or error if the *Stop* option is set.
Step (in)	Steps into functions.
Step (out)	Steps over functions.
Finish	Runs to the end of the current trigger.
Breakpoints	Shortcuts to the `Breakpoints->Line` menu item.
Variable	Performs a quick variable check.
Set Value	Modifies a variable. Type in the new value and click the button.

Running Character Version

The menu and list-driven debugger appears in a window that you can move on your screen.

When you enter the debugger, the executing trigger becomes the current trigger. To change the current trigger, use *SetCurTrg*. Breakpoints are always turned on and off via the current trigger and you can set watchpoints on all variables.

You can view variables one at a time, or in groups using the tagged variable option.

The last 32 lines of debugging information are always available via the message scroll commands.

Depending on the traps and go_field statements, QUIT itself can be trapped with unexpected results. Just press **QUIT** again.

Command Syntax

The commands are in the following general format:

[*n*] **command** [*parm* [*parm* ...]]

where:

- **command** — is the command to perform.
- *n* — is the number of times to perform command. Default is 1.
- *parm* — is an optional parameter (depends on command).

Cmd	Description
t	Prints a trace back.
s	Single steps through every statement.
n	Single steps through every statement in current trigger.
c	Continues execution.
r	Runs (restarts).
q	Quits.
f	Finishes current trigger (stops at return or end of trigger).
g	GOTO statement (to a statement number). Be careful when you leave a multi-statement line. Another goto (to the same line) moves to the correct line.
h	Hides the debugger window. Press [Return] to bring back debugger windows.
p	Prints a variable (you can use an optional index). If you do not provide additional parameters, a list is presented. If a string contains a '%' a like is performed in order to position the current item before presenting the list. If 'PARM' is given, then all the parameters are displayed.
v	Sets a variable's value (optional index can be given). If no additional parameter is given a list is presented. If a string contains a '%' a like is performed in order to position the current item before presenting the list.
d	Displays tagged variables.
m	Sets (mark) tagged variables. Use '-' to reset all.
w	Sets a watchpoint on a variable (optional index can be given). Use '-' to reset, '-' with nothing else resets all.
b	Sets a breakpoint at line number (optional trigger can be given). Use '-' to reset, '-' with nothing else resets all. When a trigger name is given the breakpoint is automatically set on the first executable line. If you don't specify parameters, the cursor is positioned within the code area and [Return] is used to toggle the breakpoints. An [F3] returns you to the command area.
x	Invokes old debugger. Should only be used by Trifox personnel.
e	Prints last error and last database error.

Cmd	Description
o	Displays / sets list of flags: • `stop on DB error` • `stop on non DB error` • `stop on Ctrl-C` (cannot continue after this) • `auto display tagged variables`

TRIMreport and Stand-Alone Applications

Most errors result in the application terminating. The exceptions are `exec_sql()` and `exec_row()` functions that are within expressions as well as TRIMpl code in the `trap()` function.

DVapp Performance Tuning

The DVapp and TRIMpl toolset, designed for heavy OLTP applications, are oriented to achieve top performance. However, nearly any TRIMpl application can benefit from a second "look." DVapp has a built-in profiler to help you with that second look. It helps you identify where the application processing time is spent.

In addition to looking at TRIMpl "gridlocks," the profiler can help you evaluate other design and programming considerations, such as:

- ***Lists*** — Each list object manages its own memory space. When you delete a row from a list, its memory is moved to a free list attached to the list object. Thus when you create a new list row or modify an existing list data column, TRIMpl looks through this free list for memory first. If you perform many deletes and your modifications are larger than the deleted data items, this free list may become large. To clean up the free list you should issue a `list_copy()` from time to time.
- ***Memory use*** — You can determine the memory use number only for an entire application and identify where the memory use increases by running the application with different data or a different number of cycles. DVapp never returns memory to the operating system so your application should reach a plateau rather quickly.

Using the Profiler

To activate the profiler, specify "`-p`" when you TRIMgen your application file (either the TRIMpl `.pl` or the DV `.gap`). TRIMrun automatically creates a profile file called `application.prf` when the application is run. The `profile.pl` TRIMpl script creates a report from this file showing the inclusive and exclusive times.

To determine how much memory your application is using, specify ***-m*** on the DVrun command line.

1. Generate your application file, for example

   ```
   dvgen myexample.gap -p
   ```

2. Run the application with its parameters:

```
trimrun.net myexample -p input output
```

3. Examine the automatically-created `.prf` file with either a simple text editor or the profiler application.

   If you choose to review `myexample.prf` with a text editor, you'll see four columns which are, from left to right: the count, the time in a specific trigger and anything that triggers has called, total time in the named trigger only, and finally, the trigger name.

Profiler Script

You must compile the profiler application, using the following code.

```
{
/
*****************************************************************/
/* Message raw profiling data                                   */
/*****************************************************************/
int     i,j;
numeric n,m;
numeric sum_exc = 0.0;
char    line[256];
list    LL,L2,PL;

/*****************************************************************/
/* Read in the raw data                                         */
/*****************************************************************/
list_mod(L2,1,parm.0);
list_mod(L2,1,"variable");
list_mod(L2,1," `");
list_mod(L2,1,"I");
list_mod(L2,1,"N");
list_mod(L2,1,"N");
list_mod(L2,1,"C");

PL = list_open(L2,10000);

list_seek(PL,0);
for (i=list_rows(PL);i;i--) sum_exc = sum_exc + list_read(PL,2);

/*****************************************************************/
/* Calculate the percentages                                    */
/*****************************************************************/
LL = list_open("5 9 7 9 7 72",0,"Count   "
                                "Inclusive times   "
                                "Exclusive times  "
                            "Trigger name
"
                             "
");
list_seek(PL,0);
for (i=list_rows(PL);i;i--) {
  list_mod(LL,1,list_curr(PL,0),
```

```
                to_char(list_curr(PL,1),"9999.9999"),
                to_char(100.0*list_curr(PL,1)/sum_exc,"999.99%"),
                to_char(list_curr(PL,2),"9999.9999"),
                to_char(100.0*list_curr(PL,2)/sum_exc,"999.99%"),
                list_curr(PL,3));
  list_next(PL);
  }

/
*****************************************************************/
/* Setup option list                                            */
/*****************************************************************/
L2 = list_open("30",0,"Profiling data for '" ^^ parm.0 ^^ "'",
                      "Total time: ~" ^^ sum_exc ^^ " seconds","",
                      "Options");
list_mod(L2,1,"Sort by inclusive times");
list_mod(L2,1,"Sort by exclusive times");
list_mod(L2,1,"Sort by invocation count");
list_mod(L2,1,"Sort by name");
list_mod(L2,1,"View list");
list_mod(L2,1,"File list");
list_mod(L2,1,"Quit");
list_seek(L2,0);

/*****************************************************************/
/* Command loop                                                 */
/*****************************************************************/
while (true) {
  list_view(L2,0); i = list_pos(L2);
  if (i == (list_rows(L2)-1)) break;
  if      (i == 0) list_sort(LL,1,!confirm("Sort
order","Ascending",true));
  else if (i == 1) list_sort(LL,3,!confirm("Sort
order","Ascending",true));
  else if (i == 2) list_sort(LL,0,!confirm("Sort
order","Ascending",true));
  else if (i == 3) list_sort(LL,5,!confirm("Sort
order","Ascending",true));
  else if (i == 4) { list_seek(LL,0); list_view(LL,0); }
  else if (i == 5) { line = prompt("Enter filename ==> ");
                     if (line != NULL) list_file(LL,line,"a"); }
  }
}
```

AppendixA
Data Dictionary

The starting point for implementing the DVdd is the DVdd itself.

