# Programming the User Interface

## Preface to Programming the User Interface

The information contained in this volume and its companion, the Symbolics User Interface Dictionary, includes everything the programmer needs to write a user interface for an application program. This includes information on:

- Creating, modifying, and controlling windows
- Obtaining user input from the keyboard and mouse
- Formatting tables, tree graphs, and other graphics output
- Using the Command Processor facility

The current volume is organized in an overview and three sections. The first section, Part 1, presents a set of top-level tools for interface programming. It explains how to use these tools to accomplish the two essential functions of an interface — presenting output and obtaining input. This section documents completely the two top-level macros, **cp:define-command** and **dw:define-program-framework**.

Part 2 explains how to build on and modify the basic structures created by the high-level tools in order to accomplish more complicated tasks. This section completely documents the four presentation-related defining macros, **define-presentation-type**, **define-presentation-translator**, **define-presentation-to-command-translator**, and **define-presentation-action.**

Part 3 discusses the organization of the Symbolics user interface substrates and how the system works, including information on low-level details. Most of this information is unnecessary unless you want to modify the behavior of some top- or middle-level facility. The purpose of this last section is to enable you to choose among a variety of means for achieving a particular end, given the overall goals of the application program.

*User Interface Dictionary* contains definitions of all the Lisp objects that constitute the user interface system, except for the major top-level defining macros included in the first volume. It has three parts: a dictionary of user-interface functions, macros, variables, and flavors; a dictionary of predefined presentation types; and a graphics dictionary.

## Using the Top-Level User Interface Programming Facilities

### Overview of User Interface Programming Facilities

Historically, the user interface component has often turned out to be the largest part of any large-sized system. The part of the program devoted to problem solving was often overshadowed by the part that took care of presenting data, reading re-

sponses, and handling mouse input and menu display. With Genera, this no longer needs to be the case.

In Genera, Symbolics offers a set of tools that allow the programmer to perform these interfacing tasks easily, by embedding standard Symbolics user interface features directly in application software. This set of tools forms a *substrate* layer upon which programmers can build the interface for their users. This substrate includes:

- A generic read/interpret/redisplay command loop that works for any interactive system.

- A mechanism for presenting many different types of data on the screen and having the system "remember" the objects and their semantics (that is, their current program-defined meaning) so that the user can access them from the displays using the mouse.

- A means of dividing up screen real estate into a programmer-controlled program frame, allowing the programmer to specify relative or absolute sizes of the divisions, the types of display, and so forth.

- An interactive layout design program that aids in the construction of these program frames.

The existence of this substrate means that, *without* doing any low-level programming, you can take advantage of such Symbolics system features as:

- User-input prompts
- Error checking of commands and their arguments
- Mouse-sensitive output
- Windows
- Menus
- Table formatting
- Screen layout
- Scrolling
- Graphics

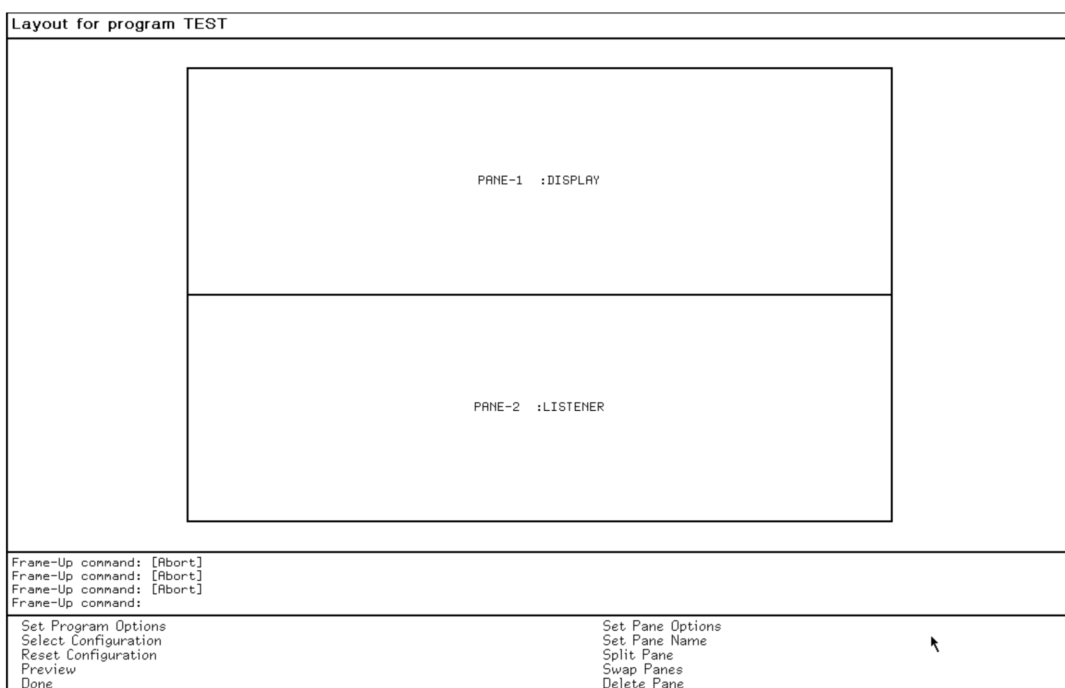
Specifically, the Symbolics Genera user interface management system provides the following facilities for constructing user interfaces to application programs.

- The **Command Processor facility** includes the means to
  - build generic read/interpret/display command loops that work for any interactive system
  - read and parse commands,
  - manage command tables, and
  - define and install command accelerators, which allow users to enter commands by pressing a single key.

  These last two features allow you to register commands and their single-key accelerators in tables that facilitate command lookup. These tables can be organized into a hierarchy so that one table can inherit commands or accelerators from other tables.

- The **Frame-Up layout designer** and the **program framework definition facilities** are tools that you can use to construct a custom-tailored window configuration and program command loop for an application. The interactive layout designer makes it easy to set up several window panes, size them, and specify their types and options by displaying a mock-up of the window geometry as shown in Figure !.

  Figure 36. A typical Frame-Up display




It also allows you to set up and initialize the application's state variables. The

program framework definition lets you specify the set of commands and the command loop in which the user interacts with the program.

- **Output facilities**, including the Showcase display facilities, are built on the presentation and Dynamic Window substrates. They allow you to control the format of displayed data, including character styles, list, table and graph formatting, progress indicators, and graphical presentations. Commands and displayed data can be retained in the output history, can be *mouse-sensitive* in appropriate contexts, and their redisplay can be controlled in several ways.

- **User input facilities**, also built on the presentation and Dynamic Window substrates, include a choice of functions that make it easy to acquire input from users in any of several ways: from the keyboard, from mouse clicks on automatically mouse-sensitive output, or selection from a variety of types of menus. All modes of interaction are available to the user simultaneously; there is no need for either the user or the application programmer to choose a particular style of interaction at any time.

- The **presentation substrate** is the basis for all output and user input facilities, including the Command Processor and the query and menu facilities. The presentation type system, an extension of the Common Lisp type system, centralizes the responsibility for parsing and printing data. It allows programs to *present* output and *accept* input in a variety of ways.

- The **Dynamic Window substrate** allows you to create and use windows that are horizontally and vertically scrollable and that retain history of their output. The characters or graphics displayed and their location on the window, along with the semantic content of the display — that is, the object's type within a given context — are remembered and used so that presented objects are usable as program input and mouse-sensitive according to the current input context.

- The **graphics substrate** comprises a collection of functions and macros that let the user create all sorts of graphic images, manipulate them within interactive streams, and use these images as *presentations*, that is, as context-dependently mouse-sensitive output and input for commands and functions.

The simplest interface to an application program is just the Lisp Listener itself. If you are writing a small program that requires no input from the user and perhaps writes to a file or just displays its results once with no need for formatting, saving, modifying, or redisplaying, then you do not need to use any of the user interface facilities. Your user has to type in a Lisp form to execute your program and that is all.

At the other extreme, you can write an interactive application program that incorporates all of the sophisticated features of the most complicated system facility, including scrollable windows, sub-windows, and text areas, mouse-sensitive text or graphics, menus of all sorts — static, pop-up, scrollable, multiple-choice — automatically self-updating displays, all sorts of mouse-invoked commands, automatic

window reconfiguration, and many other features. The Graphics Editor, Document Examiner, and Display Debugger program incorporate several of these features. The Symbolics user interface facilities allow you to incorporate these features using high-level functions, without having to understand the details of mouse and window programming.

The remaining sections of this chapter introduce the concepts and terminology of user interface programming and describe the user interface and the facilities for programming it in a bit more detail. They also direct you to the appropriate manual sections.

## User Interface Concepts

### User Interaction Paradigm

The user interfaces you can build using the Symbolics user interface substrates support a style of program interaction very different from conventional user interfaces. This interaction style affords the user a variety of modes: typed in commands, menu-selection, mouse activated commands using highlighted text or graphics — all can be available at the same time so a preferred mode can be selected.

Central to the interaction paradigm is a command processor based entirely on the user interface substrates. The Genera Command Processor manages the user interface aspects of all commands. It is similar to the executive or shell of other operating systems. The Command Processor automatically provides command name completion and help strings, as well as a sophisticated defaulting mechanism.

Top-level control for a program is provided by its *command loop*. Most programs' command loops involve the same set of steps:

1. Read a command.

2. Execute the command.

3. Update the display. This may include the redisplay of any modified data structures.

The part of the loop that reads the command builds and then parses a complete "sentence," the command.

Command sentences generally include:

| | |
|---|---|
| Verbs | Specifying the action to be performed (for example, Show File) |
| Nouns | The objects on which the specified action is to be performed (for example, a pathname argument to the Show File command) |
| Modifiers | Specializations introduced via optional, typically keyword, arguments |

Here is an example. In this case, the user has entered the entire command from the keyboard.

```
Show Directory (files [default Q:>jones>*.*.newest])       [verb]
Q:>jones>*.*.newest                                         [noun]
(keywords) :Since (a universal time in the past or a null value)
                                   "8/01/87 00:00:00" [modifier]
```

Here is another example, starting with the output produced by the command of the previous example.

```
Q:>jones>*.*.newest
2920 free, 363880/366800 used (99%, 5 partitions) (LMFS records, 1 = 4544. 8-bit bytes)

      babyl.text.2344      71 316725(8)   !   08/07/87 09:52:33 (08/07/87)
      mail.text.1          19  83691(8)   !   08/07/87 11:45:13 (08/07/87)  Mail-Server

90 blocks in 2 files

Show File Q:>Jones>mail.text.1
```

In the case above, nothing was entered from the keyboard. Instead, the user moved the mouse over the line beginning `mail.text.1` and the system surrounded that line with a box. At the same time the mouse documentation line displayed "**Mouse-L: Show File (file) Q:>Jones>mail.text; ...**" When the user clicked the left mouse button, the system displayed the Show File command and then executed it. So, in this example, the noun has been presented previously and the verb is called upon it.

Users can construct command sentences from keyboard input, mouse input, or a mixture of the two. Mouse handling with respect to the Command Processor is synchronous: mouse and keyboard input can be interleaved in the construction of a command sentence. Thus, for example, if the user types in the Show File command, the pathname argument can be supplied by clicking on a pathname that a previously executed command has caused to be presented as output.

Here is a final very brief example that illustrates how easy it can be, using top-level and intermediate-level facilities, to make commands available as graphic objects.

```
;;Function to present a circle that represents a number:
(defun present-number-as-circle (number)
  (graphics:with-room-for-graphics (t (* number 2))
    (dw:with-output-as-presentation (:object number :type 'integer)
      (graphics:draw-circle number number number))))

;;Draw a handy set of these circles:
(loop for i from 2 to 6 do (present-number-as-circle i))
```

```
;;Define a CP command that takes a number as its argument:
(cp:define-command (com-beep-n-times :command-table 'user) ((number 'integer))
    (loop repeat number do (beep)))

;;Translator that executes command when user selects eligible argument:
(define-presentation-to-command-translator number-to-beeps (integer)
                                                           (number)
  `(com-beep-n-times ,number))
```

Try evaluating this code and then click the left mouse button over one of the circles produced.

To summarize: a Genera user can access a program's command set using various mixes of form filling, arbitrary operation-operand (verb-noun) order, graphics, and text. This is true whether the commands are system commands or ones the programmer creates for an application program.

### The Presentation System

The presentation substrate is the basis for the higher-level user interface programming tools: the Command Processor, and the input and output facilities. The interface-oriented data typing of this substrate, based on *presentations* described next, forms the framework for behavioral consistency across all types of interactions. The substrate affords access to system "hooks" for mouse-sensitive display, context-sensitive input parsing, error checking, prompting, and the like.

### Presentations

The Genera SemantiCue enhanced interaction system is based on the concept of presentations. These are the user-visible representations of the target objects for interactive commands — they are what you see on the screen, either entered by the user or output by the system or by an application. The important thing about such an object is that it has a *presentation type*. Presentation types are the means of organizing the way data is collected from and returned to the user.

### Presentations, Presentation Objects, and Presentation Types

Each time output is done to a dynamic window, one or more presentations are created and saved in the window's history. The presentations record:

- The underlying object, which is the object that was displayed

- The presentation type, which tells how the object is to be classified or interpreted

- Various other data of use internally by the presentation system, such as the location and size of the presentation in the window, relationship to other presentations, and so on

The presentation type of a presentation classifies it according to its extrinsic (programmer-defined) meaning and moreover, describes the translation between the object's internal representation and its printed representation. It is this classification and translation that gives you context-dependent mouse sensitivity practically "for free." This is due to the way in which data types can be overloaded with different user semantics in different situations.

For example, you can have a presentation type called *octal integer* and one called *decimal integer*. These differ from one another because their displays are visibly different. You can, however, display the same object as either one of these presentation types; the data representing the object internally would be the same in either case. The presentation type does not matter in any way to a compiler or arithmetic processor, but it does make a difference to, say, the function that is looking for a particular type of operand for some command. Moreover, you can assign yet another presentation type to the same object in some other context — for example, the presentation type *file-name generation* — if you wish to associate with the object a different meaning to the user. In this last case, you are using the presentation type to reflect user semantics.

The presentation type system associates interactive procedures with type specifiers. An extension of the Common Lisp type system, it allows specification of not only what constitutes an object of a type but also how that kind of object is to be displayed (the type's *printer*) and how to parse user input in looking for the object (the type's *parser*). The relationship between Common Lisp types and the presentation type system is described in the chapter in Part 1 that introduces the use of presentation types: See the section "Using Presentation Types".

### Presentation Functions

Two complementary functions use presentation types: **present** and **accept**. These basic functions are used by all the other user interface facilities such as the Command Processor. **present** and **accept** convert between what the user types or sees on the display and what the system stores internally. This is analogous to what **print** and **read** do.

> Given an object and a presentation type, **present** looks up the printer function for that presentation type and makes a presentation.
>
> **accept** takes a presentation type as an argument and lets the user enter an object of that type either by typing at the keyboard or by clicking with the mouse on some appropriately sensitive object.

An object that has been displayed as some presentation type by **present** is automatically eligible as an argument of that type to **accept**.

The high-level input and output facilities built on the presentation substrate allow you to write your own functions to present and accept data. Lists of the specific tasks you can accomplish with these are included in the descriptions of those facilities. See the section "Using Presentation Types".You can use the presentation substrate directly to accomplish tasks not performed by the higher-level facilities.

See the section "Defining Your Own Presentation Types". See the section "Programming the Mouse: Writing Mouse Handlers".

**Actions and Translations**

The behavior associated with accepting an object originally displayed (presented) by some earlier output function and using it as a command operand is a special case of a general facility known as *translation*. For certain pairs of input context (that is, what is being accepted) and presentation type of the presentation, the system has a set of translation procedures. These procedures take the object in the presentation and return an operand suitable for the requested operation. The user selects from among the possible procedures by the use of specific "mouse gestures" — such as holding down the SHIFT key and pressing the middle mouse button — which are associated with the translator.

For example, you could write a translator that would translate from a user's name, which has presentation type **net:user**, to that user's init file, which has the type **pathname**. Then, when the Command Processor is prompting for a filename for a Hardcopy File command, a user could click a mouse button on some **net:user** user object (which the system displays as the user's name) and hardcopy that user's init file.

One special kind of translator does not actually return an object from **accept**, but instead associates some kind of side effect with the input-context/output-presentation pair. This side effect is some action that helps the user in the process of finding an operand for input. (For example, if the user is expected to input a filename, a useful side effect would be to list the files in a directory when the user clicks on that directory name.) This special kind of translator is called an *action*.

**Mouse Sensitivity**

To say that a displayed item on the screen is *mouse-sensitive* means that pushing a mouse button while the mouse cursor is pointing at it has some defined meaning: it will cause an action or indicate a choice. The window system usually indicates this sensitivity by highlighting the item (that is by drawing a rectangular box around its representation). Every displayed output on a window is potentially mouse-sensitive.

Making mouse-sensitive displays using Genera tools is easy and safe, in contrast to the same task in pre-Genera software releases. You no longer need to learn the innermost details of the window system to do mouse-sensitive output. A single kind of window, namely a *dynamic* window (the associated flavor is **dw:dynamic-window** or **dw:dynamic-window-pane**), remembers all of the following for each item of output:

1. The location of the item on the window

2. What object the item represents

3. What kind of item it is (that is, its presentation type), including display-related options, such as the number base, whether a list was displayed as code, a property list or data, and the like

Which presentations are mouse-sensitive at any given time depends upon two things:

1. The input context. If you are using **accept**, for example, only those presentations that are of the type you are accepting (or can be translated into that type) are mouse-sensitive.

2. Whether any modifier keys are held down. "Modifier keys" include not only the SHIFT key, but also CONTROL, META, HYPER, SUPER, or any combination of these.

Dynamic Windows remember all the output ever done on a window until the user enters the Clear Output History command. This includes output that has been scrolled off the top of the window. Dynamic Windows can be scrolled forward and back, as well as left and right (if any of the output has been written beyond the right-hand edge of the window). Any output presented with the appropriate type, even if it is not visible in the window, can become mouse-sensitive — if you scroll the window back to it and put the mouse cursor over it, it will be highlighted.

Ways to display output so that it will be mouse-sensitive in the correct context and ways to make use of this output are discussed in the chapters on the Command Processor and on presentation types.

See the section "Managing the Command Processor".
See the section "Using Presentation Types".
See the section "Defining Your Own Presentation Types".

**Note**: You can still program the mouse at the lowest level if it is really necessary. See the section "Programming the Mouse: Writing Mouse Handlers".

## The Facilities

### Command Processor Facilities

The core of the Command Processor facilities, which are built on the presentation and Dynamic Windows substrates, is the **cp:define-command** macro. Another very similar macro, **dw:define-program-command**, is included in the program framework definition facilities. Both of these macros allow you to define commands that the user can enter making use of system-provided features, such as prompting, help, completion, and the like. The commands you define are installed in a command table that you specify. The difference between the two macros is that commands defined with **dw:define-program-command** have access to program state variables and can be easily added to your program's menus, unlike those defined with **cp:define-command**.

To find out all you need to know about building your own CP commands and using them from within your programs, see the section "Managing the Command Processor". To find out how to define commands within your own program framework and how to build and manage your own command loops, see the section "Defining Commands within Your Own Framework".

### Program Frameworks

Most interactive programs consist of an "infinite" loop that repeatedly cycles between command reading, command execution, and display update. Examples of such programs are the Lisp Listener, the editor program Zmacs, and the mail reading program Zmail. Other programs, which have a similar sort of infinite loop but very different interaction styles, are the Graphic Editor and the Frame-Up Layout Designer.