The following four tables provide the foundation:

- TDD_CATEGORY
- TDD_DOMAIN
- TDD_LANGUAGE
- TDD_TRIGGER

Entries in all other DVdd tables eventually rely on definitions in one or more of these four tables. Since these four tables are the foundation, or root, of the DVdd, they must have entries before you can make entries in the other DVdd tables. The DVdd management applications enforce this relationship.

Every DVdd table has a version stamp in its first column that represents the current version with a number greater than one. Old versions have negative numbers. You can find the dates that correspond to each version in the table TDD_VERSION.

TDD_CATEGORY

This table lets you categorize DVdd entries by application group and project.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
CAT_NAME	_TDD_IDENT	C	30	N		Name
CAT_TYPE	_TDD_CODE	I	2	N		**Values** 0 Project 1 Application Group
CAT_DESC	_TDD_DESC_SHORT	C	40	Y		Brief description.

The following tables depend on TDD_CATEGORY:

- TDD_CODE.COD_CAT_PROJECT
- TDD_FORMAT.FMT_CAT_GROUPID
- TDD_FORMAT.FMT_CAT_PROJECT
- TDD_HELP.HLP_CAT_PROJECT
- TDD_LABEL.LAB_CAT_GROUPID
- TDD_LABEL.LAB_CAT_PROJECT
- TDD_TEXT.TXT_CAT_GROUPID
- TDD_TEXT.TXT_CAT_PROJECT

TDD_DOMAIN

You must specify all object datatypes in the TDD_DOMAIN table. Modifying a datatype description in this table ensures that all instances of that object are changed.

For example, the LAST_NAME domain might have originally been defined as char(20) but now must be changed to char(30). You simply make the change in the TDD_DOMAIN table and regenerate your applications and tables.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	1	4	N		Version stamp.
DOM_NAME	_TDD_IDENT	C	30	N		Name
DOM_DTY	_TDD_CODE	I	2	N		**Values**: 0 Integer 1 Character 2 Numeric 12 Datetime
DOM_LEN	_TDD_LEN	I	2	N		Length of the data. If scale is given, length is the same as precision.
DOM_SCALE	_TDD_LEN	I	2	Y		Scale of a number (decimal).

The following tables depend on TDD_DOMAIN:

- TDD_COLUMN.COL_DOM_NAME
- TDD_IFIELD.IFL_DOM_NAME

TDD_LANGUAGE

All displayed text has a language code associated with it. You store the codes, which match the ISO definitions, in the TDD_LANGUAGE table. You convert from one language to another by simply changing the language code in this table and regenerating your applications.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
LAN_NAME	_TDD_ISOLAN	C	3	N		Language code in 3-character ISO format.
LAN_DESC	_TDD_DESC_SHORT	C	40	y		Brief description.

The following tables depend on TDD_LANGUAGE:

- TDD_CODE.COD_LAN_NAME
- TDD_FORMAT.FMT_LAN_NAME
- TDD_HELP.HLP_LAN_NAME
- TDD_LABEL.LAB_LAN_NAME
- TDD_TEXT.TXT_LAN_NAME

TDD_TRIGGER

The TDD_TRIGGER table stores all your trigger code.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
TRG_NAME	_TDD_IDENT	C	30	N		Name
TRG_TYPE	I	2		N		**Values**: 20 Action (generic) 10 Main 17 Init (Window) 18 Query (Window) 19 Record (Window) 21 Table (Window) 15 Update (Window) 11 Window (Window) 16 Validate 12 Screen field 7 Report field
TRG_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
TRG_DATA	_TDD_TEXT_LONG	C	240	Y		Text data.

The following tables depend on TDD_TRIGGER:

- TDD_EVENT.EVT_TRG_NAME
- TDD_LIST.LIS_TRG_NAME
- TDD_RFIELD.RFL_TRG_NAME
- TDD_SFIELD.SFL_TRG_NAME
- TDD_SFIELD.SFL_VAL_TRG
- TDD_TRIGGERSET.TRS_INIT
- TDD_TRIGGERSET.TRS_MAIN
- TDD_TRIGGERSET.TRS_QUERY
- TDD_TRIGGERSET.TRS_RECORD
- TDD_TRIGGERSET.TRS_UPDATE
- TDD_TRIGGERSET.TRS_WINDOW

TDD_CODE

All codes are defined in TDD_CODE. For example, a Yes/No code could be defined as follows:

COD NAME	COD LAN NAME	COD SEQ	COD DEFNAME	COD DESC	COD VALUE
TDD_YESNO	ENG	0	TDD_NO	No	0
TDD_YESNO	ENG	1	TDD_YES	Yes	1

From the menu, select **TOOLS > Generate 'tdd.h' file** to write these definitions to a TRIMpl- and C- compatible header file.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
COD_NAME	_TDD_IDENT	C	30	N		Name.
COD_CAT_PROJECT	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Project ID. (e.g., customer ID)
COD_LAN_NAME	_TDD_ISOLAN	C	3	N	TDD_LANGUAGE LAN_NAME	
COD_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
COD_DEFNAME	_TDD_IDENT	I	2	N		#define name. Can be used in TRIMpl and C programs.
COD_DESC	_TDD_DESC_MEDIUM	C	80	N		Brief description.
COD_VALUE	_TDD_CODE	I	2	Y		Code value.

The following tables depend on TDD_CODE:

- TDD_CHOICE.CHO_COD_NAME
- TDD_FLAG.FLG_COD_NAME

TDD_FORMAT

TDD_FORMAT stores screen and report field masks. For example, if after you build your applications you decide to make the *Name* field wider on the screen, you simply change its definition in TDD_FORMAT and regenerate your applications.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
FMT_NAME	_TDD_IDENT	C	30	N		Name.
FMT_CAT_PROJECT	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Project ID.(e.g., customer ID)
FMT_CAT_GROUPID	_TDD_IDENT	C	30		TDD_CATEGORY CAT_NAME	Group ID. (Groups applications within a project.)
FMT_LAN_NAME	_TDD_ISOLAN	C	3	N	TDD_LANGUAGE .LAN_NAME	
FMT_DEFNAME	_TDD_IDENT	C	30	Y		#define. Can be used in TRIMpl and C programs.
FMT_DATA	_TDD_FORMAT	C	60	N		Format string.