A *program framework* organizes such an interactive program. It contains the following four related items:

- A window-layout declaration, specifying the division of the screen into panes
- A set of window display definitions, specifying type and characteristics for each pane
- A command loop as described above
- A set of commands

Symbolics' program framework facilities include a set of tools that enable you to define this organization. The centerpiece of these facilities is the macro **dw:define-program-framework**. This macro creates a flavor of program for the framework. It has a great many options, including ones that let you:

- Specify redisplay information for several window panes
- Specify how panes are to be arranged on the screen geometrically
- Define a command-defining macro for the program
- Specify state variables for the program
- Specify how to select the program

These are the types of panes you can specify:

**:interactor** Accepts forms or CP commands and displays output, like a Lisp Listener

**:display** Displays (possibly mouse-sensitive) output only, like the main pane in the file system editor

| | |
|---|---|
| **:title** | Displays a fixed title, like the title pane in the Flavor Examiner |
| **:command-menu** | Presents a fixed set of mouse-sensitive commands, like the middle pane in a Zmail window |
| **:accept-values** | Presents an accept-values type of menu, usually for selecting values of program state variables |
| **:listener** | A window that accepts forms or commands but does not display output |

The redisplay options for a pane specify whether, when, and how the contents of the pane should be updated.

The Frame-Up Layout Designer lets you build a **dw:define-program-framework** macro interactively — to help you write the code that specifies your program framework, you can call up a special window and configure your application window using menus. Once you have generated your program-framework defining macro, you can edit it to refine and augment the results. All of this is explained in the introductory chapter in Part 1. See the section "Defining Your Own Program Framework".

To find out how to create commands for your own application program framework, see the section "Defining Commands within Your Own Framework".

Other chapters, in Part 2, present information about low-level facilities for adjusting the user interface within your own program framework. For example:

See the section "Managing Your Program Frame".

See the section "Programming the Mouse: Writing Mouse Handlers".

See the section "Displaying Output: Replay, Redisplay, and Formatting".

### Output Facilities

The top-level facilities for displaying output, collectively referred to as Showcase are described in Part 1. See the section "Using Presentation Types".

Depending on how much control you want over the presentation type of a display, you have three options for facilities with which to present output:

1. Use any printing operation like **print** or **format**. This presents output as a simple default presentation type. When you use **print**, for example, on an object, that object is presented with the presentation type **sys:expression**.

2. Use **present** or **present-to-string** to specify the exact presentation type you wish to use. You might use this, for example, if you have an object that *is* an integer but *represents* a universal time. In general, **present** is used for invoking the printer function associated by your program with a presentation type on the object you are displaying.

3. Use the macro **dw:with-output-as-presentation** to specify both the presentation type and the manner in which the data is to be presented on the screen. All output drawn on the screen inside this form becomes part of the presentation. You can use this to incorporate graphics into your presentation.

However you choose to present your output, you also have control over the character style used to display it, as well as over other aspects, such as underlining, abbreviating, filling, and indenting. See the sections "Formatting Text" and "Formatting Textual Lists".

If the object you want to display in your output is a list or sequence, there are facilities to help you format it. In particular, a number of functions and macros make construction, arrangement, and labeling of tables very easy. See the section "Formatting Tables". Genera provides two functions and two macros that allow you to display data structures such as trees and lattices graphically. See the section "Formatting Graphs". Remember that any output you do, whether displayed in a nicely formatted table or in a graphical tree structure, has the potential to be selected by the user for input, and it has the potential for being automatically updated if the displayed data changes.

### Input Facilities

The top-level facilities for obtaining user input, collectively referred to as SemantiCue are described in Part 1. See the section "Using Presentation Types".

Just as there are three ways to present output, depending on how much control you want to have over its appearance and underlying type, there are three ways to accept input:

1. Presentation-based input operation, for example, **read**, can accept its input in the form of mouse clicks on sensitive items (as well as in the form of keystrokes from the keyboard). The Command Processor also accepts both whole commands and single arguments to those commands from the mouse; thus previously entered commands and other output are often sensitive while the Command Processor is waiting for you to type a command.

2. The function **accept** is used to control this behavior more precisely. You can specify not only the presentation type, but also data arguments (to restrict the range of the allowed input) and presentation arguments (to affect aspects of input interaction, such as number base). **accept** is especially powerful inside a **dw:accepting-values** form, where several calls to **accept** become a single *menu* of choices. Related to **accept**, and used in similar situations, are the functions **prompt-and-accept** and **accept-from-string**.

3. The macro **dw:with-presentation-input-context** allows you to control exactly what input is sensitive while performing any arbitrary input operations. This is the most flexible input control you can use: inside the macro's body, you specify what the keyboard-reading operation is, and also what to do when a mouse operation is performed.

There are additional facilities for accepting single objects from menus — **dw:menu-choose** and **dw:menu-choose-from-set** — and for accepting multiple objects — **dw:accept-values**, **dw:accept-variable-values**, and **dw:accepting-values**.

Several chapters in Part 2 explain how to use the advanced facilities of SemantiCue: See the section "Defining Your Own Presentation Types". See the section "Managing Your Program Frame". See the section "Programming the Mouse: Writing Mouse Handlers".

### Presentation Substrate

Two chapters in this manual discuss the presentation-related facilities in more detail and explain how to use them.

An introductory chapter in Part 1 explains how to make use of them. See the section "Using Presentation Types".

A more advanced chapter in Part 2 explains presentation types in detail and shows you how to use the facilities of the substrate to:

- Define your own presentation types
- Declare and use specialized presentation input context
- Create and use translating mouse handlers
- Dissect presentations and manipulate presentation-type arguments

See the section "Defining Your Own Presentation Types".

### Window Substrate

The *window* system is the second major source of user interface substrate facilities, after the presentation system. A window can be *static* or *dynamic*. Output to static windows is relative to an unchanging set of window coordinates: once a static window is full, it must be cleared entirely or partially before new output can be done without overwriting previous output. Dynamic Windows, on the other hand, are scrollable in both the vertical and horizontal dimensions; they have a definite origin (0, 0), but an indefinite length and width. Scrollability is a basic feature of Dynamic Windows and does not require the explicit use of special procedures as in the case of static windows.

Associated with the scrollability of Dynamic Windows are the concepts of *output history* and *viewport*. You do not have to clear a Dynamic Window to avoid overwriting previous output. New output, unless specifically directed otherwise, is appended to the bottom of the window's history, that is, at the end of all previous output to the window. The window is automatically scrolled so that the current viewport — the visible portion of the window — shows the new output. Scrolling backwards through the history makes previous output viewable.

With the use of presentation types for doing output to a Dynamic Window, not only is the previous output retained and viewable, but its semantic content is also remembered. That is, links to the objects represented by displayed presentations are maintained so that the objects themselves remain accessible and usable as current program input. This capability is central to the SemantiCue input system. In the appropriate input context (established by your program), the displayed presentations are automatically mouse-sensitive. Automatic mouse-sensitivity is another point where Dynamic Windows depart from static windows; with a static window, mouse sensitivity must be provided through explicit procedures associated with output operations.

The window substrate contains all the facilities for creating and controlling windows, both dynamic and static. This includes control of window and window pane sizes, window pane labels, margins, margin labels, borders, scrollability, scroll bars, end-of-page modes, exposure and activation, blinkers, character styles, graying, and notification handling. It also includes control of the mouse.

See the section "Using the Window System".

Higher-level aspects of window programming are covered in part 2: See the section "Defining Your Own Program Framework". See the section "Programming the Mouse: Writing Mouse Handlers".

### The Graphics Substrate

The graphics substrate contains the following facilities:

- Basic drawing functions for all sorts of graphics objects, such as lines, polygons, arcs, circles, splines, and the like, together with drawing options such as texture, filling, thickness.

- Coordinate system facilities for scaling and rotating output.

- Graphics presentation facilities that let you use graphics objects as presentations.

- Low-level facilities that let you use graphic drawing and encoding primitives.

The graphics substrate is *generic*. This means that there is a uniform interface for accessing the different facilities provided by different graphical output devices.

The graphical output devices supported in Genera are:

- The standard bitmap screen and its color analogues

- The LGP2/LGP3 laser printer

- Raster arrays intended for copying to the screen at some later time

To say that the substrate is generic does not mean that all programs written for

one device using the substrate will run on all devices. In particular, there is no guarantee that every function will degenerate cleanly when the functionality requested of it does not exist. Every attempt has been made to do this where it is practical and reasonable. For instance:

- Using the **:color** option on a black and white screen produces a stipple pattern approximating the intensity of the desired color.

- Drawing with the **:opaque nil** option on the LGP2/LGP3 draws special images to simulate this, even though the PostScript imaging model does not normally support color mixing.

In other cases, the limitations of the device are just too great.

- The LGP2/LGP3 is not capable of drawing in **:alu :flip**, because it does not necessarily retain a complete image of the current partially drawn page.

- Graphics sent to character-only devices, for example, **with-output-to-string** or ASCII terminals, is not simulated by a clever use of *, +, |, -, / and \.

If you have any concerns about the compatibility of the drawing options you are using with the devices you need to output onto, you should check the corresponding dictionary entries.

Use of the graphics substrate is documented in Part 2. See the section "Creating Graphic Output".

## Using Presentation Types

The purpose of this chapter is to introduce the general concept of the presentation type system in just enough detail so that the reader will be able to make use of the predefined types in the course of using the top-level user interface programming facilities. Information on how to create new types is presented elsewhere in this manual. See the section "Defining Your Own Presentation Types".

Recall from the introductory chapter of this volume (see the section "Overview of User Interface Programming Facilities".) that the basis of all the facilities for providing and obtaining user information are the functions **accept** and **present**, which require presentation types for the objects that they are reading or displaying. All the higher-level user interface facilities, such as the command processor, use these two functions to obtain their operands and display their output. For this reason, whenever you use these facilities to write code to obtain input from the user or to display output that might be subsequently read back in, you need to supply the appropriate presentation type information.

## The Presentation Type System: an Overview

### What a Presentation Type Is

A presentation type, like a Common Lisp data type, is a (possibly infinite) set of Lisp objects together with information on how objects in the set are to be printed and parsed. These sets constitute categories of objects acceptable as arguments for the user interface functions **accept** and **present** or for higher-level interface functions built upon those. A presentation type indicates something of the semantics of the members of the set. For example, a list may be just a list, or it may be a **sys:form**, that is, a piece of program to be evaluated.

Every structure, flavor, and Common Lisp data type is also a presentation type. The result of a call to the **present** function is called a *displayed presentation*, or, simply, a *presentation*. A presentation can be thought of as comprising the object itself and the presentation type.

A presentation type is itself a Lisp object, and the set of all presentation types is a presentation type, called **dw:presentation-type**. For more detailed information on the nature and structure of presentation-type objects: See the section "Defining Your Own Presentation Types".

The predicate **dw:presentation-type-p** can be used to ask whether a given object is a presentation type. Presentation types are arranged in a partial ordering defined by a subset relationship; this relationship can be investigated using the **dw:presentation-subtypep** function.

A presentation type specifier can be either a symbol or a list. A symbol denotes a presentation type that has no arguments. A presentation type specifier that is a list is arranged as `((name . data-arguments) presentation-arguments)`. Its parts are:

- The *name*

  The name is a symbol that names the presentation type. This name may come predefined by the system, or be defined by the user.

- The *data arguments*

  The data arguments further distinguish which objects are being described by the type. They always denote a subset of the objects that would be denoted without the data arguments. For example, when asking for an integer, you can ask for an integer in a certain range by giving data arguments to the **integer** presentation type.

- The *presentation* and *meta-presentation arguments*

  The presentation and meta-presentation arguments do not distinguish between objects. Instead, they control how the objects are presented to or accepted from the user. For example, a presentation argument to the **integer** presentation type specifies the base in which an integer should be printed or read.

  Meta-presentation arguments to presentation types are arguments that are directly understood by **accept** or **present**. They are not dependent on the parser

or printer of any particular presentation type, and can therefore be used as arguments to any type.

At present, a single meta-presentation argument is available, **:description**. Using this keyword option, you can control the prompt created by **accept** for soliciting input of a given type. This allows you to make the prompt more appropriate to the current conceptual context. For example, instead of just asking for a integer, you could do something like:

```
(accept '((integer) :description "the number of copies")) ==>
Enter the number of copies: 5
5
((INTEGER) :DESCRIPTION "the number of copies")
```

The syntax for distinguishing the different parts of a presentation type and examples are presented in a later section. See the section "How to Specify a Presentation Type".

**What Presentation Types Are For**

Presentation types are for use with the functions **accept** and **present**, which are the means of obtaining input from or displaying output for the user. They are also used by mouse handlers, which are additional facilities for letting the user manipulate objects. (See the section "Programming the Mouse: Writing Mouse Handlers".) It is the specification of presentation types for these facilities that makes possible advanced features such as context-dependent mouse sensitivity — including the mouse documentation line — automatic completion, an automatic defaulting mechanism, and the like.

The Genera presentation type system is an extension of the Common Lisp (CL) type system. Common Lisp allows the programmer to specify data types for variables and to extend the type system's primitive data types by adding new types based on those primitive types. This typing system can be used to arrange for efficient storage allocation by the compiler's code generator. In addition, the typing information arranges for the proper type-specific procedure to get called, given an overloaded operator such as a Symbolics Common Lisp (SCL) generic function like +. The current CL typing system, however, is not useful for differentiating among different types of operands to be supplied by the user interactively — mainly because it lacks arguments to control presentation independent of arguments to control set membership and it lacks a way to create subtypes that are purely semantic.

The Genera presentation type system takes the CL typing system several steps farther by adding the capability of associating one or more of the following features with a presentation type:

- a history of objects previously accepted as instances of the type,

- a specialized parsing function for recognizing an object of the type,

- a display function for an object of the type, including, for example, possible display as a graphics object,

- an allowance for user specification of different ways of displaying an object of the type — that is, of different *viewspecs*,

- functions to be used for determining whether this type is a subtype of some other type, and

- specification of various options concerning how the object is to be described by the system for the user and how it might appear in various kinds of menus.

Because the functions **accept** and **present** use these features, higher-level facilities can easily implement things like input parsing and mouse sensitivity. For example, any CP command that produces output can call **present**, specifying the presentation types of the objects to be displayed. A subsequent CP command that requires input calls **accept**, also specifying a presentation type. When the type of a displayed object matches the one required, the system makes the object's display mouse-sensitive and, if the user clicks the mouse over it, supplies the object to the command. This automatic mouse sensitivity comes about because there is an identity translator for every presentation type that translates from a presentation of that type to the object itself as acceptable input in the given context.

Mouse sensitivity and translators form an advanced topic that is covered elsewhere. (See the section "Programming the Mouse: Writing Mouse Handlers".) For now, you can simply view the mouse sensitivity provided by **accept** /**present** type matching as automatic.

When implementing an applications system, you can define and use your own presentation types according to the meanings assigned to objects in your system. In an accounting program, for example, this may mean a dollar amount, and be read with exactly two digits after the decimal point. In a numerologist's program, it is just a number represented in binary; it can be displayed in hexadecimal, octal, or binary. If both systems are displaying output on the screen at once, the numbers from the numerology program are not sensitive in the context of commands issued within the accounting program, even though they both use numbers: the numerologist was not working with dollar amounts.

Presentation types distinguish the different *uses* of a Lisp type — whether a number, a list, a flavor instance, or something else — for the external user. Not only does this differentiation appear in the syntax used to express it, but it allows SemantiCue to make appropriate quantities available for selection with the mouse (via mouse sensitivity), without also making inappropriate ones available.

### How Presentation Types Relate to Common Lisp Types

The presentation type system subsumes the CL type system: any CL type is also a presentation type. Because the presentation type that corresponds to a given CL type has the additional properties described above, it is much more useful as an object specifier.

A few specific examples can illustrate some of the benefits of this system.

Consider the CL type **string**. The CL system "knows" how to test for strings. You can supply an argument to the type specifier to indicate the size (length) of the string. You can define new types using **string** with its type specifier and one or more of **not**, **and** or **or**, using the CL **deftype** special form.

The *presentation type* **string**, on the other hand, allows you to specify additionally a list of characters that serve as delimiters during parsing of the string by the **accept** function. More importantly, the **string** presentation type supports a type history: when `(accept 'string)` is invoked, the system offers the most recently accepted string as the default.

An important difference between CL types and presentation types is the fact that there can be presentation subtypes that are defined by semantic specialization, that is, specialization according to the use the members of the type is put to. An example is the presentation type **symbol-name**, which is equivalent to, but not the same as, the presentation type **string**. This means the following things:

- **symbol-name** and **string** are equivalent types: every presentation that is a member of one is a member of the other.

- **symbol-name** inherits the input history of **string**, but **string** history does not include strings entered as symbol names.

- When **accept** is invoked specifying either **symbol-name** or **string**, objects presented as objects of either type are mouse sensitive.

The difference between the two types is in their input histories and in the fact that symbol names are parsed differently from strings.

Another example is the presentation type **fs:directory-pathname**, which is a specialization of **pathname**. In this case, **fs:directory-pathname** is a proper subset of **pathname**: objects presented as directory pathnames will be acceptable to (with mouse sensitivity) calls for pathnames, but not *vice versa*.

### How to Specify a Presentation Type

The syntax for a presentation type specifier is one of:

*presentation-type-name*

A symbol defined either by the programmer or the system to name a presentation type

((*presentation-type-name* [*data-arg1 data-arg2* ...])
[*presentation-keyword1 presentation-arg1*
*presentation-keyword2 presentation-arg2* ...]
[*meta-presentation-keyword meta-presentation-arg*])

A list of lists, where the items in square brackets are optional. The first item in the outer list is itself a list, which contains

the name of the type and zero or more data arguments. This inner list is similar (in many cases identical) to a CL type specifier. The remaining items in the outer list are keyword-value pairs of presentation and meta-presentation arguments.

(*presentation-type-name* [*data-argument1 data-argument2* ...])
A list in which the first item is the presentation type and the remaining items are data arguments.

Keep in mind that data arguments always serve to *restrict* (except in the case of **or**) the set of objects included in the type, never to expand the set. Presentation arguments never do any such restriction — they only control how an object should be presented or accepted. Presentation arguments are always keyword arguments.

Note that whenever you need to specify presentation arguments, you *must* use the double set of parentheses. They distinguish unequivocally between data arguments and presentation arguments. You can omit the outer parentheses when you do not need to supply any presentation arguments for the presentation type. You can omit both sets of parentheses when you have neither data nor presentation arguments.

Consider the **integer** type. You can control the printed representation of an integer by specifying its base:

```
(accept '((integer 0 100) :base 16)) ==>
Enter a hexadecimal integer greater than or equal to 0
and less than or equal to 64: e
14
((INTEGER 0 100) :BASE 16)
```

**:base** is a presentation argument. Internally a `14` is still returned, but externally it is displayed as an `e`.

It is important to note that use of a presentation type with no data arguments always refers to the entire set of objects of that type. For example, if you have defined a flavor named **ship** with an instance variable **tonnage,** then **ship** always refers to the entire set of **ship**s. Data arguments can only restrict that set, not extend it. Thus, if you additionally define a data argument named **tonnage** for the presentation type **ship**, then **(ship :tonnage 50000)** is always a subtype of **ship**; it can never include anything not in **ship**. See the section "Type Inheritance".

Here is a set of examples that cover most of the possible variations of presentation type specifier syntax. The examples make use of the basic presentation input/output functions **accept** and **present**, which are described in detail, after the predefined presentation types have been introduced. The presentation types used in these examples are all predefined in Genera.

**Simple Predefined Types**

```
(present 4 'integer)

(present "abcd" 'string)
```

```
(present t 'boolean)

(present (net:find-object-named :network "CHAOS") 'sys:expression)
```

Any of the presentation types in the examples above can be expressed alternately using parentheses, as

```
(present 4 '((integer)))
```

or

```
(present 4 '(integer))
```

and so forth.

**Using Data Arguments**

```
(accept '(string 5)) =>
Enter a string 5 characters long [default "right"]: abcde
"abcde"
(STRING 5)
```

Note that, when using **present**, you must not mislead the system about the object you are presenting. No checking is currently done to make sure that the object you present satisfies the requirements for the type. For example, if you enter `(present "left" '(string 5))`, "left" would be accepted as a **(string 5)** type. This would cause anomalous behavior later on.