The following tables depend on `TDD_FORMAT`:

- `TDD_EDIT.EDT_FMT_NAME`
- `TDD_RFIELD.RFL_FMT_NAME`

TDD_HELP

All help descriptions are stored in TDD_HELP. With the HLP_LAN_NAME column, you can change your help dialogs and documentation to different languages without modifying your applications. Using the following TRIMpl code, you can easily create field-level help:

```
/*************************************************************/
/* Display help (if any) on a field                          */
/*  parm.0 - field name                                      */
/*  parm.1 - field help name                                 */
/*************************************************************/
{
list LL,L2;
if (!trap( { LL = list_open("select HLP_DATA,HLP_SEQ "
              "from TDD_HELP "
              "where TDD_VERSION > 0 and "
                     "HLP_LAN_NAME = &g.language and "
                     "HLP_NAME = &parm.1 "
              "order by HLP_SEQ",1000,"Help on "^^parm.0); } )) {
  L2 = NULL;
  while (list_rows(LL)) { list_mod(L2,1,list_curr(LL,0));
list_mod(LL,0); }
  if (list_rows(L2)) {
    edit_text(L2,0,-2,-2,min(16,list_rows(L2)),80,false);
    return;
    }
  }
confirm("Help on '" ^^ parm.0 ^^ "'",
      "No help available",confirm_ok,confirm_info,confirm_okonly);
}
```

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
HLP_NAME	_TDD_IDENT	C	30	N		Name.
HLP_CAT_PROJECT	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Project ID. For VARS this might be the customer ID.
HLP_LAN_NAME	_TDD_ISOLAN	C	3	N	TDD_LANGUAGE LAN_NAME	
HLP_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
HLP_DATA	_TDD_TEXT	C	80	Y		Text data.

The following tables depend on TDD_HELP:

- TDD_COLUMN.COL_HLP_NAME
- TDD_IFIELD.IFL_HLP_NAME
- TDD_TABLE.TAB_HLP_NAME

TDD_LABEL

This table manages screen and report field labels. Note the `LAB_LAN_NAME` column, which simplifies localization efforts for applications and reports. This table also stores logical screen font information.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
LAB_NAME	_TDD_IDENT	C	30	N		Name.
LAB_CAT_PROJECT	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Project ID. For VARS this might be the customer ID.
LAB_LAN_NAME	_TDD_ISOLAN	C	3	N	TDD_CATEGORY CAT_NAME	Group ID. Used to group applications within a project.
LAB_SEQ	_TDD_SEQ	I	2	N	TDD_LANGUAGE LAN_NAME	Orders (sequences) the data correctly.
LAB_FONT	_TDD_CODE	I	2	N		The logical font. **Values:** 0 System 1 Fixed 2 ANSI 3 Field 4 Label 5 Button
LAB_TEXT	_TDD_TEXT	C	80	N		Text data.

The following tables depend on `TDD_LABEL`:

- `TDD_RFIELD.RFL_LAB_NAME`
- `TDD_SFIELD.SFL_LAB_NAME`

TDD_TEXT

This table stores screen, button, and menu, and text items.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
TXT_NAME	_TDD_IDENT	C	30	N		Name.
TXT_CAT_PROJECT	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Project ID. For VARS this might be the customer ID.
TXT_CAT_GROUPID	_TDD_IDENT	C	30	Y	TDD_CATEGORY CAT_NAME	Group ID. Used to group applications within a project.
TXT_LAN_NAME	_TDD_ISOLAN	C	3	N	TDD_LANGUAGE LAN_NAME	
TXT_DATA	_TDD_TEXT	C	80	Y		Text data.

The following tables depend on `TDD_TEXT`:

- `TDD_ACTION.ACT_TXT_NAME`
- `TDD_MENU.MEN_TXT_NAME`

TDD_EVENT

The `TDD_EVENT` table associates an action name with an actual trigger. It also marks the trigger as pre- or post-validation.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
EVT_NAME	_TDD_IDENT	C	30	N		Name.
EVT_PREVAL	_TDD_BOOLEAN	I	2	N		If true (nonzero), this event is performed before the field validation trigger (if one exists) executes. **Values**: 0 No 1 Yes
EVT_TRG_NAME	_TDD_IDENT	C	30		TDD_TRIGGER TRG_NAME	

The table `TDD_ACTION.ACT_EVT_NAME` depends on `TDD_EVENT`.

TDD_LIST

If a screen field is actually a list object, then its trigger name is stored in `TDD_LIST`.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
LIS_NAME	_TDD_IDENT	C	30	N		Name.
LIS_LEN	_TDD_LEN	I	2	N		
LIS_TRG_NAME	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	

The table `TDD_SFIELD.SFL_LIS_NAME` depends on `TDD_LIST`.

TDD_RFIELD

`TDD_RFIELD` stores the report field definitions — justification, display format, label, and trigger names. Storing the format name here makes it simple to modify a field's report display in one place and have that reflected in all reports.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
RFL_NAME	_TDD_IDENT	C	30	N		Name.
RFL_JUST	_TDD_CODE	I	2	N		Justification. **Values:** 76 Left 67 Center 82 Right
RFL_FMT_NAME	_TDD_IDENT	C	30	N	TDD_FORMAT FMT_NAME	
RFL_LAB_NAME	_TDD_IDENT	C	30	Y	TDD_LABEL LAB_NAME	
RFL_TRG_NAME	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	

The table `TDD_FIELD.FLD_RFL_NAME` depends on `TDD_RFIELD`:

TDD_CHOICE

Radio buttons and single-choice lists are described here. In addition to defining the code value to use, TDD_CHOICE also defines the default value to set for new records.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
CHO_NAME	_TDD_IDENT	C	30	N		Name.
CHO_COD_NAME	_TDD_IDENT	C	30	N	TDD_CODE COD_NAME	
CHO_DFLT	_TDD_CODE	I	2	N		Default value to use when creating a new record.

The table TDD_GROUP.GRP_CHO_NAME depends on TDD_CHOICE.

TDD_FLAG

This table describes checkboxes and multi-choice lists. In addition to defining the code value to use, TDD_FLAG also defines the default value to set for new records.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
FLG_NAME	_TDD_IDENT	C	30	N		Name.
FLG_COD_NAME	_TDD_IDENT	C	30	N	TDD_CODE COD_NAME	
FLG_DFLT	_TDD_CODE	I	2	N		Default value to use when creating a new record.

The table TDD_GROUP.GRP_FLG_NAME depends on TDD_FLAG.

TDD_ACTION

Buttons, menu items, and predefined events all use the `TDD_ACTION` table to define their label text and event. A Quit button and a Quit menu item can point to the same `TDD_ACTION`, thus sharing the same code and behavior.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
ACT_NAME	_TDD_IDENT	C	30	N		Name.
ACT_AID	_TDD_INTEGER	I	4	N		Additional Action ID.*
ACT_TXT_NAME	_TDD_IDENT	C	30	N	TDD_TEXT TXT_NAME	
ACT_EVT_NAME	_TDD_IDENT	C	30	N	TDD_EVENT EVT_NAME	

*The pre-defined variable g.aid is set to this value after a `[raw_]input()` call. If the value is (-1) `G.AID` is set to `G.KEY`. On a FORWARD EVENT and if the value is (-1) then `G.AID` is set to `G.AUX`.

The following tables depend on `TDD_ACTION`:

- `TDD_BUTTON.BUT_ACT_NAME`
- `TDD_MENU.MEN_ACT_NAME`
- `TDD_PEVENT.PEV_ACT_NAME`

TDD_EDIT

This table describes edit fields, which allow end users to type in values. The `EDT_FMT_NAME` column controls the display format and the `EDT_ATTR` column controls the input behavior.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
EDT_NAME	_TDD_IDENT	C	30	N		Name.
EDT_ATTR	_TDD_ATTR	I	4	N		Attributes. To combine several attributes binary OR the individual attributes. **Values:** 2 Fixed 8 Uppercase 16 No echo 256 Auto reset 4096 Autoskip 6384 Raw input
EDT_JUST	_TDD_CODE	I	2	N		Justification. **Values:** 76 Left 67 Center 82 Right
EDT_LEN	_TDD_LEN	I	2	N		Display length. If 0, use the format mask.
EDT_FMT_NAME	_TDD_IDENT	C	30	N	TDD_FORMAT FMT_NAME	

The table `TDD_SFIELD.SFL_EDT_NAME` depends on `TDD_EDIT`.