Other examples:

```
(accept '(member harpo chico groucho))

(accept '(dw:member-sequence (descartes sartre sinatra)))

(accept '(type-or-string net:host))

(present :all '(token-or-type (("None" . :none)
                                ("All" . :all))
                               (member Abel Baker Charlie)))

(accept '((integer (2) (9)))) ;The parens around the arguments
                              ;indicate exclusive limits
```

The predefined types **member**, **dw:member-sequence**, **type-or-string** and **token-or-type** are all defined in the "Dictionary of Predefined Presentation Types".

**Using Presentation Arguments**

```
(present (+ 7 5)  '((sys:expression) :base 8 :radix t))

(present 1420088400  '((time:universal-time) :timezone -1))
```

Many other presentation type arguments have to do only with how items are parsed for **accept**. There are more examples shown as part of that function's definition.

**Compound Types**

```
(accept '(and integer (satisfies oddp)))
```

alternatively:

```
(accept '((and ((integer)) ((satisfies oddp)))))

(accept '((and ((number)) ((not integer)))))
```

**All Types of Arguments**

```
(accept '((dw:member-sequence ("a" "e" "i" "o" "u"))
           :description "a vowel"))
=>Enter a vowel:

(accept '((alist-member :alist (("Number 1" . first)
                                ("Number 2" . second)))
          :convert-spaces-to-dashes t))
=>Enter Number-1 or Number-2: Number-1

(cp:define-command (com-draw-rectangle :command-table 'user)
   (&key
    (alu '(dw:member-sequence (:draw :erase :flip))
         :default :draw)
    (thickness '((integer 1 10) :description "Width, pixels")
               :default 1)
    (dashed 'boolean :default t)
    (dash-pattern '((sequence integer) :echo-space nil)
                  :default '(5 5)))
   (graphics:with-room-for-graphics (t 100)
   (graphics:draw-rectangle 0 100 100 0 :alu alu
                                  :thickness thickness :filled nil
                                  :dashed dashed
                                  :dash-pattern dash-pattern)))
```

## Predefined Presentation Types

The Genera SemantiCue system includes a large number of predefined presentation types. Your programs can use these to specify argument types for your own user interface functions in many cases, thus avoiding the need to define your own presentation types.

Because many of the predefined types are defined for internal system use, only a small proportion of them are of interest to the applications programmer. The Common Lisp types **hash-table** and **compiled-function**, for example, do not normally occur in end-user-visible places.

In this section, we list the predefined types likely to be used by application programmers. Some, like **integer**, **string**, and **boolean**, are encountered frequently in all kinds of programs. Many others, like **sys:code-fragment** and **net:network**, are more specialized in their uses.

In any case, all of the types included here are also documented as individual dictionary entries. See the section "Dictionary of Predefined Presentation Types". Also, many of them are defined in the file sys:dynamic-windows;standard-presentation-types.lisp, where you can look for models when defining your own types. The dictionary entry for each type notes whether it is one of those included in this file.

There are three categories of documented types:

- Common Lisp Presentation Types (a subset of SCL types)
- Other Symbolics Common Lisp Presentation Types (besides CL types)
- Other Presentation Types

The category "Other Presentation Types" includes potentially useful types exported from packages other than Symbolics Common Lisp; most of them are in the specialized-use category.

**Common Lisp Presentation Types**

| | |
|---|---|
| **and** | **package** |
| **character** | **pathname** |
| **integer** | **satisfies** |
| **keyword** | **sequence** |
| **member** | **string** |
| **not** | **symbol** |
| **null** | **symbol-name** |
| **number** | **t** |
| **or** | |

When you use one of the Common Lisp types as a presentation type, you must specify it using the correct syntax. See the section "How to Specify a Presentation Type". Note that **t** is the supertype of all other types. The type **t** has no parser of its own, however, so you can not use it with **accept**.

Also note the compound types **and** and **or**. These provide a way of extending the type system by combining types. Another Common Lisp type, **satisfies**, is usually used within compound types based on **and**.

Here is an example showing how to accept only odd integers:

```
(accept '(and integer (satisfies oddp))) ==>
Enter an integer satisfying ODDP: 53
53
(AND INTEGER (SATISFIES ODDP))
```

Calling **present** with objects as **or** types is not useful; these are useful only for

**accept**. Several extensions via the **or** type are already among the predefined Symbolics Common Lisp types, listed below. In general, you should call **present** with the most specific type available.

**Other Symbolics Common Lisp Presentation Types**

| | |
|---|---|
| **alist-member** | **inverted-boolean** |
| **boolean** | **null-or-type** |
| **character-face-or-style** | **sequence-enumerated** |
| **character-style** | **subset** |
| **character-style-for-device** | **token-or-type** |
| **instance** | **type-or-string** |

Three of the compound types mentioned above, **null-or-type**, **token-or-type**, and **type-or-string**, are compound types. **sequence-enumerated** is also a compound type, one for accepting or presenting a sequence of objects, each of a specified presentation type. Using **alist-member** to accept an object is similar to using a menu; the object is represented by a user-visible string different from its internal representation. **subset** provides a way of accepting or presenting zero or more objects in a set.

The **instance** presentation type is typical of the many Common Lisp types (like **hash-table** and **compiled-function**, mentioned at the beginning of this section) unlikely to be useful in many situations. For one thing, you cannot type the name of an instance at an **accept** function; it either has to be entered via a mouse click on a previously presented **instance** object, or through **accept**'s default mechanism. It is documented as a dictionary entry merely as an example of such types. You should not ordinarily need it.

The remaining presentation types, listed below, provide potentially useful I/O capabilities, spread across a broad spectrum of system software and functionality. We encourage you to study this list, and the corresponding dictionary entries, for types of use in your applications. Only four of these types are discussed here, **sys:expression**, **sys:form**, **sys:code-fragment** and **dw:no-type**.

Other Presentation Types

| | |
|---|---|
| **cp:command** | **si:input-editor** |
| **dw:accept-values-sample** | |
| **dw:member-sequence** | **sct:system** |
| **dw:no-type** | **sct:system-version** |
| **dw:out-of-band-character** | **sys:code-fragment** |
| **dw:raw-text** | **sys:expression** |
| **fs:directory-pathname** | **sys:font** |
| **fs:wildcard-pathname** | **sys:form** |

| | |
|---|---|
| **net:host** | **sys:flavor-name** |
| **net:local-host** | **sys:function-spec** |
| **net:namespace** | **sys:generic-function-name** |
| **net:namespace-class** | **sys:stack-frame** |
| **net:network** | **time:time-interval** |
| **neti:local-network** | **time:time-interval-60ths** |
| **net:object** | **time:timezone** |
| **sys:printer** | **time:universal-time** |
| **neti:protocol-name** | **dw:dynamic-window** |
| **neti:site** | **tv:window** |
| **net:user** | **zwei:buffer** |

Certain presentation types are standins for a hierarchy of more specific presentation types determined by the type of the object presented. Handler matching and mouse sensitivity use the more specific type. These standin presentation types are **sys:expression**, **sys:form**, and **sys:code-fragment**. We mentioned earlier that the number of presentation types is large, including all structures, flavors, and a variety of little-used (for program I/O) Common Lisp types. The standin types are the links between these types and the presentation system.

To consider one case, **sys:expression** is a supertype of all Common Lisp types (except **t**), and is the type from which they inherit their printer and parser functions when these are not otherwise defined for them. The **instance** type, for example, inherits from **sys:expression** and, through **instance**, so do all flavors. (The undocumented **structure** presentation type plays an analogous role for all structures.)

All values printed by the Lisp Listener are **sys:expression** presentations. For purposes of mouse sensitivity, however, handlers whose *from-presentation-type* is a supertype of the actual type of the value are considered, as well as handlers whose *from-presentation-type* is a supertype of **sys:expression**. Specifically, the result of `(present 5 'sys:expression)` will be treated by mouse handlers as though it were the type **fixnum**, and `(present 5 'sys:form)` will be treated as though it were `(and form fixnum)`.

**sys:expression** can provide any of its subtypes with a type history as well as a printer and parser. Some of the Common Lisp presentation types listed in an earlier table make use of **sys:expression**'s type history, for example. This is true of the **integer** presentation type. Through the **number** presentation type, to which it and all other numeric types are subtype, it has access to the history of **sys:expression** objects previously accepted.

The expression history is the source of default values offered when types inheriting this history are accepting objects. When used by **integer**, the expression history is "pruned" of non-integer objects; an appropriate default value can thereby be offered. Other types with access to the expression history benefit from a similar pruning process.

**dw:no-type**, as its name might suggest, is not really a presentation type. Rather, it is a type for use by mouse handlers that are intended to be active only over blank areas of a window, not over presentations.

All presentation types listed in the tables in this section are documented in a separate dictionary. See the section "Dictionary of Predefined Presentation Types".

### Using Presentation Types for Output

The reason for using presentation functions for doing program output is so that the objects presented will be acceptable to input functions. Suppose, for example, you present an object as a microcode version number. When a command that takes a microcode version number as an argument is issued or when a mouse translation function is "looking for" such a thing, the system will make that object mouse sensitive and the parsers in question will accept the object as input.

A presentation includes not only the display itself, but also the object presented and its presentation type. When presentations are output to Dynamic Windows, the object and presentation type are "remembered" — that is, the object and type of the display at a particular set of window coordinates are recorded in the window's *output history*. Because this information remains available, previously presented objects are themselves available for mouse input to functions for **accept**ing objects. There is an extensive collection of facilities for generating output, which can be textual or graphical and which can make use of the presentation system or not. See the section "Presenting Formatted Output".

SemantiCue provides three top-level facilities for presentating output: **present**, **present-to-string**, and **dw:with-output-as-presentation**. **present** is the basic function for outputting presentation objects. **present-to-string** is a special-purpose function intended to present an object as an output string in such a way that it can subsequently be **accept**ed as input. **dw:with-output-as-presentation** is a powerful macro that you can use to control the appearance of the output object. If appropriate in your application, you can use this macro to present a string as a graphic display, for example, and still have the string object be available for program input via the mouse.

The Showcase display facilities include a variety of top-level functions and macros for generating formatted and redisplayable output. These are described in later chapters, "Presenting Formatted Output" and "Displaying Output: Replay, Redisplay, and Formatting".

#### The present Function

The **present** function does not determine the exact form that the output takes, that is, its printed representation. This is determined by the presentation type that you specify. The definition of the presentation type includes a printer function specifying the details of the output display. The following examples show presentations of **inverted-boolean** and **character-style**:

```
(present t '(inverted-boolean)) ==> No
#<DW::DISPLAYED-PRESENTATION T (INVERTED-BOOLEAN) 507772242>
```

```
(present
 (si:parse-character-style
  '(:swiss :bold :large))) ==> SWISS.BOLD.LARGE
#<DW::DISPLAYED-PRESENTATION #<CHARACTER-... CHARACTER-STYLE 507774656>
```

You have the option of defining your own presentation type, with its own printer function, but many, like the two example types above, have already been defined for you. (For a list of predefined types, see the section "Predefined Presentation Types". Reference documentation for each listed type is included in a user interface dictionary. See the section "Dictionary of Predefined Presentation Types".)

**The present-to-string Function**

The **present-to-string** function converts an object of a specified presentation type into a string. It has options that allow you to specify a string to hold the output, an index into the string, and whether or not the object should be presented so that it can be parsed later by **accept**. Example:

```
(setq output-string
       (present-to-string
        '(si:com-show-file (#P"V:>birch>lispm-init.lisp.newest"))
        'cp:command :acceptably t)) ==>
"Show File V:>birch>lispm-init.lisp.newest"
```

Note that the presentation type of the result is **sys:expression**, not **string** and not **cp:command**. You cannot use (accept 'cp:command) to use it as input. Instead, use:

```
(accept-from-string 'cp:command output-string) ==>
(SI:COM-SHOW-FILE (#P"V:>birch>lispm-init.lisp.newest"))
((CP:COMMAND :COMMAND-TABLE #<CP::COMMAND-TABLE User 410016565>))
```

**The dw:with-output-as-presentation Macro**

If you wish to output a presentation but want to specify exactly how you want it to look, the **dw:with-output-as-presentation** macro provides such a capability. It uses your code to create the display rather than the printer belonging to the specified presentation type. The following function of two arguments presents the first argument, *object*, with presentation type *type* as a pair of overlapping circles. (**:single-box t** is specified here so that the graphical objects produced together form a single presentation.)

```
(defun present-object-of-type
        (object type &optional (stream *standard-output*))
  (dw:with-output-as-presentation
     (:single-box t :stream stream :type type :object object)
   (graphics:with-room-for-graphics (t 50)
    (graphics:draw-circle 50 20 25)
    (graphics:draw-circle 70 20 25))))
```

Try calling this function with "ABC" as the first argument and 'string as the second. Then, do (accept 'string) and click on the graphic. You will see that a perfectly normal **string** object is returned, despite its unorthodox presentation. (A sep-

arate graphics facility is provided for creating graphic presentations. See the function **graphics:with-output-as-graphics-presentation**. Its functionality is the same as that of **dw:with-output-as-presentation**.)

**dw:with-output-as-presentation** is usually used for outputting text — for example, to make stack frames, which have no particular representation, mouse-sensitive in the debugger.

## Using Presentation Types for Input

### Accepting Single Objects

These are the facilities for accepting single objects:

- **accept**
- **prompt-and-accept**
- **accept-from-string**
- **dw:menu-choose**
- **dw:menu-choose-from-set**

The primary facility for accepting input of presentations is the Symbolics Common Lisp function **accept**. Presentations can be accepted via keyboard or mouse input. Characters typed in at the keyboard in response to an **accept** prompt are parsed, and the presentation they represent is returned to the calling function. Alternatively, if a presentation of the type specified by the **accept** call has previously been displayed, the user can click on it with the mouse and **accept** returns it directly (that is, no parsing is required).

Examples:

```
(accept '(string) ==>
Enter a string: abracadabra
"abracadabra"
STRING

(accept 'string) ==>
Enter a string [default abracadabra]: abracadabra
"abracadabra"
STRING
```

In the first **accept** function, "abracadabra" was typed at the keyboard. In the second **accept**, the user clicked on the keyboard-entered string of the first function. In both cases, the string object was returned.

Typically, not just any kind of object is acceptable as input. Only an object of the presentation type specified in the current **accept** function can be input. The

**accept** function establishes the current *input context*. For example, if the call to **accept** specified an integer presentation type, only a typed in or a displayed integer would be acceptable. Numbers displayed as integer presentations would, in this input context, be mouse-sensitive, but those displayed as part of some other kind of presentation, for example, a file pathname, would not. Thus, **accept** controls the input context and thereby the mouse sensitivity of displayed presentations.

A mouse gesture on a presentation of a type different from the input context may cause translation to an acceptable object. For example, you could make a presentation of a file pathname translate to an integer — say, its length — if you want. It is very common to translate to a command that operates on a presented object.

We say above that the range of acceptable input is, typically, restricted. How restricted is strictly up to you, the programmer. Using compound presentation types like **and** and **or**, and other predefined or specially devised presentation types gives you a high degree of flexibility and control over the input context. Consider the following example:

```
(accept '(or (integer 1 4)
             (dw:member-sequence
                ("one" "two" "three" "four")))) ==>
Enter an integer greater than or equal to 1 and
less than or equal to 4 or one, two, three, or four: three
"three"
(DW:MEMBER-SEQUENCE ("one" "two" "three" "four"))

(accept '(or (integer 1 4)
            (dw:member-sequence
                ("one" "two" "three" "four")))) ==>
Enter an integer greater than or equal to 1 and
less than or equal to 4 or one, two, three, or four: 4
4
(INTEGER 1 4)
```

The particular combination of types used above might not have any practical use, but it does begin to illustrate what the possibilities are. Notice that **accept** took care of devising a prompt. You could override this if you wanted to, but in most cases it comes up with something reasonable. Notice that **accept** returns two values: the object and its presentation type.

The parser used by **accept** for parsing strings into presentation objects is not part of the **accept** function itself. Rather, each presentation type has its own, type-specific parser that **accept** calls to parse objects of that type. The parser function is included in the form that defines a presentation type. You may write your own presentation types, including the parsers (and printers) that go with them, but a sizeable set of types has already been defined for you. See the section "Predefined Presentation Types". Each type is documented in a user interface dictionary. See the section "Dictionary of Predefined Presentation Types". See the section "The Recursive Behavior of Accept".

Ancillary functions for accepting single objects include **prompt-and-accept** and **accept-from-string**. The first is the presentation-system equivalent of **prompt-and-read**. It is similar to **accept**, taking the same keyword options, but differs in its letting you use the **format** function to generate the input prompt. **accept-from-string** is the presentation-system equivalent of **read-from-string**.

Two **accept**-based menu facilities are included among the facilities for accepting single objects. The **dw:menu-choose** function is a menu-generating facility for use with Dynamic Windows. It displays a list of choices in a conventional menu format and returns the value associated (in your code) with the selected choice. For example,

```
(dw:menu-choose '(("Prize" :value 440
                    :documentation "Color television")
                   ("Alternate Prize" :value 450
                    :documentation "Video cassette recorder")))
```

**dw:menu-choose** differs from the second listed menu facility, **dw:menu-choose-from-set**, in its ability to create menus of items in the "general list" form. (See the section "The Form of a Menu Item".) **dw:menu-choose-from-set** is intended primarily for creating menus from a simple list of objects. Compare this example with that above.

```
(setq item-list '(Size Weight))
(dw:menu-choose-from-set item-list 'sys:expression)
```

When considering menus for your applications, bear in mind that a dynamic window with displayed presentations can be regarded as menus of input possibilities. You may not need to construct a menu in the strict sense of **dw:menu-choose** to provide your users with the convenience that mouse acceleration of data entry provides.

### Accepting Multiple Objects

Facilities for accepting multiple objects:

- **dw:accept-values**
- **dw:accept-variable-values**
- **dw:accepting-values**

The function **dw:accept-values** is similar to **accept**. It differs in that it accepts a series of objects from the input stream, not just one object, and also in that the mechanism by which the user makes a choice is different. The presentation type of each input object is specified independently. In the following example, an **integer** and a **pathname** object are sought:

```
(dw:accept-values '((integer :prompt "Half-life"
                     :default 24000)
                    (pathname :prompt "Log file"))
        :prompt "Atomic experiment") ==>
Atomic experiment
Half-life: 24000
Log file: Y:>curie>atom-data.log
ABORT aborts, END uses these values ==>
24000
#P"Y:>CURIE>atom-data.log"
```

The **dw:accept-variable-values** function is like **dw:accept-values**, but instead of returning a series of the user-entered values, it assigns these values to a set of special variables. It does this as the values are entered, not after the function returns. You have the option of constraining user choices for certain variables to a predefined set. **dw:accept-variable-values** is used primarily for compatibility with the older **user::choose-variable-values** function. Example:

```
(setq some-variables '((*desk* "Desk Style" symbol)
                       (*chair* "Chair Style" symbol)
                       (*size* :enter-type number)))
(dw:accept-variable-values some-variables)
```

**dw:accepting-values** is a macro that takes all calls to **accept** within its body and puts the prompts into a single, multiple-prompt display like the one shown in the example above. It is the most versatile of the three and the one recommended for general use. One of its big advantages over the previous functions is that the multiple-prompt display can be modified at runtime, in response to values entered by your user to earlier prompts in the display. In other words, the values you solicit from your users can change "on the fly", at runtime, depending on the values already received. The following is a simple example:

```
(defun return-host-or-printer ()
 (fresh-line)
 (let (choice
        (stream *query-io*))
  (dw:accepting-values (stream :own-window t)
   (setq choice (accept '(member host printer)
                        :default 'printer
                        :stream stream
                        :prompt "Send file to host or printer?"))
   (case choice
    (host (accept 'neti:host :stream stream))
    (printer (accept 'sys:printer :stream stream))))))
```

For other examples, see the file `sys:examples;accepting-values.lisp`. For information on how end users interact with multiple-accept menus: See the section "Using the Mouse and the Keyboard on Menus".