TDD_TABLE

Database table descriptions reside in TDD_TABLE. Along with table name and description, you can store an optional TAB_EDITAPP, which is called from the DVdd to edit the table.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
TAB_NAME	_TDD_SQLID	C	18	N		SQL Identifier.
TAB_EDITAPP	_TDD_TEXT	C	80	Y		Table-editing application. This .run file. If no name is given the default grid editor application is used.
TAB_DESC	_TDD_DESC_LONG	C	240	Y		Brief description.
TAB_HLP_NAME	_TDD_IDENT	C	30	Y	TDD_HELP HLP_NAME	

The following tables depend on TDD_TABLE:

- TDD_COLUMN.COL_TAB_NAME
- TDD_FORKEY.FKY_TAB_MASTER
- TDD_FORKEY.FKY_TAB_SLAVE
- TDD_INDEX.IDX_TAB_NAME
- TDD_LOV.LOV_TAB_NAME

TDD_STEXT

This table defines screen text items. In addition to the text name, you also define the font used to display the text.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
STX_NAME	_TDD_IDENT	C	30	N		Name.
STX_FONT	_TDD_CODE	I	2	N		The logical font. **Values can be:** 0 System 1 Fixed 2 ANSI 3 Field 4 Label 5 Button
STX_TXT_NAME	_TDD_IDENT	C	30	N	TDD_TEXT TXT_NAME	

No tables depend on this table.

TDD_MENU

TDD_MENU stores menu definitions. Because this table is very complicated and it is easy to make mistakes that render your menus unusable, you should always use DVmenu to create and modify your menu definitions. DVmenu can be run either as a stand-alone product or from within DVapp.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
MEN_NAME	_TDD_IDENT	C	30	N		Name.
MEN_SNAME	_TDD_IDENT	C	30	Y		Submenu name.
MEN_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
MEN_ACT_NAME	_TDD_IDENT	C	30	Y	TDD_ACTION ACT_NAME	
MEN_MEN_SNAME	_TDD_IDENT	C	30	Y	TDD_MENU MEN_SNAME	Submenu name.
MEN_TXT_NAME	_TDD_IDENT	C	30	Y	TDD_TEXT TXT_NAME	

The following table depends on `TDD_MENU`:

- `TDD_MENU.MEN_MEN_SNAME`
- `TDD_TRIGGERSET.TRS_MEN_NAME`

TDD_BUTTON

This table defines action buttons. You can specify an optional bitmap to display in the button, otherwise the generator uses the underlying `BUT_ACT_NAME`'s `TDD_ACTION` label.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
BUT_NAME	_TDD_IDENT	C	30	N		Name.
BUT_ACT_NAME	_TDD_IDENT	C	30	N	TDD_ACTION ACT_NAME	
BUT_BITMAP	_TDD_IDENT	C	30	Y		If specified, the button is displayed with the named bitmap. If NULL, then the label from the underlying action is used.

No tables depend on `TDD_BUTTON`.

TDD_GROUP

Checkboxes, radio buttons, and lists are all group objects. In the TDD_GROUP table, you specify object type by using GRP_ATTR flags and GRP_FLG_NAME and GRP_CHO_NAME fields. For example, if GRP_FLG_NAME is set, then the field is a multi-choice field. If GRP_ATTR has the **List** bit on, then it is a multi-choice list. Add the **drop-down** bit and it is now a drop-down multi-choice list.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
GRP_NAME	_TDD_IDENT	C	30	N		Name.
GRP_ATTR	_TDD_ATTR	I	4	N		Attributes. To combine several attributes binary OR the individual attributes. **Values:** 1 Button 2 List 8 Drop Down 16 Horizontal
GRP_FLG_NAME	_TDD_IDENT	C	30	Y	TDD_FLAG FLG_NAME	
GRP_CHO_NAME	_TDD_IDENT	C	30	Y	TDD_CHOICE CHO_NAME	

The table TDD_SFIELD.SFL_GRP_NAME depends on TDD_GROUP.

TDD_INDEX

Define your database indexes here.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
IDX_NAME	_TDD_SQLID	C	18	N		SQL Identifier. Following the ANSI standard, this field is 18 characters long to ensure safe portability.
IDX_TAB_NAME	_TDD_SQLID	C	18	N	TDD_TABLE TAB_NAME	Table name.
IDX_UNIQUE	_TDD_BOOLEAN	I	2	N		**Values**: 0 No 1 Yes

The table TDD_XCOLUMN.XCO_IDX_NAME depends on TDD_INDEX.

TDD_SFIELD

This table is the main DVdd table for screen fields. It stores screen field attributes and user-defined attributes as well as the names of the label, field, and validation triggers, and edit, list, and group names. If entries in more than one of the edit, list, and group columns appear, users are prompted to pick one.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
SFL_NAME	_TDD_IDENT	C	30	N		Name.
SFL_ATTR	_TDD_ATTR	I	4	N		Attributes. To combine several attributes binary OR the individual attributes. **Values**: 0 - none 1 Calculate 2 Primary 4 Initialize
SFL_UATTR	_TDD_ATTR	I	4	N		User-defined attributes. **Values**: 0 - none 1 Calculate 2 Primary 4 Initialize
SFL_EDT_NAME	_TDD_IDENT	C	30	Y	TDD_EDIT EDT_NAME	
SFL_LIS_NAME	_TDD_IDENT	C	30	Y	TDD_LIST LIS_NAME	
SFL_GRP_NAME	_TDD_IDENT	C	30	Y	TDD_GROUP GRP_NAME	
SFL_LAB_NAME	_TDD_IDENT	C	30	Y	TDD_LABEL LAB_NAME	
SFL_VAL_TRG	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
SFL_TRG_NAME	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	

The table `TDD_FIELD.FLD_SFL_NAME` depends on `TDD_SFIELD`.

TDD_TRIGGERSET

Use the TDD_TRIGGERSET table to create sets of triggers. DVapp windows have several triggers, such as window, update, record, and table. This table groups these triggers, creating templates for window types. For example, a user may define a read-only, an insert-only, and an update triggerset. Then when someone creates a new window in an application, picking the correct triggerset automatically associates the triggers with the desired behavior.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
TRS_NAME	_TDD_IDENT	C	30	N		Name.
TRS_MEN_NAME	_TDD_IDENT	C	30	Y	TDD_MENU MEN_NAME	
TRS_MAIN	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
TRS_WINDOW	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
TRS_UPDATE	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
TRS_INIT	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
TRS_QUERY	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	
TRS_RECORD	_TDD_IDENT	C	30	Y	TDD_TRIGGER TRG_NAME	

TDD_PEVENT.PEV_TRS_NAME depends on TDD_TRIGGERSET.

TDD_PEVENT

The window and key events reside in TDD_PEVENT. Note the PEV_DEFNAME column which is used to create TRIMpl- and C-compatible header files with the predefined event definitions. DVapp applications use these files to find out what event has occurred.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
PEV_NAME	_TDD_IDENT	C	30	N		Name.
PEV_TRS_NAME	_TDD_IDENT	C	30	Y	TDD_TRIGGERSET TRS_NAME	
PEV_MODE	_TDD_CODE	I	2	N		**Values:** 0 Record 1 Query
PEV_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
PEV_DEFNAME	_TDD_IDENT	C	30	Y		#define name. Can be used in TRIMpl and C programs.
PEV_ACT_NAME	_TDD_IDENT	C	30	N	TDD_ACTION ACT_NAME.	

No tables depend on TDD_PEVENT.

TDD_FIELD

TDD_FIELD defines the screen and report field names for a column or independent field.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
FLD_NAME	_TDD_IDENT	C	30	N		Name.
FLD_SFL_NAME	_TDD_IDENT	C	30	Y	TDD_SFIELD SFL_NAME	
FLD_RFL_NAME	_TDD_IDENT	C	30	Y	TDD_RFIELD RFL_NAME	

The following table depends on TDD_FIELD:

- TDD_COLUMN.COL_FLD_NAME
- TDD_IFIELD.IFL_FLD_NAME

TDD_COLUMN

Use this table to define your database columns.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
COL_NAME	_TDD_SQLID	C	18	N		SQL Identifier.
COL_TABL_NAME	_TDD_SQLID	C	18	N	TDD_TABLE TAB_NAME	SQL Identifier.
COL_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.
COL_NULLS	_TDD_BOOLEAN	I	2	N		**Values:** 0 No 1 Yes
COL_DOM_NAME	_TDD_IDENT	C	30	N	TDD_DOMAIN DOM_NAME	
COL_FLD_NAME	_TDD_IDENT	C	30	Y	TDD_FIELD FLD_NAME	
COL_HLP_NAME	_TDD_IDENT	C	30	Y	TDD_HELP HLP_NAME	

The following tables depend on `TDD_COLUMN`:

- `TDD_FORKEY.FKY_COL_MASTER`
- `TDD_FORKEY.FKY_COL_SLAVE`
- `TDD_FORKEY.FKY_VCO_SLAVE`
- `TDD_LOV.LOV_COL_NAME`
- `TDD_XCOLUMN.XCO_COL_NAME`
- `TDD_XCOLUMN.XCO_TAB_NAME`

TDD_IFIELD

Independent fields, fields that are not related to database table columns, are defined here.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
IFL_NAME	_TDD_IDENT	C	30	N		Name.
IFL_DOM_NAME	_TDD_IDENT	C	30	N	TDD_DOMAN DOM_NAME	
IFL_FLD_NAME	_TDD_IDENT	C	30	N	TDD_FIELD FLD_NAME	
IFL_HLP_NAME	_TDD_IDENT	C	30	Y	TDD_HELP HLP_NAME	

No tables depend on `TDD_IFIELD`.

TDD_FORKEY

The foreign key table is used by most of the DVdd utility programs to keep the internal DVdd structures correct. It is also used by DVapp to perform automatic foreign key checking.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
FKY_TAB_MASTER	_TDD_SQLID	C	18	N	TDD_TABLE TAB_NAME	Table name.
FKY_COL_MASTER	_TDD_SQLID	C	18	N	TDD_COLUMN COL_NAME	Column name.
FKY_VAL_MASTER	_TDD_CODE	I	2	Y		Additional value predicate. If not NULL, FKY_VCO_SLAVE indicates the column to which the value should be equal.
FKY_TAB_SLAVE	_TDD_SQLID	C	18	N	TDD_TABLE TAB_NAME	Table name.
FKY_COL_SLAVE	_TDD_SQLID	C	18	N	TDD_COLUMN COL_NAME	Column name.
FKY_VCO_SLAVE	_TDD_SQLID	C	18	Y	TDD_COLUMN COL_NAME	Column name.
FLY_LOV_DISPCOL	_TDD_COLUMN	I	2	Y		Column to display during default LOV processing. NULL indicates all columns. A value greater than the number of columns in the LOV table shows the return column. Otherwise, the column indicated is shown.

No tables depend on `TDD_FORKEY`.

TDD_LOV

List of values definitions are stored in TDD_LOV. You have control over which columns appear in the LOV popup as well as any additional WHERE and ORDER BY clauses.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
LOV_TAB_NAME	_TDD_SQLID	C	18	N	TDD_TABLE TAB_NAME	Table name.
LOV_COL_NAME	_TDD_SQLID	C	18	N		Column name.
LOV_SEQ	_TDD_SEQ	I	2	N	TDD_COLUMN COL_NAME	Orders (sequences) the data correctly.
LOV_SEL_COLUMNS	_TDD_TEXT_LONG	C	240	Y		Specifies the columns (in a comma-separated list) to display in the list of values popup. If not specified all columns display.
LOV_SEL_WHERE	_TDD_TEXT_LONG	C	240	Y		Optional WHERE clause. The WHERE is added automatically.
LOV_SEL_ORDER	_TDD_TEXT_LONG	C	240	Y		Optional ORDER BY clause. The ORDER BY is added automatically.

No tables depend on the TDD_LOV table.

TDD_XCOLUMN

TDD_XCOLUMN stores the index columns used to build indexes.

Name	Domain	Dty	Len	Nulls	Foreign Key (Table & Col)	Description
TDD_VERSION	_TDD_VERSION	I	4	N		Version stamp.
XCO_IDX_NAME	_TDD_SQLID	C	18	N	TDD_INDEX IDX_NAME	
XCO_TAB_NAME	_TDD_SQLID	C	18	N	TDD_COLUMN COL_TAB_NAME	
XCO_COL_NAME	_TDD_SQLID	C	18	N	TDD_COLUMN COL_NAME	
XCO_SEQ	_TDD_SEQ	I	2	N		Orders (sequences) the data correctly.

No tables depend on TDD_XCOLUMN

Appendix B
Intialization Files

Most of the Trifox tools and sub-systems read configuration and initialization data from special `.ini` files. These files typically have the same format:

```
option                  value
```

The *option* is the name of the initialization option, setting name, or parameter. Lines with un-recognized options are ignored.

Value is the value of the option. Depending on *option* the *value* can be a number, a yes/no, or a text string. The value can also represent one or more environment variables expressed as:

```
$(name)
```

The environment variable(s) are expanded before the value is evaluated.

The files support text strings as values, but they must be enclosed in double quotes ("), SQL-style, if blanks or quotes are part of the string. If no ending quote mark is provided, the string is terminated with a `\n`.

If an option is not found in the file, then the default value is used.

The various relevant `.ini` files are described in detail in the following section(s).

Edit them using any ascii-based text editor. If you are reinstalling a product, we recommend you edit a ‘‘clean’’ copy of each `.ini` file, rather than modifying an existing one from your environment.

VORTEXclient applications all look for necessary initialization parameters in the following priority:

1. Current working directory, regardless of any env_var setting.
2. The lib subdirectory under `$TRIM_HOME`, which is specified according to operating system rules.

dv.ini

`dv.ini` is used by DesignVision.

auto_start

Type	yes/no
Default	no
Description	This setting overrides the automatic start and displays the Slave Start Dialog box when you start DV or TRIM with the -i option.