## Distinguishing Queries When Accepting Multiple Values

When you are using a **dw:accepting-values**-like function, you must provide some means for the system to be able to tell which question is which. The system needs this in order to know when a question has appeared or been removed conditionally. Here is an example of what goes wrong if you do not supply distinguishing information:

*Wrong:*

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect (accept 'integer)))
```

When you run the example, you will notice that, no matter which entry you click on, the input is accepted only for the last item and that the new value goes into them all.

The usual means of distinguishing one query from the others is by its prompt. It is always good practice to give different queries different prompts so that the user (as well as the system) can tell them apart. If the queries do not have unique prompts, use the **:query-identifier** option to **accept** to distinguish them. Identity will be based on **equal**. Here are examples illustrating each option.

*Right:*

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect (accept 'integer
                         :prompt (format nil "Number #~D" i))))
```

or

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect (accept 'integer :query-identifier i)))
```

Of the two examples above, the first is preferable because it supplies more information to the user. This next example shows a valid case of the use of identical prompts — here, for the subfield queries.

```
(dw:accepting-values ()
  (loop for i from 1 to 5
        collect (let ((num (accept 'integer
                         :prompt (format nil "Number #~D" i))))
                   (list num
                         (when (oddp num)
                           (accept 'boolean
                                   :query-identifier (list :subfield i)
                                   :prompt "  Subfield for that"))))))
```

### Complex Formatting of Accepting-Values Queries

Normally, **dw:accepting-values** puts each question on a separate line, by following the printing of the default with a **terpri**. You can suppress this, or use those advanced formatting facilities that work properly with incremental redisplay to get a

more complex layout. Non-keyboard queries can be given different graphical formats.

When more than one query appears on the same horizontal line, entering from the keyboard may erase several values, which will be redisplayed after the new value has been entered.

Give the **:newline-after-query nil** option to those **accept**s within the **dw:accepting-values** that you do not want to be followed by a newline. You may need to explicitly put in some **terpri**s after the last query and before the end/abort questions. Keep in mind the redisplay requirements on non-query output that you may also wish to mix in. See the section "Displaying Output: Replay, Redisplay, and Formatting".

Example:

```
(defun two-queries-on-the-same-line ()
  (dw:accepting-values ()
    (let ((count 1)
          (strength 50.0))
      (setq count (accept 'integer
                          :prompt "Perform "
                          :prompt-mode :raw
                          :newline-after-query nil
                          :default count))
      (setq strength (accept '(float 0.0 100.0)
                             :prompt (format nil " iteration~P at " count)
                             :prompt-mode :raw
                             :newline-after-query nil
                             :default strength))
      (dw:redisplayable-format t "%~%")
      (values count strength))))
```

For additional examples of formatting accepting values queries: See the section "Redisplaying with **dw:accepting-values** Forms".

### Table of Top-level Presentation Type Facilities

**present** *object* &optional (*presentation-type* **(type-of dw::object)**) &key (*stream* ***standard-output***) (*acceptably* **nil**) (*sensitive* **t**) *for-context-type (original-type* **nil***)* (*form* **nil**) (*location* **nil**) (*single-box* **nil**) (*allow-sensitive-inferiors* **t**) *(allow-sensitive-raw-text* **t***)*
Outputs *object* to *stream* as a presentation of type *presentation-type*.

**present-to-string** *object* &optional *(presentation-type* **(type-of dw::object)***)* &key *(original-type* **dw:presentation-type***) (string* **nil***) (index* **nil***) acceptably for-context-type*
Outputs *object*, a presentation object of type *presentation-type* to *string* or returns an new string.

**dw:with-output-as-presentation** (&key *stream object type form location single-box* (*allow-sensitive-inferiors* **t**)) &body *body*
Outputs *object*, a presentation object of type *type*, to *stream* in a manner specified by *body*, which is code that generates some type of output display.

**accept**

*presentation-type* &key (*stream* ***query-io***) (*prompt* **:enter-type**) (*prompt-mode* **:normal**) (*original-type* **dw:presentation-type**) *activation-chars additional-activation-chars blip-chars additional-blip-chars* (*inherit-context* **t**) (*default* **t** ) (*provide-default* **'dw::unless-default-is-nil**) (*default-type* **dw::original-type**) (*display-default* **dw::prompt**) *present-default history* (*prompts-in-line* **dw::*accept-active***) (*initially-display-possibilities* **nil**) *input-sensitizer* (*handler-type* **#'dw::presentation-type-find-parser**) *query-identifier* (*separate-inferior-queries* **nil**) (*confirm* **nil**)
Reads a printed representation of a Lisp object of type *presentation-type* from *stream*.

**prompt-and-accept** *presentation-type-or-args* &optional *format-string* &rest *format-args*
Reads a printed representation of a Lisp object of type *presentation-type* from *stream*, using the **format** function to create the prompt.

**accept-from-string** *presentation-type string* &rest *args* &key *index* (*start* **0**) *end* &allow-other-keys
Reads a printed representation of a Lisp object of type *presentation-type* from *string*.

**dw:menu-choose** *item-list* &key (*prompt* **nil**) (*default* **nil**) (*presentation-type* **nil**) (*printer* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) (*alias-for-selected-windows* **nil**) (*minimum-width* **nil**) (*minimum-height* **nil**)
Constructs a menu from a list of items and returns the value associated with the selected item.

**dw:menu-choose-from-set** *list presentation-type* &key (*printer* **nil**) (*prompt* **nil**) (*default* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) (*alias-for-selected-windows* **nil**) (*minimum-width* **nil**) (*minimum-height* **nil**)
Constructs a menu from the objects contained in *list* of type *presentation-type* and returns the selected object.

**dw:accept-values** *descriptions* &key (*prompt* **nil**) (*near-mode* **'(:mouse)**) (*stream* ***query-io***) (*own-window* **nil**) (*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**)
Reads a series of printed representations of Lisp objects, each of which is of the type specified in the list *descriptions*, from *stream* and returns one value for each object read.

**dw:accept-variable-values** *variables* &key (*prompt* **"Choose Variable Values"**) (*near-mode* **'(:mouse)**) (*delayed* **t**) (*stream* ***query-io***) (*own-window* **nil**)

(*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**)
Reads a series of printed representations of Lisp objects from *stream* and assigns the values read to the elements of *variables*.

**dw:accepting-values** *(*&optional *(stream* **'*query-io***)* &key *:own-window (:display-exit-boxes* **(not dw::own-window)***) (:temporary-p* **dw::own-window***) (:label* **"Multiple accept"***) (:near-mode* **'(:mouse)***) :initially-select-query-identifier :resynchronize-every-pass :queries-are-independent (:changed-value-overrides-default* **t***) (:query-entry-mode* **:inline***))* &body *body*
Causes all calls to **accept** within *body* to appear in a single, **dw:accept-variable-values** like menu that can be modified dynamically.

**dw:accept-values-command-button** *(*&optional *(stream* **'*standard-output***))* *prompt* &body *conditional-forms*
Used within **dw:accepting-values** form, this function displays *prompt* on *stream* and creates an area, the "button," wherein when a mouse button is clicked the *conditional-forms* are evaluated.

**dw:accept-values-fixed-line** *string* &optional *(stream* ***query-io****)*
Used within a **dw:accepting-values** form, this function displays *string* on *stream*. For an example: **dw:accept-values-command-button**.

**dw:accept-values-display-exit-boxes** &key *(stream* ***query-io****) (level* **:top-level***)*
Used within **dw:accepting-values** form, displays the exit boxes "ABORT aborts, END uses these values" on *stream*.

**dw:accept-values-for-defaults** *continuation*
Runs *continuation* with a stream argument, causing calls to **accept** to return their defaults.

**dw:accept-values-into-list** *descriptions* &key *:prompt (:near-mode* **'(:mouse)***) (:stream* ***query-io****) :own-window (:temporary-p* **dw::own-window***) :initially-select-query-identifier*
Performs the same operation as **dw:accept-values**, but returns a list rather than multiple values.

## Managing the Command Processor

This chapter introduces the Command Processor (CP) and explains how to:

- Create and install your own CP commands
- Manage command tables
- Execute CP commands from within your programs
- Get input into your program from a CP command loop

The Command Processor is the top-level function run in all Lisp Listener windows. In writing your own CP commands, you are adding your own features to the existing Symbolics system. To go beyond this — to create your own windows and create

commands to work within your own framework — you should use the program framework facilities, rather than those described in this chapter. The program framework facilities include, among many other things, a command-defining macro that is very similar to the macro that is central to this chapter, **cp:define-command**.

You should study the information contained in this chapter about defining commands, since most of it applies both to CP commands and to application-program commands; for information on application-specific command definition, see the section "Defining Commands within Your Own Framework". (Application-program commands are defined by program-framework-supplied macros and are different in several ways from CP commands.)

There is additional information in Part 2 of this manual that explains how to use some of the more advanced features of the Command Processor, such as translating mouse handlers. See the section "Programming the Mouse: Writing Mouse Handlers".

The sections that follow assume familiarity with user interface terminology. To refresh yourself on this topic, see the section "User Interface Concepts".

### Introduction to the Command Processor

The Command Processor parses user input, executes commands, and then updates the screen display. A common command processing mechanism manages all commands, including those that cannot be entered with the mouse, and those that would be difficult or impossible to enter without some mouse-sensitive items to accelerate them (for example, graphic presentations). This mechanism supplies the same *help*, *mouse documentation*, and *completion* facilities to your own program commands that it supplies to system commands. The Command Processor recognizes commands entered in one of the following ways:

- *By typing the command name at the keyboard.* The system software automatically parses the typed input. In particular, since the system has access to a *command table* that lists all possibly relevant commands, it can perform command completion, and it can respond to a request for help by displaying (mouse-sensitively) all the possible commands. It can also provide prompts and help messages on request.

- *Using a single-key command accelerator.* A command definition can specify that when the user presses a particular key the command is executed. A list of single-key command accelerators is kept in the same command table as the command names.

- *Using a translating mouse handler.* Once a CP command has been defined, you can write a presentation-to-command translator to make the command available on a mouse gesture. (See the section "Mouse Handler Facilities".) You do not need to do anything special in defining the command for this to work. (Presentations of the type **cp:command** have a system-supplied translator that invokes

the presentation's associated command when the user clicks Left on the presentation.)

- *Choosing the command from a pop-up menu*. The use of pop-up menus for user input is discussed in another section. (See the section "Using Menus".) Again, you do not need to do anything special in defining the command for these to work.

- *Choosing the command from a command menu*. Command menus are an option of the program framework definition facility. They display a collection of presentations that invoke commands when the user clicks the mouse on one. **Note**: Command menus are not actually a CP feature; they are available only within application program frameworks.

When a command is recognized, the command's parsing function, which the command definition sets up, will:

1. Generate a prompt for the argument(s)

2. Make defaults available

3. Cause previously displayed objects of the required types to be mouse-sensitive

4. Make help available

The Command Processor uses **accept** to read arguments from the user. This is why, when you define a command you must specify the presentation types of the arguments. The Command Processor also displays, upon request, a list of possible modifiers to the command and accepts those from the user in a similar way.

When the command is finally complete and has been entered, the processor executes it and then performs redisplay in the application framework. As part of the accepting and redisplay process, the system software (specifically, the interactive stream for input and display) saves each object that is presented or accepted. These objects, or *presentations*, become available as arguments for subsequent commands. They remain so until the user specifically deletes (kills) the output history. The types of the objects and their locations on the screen are saved along with them so the system software can automatically handle their context-dependent mouse sensitivity.

The Command Processor is built upon the presentation type substrate. This is why, in describing it we keep referring to things like presentations, presentation types, accepting, presenting, translating, and the like. It is the intent, nevertheless, of this introductory chapter, to introduce the reader only to the highest-level aspects of command definition, as far as that is possible. Reference to the details of the presentation substrate will become inevitable, however, as soon as you want to make use of any of the more advanced facilities, such as translating mouse handlers or your own parsers for presentation types.

## Defining and Installing Commands

### What a CP Command Is

These are examples of system CP commands:

```
Show Herald
Edit File v:>leb>lispm-init.lisp
Login leb :Init File None
```

To the user, a CP command is input that invokes an operation.

Functionally, a CP command consists of two items: a parsing function that recognizes the command when the user issues it and accepts arguments and an execution function that accomplishes the action the command calls for. The parsing function is generated automatically by the defining macro **cp:define-command**. You supply the execution function directly in the body of the macro.

Defined as a Lisp object, a CP command is a presentation object, or more simply, a presentation. That is to say, it is a user-visible representation of some object — in this case, an object of presentation type **cp:command**. You can use the Presentation Inspector to find out more about this particular type of displayed presentation. See the section "Presentation Inspector".

### The cp:define-command Macro

A single macro, **cp:define-command**, allows you to create a new Command Processor command, specify what it is to do, and specify how to help the user enter it. If you are creating commands within your own program framework, you should be using a very similar macro, **dw:define-program-command**, which is documented elsewhere. See the section "Defining Commands within Your Own Framework".

Using just the basic facilities of **cp:define-command**, you can define a command with:

- An optional user-visible name and the name of the Lisp function.

- An optional command table to put it in.

- A function implementing the body of the command.

- Two optional keywords specifying (1) if it should give the user an output-destination option and (2) whether it returns any values.

- Arguments (possibly none).

- An optional list of arguments for the implementing function. (This option is only for use by other top-level facilities, such as **dw:define-program-framework**.)

In the command definition, you must specify the presentation type of each argument. This sets up the input-context mechanism for the command: the presentation type determines how user input should be parsed to extract the argument. The syntax for specification of a presentation type is an extensive topic in its own right

and is explained elsewhere. See the section "How to Specify a Presentation Type". See the section "Defining Your Own Presentation Types". Each argument can have its own set of options. These options are incorporated into the parsing function the macro creates to control documentation, prompting, and defaulting. They are:

- Documentation option: a help string
- Prompting options:
  - a prompt
  - the prompt mode (display literally or transform)
  - a Boolean specifying "do [not] display the default"
- Default options:
  - the default
  - the default's presentation type — for use when the type being read is not determinate as, for example, with `'(token-or-type (("One" 1)) integer)`.
  - a Boolean specifying "the default is [not] provided"
- Options for keyword arguments:
  - a default value supplied only if the user enters the keyword and does not supply a value
  - a user-visible keyword name
- Other options
  - a Boolean specifying "do [not] require confirmation for this argument"
  - a predicate, evaluated at command time, to make reading of the argument optional

It might be instructive here to compare the **cp:define-command** macro with the **defun** special form. Both forms create a named object that specifies an action to be performed on a given list of arguments (ignoring, of course, the whole issue of whether and how arguments are evaluated). Both functions and commands can have positional arguments and keyword arguments and both are stored in an internal table. Any of the arguments of either type of object can be defaulted if so specified by the definition. Here the similarities end.

Unlike function arguments, CP command arguments have defined types, which must be specified when the command is defined. CP commands cannot have option-

al positional arguments or **&rest** arguments. All CP arguments must be explicitly listed in the defining macro, though when the command is invoked the user does not necessarily have to enter each argument.

Also, the syntax for keywords is quite different. In **defun**, an argument list looks like this:

```
(positional-arg1 positional-arg2 &key keyword-name1 keyword-name2)
```

In **cp:define-command** it looks like this:

```
((positional-arg1 type options) (positional-arg2 type options)
&key (keyword-arg1 type options) (keyword-arg2 type options))
```

From this we can see that the overall argument list syntax is similar; it is the syntax of the individual arguments that is different.

The next two sections present the detailed definition of the **cp:define-command** macro and a collection of examples that illustrate the use of all the options.

**cp:define-command** *name-and-options arguments* &body *body* *Function*

Defines a Command Processor command.

*name-and-options* Either the symbol to be used as the command name or a list whose first element is the name symbol and succeeding elements are alternating keyword-value pairs. The name symbol gets defined as a Lisp function. Commands are defined as functions and so share the function-name namespace. To distinguish command names from other kinds of names, we recommend that the prefix *com-* be used; the user-visible command name will not include the prefix.

These are the keywords that may be included in the *name-and-options* list:

**:name** Specifies the string serving as the user-visible command name. The default name is the result of calling **string-capitalize-words** on the print name of the symbol that is the first element of the *name-and-options* list; if the name begins with the substring "com-", that substring is omitted.

This option is useful for controlling capitalization within command names.

Example:

```
(cp:define-command
   (com-convert-files-to-vc-file
           :name "Convert Files to VC File" ...
```

Without the **:name** option, completion would result in "Convert Files To Vc File".

**:command-table**

Specifies the command table, or a symbol/string naming the command table, into which the command is to be installed. For example, to install a command into the "Global" command table, you could supply `"global"`, `'global`, or the form `(cp:find-command-table 'global)`. The first syntax — that is, a string — is preferred.

This option is evaluated. If not supplied, the command is not installed in a command table; to install the command subsequently, use the function **cp:install-commands**. Example:

```
(cp:define-command
   (com-say-goodbye :command-table 'user) ()
      (print "Goodbye"))
```

Or, alternatively:

```
(cp:define-command
   (com-say-goodbye :command-table "user") ()
      (print "Goodbye"))
```

For more information on command tables, see the section "Managing Command Tables".

**:comtab** This is a synonym for the **:command-table** option to **cp:define-command**. Use of **:command-table** is preferred. **:comtab** results in a warning from the compiler.

**:provide-output-destination-keyword**

Boolean option specifying whether to provide the **:output-destination** modifier. The default is **t**; this allows the user of the command to redirect the output of the command to a place other than the window.

To override the default action (if, for example, your command does not produce any useful output or you are using the **:explicit-arglist** option), specify a value of **nil** for this option.
Example:

```
(cp:define-command
   (com-say-goodbye
        :command-table 'user
        :provide-output-destination-keyword nil) ()
      (print "Goodbye"))
```

creates a parser function for the command that does not make use of an output-destination argument.

Many system commands provide the `:Output Destination` option. To get an idea of how this works, try, for example:

```
Command:  Show Directory (files [default
SYS:EXAMPLES;*.*.NEWEST]) SYS:EXAMPLES;*.*.NEWEST
(keywords) :Output Destination (a destination)
Buffer (an editor buffer) test-buffer
```

When the user supplies a value for **:output-destination**, the system binds ***standard-output*** to that value. This means that output done to ***standard-output*** either explicitly or implicitly — as with `(print "foo")` or `(format t "xyz")` — will be redirected, but output to other streams, such as ***terminal-io***, ***query-io***, ***error-output***, or user-created streams not connected to ***standard-output*** will not be redirected.

**:values** Boolean option specifying whether the command returns values; the default is **nil**.

(Note that even if this option is **nil**, the values returned by executing the command are stored in **cp:*last-command-values***.)

A CP command does not return any values unless you specify the **:values** option to be **t**. Thus in the following example, if `:values t` had not been included, `Show Test` would have displayed `Format was :ONE` and returned no values, in spite of the last line of the definition.