Example The following specifies that the Slave Start Dialog box should appear when the DesignVision application starts:

```
auto_start      no
```

busy_alarm

Type number

Default 0

Description Number of seconds until an alarm goes off.

The alarm is initialized by the first `[raw_]input()` function called. It is always turned off before each `[raw_]input()` and turned back on when returning from input, so it is on during actual TRIMpl processing.

The TRIMpl function `alarm()`, when used, overrides this setting.

Example The following specifies that something extraordinary might happen in five minutes (300 seconds)

```
busy_alarm 300
```

cgi_timeout

Type number

Default 300

Description Specifies the maximum amount of seconds to wait for a cgi response.

Example The following specifies that the maximum wait is 10 minutes:

```
cgi_timeout      600
```

columns

Type number

Default 64

Description Specifies the maximum number of columns allowed in query result.

Example The following specifies that the maximum number of columns allowed in a query result is 128:

```
columns      128
```

db_cursors

Type number

Default 32

Description Number of database cursors to allocate for each database connection.

Example The following specifies that 32 database cursors are allocated for each database connection:

```
db_cursors 32
```

db_cursor_cache

Type yes/no

Default yes

Description Cache database cursors.

Example The following specifies that database cursors should ***not*** be cached:

```
db_cursor_cache no
```

db_message_size

Type number

Default 128

Description Length (in characters) of the database error message buffer.

Example The following specifies that the buffer is 256 characters long:

```
db_message_size 256
```

db_trim_blanks

Type yes/no

Default yes

Description Trim trailing blanks from fetched database CHAR data.

Example The following specifies that trailing blanks are trimmed:

```
db_trim_blanks yes
```

decimal_key

Type string

Default Codepage or keyboard

Description String to return when the numeric keypad decimal separator key is pressed.

Example The following specifies that "." must be returned regardless of the current codepage or keyboard:

```
decimal_key .
```

dynamic_cursors

Type number

Default 0

Description Sets the number of dynamic cursors to allocate. Dynamic cursors are non-constant SQL statements, eg. concatenated strings, and are typically not cached. If the number of dynamic_cursors is exceeded, then those statements will not be cached which is the default action.

Example The following specifies that a maximum of 128 dynamic cursors will be allocated:

```
dynamic_cursors 128
```

env_vars

Type text

Default non

Description A comma separated list of environment variables that the DV thin client sends to the application server. The application server, in turn, sets the variables as appropriate when it starts the application. The values in this list are operating system and application-specific. They can contains variables for multiple operating systems.

Example The following sets `TRIM_HOME` on a Unix machine and a value called `CIS.INI` on an NT machine:

```
TRIM_HOME=/usr/rad,CIS.INI=c:\proj\lib\cis.ini
```

fetch_buffer_size

Type number

Default 4096

Description Database fetch buffer size (in bytes).

Example The following specifies that the fetch buffer is 2048 bytes large:

```
fetch_buffer_size    2048
```

fnc_path

Type text

Default none

Description Specifies the location of `*.fnc` files during generation. The first character defines the path “delimeter” for the keyword definition.

Example The following specifies that `*.fnc` files are located in the current working directory or the subdirectory lib specified in `TRIM_HOME`:

```
fnc_path  ;.; $(TRIM_HOME)/lib
```

heap_block_size

Type number

Default 1008

Description Minimum heap block size (in bytes) to allocate for the DesignVision session.

Example The following specifies a heap block at least 504 bytes large:

```
heap_block_size  504
```

hostname

Type text

Default none

Description The thin client uses this option to identify the application server, either by IP address or by machine name.

Example The following specifies the hostname:

```
hostname   mynt
```

html_listload_hide

Type number

Default 100

Description If a display list has more than this number of rows, then it is hidden while all the rows are loaded. This eliminates annoying flicker during browser list loading.

Example The following specifies a maximum list size of 200 rows:

```
html_listload_hide  200
```

html_min_fld_width

Type number

Default none

Description The minimum display width for a field.

Example The following specifies a minimum width of 5 cells:

```
html_min_fld_width  5
```

include_path

Type text

Default none

Description Specifies the location of #include files during generation. The first character defines the path "delimeter" for the keyword definition.

Example The following specifies that #include files are located in the `c:\trim\include` and `d:\myincludes` directories:

```
include_path  ;c:\trim\include;d:\myincludes
```

list_view_max_width

Type number

Default 1000

Description Width of list_view in characters, 80 - 8000. If the list is wider, three dots "..." will be appended to the rows.

Example The following specifies that the list_view width is limited to 128 characters:

```
list_view_max_width   128
```

logical_cursors

Type number

Default 128

Description Number of logical cursors to allocate per database connection.

Example The following specifies that 56 logical cursors are allocated for the application:

```
logical_cursors   56
```

message_file

Type text

Default trim.msg or dv.msg

Description Lets you specify your own message file.

Example The following specifies that messages are in a subdirectory, lib, in the TRIM_HOME directory:

```
message_file      $(TRIM_HOME)/lib/dv.msg
```

message_server

Type text

Default none

Description This parameter identifies the name of the Trifox message executable. This executable logs all messages between the thin client and application server when the client is run in trace mode.

Example The following specifies that the executable dvmsg32.exe logs messages.

```
message_server      dvmsg32.exe
```

mux_default_db_id

Type number

Default 0

Description Specifies the database when signing up with VORTEXaccelerator/ VORTEXmonitor. Actual slaves may override the setting if they are different. See `trim.h` for values.

Example The following specifies that Rdb is the default database

```
mux_default_db_id 1
```

packetsize

Type number

Default 8192

Description Specifies the network packet size as well as the size of the send buffers on each side of the connection. Sends are accumulated and the buffer is flushed when the contents reach the limit, or if it is not full, a read triggers a send. This option can have a significant impact on network performance by keeping traffic efficient.

Example The following specifies that the buffers should flush when they reach 4096 bytes:

```
packetsize    4096
```

parameters

Type text

Default none

Description The thin client passes the items specified in this option to the application server, which uses them for the application associated with the thin client. The option can contain any number or type of paramters; it is entirely application-dependent.

Example The following specifies the database connect string for the application:

```
parameters      net:niklas/back
```

port

Type number

Default none

Description Port defines the vtxnet service port and overrides the value specified in the operating system's service file, if defined.

Example The following specifies a connection on port 1958:

```
port    1958
```

program

Type text

Default none

Description The thin client uses this option to identify the application that the application server should start.

Example The following specifies that the server start an application called `/usr2/bin/trimrun.g2s`:

```
program   /usr2/bin/trimrun.g2s
```

run_path

Type text

Default none

Description Where to look for run files. The first character identifies the path delimeter.

Example The following indicates that programs can be in either `c:\trifox` or `e:\bin`:

```
run_path  ;c:\trifox;e:\bin
```

shmem_seg_size

Type number

Default 64

Description Minimum shared segment size to allocate (in kilobytes).

Example The following specifies that the shared segment size is never less than 128:

```
shmem_seg_size    128
```

sql_strip_comments

Type yes/no

Default yes

Description Controls whether or not SQL comments are removed from constant SQL statements. This allows Oracle hints to be passed to the driver.

Example The following specifies that SQL comments are not to be stripped:

```
sql_strip_comments    no
```

sql_xlate_file

Type string

Default none

Description The name of a file that contains the SQL translations, or function mappings.