```
(cp:define-command
   (com-show-test :command-table "User" :values t)
        (&key (format '(member :one :two :three))
                              :default :one)
     (format t "~&Format was ~S.~%" format)
     (values 1 2))
```

**:explicit-arglist**

This option is intended only for use by other top-level macros, such as **dw:define-program-framework**. It specifies explicitly the argument list of the function implementing the body of the command. By default, the argument list of this function corresponds to the arguments specified as arguments to the command.

Use of this option is *not* recommended: either list all the arguments specifically in the command definition's argument specification or use **dw:define-program-framework** to set up a framework in which needed variables are defined. This example is included to point out an interaction between **:explicit-arglist** and **:provide-output-destination-keyword**:

```
(cp:define-command
   (com-test-args
        :command-table "user"
        :provide-output-destination-keyword nil
                             ;Required unless
                             ;output-destination
                             ;is in explicit arglist
        :explicit-arglist (&optional from-pathname
                           &key (to-pathname
                                 from-pathname
                                 to-pathname-p)))
      ((from-pathname 'pathname)
         &key
       (to-pathname 'pathname :default from-pathname))
      (format t "~A ~A ~S~%"
           from-pathname to-pathname to-pathname-p))
```

If you need to have an output destination, you can use something like:

```
:explicit-arglist (&optional from-pathname
  &key (to-pathname from-pathname to-pathname-p)
       ((:output-destination cp::.output-destination.))
       &allow-other-keys)
```

*arguments* The list of command arguments. Each element of the list is itself a list of the form (*arg-name presentation-type options*) where *arg-name* is the name of the argument; *presentation-type* is the presentation-type of the argument; and *options* are keyword options to the argument. Note that *presentation-type* is evaluated, and should typically be quoted; for example, `'pathname` or `'(integer 0 10)`.

If you need to specify arguments whose types do not correspond to any of the predefined presentation types (see the section "Predefined Presentation Types"), you must first define the presentation types you will use. The procedure for defining presentation types is described in Part 2 ( see the section "Defining Your Own Presentation Types"). The examples presented in this chapter make use of predefined types only.

These are the keywords that can be included in the argument specification list:

**:prompt** Specifies either a string to be used as a prompt for the argument or a form which when evaluated returns such a string. If a default argument is displayed, the prompt appears before the default. This is the same as the **:prompt** argument to **accept**, which function **cp:define-command**

intrinsically makes use of. See the section "The Recursive Behavior of Accept".
The prompt specified is always displayed for a positional argument, but only for a keyword argument if the keyword has been entered. An example of a form rather than a string is something like the second prompt here:

```
(cp:define-command (com-hardcopy-record-file)
    ((file 'pathname :prompt "record file")
     (printer 'printer :prompt
          (format nil "~A printer" (send file :canonical-type))))
```

**:prompt-mode**

Specifies either the **:normal** or **:raw** mode for prompts. If **:normal**, the prompt you supplied using the **:prompt** option (or the default prompt) is transformed into a prompt suitable for a command line — it is enclosed in parentheses, the default is appended, and so on. If **:raw**, your prompt is used without transformation. We recommend that you avoid using **:raw** in order to keep user interfaces uniform throughout the system. The default is **:normal**. This is the same as the **:prompt-mode** argument to **accept**, which function **cp:define-command** intrinsically makes use of. See the section "The Recursive Behavior of Accept".

Keep in mind that the whole command is being accepted and hence each argument is within a recursive call. Define and issue the following command and notice, when you type spaces, the differences between the prompt displays.

```
(cp:define-command
  (com-draw-circle :name "Draw a Circle"
                   :command-table 'global)
    ((x '(integer 50 100)
          :documentation "x-coordinate of center"
          :prompt "Center x"
          :default 50)
     (y '(integer 50 100)
          :documentation "y-coordinate of center"
          :prompt "Center y"
          :default 50)
     (radius '(integer 0 50)
          :documentation "Radius "
          :prompt "Radius "
          :prompt-mode :raw
          :default 100))
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-circle x y radius)))
```

**:default** Specifies the default value for the argument.

If no default is specified, the current default — taken from the presentation history — for the presentation type of the argument is used. (Access to the current default for a presentation type is available through **dw:presentation-type-default**.) This is the same as the **:default** argument to **accept**, which function **cp:define-command** intrinsically makes use of. See the section "The Recursive Behavior of Accept".

**:default-type**

Specifies the default presentation type of the object accepted as an argument value.

This option is useful only when used in conjunction with the **:default** option. When the type of the argument being read is ambiguous— for example, if you are using an **or** presentation type — specifying the **:default-type** option tells the Command Processor how to present the given default; that is, which presentation-type printer to use. This is the same as the **:default-type** argument to **accept**, which function **cp:define-command** intrinsically makes use of. See the section "The Recursive Behavior of Accept".

Example:

```
(number-or-string '(or integer string)
  :default 3 :default-type 'integer)
```

**:provide-default**

Specifies whether a default is provided for the argument. "Providing" a default means that a default value will be provided automatically by the system (either from a history or from the **:default** or **:mentioned-default** argument) if the user types a SPACE, END, or RETURN when a value is requested. If a default is not "provided," then the user must explicitly enter a suitable value. This is the same as the **:provide-default** argument to **accept**, which function **cp:define-command** intrinsically makes use of. See the section "The Recursive Behavior of Accept".

Usually you do not need to supply the **:provide-default** option. This is because the value of **:provide-default** is **dw::unless-default-is-nil**, which causes the following behavior:

If the presentation type being requested is any other type, then a default is provided unless the default value is **nil** — this makes it easy for an application program to force the user to input a value the first time the program is run, but provide a default value thereafter, as for example,

```
(defvar *default-pathname* nil)

;The nil forces the user to enter a value
;the first time.

(cp:define-command (com-crunch-file :command-table 'user)
    ((path 'pathname :default *default-pathname*))
   (setq *default-pathname* path)
   ...)
```

If the presentation type being requested is **boolean**, then a default is always provided.

You need to supply the **:provide-default** option in these cases:

If you want to demand a "Yes" or "No" input rather than supplying a default, specify **:provide-default nil**.

If you have a presentation type that includes **nil** as a valid value, for example, **null-or-type**, then you need to supply **:provide-default** with a suitable boolean value to get the behavior you want.

It is acceptable to specify **:provide-default t** even when **:default** is not supplied and there is no suitable object available from history. In such a case, the behavior is the same as if **:provide-default nil** had been specified.

**:display-default**

Specifies whether the default is printed in the prompt. The default value for this option is **t**. If no default is provided, (see **:provide-default**) then no default is displayed, regardless of this option. This is the same as the **:display-default** argument to **accept**, which function **cp:define-command** intrinsically makes use of.See the section "The Recursive Behavior of Accept".

Example:

```
(cp:define-command
   (com-display-default-example :command-table 'user)
      ((argument 'boolean :default nil
                        :display-default nil))
      (print argument))
```

**:documentation**

Specifies a string to use as the help message for the argument. The message is displayed if, after typing the command name and any preceding positional arguments, the user presses the HELP key.

Use this option to supplement the information automatically displayed for the user by the parser, which puts together a prompt based on the presentation type of the argument and other options such as **:prompt**. For example, in the definition of the Edit Definition command, the first item in the arguments list is:

```
(name 'function-spec
   :display-default nil :confirm t :prompt "name"
   :documentation
"Name of something (e.g., a function) whose definition to edit")
```

Note that the documentation string, unlike the prompt string, is not automatically displayed: the user has to press the HELP key to get it.

**:when** Specifies a form containing a predicate to be evaluated at command-line reading time. This option provides simple control over what arguments the command line reads; if the predicate returns **nil**, the argument is not read. The predicate can refer to any positional arguments already read. If you require finer control over the reading of arguments you can use the middle-level macro **cp:define-command-and-parser**.

Example:

```
(cp:define-command (com-when-example :command-table "user")
   ((type '(member integer any)
          :default 'integer)
   (number 'integer :when (eq type 'integer)))
  (format t "~S ~S" type number))
```

This command only reads the *number* argument when the *type* argument is integer.

**:mentioned-default**

For keyword arguments only: specifies a form to be evaluated and used as the default value for the argument, but only if the user types the argument name.

The form can refer to parameters defined for any positional arguments (but not keyword arguments) specified prior to this argument specification. At the time the form is evaluated, these parameters are bound to the values of arguments already accepted.

The default value used depends on what combination of **:default** and **:mentioned-default** options is supplied:

Both — Use the value of **:mentioned-default** if the user types the name of the argument; otherwise, use the value of **:default**.

**:mentioned-default** only — If the user types the argument name, use the value of **:mentioned-default**; otherwise, the default is **nil**.

**:default** only — Use the value of **:default**.

Neither — If the user does not type the argument name, the default is **nil**. If the user types the name, if the type has no presentation history, the argument has no default and the user has to supply a value; if there is a history, then the last value supplied is the default.

Note especially how the two argument default options interact for keywords. There are times when you do want different values for them. For example:

```
(cp:define-command
   (com-print-herald :command-table "User")
      (&key
        (detailed 'boolean
                :default nil
                :mentioned-default t
                :documentation
"Whether to print version information in full detail"))
   (print-herald :verbose detailed))
```

The user can type "Print Herald" followed by RETURN for a brief display or "Print Herald :d" RETURN for details — pressing "y" is not necessary.

**:name** For keyword arguments only: specifies a string serving as the user-visible name of the argument.

Without this option, the user-visible name is the result of applying **string-capitalize-words** to the argument name. The **:name** option is useful when **string-capitalize-words** does not provide a satisfactory capitalization or when for some reason you want the user-visible name to be different from the argument name. Example:

```
(cp:define-command (com-key-name
                    :command-table 'user)
  (&key (arg1 '(integer 1 10)
         :name "Copies"
         :default 1
         :prompt "Number of copies (1-10)"))
   (print arg1))
```

Immediately after definition, typing "Key Name :C" followed by a space displays

```
Key Name (keywords) :Copies (Number of copies (1-10)) [default 1])
```

**:confirm** Boolean option specifying whether the argument requires confirmation by the user; the default is **nil**.

When `:confirm t` is specified, if the command line is terminated before the argument has been read, the prompt for the argument is printed (as well as the prompts and defaults for all unread arguments before this one on the command line), and the user must again terminate the command line.

This mechanism ensures that the user is aware that the argument is being specified automatically, and that the

default value, if available, is displayed. This is especially useful when a command is invoked from a command menu because it stops instant activation with the default. (All destructive system commands, for example, Delete File, use `:confirm t` for their critical arguments.)

Example:

```
(cp:define-command
   (com-confirm-example :command-table 'user)
      ((right 'boolean :prompt "right" :default t )
      (proper 'boolean :prompt "proper" :confirm t))
      (if (and right proper) (print "ok")))
```

## cp:define-command Examples

### No command options, no arguments

```
(cp:define-command com-issue-greeting () (print "Hello!"))
```

defines a command but does not install it in any command table. (If you type this at a Lisp listener and then try entering Issue Greeting, you will be told that no such command exists.) Note that the command name stands alone and that the argument list is empty. You can install this command immediately after entering it with

```
(cp:install-commands 'user '(com-issue-greeting))
```

### Extended Example: Most Command and Argument Options

```
(cp:define-command (com-augmented-create-fep-file
                             :name "Create FEP File"
                             :command-table 'global
                             :provide-output-destination-keyword nil
                             :values t)
   ((file-name
     'fs:fep-pathname
     :default (fs:make-pathname
                 :host "FEP" :directory :root
                 :name "temporary" :type "temp")
     :prompt "FEP file"
     :confirm t
     :documentation "FEP file to create")
   (size
    'cl:integer
    :prompt "Enter size in blocks - "
    :default 100
    :prompt-mode :raw
    :confirm t
    :documentation "Size of the new fep file in blocks")
   (zero-out
    'boolean
    :default nil
    :prompt "Zero out?"
    :documentation "Fill the new file with zeros?"
    :when (< size 10))
   &key
   (unit-no
    'boolean
    :name "Unit Number"
    :prompt "Request unit?"
    :documentation "Do you want the file's unit number?"
    :default nil
    :mentioned-default t))
 (condition-case (error)
      (with-open-file (fep-file file-name :direction :block)
       (send fep-file :allocate size :zero-p zero-out)
       (if unit-no
          (values (send (send fep-file :file-access-path) :unit))
          (values)))
    (fs:fep-no-more-room
     (format *error-output*
             "There is not enough room in ~A for a ~D block file."
             (send file-name :host) size))
    (error (send error :report error-output))))
```

### Managing Command Tables

A *command table* is an object of flavor **cp:command-table** that identifies a set of commands that are permissible in some context. The flavor has instance variables, most of which are specified by a corresponding **cp:make-command-table** init option:

**name** The command table name, a string.

**inherit-from** A list of command tables from which this table inherits commands and accelerators.

**command-table-delims**
A list of the characters delimiting commands.

**kbd-accelerator-table**
A structure that holds an accelerator-table object and a set of keyword/value pairs describing the accelerator table. (The init option, **:kbd-accelerator-p**, is a Boolean specifying whether to create such a list, and the option **:accelerator-case-matters** specfies whether the single-key accelerators are case-sensitive.)

**menu-accelerator-table**
For a command table associated with a program framework that has a command menu, this is a list of command-menu handlers. For ordinary CP command tables, this is **nil**.

**command-aarray** Each element of this array is a pair consisting of the name of a command (a string) and the command's function.

The init option **:command-table-size** specifies an estimated initial size for the command table, and the option **:if-exists** specifies what to do if a command table with the specified name already exists.

When a command loop reads a command, it checks it against the set of permissible commands, as determined by the command table that is the value of the **:command-table** option to the reading function. In the Lisp Listener, for example, the reading function uses the "Global" command table.

Command tables can be arranged in a hierarchy, so that subordinate command tables inherit commands from their superiors. The set of permissible commands for a command table includes the commands in that command table and the commands in all superior command tables.

The variable **cp:*command-table*** is bound to the current command table in Lisp Listeners and **break** loops. (This is also the default command table for the lower-level functions, **cp:read-command** and **cp:read-command-or-form**.) The value of this variable is normally the "User" command table. The "User" command table inherits from "Global" command table, so when you type "help" to the CP prompt in a Lisp Listener, all of the commands in both of these tables are listed as available.

When you define a command, you can specify that it should be available in the "User" or "Global" table, or in an application-specific table, using the **:command-**

**table** option to **cp:define-command**. You can also use the **cp:install-commands** function to install one or more commands in a command table that you specify. To remove a command, you can use **cp:command-table-delete-command-name** or **cp:command-table-delete-command-symbol**.

The Command Processor maintains a global registry of all command tables. You find a command table by using the function **cp:find-command-table**, which returns a command table object when given the name of the table as a symbol or string. You can use the function **cp:command-in-command-table-p** to find out whether a command is available to a particular table, that is, whether it is in the table or in a table from which it inherits.

Use **cp:make-command-table** to create a command table, and **cp:delete-command-table** to delete one.

**Example of creating a command table:**

```
(cp:make-command-table "Local"
   :if-exists :update-options
   :inherit-from '("user")
   :command-table-size 10
   :kbd-accelerator-p nil)
```

For more information about command tables within application programs, see the section "Commands and Command Menus".

### System Command Tables

Many activities in Genera have their own command tables. Others have commands that they inherit from globally defined system command tables, while others have both. Figure ! illustrates how a sampling of command tables from the various facilities inherit their commands.

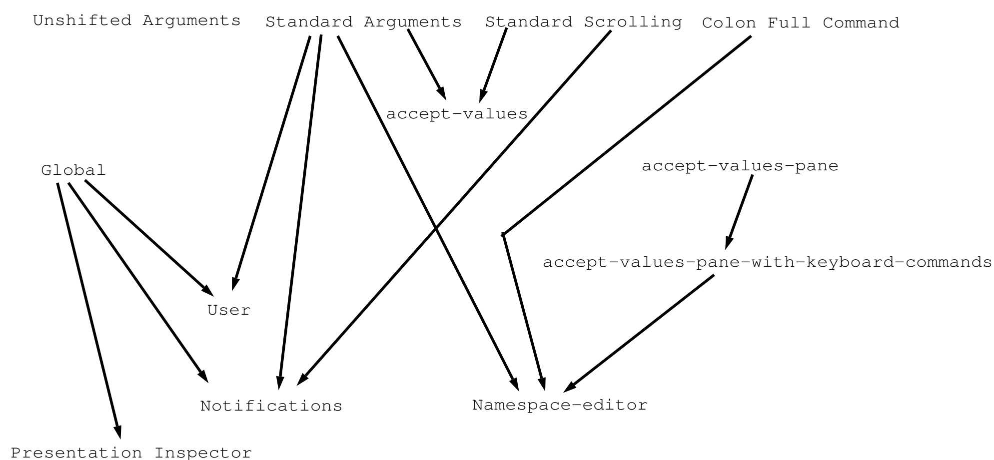



Figure 37. A sampling of system command tables

Application programs that you write will have their own command tables, which you can arrange to have inherit from any other command table you specify. This section describes the system command tables that you may find especially useful.

### Command Table "Colon Full Command"

The "Colon Full Command" command table contains only the three keyboard accelerators:

| | |
|---|---|
| : | **cp:read-full-command** |
| Meta-X | **cp:read-full-command** |
| Control-Meta-Y | **cp:yank-and-read-full-command** |

Since these commands are often desirable in application programs, this command table is offered as an option in the Frame-Up facility.

### Command Table "Standard Scrolling"

The "Standard Scrolling" command table contains the four keyboard accelerators:

| | |
|---|---|
| Scroll | scroll-window-forward-y-command-accelerator |
| Meta-Scroll | scroll-window-backward-y-command-accelerator |
| Super-Scroll | scroll-window-forward-x-command-accelerator |
| Super-Meta-Scroll | scroll-window-backward-x-command-accelerator |

It also contains the Scroll Window command and also CONTROL- versions for any typeout window. These scroll the "user" application window.

### Command Table "Standard Arguments"

The "Standard Arguments" command table contains the set of shifted (that is, those pressed while CONTROL or META are held down) standard arguments available for accelerated keyboard commands. These are used, for example, with commands in the text editor. Your program's command table can inherit from this one so that these arguments can be used. See the function **cp:define-command-accelerator**.

| | |
|---|---|
| Control-Minus-sign | sign-argument-command |
| Meta-Minus-sign | sign-argument-command |
| Control-Meta-Minus-sign | sign-argument-command |
| Control-0 | digit-argument-command |
| through | |
| Control-9 | digit-argument-command |
| Meta-0 | digit-argument-command |
| through | |
| Meta-9 | digit-argument-command |
| Control-Meta-0 | digit-argument-command |
| through | |
| Control-Meta-9 | digit-argument-command |

| | |
|---|---|
| Control-Infinity | infinity-argument-command |
| Meta-Infinity | infinity-argument-command |
| Control-Meta-Infinity | infinity-argument-command |
| Control-U | control-u-argument-command |

### Command Table "Unshifted Arguments"

The "Unshifted Arguments" table contains the set of unshifted (that is, those unmodified by CONTROL or META) standard arguments available for accelerated keyboard commands. These are used, for example, with commands in the text editor. Your program's command table can inherit from this one so that these arguments can be used. See the function **cp:define-command-accelerator**.

| | |
|---|---|
| Minus-sign | sign-argument-command |
| 0 | digit-argument-command |
| through | |
| 9 | digit-argument-command |
| Infinity | infinity-argument-command |

### Command Table "Marked Text"

This table contains the following accelerators:

| | |
|---|---|
| Super-W | kill-ring-push-all-marked-text-command-accelerator |
| Meta-W | kill-ring-push-all-marked-text-command-accelerator |
| Super-G | clear-marked-text-command-accelerator |

There are also two long-named commands in this table: Clear Marked Text, Kill Ring Push All Marked Text

### Command Table "Input Editor Compatibility"

This table contains the following accelerators:

| | |
|---|---|
| Return | noop-command-accelerator |
| Space | noop-command-accelerator |
| Refresh | refresh-command-accelerator |

There are also two long-named commands: Noop and Refresh.

### Command Table "Global"

The "Global" command table contains most system commands, such as Clean File, Edit Directory, Reset Network, and Start GC. There are approximately two hundred commands in this table, which is the one your application should inherit from if you want it to be able to access all the system commands.

## Command Table "User"

The "User" command table is primarily for user-defined commands. When you define your own special-purpose CP commands, you should put them in this table, unless you want them available globally in all programs that inherit from the "Global" command table.

## Accelerating Commands

A command accelerator allows you to define a single keystroke that will invoke a command when the user presses that key. For example, suppose you are writing an application program that has an Exit command that buries the program frame. You could put this command on the key E or X. A user would merely have to press the E or X key to exit the program. Zmail and the Lisp debugger are examples of programs that make use of accelerated commands.

You define a command accelerator with the macro **cp:define-command-accelerator**. In order for the keyboard accelerator to work, you must have set the **:kbd-accelerator-p** option to **t** when you made the command table for your application.

**Note**: single-keystroke command accelerators are not currently available in the top-level Command Processor (such as that used by the Lisp Listener); this feature is for use within your application programs.

**Example: defining an accelerated command**

```
(cp:define-command com-exit () (send dw:*program-frame* :bury))

(cp:define-command-accelerator com-exit
                           my-command-name (#\c-E)(:activate t)()
                                       '(com-exit))
```

Note that the Exit command of the example is available *only* by means of the single-key accelerator, since no command table was specified in the defining macro. In the usual case, where you want both the command in the command table and a single-key accelerator within an application program, use **dw:define-program-command** or, better still, the **dw:define-***program-name***-command** macro provided by **dw:define-program-framework**. Either of these macros allows you to define both the command and the accelerator within a single form.

Be aware that your program can inherit any command accelerators already existing in other command tables. If your program inherits these tables via the **:command-table** option to **dw:define-program-framework** or by the **:inherit-from** option to **cp:make-command-table**, the installed accelerators come along with the commands they accelerate. Take care that your accelerators make mnemonic sense and do not introduce possible conflicts with command names. Remember that you can use the CONTROL, META, SUPER, and HYPER keys to modify your single-character accelerator.

The presence of accelerators does not require reading via single keystroke commands: even if you inhibit accelerators, you can read without them in your command loop.

### Executing Commands from within a Program

**cp:execute-command** provides a convenient way to execute CP commands from within a program. It lets you express command arguments as strings, saving you the trouble of supplying arguments in the exact form expected by the command execution function. For example, to hardcopy a file, you can use:

```
(cp:execute-command "Hardcopy File"
                    "sys:examples;arrow.lisp"
                    "Audubon"
                    :body-character-style "fix.roman.large")
```

If the command name is a string, it is looked up in the command table in **cp:*command-table***. If the command can not be found there, then you must supply the command execution function as a symbol, thus

```
(cp:execute-command 'hci::com-hardcopy-file
                    "sys:examples;arrow.lisp" "Audubon"
                    :body-character-style "fix.roman.large")
```

An alternative (somewhat less readable) way to accomplish the same result as the above example would be to call the command execution function directly:

```
(let* ((file `(,(parse-namestring "sys:examples;arrow.lisp")))
       (printer
          (accept-from-string 'sys:printer "Audubon"))
       (style
          (accept-from-string `((character-style-for-device
                                   :device
                                     ,(send printer
                                        :display-device-type)))
                               "fix.roman.large")))
  (hci::com-hardcopy-file file printer
                          :body-character-style style))
```

**Note**: This latter usage is *not* recommended.

A common application of **cp:execute-command** is in lispm-init files and similar "script" files.

### Getting Input to Your Program From the Command Loop

The special variable **cp:*last-command-values*** is bound to a list containing the values returned by the most recently executed CP command. System CP commands do not, as a rule, return any values, but you can define your own commands that do. A simple example is:

```
(cp:define-command (com-show-test :command-table "User" :values t)
    (&key (format '((member :one :two :three))
              :default :one))
   (format t "~&Format was ~S.~%" format)
   (values 1 2))
```

To execute this command and use the returned values within your program you

can use something like:

```
(cp:execute-command "show test")
(setq test-values cp:*last-command-values*)
```

The above example is an alternative to

```
(multiple-value-bind (test-values) (cp:execute-command "show test") . . . .
```

Note the difference between what a CP command might display or send to a stream and the *values* it returns. For example, suppose you want to execute a CP command such as Show File and have the output displayed in your program frame rather than in the Lisp Listener. You can do this using **cp:execute-command** with output directed to one of your program's panes:

```
(let ((pane (dw:get-program-pane 'pane-1)))
  (cp:execute-command 'com-show-file "V:>User>file.text"
                      :output-destination (list pane)))
```

The following table summarizes the top-level facilities for managing the Command Processor. Refer to the user interface dictionary for the details of these. See the document *User Interface Dictionary*.

### Table of Basic Command Facilities

This table includes the top-level command processor management functions, macros, and variables. Middle-level and lower-level facilities are described elsewhere. See the section "Managing Your Program Frame".

**cp:define-command** *name-and-options arguments* &body *body*

Defines a CP command named *name* that reads in from the user a set of parameters bound to variables specified in *arguments*. When invoked, the command executes the code in *body*. Command options are **:name**, **:command-table, :explicit-arglist**, **:provide-output-destination-keyword**, and **:values**. *arguments* is a list of lists of the form (*arg-name presentation-type options*). Argument options are **:documentation**, **:prompt, :prompt-mode**, **:default**, **:mentioned-default**, **:when**, **:name**, **:default-type**, **:provide-default**, **:display-default**, and **:confirm**.

**cp:define-command-accelerator** *name command-table characters options arglist* &body *body*

Creates a single-keystroke command accelerator for a command named *name* that executes the code in *body* with arguments from *arglist*; enters it in *command-table*; and assigns it to *characters*, a list of every character that will invoke the command.

**cp:execute-command** *command-name* &rest *command-arguments*

Invokes the CP command *command-name*, with the arguments *command-arguments*, from within a program.

**cp:make-command-table** *name* &rest *init-options* &key (*if-exists* **:error**) &allow-other-keys

Creates a command table named *name*. *init-options* is a list of keyword-value pairs, in which possible keywords are **:inherit-from**, **:command-table-delims**, **:command-table-size**, **:kbd-accelerator-p**, **:accelerator-case-matters**, and **:if-exists**. Returns the command-table object.

**cp:delete-command-table** *command-table-or-name*

Removes the command table, *command-table-or-name*, from the command table registry. (Returns **t** if successful.)

**cp:find-command-table** *name* &key (*if-does-not-exist* **:error**)

Returns the command-table object bound to *name*.

**cp:install-commands** *command-table new-commands*

Installs the list of commands, *new-commands*, in *command-table*. (Returns **nil** if successful.)

**cp:command-table-delete-command-name** *command-table command-name* &key *(:if-does-not-exist* **:error***)*

Removes the command named *command-name* (a string) from *command-table*.

**cp:command-table-delete-command-symbol** *command-table command-symbol* &key *(:if-does-not-exist* **:error***)*
Removes *command* from *command-table*.

**cp:command-in-command-table-p** *command-symbol command-table* &optional *(need-name* **t***)*
Returns a list containing: **t** if *command-symbol* is in *command-table* or a superior, the name of the command table (or **nil** if *need-name* is **nil**), and the command table in which *command-symbol* was found.

**cp:*command-table***
Bound to the current command table.

**cp:*last-command-values***
Bound to the value(s) returned by the most recent CP command.

**cp:command** *(&key command-table *command-table* command-table-p) &key wait-for-activation t*
Presentation type for accepting or presenting a command processor command.

**cp:choose-command-arguments** *command-name* &rest *args* &key *(initial-arguments* **nil***) (start* **(length cp::initial-arguments)***) (end* **nil***) (command-table* **cp:*command-table****) (stream* ***standard-input****) (typeout-stream* **nil***) (help-stream* **cp::typeout-stream***) (prompt-mode* **:normal***) (near-mode* **'(:mouse)***) (mode* **:accept-values***) (typeout-stream* **nil***) (help-stream* **cp::typeout-stream***) (prompt* **nil***) (own-window* **nil***) (full-rubout* **nil***) (erase-input-editor* **nil***) (initially-select-query-identifier* **nil***)*
Returns arguments for the command *command-name* in one of three possible modes.

**cp:command-table-install-command** *command-table command-symbol* &optional *command-name*
Installs the command *command-symbol* in command table *command-table* and, optionally, gives it the name *command-name* (a string).

**cp:*default-command-accelerator-echo***
Controls whether accelerated commands are echoed on the command line when their single-key accelerators are pressed.

**dw:*display-ellipsis-help***
Controls the presentation of a help message explaining the meaning of a notation such as "`Foo ... (7)`" to indicate 7 possibilities beginning with "`Foo`".

## Presenting Formatted Output

This chapter explains how to use basic facilities to produce formatted output. To find out how to produce output that can be redisplayed and reformatted, see the section "Displaying Output: Replay, Redisplay, and Formatting".

## Introduction to Output Formatting

The output formatting facilities take data that is either given to them or produced within them and display that data in some specified way, for example, in a row or column of items, in a table, or in a graph. The implementation of such display is a two-pass process: first, the program must determine how much space the various items of the display are to occupy, then it can actually perform the output.

The formatted output facilities are designed to allow output without perturbing the flow of control of the program — that is, without requiring the user to produce some intermediate data structure to represent the individual items being formatted. (Facilities that do operate in that manner, such as **format-item-list**, are indeed provided, but they are not the only interface, and they are built from the procedural interfaces in the obvious way.) To avoid the need for such user-defined structures, your output formatting code iterates through its own data structures, doing output in a special dynamic context so that the output is properly formatted. In fact, the part of your code that does the output runs twice: the first time in a way that only remembers how large an area is required, and a second time when the layout has been determined, to draw on the screen.

Because some of the output code runs twice, there are restrictions on the side effects of code doing output with an output formatting facility. In particular, your code must be written with the understanding that it will be run more than once. In simplest terms, the code should have no side effects other than doing output to the designated stream. More liberally, it must limit its side effects to those that are impervious to multiple execution. We include several sections that explain how to write output code that addresses these issues:

"Out-of-Order Evaluation"
"Variable Snapshotting"
"Cursor Position and Formatting Output Macros"

We begin, however, with "Basic Facilities for Program Output" .

## Naming Conventions for Program Output Macros

The naming of macros for program output has followed certain conventions. Facility names prefixed with "with-" are macros that bind the environment but do not directly participate in generating output. They establish a local environment for output. Code in the bodies of such facilities is responsible for creating the output. After output is completed, the local environment goes away.

A good example is **with-character-style**. Code in the body of the macro has the job of generating characters. The macro ensures that they are output in the specified style. When the macro is finished executing, the default character style for the output stream used remains the same as before the macro was invoked.

Facility names in which the first word ends in "ing" are also macros that bind the local environment and let it go again when output is completed. In addition to this, however, they make a significant contribution to the output display, generally adding to whatever is generated in their bodies. **surrounding-output-with-border**, for example, makes an obvious contribution to the display.

### Basic Facilities for Program Output

Genera's display facilities, the high-level formatting macros in particular, are collectively known as Showcase. The Showcase facilities are intended to make generating useful and attractive displays a simpler task than if you had to do all the formatting yourself. You get to spend more time on application-specific needs for program output, and less on the requirements that most applications have in common.

Recall that the primary output facilities are those for *presenting* objects. These include the the functions **present** and **present-to-string**, and the macro **dw:with-output-as-presentation**, all of which output objects as *presentations*. The use of these primary output facilities is described in "Using Presentation Types for Output".

Output facilities described in this chapter include macros for controlling character and line output, and a large number of formatting macros. The character output facilities provide control over character style or style components (family, face, and size). (For more information on character styles, see the section "Character Styles".) Facilities for controlling line output let you specify underlining, filling, abbreviation, and truncation.

The formatting macros are high-level facilities for creating textual lists, tables, and graphs. The textual list facilities accept a sequence of objects from you and provide item delimiters, like commas, and a conjunction between the final two items. The table facilities let you create two-dimensional displays of simple or compound objects; they give you detailed control over layout. Two graph formatting facilities are available; both are for constructing hierarchical graphs showing the connections among object nodes. You can write your own output formatting macros. See the section "Writing Formatted Output Macros". You can also write code that enables your formatted output to be automatically replayed or redisplayed. A chapter in Part 2 explains how to do these ( see the section "Displaying Output: Replay, Redisplay, and Formatting").

### Formatting Text

The facilities for formatting text include means to control both character style and line output.

### Controlling Character Style

Four macros control character style: one controls all three aspects of the style and each of the other three controls the individual aspects, family, face, and size. The

macros are:

**with-character-style**
**with-character-family**
**with-character-face**
**with-character-size**

The final character style of the output characters is the result of merging the macro-specified style against the default style set for the output stream. (For more information on character styles, see the section "Character Styles".) These formatting macros have a common keyword argument, **:bind-line-height**, that causes the height of the line containing the character output to be based on the style specified in the macro.

The following example shows the use of **with-character-face** to italicize the column headings in a table:

```
(defun table-with-italicized-heads ()
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings (())
      (with-character-face (:italic)
        (formatting-cell ()
          "Number")
        (formatting-cell ()
          "Square")))
    (loop for i from 1 to 10
          as square = (* i i)
          do
      (formatting-row ()
        (formatting-cell (nil :align :center)
          (princ i))
        (formatting-cell (nil :align :right)
          (princ square))))))

(table-with-italicized-heads)
```

| *Number* | *Square* |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |
| 6 | 36 |
| 7 | 49 |
| 8 | 64 |
| 9 | 81 |
| 10 | 100 |

### Controlling Line Output

**with-underlining** Adds underlines to character output.

**abbreviating-output**
Terminates output and supplies ellipses if the output is wider or taller than specified limits.

**filling-output** Prevents the breaking of lines in the middle of words; it inserts newlines at appropriate points.

**indenting-output** Lets you insert space or a string at the beginning of each new line of character output.

Here is an example using **abbreviating-output**:

```
(defun abbrev-test (width height)
  (abbreviating-output (() :width width :height height
                           :show-abbreviation t)
    (loop for row from 1 to 20 do
      (terpri)
      (loop for col from 1 to 100 do
        (format T " ~d:~d" row col)))))

(abbrev-test 42 10)

 1:1 1:2 1:3 1:4 1:5 1:6 1:7 1:8 1:9 1:...
 2:1 2:2 2:3 2:4 2:5 2:6 2:7 2:8 2:9 2:...
 3:1 3:2 3:3 3:4 3:5 3:6 3:7 3:8 3:9 3:...
 4:1 4:2 4:3 4:4 4:5 4:6 4:7 4:8 4:9 4:...
 5:1 5:2 5:3 5:4 5:5 5:6 5:7 5:8 5:9 5:...
 6:1 6:2 6:3 6:4 6:5 6:6 6:7 6:8 6:9 6:...
 7:1 7:2 7:3 7:4 7:5 7:6 7:7 7:8 7:9 7:...
 8:1 8:2 8:3 8:4 8:5 8:6 8:7 8:8 8:9 8:...
...
```

### Formatting Textual Lists

Showcase provides the following facilities for formatting "textual" lists:

**format-textual-list**
**formatting-textual-list**

A textual list is simply a list of comma-separated items, for example: "1, 2, 3, and 4". You provide the items for the list, and the facilities take care of inserting the commas and the "and" before the final item.

**format-textual-list** is the function for printing textual lists. **formatting-textual-list** is the environment-binding macro for doing the same thing. What this and similar formatting macros provide that the functions do not is flexibility. In this case, the **format-textual-list** function requires that an explicit sequence object provide the

items for formatting, for example:

```
(defun simple-list-formatter ()
  (fresh-line)
  (format-textual-list '(1 2 3 4) #'princ :conjunction "and"))
(simple-list-formatter)
1, 2, 3, and 4
```

**formatting-textual-list**, on the other hand, lets you write code to sequence through the items using whatever data structure you choose, for example:

```
(defun simple-list-formatting ()
  (fresh-line)
  (formatting-textual-list (t :conjunction "and")
    (loop for i from 1 to 4
          do
      (formatting-textual-list-element ()
        (princ "Number ")
        (princ i)))))
(simple-list-formatting)
Number 1, Number 2, Number 3, and Number 4
```

As shown in the above example, **formatting-textual-list-element** controls the printing of one item for display by **formatting-textual-list**.

### Formatting Tables

There are five top-level facilities for displaying output in tabular form:

**format-item-list**
**formatting-item-list**
**format-sequence-as-table-rows**
**formatting-multiple-columns**
**formatting-table**

These allow you either to display an existing sequence or list in tabular format or to bind an environment such that data created within that environment is displayed in a table. **formatting-multiple-columns**, for example, displays what would otherwise be a single column of output in a multiple-column format:

```
(defun quick-table ()
  (fresh-line)
  (formatting-multiple-columns ()
    (loop for i from 0 to 79
          do
      (prin1 i)
      (terpri))))
(quick-table)
0  3  6  9  12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78
1  4  7  10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79
2  5  8  11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77
```

(If you try this example, be aware that your display might not look like the one above; the width of the output window affects the number of columns.)

**format-sequence-as-table-rows** takes a sequence of elements and outputs each element on its own row. **format-item-list** and **formatting-item-list** are also used for generating tables of simple items but, through a variety of keyword options, provide much finer control over the appearance of the table than do the first two facilities.

**formatting-table** provides the greatest flexibility for constructing tables. Five mid-level facilities can be used within a **formatting-table** form to control separately the formatting of rows, columns, and individual cells:

**formatting-column-headings**
**formatting-column**
**formatting-row**
**format-cell**
**formatting-cell**
**graphics:formatting-graphics-cell**

The following example creates a table of network servers:

```
(defun server-table ()
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings ()
      (with-character-face (:italic)
        (with-underlining ()
          (formatting-cell ()
            (write-string "Protocol"))
          (formatting-cell ()
            (write-string "Medium"))
          (formatting-cell ()
            (write-string "No. of Arguments")))))
    (loop for server in neti:*servers* do
      (formatting-row ()
        (formatting-cell ()
          (format t "~a"
                  (neti:server-protocol-name server)))
        (formatting-cell ()
          (format t "~a"
                  (neti:server-medium-type server)))
        (formatting-cell (*standard-output* :align :right)
          (format t "~a"
                  (neti:server-number-of-arguments server)))))))
(server-table)
```

```
==>
```

| *Protocol* | *Medium* | *No. of Arguments* |
|---|---|---|
| MANDELBROT | BYTE-STREAM | 1 |
| UNIX-RWHO | UDP | 1 |
| IEN-116 | UDP | 2 |
| TCP-FTP | BYTE-STREAM | 4 |
| TFTP | UDP | 1 |
| CHAOS-FOREIGN-INDEX | CHAOS | 1 |
| RTAPE | BYTE-STREAM | 1 |
| CONVERSE | BYTE-STREAM | 2 |
| SEND | BYTE-STREAM | 1 |
| SMTP | BYTE-STREAM | 3 |
| CHAOS-MAIL | BYTE-STREAM | 1 |
| CONFIGURATION | BYTE-STREAM | 1 |
| NFILE | BYTE-STREAM-WITH-MARK | 2 |
| QFILE | CHAOS | 1 |
| CHAOS-SCREEN-SPY | CHAOS | 1 |
| NOTIFY | CHAOS | 1 |
| CHAOS-ROUTING-TABLE | CHAOS | 1 |
| CHAOS-STATUS | CHAOS | 1 |
| ECHO-XCN-TOKEN-LIST | TRANSACTION-TOKEN-LIST | 1 |
| 3600-LOGIN | BYTE-STREAM | 3 |
| SUPDUP | BYTE-STREAM | 3 |
| TELNET | BYTE-STREAM | 3 |
| TTY-LOGIN | BYTE-STREAM | 3 |
| NAMESPACE-TIMESTAMP | DATAGRAM | 3 |
| NAMESPACE | BYTE-STREAM | 0 |
| BAND-TRANSFER | BYTE-STREAM | 2 |
| WHO-AM-I | DATAGRAM | 2 |
| PRINT-DISK-LABEL | BYTE-STREAM | 1 |
| EVAL | BYTE-STREAM | 1 |
| NAME | BYTE-STREAM | 2 |
| ASCII-NAME | BYTE-STREAM | 2 |
| LISPM-FINGER | DATAGRAM | 1 |
| UPTIME-SIMPLE | DATAGRAM | 2 |
| TIME-SIMPLE-MSB | DATAGRAM | 2 |
| TIME-SIMPLE | DATAGRAM | 2 |
| RESET-TIME-SERVER | DATAGRAM | 0 |

```
NIL
```

## Formatting Graphs

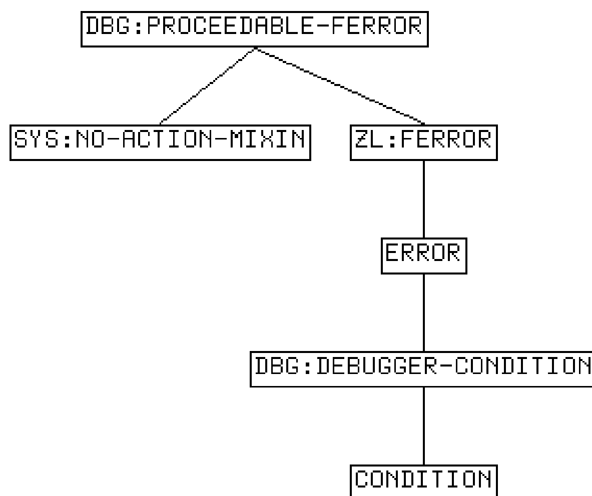
There are four top-level graph formatting facilities:

**format-graph-from-root**
**formatting-graph**

**formatting-graph-node**
**dw:find-graph-node**

**format-graph-from-root** and **formatting-graph** create graphs showing the connections between nodes. The **format-graph-from-root** function generates a graph from your specification of a root node and its descendants. Here, for example, is a flavor-component grapher built on **format-graph-from-root**:

```
(defun graph-flavor-components (flavor-name)
 (labels ((component-flavors (flavor-name)
            (flavor::flavor-local-components
             (flavor:find-flavor flavor-name))))
  (fresh-line)
  (format-graph-from-root flavor-name
                          #'(lambda (thing stream)
                              (present thing 'flavor:flavor-name
                                       :stream stream))
                          #'component-flavors
                          :dont-draw-duplicates t)))
(graph-flavor-components 'dbg:proceedable-ferror)
```




If you run this function on complex flavors, by the way, you will get a chance to exercise the horizontal scrolling capability of Dynamic Windows. This also illustrates the point that the graph formatters (and **formatting-table** as well) have built-in the functionality provided by **dw:with-output-truncation** to other kinds of output. That is, output generated using these macros that exceeds the width of the output window does not wrap around as character output ordinarily would. Rather, the user's view of the output is truncated by the right margin of the window, but can be made visible by horizontal scrolling.

**formatting-graph** works similarly to **format-graph-from-root**, but lets you specify a number of nodes and their connections, not just one node and its descendants. This allows the creation of more complex graphs than possible to create with **format-graph-from-root**. (For an example, see the function **formatting-graph-node**.) Creating node objects within **formatting-graph** is the job of **formatting-graph-node**.

Note that neither of the graph formatting facilities can be used for generating graphs containing cycles, but you can write your own function to do this using other formatting facililities.

## Using the Formatted Output Facilities: Programming Hints

### Nesting Formatted Output

Most simple combinations of formatted output macros work inside of one another. Here is an example that formats several very simple tables with no headings of any sort. Note that this task does not require the use of **formatting-table**.