Example The following, from a file called `db2.fms` (DB2 function mapping specs) specifies that DECODE replaces a string that begins CASE …:

```
DECODE  CASE WHEN $0=$1 THEN $2 #2[WHEN $0=$3 THEN $4 ]ELSE ~0
END
```

stack_size

Type number

Default 8000

Description This defines the amount of local variable stack in bytes

Example The following defines the local variable stack size to 16000 bytes:

```
stack_size     16000
```

trap_intern_errors

Type yes/no

Default no

Description Controls whether or not internal errors can be caught by the trap() function.

Example The following specifies that internal errors can be trapped:

```
trap_intern_errors    yes
```

uppercase

Type yes/no

Default yes

Description Sets case for all identifiers.

Example The following specifies that identifiers are not case-sensitive:

```
uppercase No
```

uppercase_sql

Type yes/no

Default yes

Description Specifies whether SQL identifiers in constant statements are forced to be uppercase.

Example The following, required for Sybase databases which are case-sensitive, specifies that identifiers on constant statements are not automatically made uppercase:

```
uppercase_sql  No
```

working_directory

Type text

Default none

Description Specifies the default working directory for the host program.

Example The following specifies that the host program's default working directory is `/usr/bin/work`:

```
working_directory     /usr/bin/work
```

xaml_cell_height

Type numeric

Default none

Description Specifies the character cell height.

Example The following specifies that the character cell height is 23 pixels:

```
xaml_cell_height  23
```

xaml_cell_width

Type numeric

Default none

Description Specifies the character cell width.

Example The following specifies that the character cell width is 10 pixels:

```
xaml_cell_width  10
```

xaml_fld_add_width

Type numeric

Default none

Description Specifies the extra pixels to add to each field.

Example The following specifies that each field will have an extra 8 pixels of width:

```
xaml_fld_add_width  8
```

xaml_fld_max_margin

Type numeric

Default none

Description Specifies the extra pixels to add to each character in fields that are less than xaml_fld_max_margin characters. The number of margin pixels decreases based on the size of the field.

Example If xaml_fld_max_margin is set to 5

```
xaml_fld_max_margin  5
```

then in all character based fields less than 5 characters, the following adjustments will occur:

1 char: 10 pixels (2 * 5)
2 chars: 8 pixels (2 * (5 - 1))
3 chars: 6 pixels (2 * (5 - 2))
4 chars: 4 pixels (2 * (5 - 3))

xaml_fld_min_width

Type numeric

Default none

Description Specifies the minum field width in pixels.

Example The following specifies that each field will have aminimum width of 20 pixels:

```
xaml_fld_min_width  20
```

xaml_font_n

Type text

Default The default fixed font is “ FontFamily=’Coutier New’”. None of the other fonts have default values.

Description Specifies the font to use for

xaml_font_0 system
xaml_font_1 fixed
xaml_font_2 ansi
xaml_font_3 field
xaml_font_4 label
xaml_font_5 button

Example The following specifies the fonts to use for the different object types::

```
xaml_font_0 “FontFamily=’Arial Black’”
xaml_font_1 “FontFamily=’Courier New’”
xaml_font_3 “FontFamily=’Verdana’”
xaml_font_4 “FontFamily=’Trebuchet MS’”
xaml_font_5 “FontFamily=’Comic Sans MS’
FontSize=’12’”
```

Example

The following is a sample `dv.ini` file on a Unix machine.

```
rem ---------- Trifox Virtual DV specifics
env_vars       TRIM_HOME=/usr3/rad/dac,WRCIS.INI=/usr3/rad/dac/lib/wrcisth.ini
hostname       192.0.2.99
working_directory /usr3/rad/dac/cis
#default_file c:\daspects\lib\big4caps.vgd
message_server dvmsg.exe
program        /usr2/trim/bin/trimrun.g2s
parameters     start
packetsize     1024                      -- network packet size (send/recv)
port           1958                      -- port (overrides /etc/services)
rem -------------- TRIM generics
heap_block_size    1008                    -- heap block size (in bytes)
fetch_buffer_size  4096                    -- fetch buffer size (in bytes)
db_message_size    300                     -- max DB message length
columns            256                     -- max # of database columns
logical_cursors    512                     -- max # of logical cursors
db_cursors         64                      -- max # of DB cursors
db_cursor_cache    yes                     -- cache the db cursors
db_trim_blanks     yes                     -- trim trailing blanks from DB
uppercase          yes                     -- true if to uppercase idents
uppercase_sql      no                      -- true if to uppercase SQL stmts
include_path       ;.;f:\trim\lib
run_path           ;.;f:\trim\run
```

trim.ini

`trim.ini` is used by all the TRIM tools except TRIMqmr.

columns

Type number

Default 64

Description Maximum number of columns allowed for the associated `connect()`.

Example The following specifies that a query can contain no more than 24 columns:

```
columns  24
```

db_cursor_cache

Type yes/no

Default yes

Description Caches database cursors.

Example The following specifies that database cursors should not be cached:

```
db_cursor_cache  no
```

db_cursors

Type number

Default 32

Description Number of database cursors to allocate for the associated `connect()`.

Example The following specifies that 15 cursors are allocated for each database connection:

```
db_cursors  15
```

db_message_size

Type number

Default 128

Description Length (in characters) of the database error message buffer.

Example The following specifies that the buffer is 256 characters long:

```
db_message_size  256
```

db_trim_blanks

Type yes/no

Default yes

Description Trim trailing blanks from fetched database CHAR data.

Example The following specifies that trailing blanks are trimmed:

```
db_trim_blanks  yes
```

fetch_buffer_size

Type number

Default 2048

Description Database fetch buffer size (in bytes).

Example The following specifies a buffer 1024 bytes large:

```
fetch_buffer_size  1024
```

heap_block_size

Type number

Default 1008

Description Minimum heap block size (in bytes) to allocate for the DesignVision session.

Example The following specifies a heap block at least 504 bytes large:

```
heap_block_size  504
```

include_path

Type text

Default none

Description Specifies the location of #include files during generation. The first character defines the path“delimeter” for the keyword definition.

Example The following specifies that #include files are located in the locations `c:\trim\include` and `d:\myincludes`:

```
include_path  ;c:\trim\include;d:\myincludes
```

logical_cursors

Type number

Default 128

Description Number of logical cursors to allocate for the connection.

Example The following specifies that 56 logical cursors are allocated for the connection:

```
logical_cursors   56
```

message_file

Type string

Default trim.msg or dv.msg

Description Lets you specify your own message file at runtime.

Example The following specifies that messages are in a subdirectory , lib, in the TRIM_HOME directory:

```
message_file      $(TRIM_HOME)/lib/trim.msg
```

mux_default_db_id

Type number

Default 0

Description Specifies the database when signing up with VORTEXaccelerator. Actual slaves may override the setting if they are different. See `trim.h` for values.

Example The following specifies that Rdb is the default database:

```
mux_default_db_id 1
```

run_path

Type text

Default none

Description Where to look for run files. The first character identifies the path delimeter.

Example The following indicates that programs can be in either `c:\trifox` or `e:\bin`:

```
run_path  ;c:\trifox;e:\bin
```

shmem_seg_size

Type number

Default 64

Description Minimum shared segment size to allocate (in kilobytes).