```
(defun nested-formatting-1 (&optional (ntables 2) (nrows 3) (ncols 2))
  (fresh-line)
  (formatting-item-list (t :n-rows 1)
    (dotimes (i ntables)
      (formatting-cell ()
        (formatting-item-list (t :n-rows nrows :n-columns ncols)
          (dotimes (row nrows)
            (dotimes (col ncols)
              (formatting-cell ()
                (format t "Cell ~D,~D" row col)))))))))
(nested-formatting-1)
```

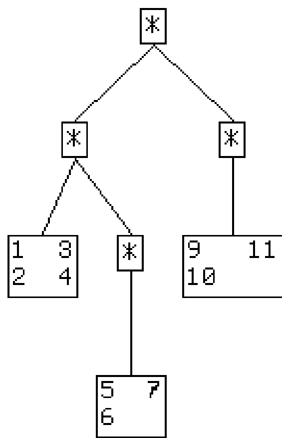
Here is another example that formats the same tables, but this time with table headings as well as row and column headings.

```
(defun nested-formatting-1a (&optional (ntables 2) (nrows 3) (ncols 2))
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings ()
      (dotimes (table ntables)
        (with-character-style ('(nil :bold :large) t :bind-line-height t)
          (formatting-cell (t :align-x :center)
            (format t "Table #~D" table)))))
    (formatting-row ()
      (dotimes (table ntables)
        (formatting-cell ()
          (formatting-table ()
            (formatting-column-headings ()
              (dotimes (col ncols)
                (formatting-cell ()
                  (format t "Column #~D" col))))
            (dotimes (row nrows)
              (formatting-row ()
                (dotimes (col ncols)
                  (formatting-cell ()
                    (format t "Cell ~D,~D" row col))))))))))))
(nested-formatting-1a)
```

```
      Table #0            Table #1
Column #0 Column #1  Column #0 Column #1
Cell 0,0  Cell 0,1   Cell 0,0  Cell 0,1
Cell 1,0  Cell 1,1   Cell 1,0  Cell 1,1
Cell 2,0  Cell 2,1   Cell 2,0  Cell 2,1
```

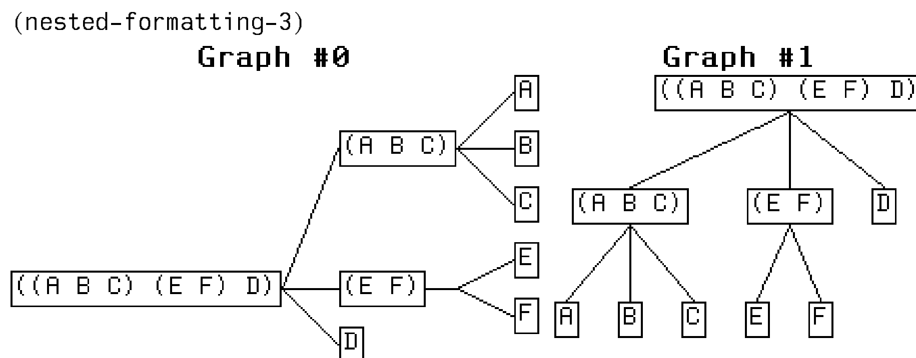
Here is an example that shows how **format-item-list** can be used within a **format-graph-from-root** form to format a node of the graph. Note how the functional interface to **format-item-list** differs from **formatting-item-list** in its defaults.

```
(defun nested-formatting-2
               (&optional (tree '(((1 2 3 4) ((5 6 7))) ((9 10 11)))))
 (fresh-line)
  (format-graph-from-root tree
     #'(lambda (node stream)
          (if (consp (first node))
              (with-character-size (:large stream :bind-line-height t)
                 (princ "*" stream))
              (format-item-list node :stream stream
                                     :optimal-number-of-rows 2
                                     :fresh-line nil
                                     :additional-indentation 0)))
     #'(lambda (node)
          (and (consp (first node)) node))))
(nested-formatting-2)
```




This last example shows how a formatted graph can be included within a table:

```
(defun nested-formatting-3 (&optional (tree '((a b c) (e f) d)))
 (fresh-line)
 (formatting-table ()
  (formatting-column-headings ()
   (dotimes (graph 2)
    (with-character-style ('(nil :bold :large) t :bind-line-height t)
     (formatting-cell (t :align-x :center)
      (format t "Graph #~D" graph)))))
  (formatting-row ()
   (dolist (orientation '(:horizontal :vertical))
    (formatting-cell ()
     (format-graph-from-root tree #'prin1
                                    #'(lambda (code) (unless (atom code) code))
                                    :orientation orientation))))))
(nested-formatting-3)
```




### Filling Output Inside Table Cells

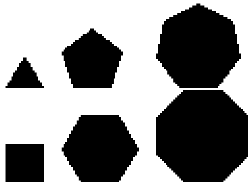
Unless you specifically set it, the width of the column of a table is determined by the output that goes into it. So if you do not set it, there is no way to fill to the full column width, since the filled output determined that width. On the other hand, if you specify the width of the filling output via the **:fill-column** argument, the output will automatically be filled to fit that width. Example:

```
(defun character-table ()
  (fresh-line)
  (let ((fill-col
          (round (* (send *standard-output* :inside-size) 2/3))))
    (formatting-table ()
      (dolist (symbol (subseq sys:area-list 0 10))
        (formatting-row ()
          (format-cell symbol #'prin1)
          (formatting-cell ()
            (format-textual-list (coerce (string symbol) 'list)
                   #'prin1 :filled t :fill-column fill-col)))))))
(character-table)
```

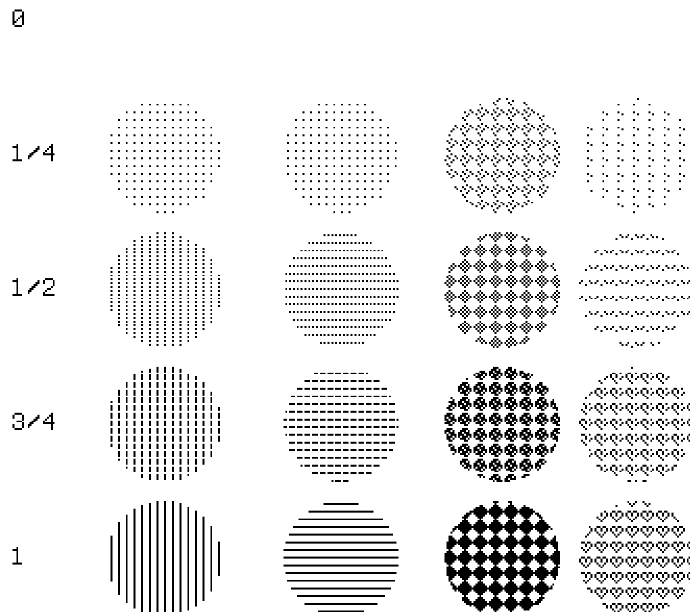
### Graphic Output within Tables

For the most part, graphic output works just like any other output inside a table. The special form **graphics:formatting-graphics-cell** gives a primary quadrant of the whole cell, the extent of the cell being the output drawn, regardless of any translation. Example:

```
(defun shapes ()
 (fresh-line)
 (formatting-item-list (t :n-rows 2 :row-wise nil)
  (loop for i from 3 to 8 do
    (graphics:formatting-graphics-cell ()
     (graphics:draw-regular-polygon 0 0 1 0 i :scale 20)))))
(shapes)
```



```
(defun grays ()
 (fresh-line)
 (let ((stipples (map 'list #'symbol-value
                        'stipples:(vertical-lines horizontal-lines
                                                   filled-diamonds hearts))))
  (formatting-table (t :inter-row-spacing 10)
   (formatting-row ()
    (format-cell "Gray" #'princ)
    (dolist (stipple stipples)
     (format-cell stipple #'princ)))
   (loop for gray from 0 to 1 by 1/4 do
     (formatting-row ()
      (format-cell gray #'prin1 :align-y :center)
      (dolist (stipple stipples)
       (graphics:formatting-graphics-cell
        (t :align-x :center :align-y :center)
        (graphics:draw-circle 0 0 30 :gray-level gray
                                     :stipple stipple))))))))
(grays)
```




Note that the polygons with more than four sides and all the circles extend into the negative $x$ region, but are still floated inside the cell.

**Controlling Location and Other Aspects of Output**

The following facilities allow you to control where in the output history of a Dynamic Window your output appears, as well as some other aspects of output history.

**(flavor:method :clear-window dw:dynamic-window)**
**(flavor:method :clear-history dw:dynamic-window)**
**(flavor:method :clear-region dw:dynamic-window)**
**(flavor:method :delete-displayed-presentation dw:dynamic-window)**
**(flavor:method :visible-cursorpos-limits dw:dynamic-window)**
**(flavor:method :set-viewport-position dw:dynamic-window)**
**(flavor:method :y-scroll-position dw:dynamic-window)**
**(flavor:method :y-scroll-to dw:dynamic-window)**
**(flavor:method :x-scroll-position dw:dynamic-window)**
**(flavor:method :x-scroll-to dw:dynamic-window)**
**dw:with-output-recording-disabled**
**dw:with-own-coordinates**
**dw:with-output-truncation**
**surrounding-output-with-border**
**dw:displayed-presentation-set-highlighting**
**dw:displayed-presentation-clear-highlighting**

Controlling the location of output is especially important in the case of dynamic — as opposed to static — windows, because it is often impossible to know in advance where the visible portion of the window will be at the time any given output is displayed. One way of handling this situation is with the **:visible-cursorpos-limits** method, as illustrated by the following example:

```
(defun graphic-output-to-dynamic-window-1 ()
  (let ((width 100) (height 50) (start-x 200) (start-y 150))
    (multiple-value-bind (x1 y1 x2 y2)
        (send *standard-output* :visible-cursorpos-limits)
        (send *standard-output* :draw-rectangle width height
                             (+ x1 start-x) (+ y1 start-y)))))
```

In this example, we are asking the window the coordinates of the current viewport, and using these as offsets to adjust where we send output.

The **dw:with-own-coordinates** macro has a similar purpose. That is, it allows you to avoid using absolute coordinates, and to use coordinates relative to the current viewport instead:

```
(defun graphic-output-to-dynamic-window-2 ()
  (let ((width 100) (height 50) (start-x 200) (start-y 150))
    (dw:with-own-coordinates ()
      (send *standard-output* :draw-rectangle width height
                                              start-x start-y))))
```

Another capability of **dw:with-own-coordinates** is the disabling of output recording. That is, through a keyword option to this macro, you can prevent output from being recorded in the output history of the window to which it is sent. (This capability is also provided by **dw:with-output-recording-disabled**.) It defeats one of the

main advantages of Dynamic Windows, but is sometimes useful, particularly when doing graphic output. Try calling the following example with **t**, to enable output recording, then **nil**, to disable it:

```
(defun moving-arrow (t-or-nil)
 (dw:with-own-coordinates (t :enable-output-recording t-or-nil)
  (do ((x 100 (+ x 4))
       (y 100 (+ y 2)))
      ((> x 500) 'done)
   (graphics:draw-arrow 100 100 x y :alu :flip
                              :arrow-base-width 20
                              :arrow-head-length 35)
   (graphics:draw-arrow 100 100 x y :alu :flip
                              :arrow-base-width 20
                              :arrow-head-length 35))
  (graphics:draw-arrow 100 100 500 300 :alu :flip
                           :arrow-base-width 20
                           :arrow-head-length 35)))
(moving-arrow t)
```

First note the speed with which the arrows are drawn. Now try scrolling backwards and forwards over the output and observe the effects.

**dw:tracking-mouse** lets you track the current position of the mouse cursor, useful in graphic applications, and other mouse events as well. In conjunction with the mouse handler facilities, it provides the primary interface between your programs and the mouse. An example showing its use in a drawing function is presented elsewhere, see the section "Creating Graphic Output". **dw:with-output-truncation** is necessary for taking advantage of the horizontal scrolling capability of Dynamic Windows. With it you can prevent the wrapping of character output and let the user's view of the output be truncated by the right (or bottom) margin of the window. The truncated output is accessible through scrolling. This also applies to graphic presentations that would otherwise be too big if limited to the size of a window. (Note that **formatting-table** and the two graph formatters, **formatting-graph** and **format-graph-from-root**, include this kind of functionality as a built-in feature.)

For a simple demonstration of output truncation, try calling the following function first with **t**, and then with **nil**:

```
(defun truncation-test (t-or-nil)
  (dw:with-output-truncation (t :horizontal t-or-nil)
    (loop repeat 200 do (write-char #\a))))
(truncation-test t)
(truncation-test nil)
```

**surrounding-output-with-border** lets you enclose any other kind of output — tables, graphics, whatever — in a rectangular, oval, circular, or diamond-shaped border. To see the different shapes, try calling the following function with `:rectangle`, `:oval`, `:circle`, or `:diamond`.

```
(defun shape-test (shape)
  (fresh-line)
  (surrounding-output-with-border
      (*standard-output* :shape shape)
      (present tv:selected-window)))
(shape-test :oval)
(Editor Typeout Window 1)
```

**dw:displayed-presentation-set-highlighting** and **dw:displayed-presentation-clear-highlighting**, as their names suggest, let you highlight and clear the highlighting of displayed presentations. This highlighting, unrelated to mouse sensitivity, is done by either underlining the presentation or putting it into reverse video. This is the same highlight that region marking (CONTROL-LEFT) uses, thus it affects META-W, for example.

### Output Formatting Spacing Parameters

The spacing parameters listed for the following output formatting macros are specified and treated in a special way:

**:inter-column-spacing** of **formatting-table**
**:inter-column-spacing** of **formatting-item-list**
*indentation* of **indenting-output**
**:fill-column** of **filling-output**

Each of these parameters can be specified in one of the following possible ways:

As an integer — If the output stream is one whose device units are smaller than single characters (pixels, for example) and if the integer is less than ten, it is interpreted as a number of character spaces; otherwise, if the number is greater than ten, it is interpreted as a number of device units. Note that the requirement that this number be an integer precludes the specification of spacing as a fraction of a character size; use the list method below to get fractional character spacing. (Ten is the number of pixels in a device character.)

As a string — The spacing is the width of the string.

As a function — The spacing is the amount of space the function would consume when called on the stream.

As a list — The list is of the form (*number unit*), where *unit* is one of **:pixel** or **:character**. **'(3 :character)** is different from **(* 3 (send stream :char-width))** or just **3**, in that the character width of whatever stream is really used to do the formatting is correctly used. **'(4 :pixel)** is different from just **4** in that it is not subject to the special interpretation of small numbers (< 10) normally applied.

These first two examples produce output that appears identical:

```
(defun example-1 ()
 (formatting-table (t :inter-column-spacing
                        (* 3 (send *standard-output* :char-width)))
  (formatting-row ()
   (loop for item in
             '("AAAAAA" "BBBBBB" "CCCCCC" "DDDDDD" "EEEEEE")
         do (format-cell item #'princ)))))
(example-1)
         BBBBBB   CCCCCC   DDDDDD   EEEEEE

(defun example-2 ()
 (formatting-table (t :inter-column-spacing 3)
  (formatting-row ()
   (loop for item in
             '("AAAAAA" "BBBBBB" "CCCCCC" "DDDDDD" "EEEEEE")
         do (format-cell item #'princ)))))
(example-2)
         BBBBBB   CCCCCC   DDDDDD   EEEEEE
```

In this example, the integer is interpreted as pixels rather than as characters:

```
(defun example-3 ()
 (formatting-table (t :inter-column-spacing 13)
  (formatting-row ()
   (loop for item in
             '("AAAAAA" "BBBBBB" "CCCCCC" "DDDDDD" "EEEEEE")
         do (format-cell item #'princ)))))
(example-3)
        BBBBBB  CCCCCC  DDDDDD  EEEEEE
```

Here, in order to achieve fractional character spacing, we force the units (which would have been assumed to be characters) to be interpreted as pixels:

```
(defun example-4 ()
 (formatting-table (t :inter-column-spacing '(8 :pixel))
  (formatting-row ()
   (loop for item in
             '("AAAAAA" "BBBBBB" "CCCCCC" "DDDDDD" "EEEEEE")
         do (format-cell item #'princ)))))
(example-4)
       BBBBBB CCCCCC DDDDDD EEEEEE
```

**(send stream :char-width)** would not work for the following example, because the encapsulating stream would not "know" about character width at the appropriate time. The **3** is interpreted as number of characters.

```
(defun example-5 ()
 (with-open-stream
  (stream
   (hardcopy:make-hardcopy-stream
    hardcopy:*default-text-printer* :landscape-p t))
  (formatting-table (stream
                          :inter-column-spacing 3)
   (formatting-row (stream)
    (loop for item in
              '("AAAAAA" "BBBBBB" "CCCCCC" "DDDDDD" "EEEEEE")
          do (format-cell item #'princ :stream stream))))))
(example-5)
```

If **example-5** had used inter-column-spacing of **'(3 :character)** for an LGP2, there would have been no visible difference in the result. This might not be the case for other hardcopy devices.

### Out-of-Order Evaluation

In most cases, even though some output-producing code is run twice, it is still run in the order in which it appears lexically. In limited specific cases, though, the part of the program that does an individual item is separated and run the second time in a different order. The main example of this is the use of the **:row-wise nil :output-row-wise t** options to **formatting-item-list**. These options are used for laying items out in columns, left to right, rather than rows, top to bottom; it does actual output in rows since that involves less cursor motion. In this case, the first pass to determine the size of the items in the menu will be in the normal order. However, for the second pass, they will be taken in the order necessary to produce the desired inversion.

Here is a concrete example:

```
(defun what-order (l row-wise)
  (terpri)
  (stack-let ((things (make-array 100 :fill-pointer 0)))
    (formatting-item-list (t :row-wise row-wise :output-row-wise t)
      (dolist (x l)
        (formatting-cell ()
          (vector-push-extend x things)
          (princ x))))
    (coerce things 'list)))
  (what-order '(a b c d e f) t)

(A B C D E F A B C D E F)
  (what-order '(a b c d e f) nil)

(A B C D E F A D B E C F)
```

Note the double evaluation: Each element appears twice in the result. If you use the **format-item-list** functional interface, the problem is similar, but not quite so

severe, since the function given for printing is more likely to stand alone. The function may be called on the items in a table or menu multiple times, and not in the order in which they occur in the argument sequence.

### Variable Snapshotting

Variable snapshotting is saving the values of lexical variables at one time and using the saved values at a later time, for example, when output-producing code is run a second time. To understand the need for variable snapshotting, consider the following programs.