Example The following specifies that the shared segment size is never less than 128:

```
shmem_seg_size  128
```

sql_strip_comments

Type yes/no

Default yes

Description Controls whether or not SQL comments are removed from constant SQL statements. This allows Oracle hints to be passed to the driver.

Example The following specifies that SQL comments are not to be stripped:

```
sql_strip_comments     no
```

sql_xlate_file

Type string

Default none

Description The name of a file that contains the SQL translations, or function mappings.

Example The following, from a file called db2.fms (DB2 function mapping specs) specifies that DECODE replaces a string that begins CASE …:

```
DECODE  CASE WHEN $0=$1 THEN $2 #2[WHEN $0=$3 THEN $4 ]ELSE ~0 END
```

stack_size

Type number

Default 8000

Description This defines the amount of local variable stack in bytes

Example The following defines the local variable stack size to 16000 bytes:

```
stack_size      16000
```

uppercase

Type yes/no

Default yes

Description Indicates that all identifiers are case-sensitive.

Example The following specifies that identifiers are not case-sensitive:

```
uppercase No
```

uppercase_sql

Type yes/no

Default yes

Description Specifies whether SQL identifiers in constant statements are forced to be upper case.

Example The following, required for Sybase databases which are case-sensitive, specifies that identifiers on constant statements are not automatically made upper case:

```
uppercase_sql  No
```

Example

```
rem -------------- TRIM generics
heap_block_size    16000                  -- heap block size (in bytes)
message_file       $(TRIM_HOME)/lib/trim.msg
fetch_buffer_size  32768                    -- fetch buffer size (in bytes)
stack_size         8000
```

```
db_message_size    300                      -- max DB message length
columns            768                      -- max # of database columns
logical_cursors    4096                     -- max # of logical cursors
shmem_seg_size     128                      -- min shared seg size (in Kbytes)
db_cursors         64                       -- max # of DB cursors
db_cursor_cache    yes                      -- cache the db cursors
db_trim_blanks     yes                      -- trim trailing blanks from DB
sql_xlate_file     $(TRIM_HOME)/lib/db2.fms -- SQL translate file
uppercase          yes                      -- true if to uppercase idents
uppercase_sql      yes                      -- true if to uppercase SQL stmts
sql_strip_comments yes                      -- true if to strip SQL comments
include_path       ;.;$(TRIM_HOME)/lib
fnc_path           ;.;$(TRIM_HOME)/lib
run_path           ;.;$(TRIM_HOME)/run;$(TRIM_HOME)/dd;/usr2/bin
os_add_path :trifoc:niklas
rem
rem NOTE: first character in ‘include_path’ and ‘run_path’ is the path delimiter
rem
```

Appendix C
DV File Structure

A subset of the tree structures is independent of the type of application — for example, all database access functions. The differences in the structures are mainly in supplied functions; for example, report writing functions such as paginate are meaningless in screen applications.

DesignVision and TRIMtools, its predecessor, uses an environment variable, `TRIM_HOME`, to locate supplementary files. Within its default directory, DesignVision creates four subdirectories — LIB, TERM, RUN, and BIN — that comprise the development environment.

LIB Subdirectory

The LIB, or library, subdirectory, contains a number of files that store settings, including yours, for application development.

Filename	Contains
trim.cc / dv.cc	Printer control code used by the `pset()` function in report designs (`.rep` files). Each line in the file contains a symbol followed by a quoted string of hexadecimal characters that represent an escape code sequence.
trim.dft / dv.dft	Various sections of code that is executed by default functions including code for the Global window, user-defined windows, key triggers, and update triggers.
trim.fnc / dv.fnc	General-purpose subroutines that can be called by any design. Each subroutine entry consists of a title line that has the subroutine namewith an optional suffix, followed by the code in braces ({ }), followed by the record separator.
trim.h / dv.h	Symbol definitions for DVapp and DVreport designs as well as stand-alone triggers. Symbols are defined, like `C` symbols, by beginning a line with a `#define` followed by a symbol name and a number or quoted string.
dv.img	Used to preload graphics into DVslave's graphic cache. Each line is in the format: <image name> <image file name> <image name> is the name the image will have when referenced within the application. <image file name> is the name of the file (on the Windows machine) to be read into the internal image buffers.

Filename	Contains
trim.key / dv.key	The labels that identify the key triggers in the DVapp form designer. You can edit the labels to reflect different key mappings and check the labels in DVapps's designer `DEFINE KEY` dialog.
trim.uat / dv.uat	User-defined attribute labels. Each line has a single label and the label's position (the line number) in the file indicated the bit to which the user attribute corresponds. For example, if `trim.uat` contains only `CALCULATE, ROUNDUP, ROUNDDOWN` these three user attribute options on the DEFINE FIELD dialog. The built-in function `field_attr()` returns a number that corresponds to the sum of the value of each user attribute: The first user attribute is equal to `1`, the second to `2`, the third to `4`, the fourth to `8`, and so on, based on the result of 2 raised to the power of the position in the `.uat` file.
trim.msg / dv.msg	The error messages for DesignVision. The file is not ascii and can not be edited. It must be located in the TBU_HOME lib directory.
trim.csf	OBSOLETE.
dv.ini	Initialization values for operating characteristics of DesignVision.

Appendix D
Migration Issues

Several issues influence the migration of a character-based DVapp or TRIMapp application to a windows DVapp one.

Support Files

Character-based DVapp (TRIMapp) uses a set of support files usually located in the `lib` directory located under the `$TRIM_HOME` directory. The files are all prefixed with `trim.` and suffixed with the following file extentions:

TRIM Extensions	Description
csf	OBSOLETE.
dft	Window default trigger code.
fnc	TRImpl function library.
h	TRImpl header file that contains defines.
ini	TRIM initialization values.
key	Names of key triggers.
msg	Error and informational messages.
uat	Names of user attributes.
vis	Names of visual attributes.

The window version of DVapp uses the above files plus the following files. For the window version, all files have the `dv.` prefix:

TRIM Extensions	Description
kma	Windows-to-TRIMapp key mapping.
men	Default window menu.
pev	Names of the predefined events.

The difficulty arises in integrating any modifications that you may have made to the character-based files into the window-based files. This difficulty is most evident in the `dft` file, which has default trigger entries for the pre-defined events. You can add the modifications to the existing `trim.dft` either by using an editor (being careful to keep the ASCII 30 record separator) or by using the `car` utility.

If you intend to migrate character-based applications to window-based ones, you must modify the trigger code for pre-defined event 12, MDI. The default code shipped with the kit references `G.parent_aux`, a variable defined in the new MAIN trigger in

`dv.dft` but which did not exist in character-based `trim.dft`. Simply remove this code line in `trim.dft`.

Just copy your existing `trim.fnc` to `dv.fnc`. The default `dv.h` file includes `trim.h` and incorporates all necessary information.

Global Window

Because DVapp does not automatically create a global window, any of the functions you copied from your old `trim.fnc` into `dv.fnc` that reference `G.msg` do not work. The message line concept has been replaced by a status line and a message server with the `display_msg` function in `dv.fnc`. Call this function when you want to write a message to the ‘‘message" line. In addition, any references to `G.fnn` where *nn* is 1 through 10, do not work since these fields no longer exist.

Symbols

A

B

C

W

X