```
(map 'list #'funcall
     (loop for i from 1 to 10 collect #'(lambda () i)))
=> (11 11 11 11 11 11 11 11 11 11)

(map 'list #'funcall
     (let ((continuations nil))
       (dotimes (i 10)
         (push #'(lambda () i) continuations))
       (nreverse continuations)))
=> (10 10 10 10 10 10 10 10 10 10)

(map 'list #'funcall
     (let ((continuations nil))
       (dolist (x '(a b c d))
         (push #'(lambda () x) continuations))
       (nreverse continuations)))
=> (D D D D)

(map 'list #'funcall
     (map 'list #'(lambda (x) #'(lambda () x))
          '(a b c d)))
=> (A B C D)
```

In all but the last case, the program does not perform as expected. A strict interpretation of the definition of **dolist** and **dotimes** requires that there be a single iteration variable that is set each time around the loop to the next value. The semantics of the complete version of **loop**, with multiple variables of iteration, makes this even more imperative. All the closures collected are closed over the same variable *i* or *x*, not ten different variables. Hence they all return the same value when called. A **setq** after a closure is created changes the value of the enclosed variable.

Unfortunately, this collecting of continuations is exactly how the formatted output macros accomplish their layout, as the following examples show.

```
(defun with-snapshotting (l)
  (terpri)
  (formatting-item-list ()
    (dolist (x l)
      (formatting-cell ()
        (princ x)))))
(with-snapshotting '(a b c d e f g h i j))

A B
C D
E F
G H
I J
(defun without-snapshotting (l)
 (terpri)
 (formatting-item-list ()
  (dolist (x l)
   (formatting-cell (t :dont-snapshot-variables (x))
    (princ x)))))
(without-snapshotting '(a b c d e f g h i j))

J J
J J
J J
J J
J J
```

So that the formatting output macros work correctly when they are used in conjunction with the simplest iteration macros, the output macros create new variables with the same names to hold an unchanging value. Compare this example with the previous **map** examples:

```
(map 'list #'funcall
       (loop for i from 1 to 10
              collect (let ((i i))
                         #'(lambda () i))))
```

Be careful that snapshotting does not cause problems for legitimate uses of shared lexical variables: the macros cannot distinguish these from variables of iteration. Here is an example of this sort of problem:

```
(defun show-some-hash-elements (table)
 (terpri)
 (let ((items-output nil))
  (formatting-table ()
   (maphash #'(lambda (key item)
                (when (oddp key)
                 (pushnew item items-output)
                 (formatting-row ()
                  (formatting-cell ()
                   (princ key))
                  (formatting-cell ()
                   (princ item)))))
            table))
  (sort items-output #'<)))
(show-some-hash-elements (make-hash-table
        :initial-contents '(1 2 2 3 3 4 4 5 5 6)))

5 6
3 4
1 2
```

This program will get a compiler warning because it is writing to a snapshotted variable. It also does not work, as it returns **nil** instead of the derived list of items. In order to get the code to function correctly, you must inhibit the snapshotting of the *items-output* variable as follows:

```
(defun show-some-hash-elements (table)
 (terpri)
 (let ((items-output nil))
  (formatting-table (t :dont-snapshot-variables (items-output))
   (maphash #'(lambda (key item)
                (when (oddp key)
                 (pushnew item items-output)
                 (formatting-row ()
                  (formatting-cell ()
                   (princ key))
                  (formatting-cell ()
                   (princ item)))))
            table))
  (sort items-output #'<)))
(show-some-hash-elements (make-hash-table
                    :initial-contents '(1 2 2 3 3 4 4 5 5 6)))

5 6
3 4
1 2
```

### Cursor Position and Formatting Output Macros

Most formatted output macros that produce blocks of output such as tables or graphs put the upper left corner at the current cursor position and leave the cursor just below the bottom left corner of the output. This means that if you want a table to be flush left in a window, you should do a **(fresh-line)** before starting the output. Since a program frame redisplay function or Command Processor command is called with the cursor already on a fresh line, you do not actually need the **(fresh-line)** in all situations.

When you are debugging in the Lisp Listener by typing forms, the position at the start of form evaluation is just after the form on the same line. So, when you are debugging, the absence of an initial **(fresh-line)** form may cause your output to start in the middle of the window, displaced to the right by the width of the last line of the Lisp form you typed in.

**Note**: The various kinds of table formatting output macros perform their own newline output where needed. They do the conditional decisions necessary for this. Do not include newline commands inside formatting output macros unless you are creating multi-line cells — this includes use of ~% and ~&, as well as **fresh-line** and **terpri**.

### Table of Program Output Facilities

**with-character-style** (*style* &optional (*stream* **t**) &key *bind-line-height*) &body *body*
Binds the local environment such that character output is in the specified style.

**with-character-family** (*family* &optional (*stream* **t**) &key *bind-line-height*) &body *body*
Binds the local environment such that character output is in the specified family.

**with-character-face** (*face* &optional (*stream* **t**) &key *bind-line-height*) &body *body*
Binds the local environment such that character output is in the specified face.

**with-character-size** (*size* &optional (*stream* **t**) &key *bind-line-height*) &body *body*
Binds the local environment such that character output is of the specified size.

**with-underlining** (&optional *stream* &key (*underline-whitespace* **t**)) &body *body*
Binds the local environment such that character output is underlined.

**abbreviating-output** (&optional *stream* &key *width height lozenge-returns newline-substitute show-abbreviation abbreviate-initial-whitespace*) &body *body*
Binds local environment such that character output is abbreviated.

**filling-output** *(*&optional *stream* &key *:fill-column (:fill-characters* **'(#\Space)***) :after-line-break :after-line-break-initially-too :dont-snapshot-variables)*

&body *body*
Binds local environment such that character output is filled.

**indenting-output** (*stream indentation*) &body *body*
Binds local environment to control the insertion of spaces or other characters at the beginning of each newline output to a stream.

**format-textual-list** *sequence function* &key (*separator* ",") *finally if-two filled after-line-break conjunction* (*stream* ***standard-output***)
Outputs a sequence of items as a textual list.

**formatting-textual-list** (&optional *stream* &key (*separator* ",") *finally if-two filled after-line-break conjunction*) &body *body*
Binds local environment to output a sequence of items as a textual list.

**formatting-textual-list-element** (&optional *stream*) &body *body*
Controls the printing of items output as textual list elements within a **formatting-textual-list** macro.

**format-item-list** *list* &key (*stream* ***standard-output***) *printer presentation-type (key #'identity)* (*fresh-line* **t**) (*return-at-end* **t**) (*order-columnwise* **t**) (*optimal-number-of-rows* **si:*optimal-number-of-rows***) (*additional-indentation* **2**) (*equalize-column-widths* **nil**) *max-width max-height*
Displays the elements of a list in a tabular format.

**formatting-item-list** (&optional *stream* &key *inter-row-spacing inter-column-spacing row-wise output-row-wise n-rows n-columns inside-width inside-height max-width max-height*) &body *body*
Binds local environment to output a list of items created in the body of the macro as a table.

**format-sequence-as-table-rows** *sequence printer* &rest *options* &key (*stream* ***standard-output***) &allow-other-keys
Displays the elements in a sequence as a series of table rows.

**formatting-multiple-columns** (&optional *stream* &key *number-of-columns*) &body *body*
Binds the local environment such that the lines of text generated by *body* are output in a multiple-column format.

**formatting-table** (&optional *stream* &key *equalize-column-widths extend-width extend-height inter-row-spacing inter-column-spacing multiple-columns* (*multiple-column-inter-column-spacing* **dw::inter-column-spacing**) (*equalize-multiple-column-widths* **nil**) (*output-multiple-columns-row-wise* **nil**)) &body *body*
Binds local environment to output items in a tabular format.

**formatting-column-headings** (&optional *stream* &rest *options*) &body *forms*
Controls the display of column headings within a **formatting-table** macro.

**formatting-column** (&optional *stream* &rest *options*) &body *body*
Controls column layout within a **formatting-table** macro.

**formatting-row** (&optional *stream* &rest *options*) &body *body*
Controls row layout within a **formatting-table** macro.

**format-cell** *object printer* &key *(stream* ***standard-output****)* *align-x align-y*
Controls the printing of a table element within a **formatting-table** or **formatting-item-list** macro.

**formatting-cell** *(*&optional *stream* &key *align-x align-y dont-snapshot-variables)* &body *body*
Binds local environment to control the printing of a table element within a **formatting-table** macro.

**graphics:formatting-graphics-cell** *(*&optional *stream* &key *(:align-x* **:left***)* *(:align-y* **:bottom***)* *(:float-origin* **t***)* *:dont-snapshot-variables)* &body *body*
Binds local environment to control the printing of a graphical table element within a **formatting-table** or **formatting-item-list** macro.

**format-graph-from-root** *root-object object-printer inferior-producer* &key *(stream* ***standard-output****)* *(dont-draw-duplicates* **nil***)* *(key* **#'identity***)* *(test* **#'eql***)* *(root-is-sequence* **nil***)* *(direction* **:after***)* *(default-drawing-mode* **:line***)* *(default-drawing-options* **nil***)* *(cutoff-depth* **nil***)* *(border* **'(:shape :rectangle)***)* *(orientation* **dw:*default-graph-orientation****)* *(balance-evenly* **dw:*default-graph-balance-evenly****)* *(row-spacing* **dw:*default-graph-row-spacing****)* *(within-row-spacing* **dw:*default-graph-within-row-spacing****)* *(column-spacing* **dw:*default-graph-column-spacing****)* *(within-column-spacing* **dw:*default-graph-within-column-spacing****)* *(branch-point* **dw:*default-graph-branch-point****)* *(allow-overlap* **dw:*default-graph-allow-overlap****)*
Constructs and displays a tree graph, given the root of a tree, a function to display a node, and a function to extract a node's inferiors.

**formatting-graph** *(*&optional *stream* &key *(orientation* **dw:*default-graph-orientation****)* *(inverted-center* **dw:*default-graph-inverted-center****)* *(balance-evenly* **dw:*default-graph-balance-evenly****)* *(row-spacing* **dw:*default-graph-row-spacing****)* *(within-row-spacing* **dw:*default-graph-within-row-spacing****)* *(column-spacing* **dw:*default-graph-column-spacing****)* *(within-column-spacing* **dw:*default-graph-within-column-spacing****)* *(branch-point* **dw:*default-graph-branch-point****)* *(allow-overlap* **dw:*default-graph-allow-overlap****)* *(default-drawing-mode* **:line***)* *default-drawing-options dont-snapshot-variables)* &body *body*
Binds the local environment to output a graph connecting node objects generated in the body of the macro.

**formatting-graph-node** (&optional *stream* &key *id connections* (*drawing-mode* **t**)) &body *body*
Binds local environment to create node objects for use by the **formatting-graph** macro.

**dw:find-graph-node** *redisplay-helper-stream id* &key (*key* **#'identity**) (*test* **#'eql**)
Searches for a node object given its symbol and the output stream on which it is to be displayed and returns the object if it finds it.

**dw:erase-displayed-presentation** *displayed-presentation window* &optional *recursive as-single-box (clear-inferiors* **t***)*
Erases the specified *displayed-presentation* from *window* and removes it from the output history.

**(flavor:method :clear-window dw:dynamic-window)**
Scrolls the window forward in the vertical dimension far enough to eliminate previous output from the current display.

**(flavor:method :clear-history dw:dynamic-window)**
Eliminates all items in the output history of the window, and resets the viewport to the top of the history.

**(flavor:method :clear-region dw:dynamic-window)** *left top right bottom*
Clears the output display in a rectangular area of the window.

**(flavor:method :delete-displayed-presentation dw:dynamic-window)** *displayed-presentation*
Deletes an already displayed presentation from a Dynamic Window's output history and display.

**(flavor:method :visible-cursorpos-limits dw:dynamic-window)** &optional (*unit* **:pixel**)
Returns the limits of the current viewport as absolute window locations.

**(flavor:method :set-viewport-position dw:dynamic-window)** *new-left new-top*
Scrolls the window to a specified location in the window's output history.

**(flavor:method :y-scroll-position dw:dynamic-window)**
Returns: absolute location of current viewport's top edge, its vertical extent, window's minimum y-coordinate, and absolute location of viewport's bottom edge.

**(flavor:method :y-scroll-to dw:dynamic-window)** *position type*
Scrolls the window to a specified y-coordinate.

**(flavor:method :x-scroll-position dw:dynamic-window)**
Returns: absolute location of current viewport's left edge, its horizontal extent, window's minimum x-coordinate, and absolute location of viewport's right edge.

**(flavor:method :x-scroll-to dw:dynamic-window)** *position type*
Scrolls the window to a specified x-coordinate.

**dw:with-output-recording-disabled** *(*&optional *stream)* &body *body*
Binds the local environment so that output produced by *body* is not recorded in the history of the dynamic window associated with *stream*.

**(flavor:method :with-output-recording-disabled dw:dynamic-window)** *continuation xstream*
Disables output recording on a specified window for a specified continuation.

**dw:with-own-coordinates** (&optional *stream* &key *left top right bottom* (*clear-window* **t**) (*erase-window* **nil**) (*enable-output-recording* **t**)) &body *body*
Binds the local environment such that output to a Dynamic Window is in a refreshed area and the coordinate system is relative to the current viewport.

**dw:tracking-mouse** *(*&optional *stream* &key *(:whostate* **"Track Mouse"***) :who-line-documentation-string :who-line-more-documentation-string :multiple-window)* &body *clauses*
Tracks the mouse in the user process.

**dw:with-output-truncation** (&optional *stream* &rest *options*) &body *body*
Binds the local environment to allow textual output to extend beyond the bottom and right borders of the output window.

**surrounding-output-with-border** (&optional *stream* &key (*shape* **:rectangle**) (*thickness* **1**) (*margin* **1**) (*pattern* **t**) *(gray-level* **1***) (opaque* **nil***)* (*filled* **nil**) *alu (label* **nil***) (label-position* **:bottom***) (label-separator-line* **nil***) (label-separator-line-thickness* **1***) (label-alignment* **:left***)* (*width* **nil**) (*height* **nil**) (*move-cursor* **t**)) &body *body*
Binds the local environment such that output generated in the body of the macro is enclosed within a border.

## Defining Your Own Program Framework

### Introduction to Program Frameworks

If you have not read the introductory sections describing the Genera user interaction paradigm and introducing program frameworks, it might be helpful to do so now. See the section "User Interaction Paradigm". See the section "Program Frameworks".

A program framework organizes the interaction portion of an interactive program. The most elaborate framework can consist of:

- A command loop
- A set of commands
- A window-layout declaration
- A set of window display definitions

Genera provides a top-level facility, **dw:define-program-framework**, for creating a program framework that aids the programmer in:

- Establishing and managing the command loop
- Managing the program's state variables
- Managing screen real estate, by means of a window called a *program frame*

A simple program framework might consist of just a command loop and a set of commands; it is not a requirement that an interactive program have its own window. In fact, you can use the Genera program framework facility simply to create a few related commands. Doing so gives you the state-variable, redisplay, and command-management facilities at the cost of minimal programming effort.

Genera includes an interactive code-building utility, the Frame-Up Layout Designer, that produces code written as a single definition using **dw:define-program-framework**. You can use Frame-Up, or you can write your own code using **dw:define-program-framework** to organize an interactive program, or you can generate a preliminary version with Frame-Up and then edit the resulting definition to produce code tailored to your application. A later section describes in detail the use of Frame-Up. See the section "Using Frame-Up". Before presenting **dw:define-program-framework** in detail, we need to define more precisely the various concepts involved.

### The Top-Level Command Loop

A typical top-level command loop involves these steps:

1. Read a command.
2. Execute the command.
3. Update the display.

An option to **dw:define-program-framework**, called **:top-level**, allows you to specify a function to perform these steps or any others you might want. The function specified by **:top-level** also lets you control what happens when your program is selected for the first time. The default of this option, the function **dw:default-command-top-level**, takes care of the first two of the above steps in a manner

similar to that of the Lisp listener command loop, and it handles the initial redisplay of the program frame, if there is one. This includes the automatic provision of a large number of capabilities, including:

- Command recognition, with completion and all the associated help, documentation, and input editing facilities. This makes use of a program-specific command table, which is set up for you by **dw:define-program-framework**. Additionally, you can specify other command tables from which your program is to inherit commands. You define the commands themselves by making use of a command definition macro that is automatically created for you as part of the framework definition.

- Command acceleration by means of

  ° program menus, which are set up by the framework definition

  ° single-keystroke accelerators, enabled by the framework definition, and set up by program command definitions

  ° mouse translations or actions (see the section "Actions and Translations".)

- Handling of `**MORE**` breaks

- Optional evaluation of Lisp forms and maintenance of the top-level-loop variables, *, +, and the like.

Note that you do not have to make use of all these features: You can have a program, for example, that does not make use of keyboard input at all — it just uses a command menu that the user clicks the mouse on. Alternatively, you might have only single-keystroke commands and make no use of the mouse.

Another option to **dw:define-program-framework**, called **:command-evaluator**, allows you to specify some additional action to be performed before or after a command has been read.

The third step of a command loop, redisplaying data, is handled on a per-program-frame basis. Each pane within a program frame (window) for which redisplay makes sense can have its own redisplay function, defined by the pane's **:redisplay-function** option, as well as a set of options specifying whether and under what circumstances redisplay should be performed and whether the redisplay should be incremental.

Writing your own top-level command loops and redisplay functions is discussed in Part 2. See the section "Managing Your Program Frame". See the section "Displaying Output: Replay, Redisplay, and Formatting".

**Program Panes**

**dw:define-program-framework**, if you so specify, creates and initializes a dynamic window that is dedicated to your program. Such a window is called a *program frame*. A program frame is a window resource of type **dw:program-frame** and is an object of flavor **dw:program-frame**. You can split this window interface to your application into any number of subwindows, called panes. The Frame-Up Layout designer allows you to do splitting, swapping, sizing, and deletion of a program frame's panes interactively. You can further alter the layout of panes by editing the code generated by Frame-Up.

There are six types of program frames, each based on a dynamic pane flavor:

Title Pane — A constant display telling the user what the window is for.

Interactor Pane — A pane for interactive input/output. This holds your command history.

Lisp Listener Pane — Another, taller, pane for interactive input/output.

Display Pane — A pane for output display. This is where your display is done. You might have several display panes in your window.

Command Menu Pane — A menu of mouse-sensitive commands.

Accept Values Pane — Another kind of menu pane for accepting variable, user-specified values — typically, state variables.

Appropriate options for each type of pane control such factors as the pane name, its height, whether a typeout window can appear, and the pane's redisplay function.

### Program State Variables

One of the things that **dw:define-program-framework** does is to define a flavor named after your program. The purpose of this is to allow you to have variables to which your program has direct access. These variables are writable instance variables of the program's associated flavor. They are called the program's *state variables*. Their bindings are preserved between activations of the program.

You define and initialize state variables by means of a keyword option to **dw:define-program-framework**. This is a handy alternative to having to declare your program data as special variables. More important, it allows multiple instances of your program to co-exist without interfering with each other. See the section "Kinds of Variables". An important and useful consequence of this is that program functions can be written as methods of the program flavor and thereby have direct access to its instance variables, including your state variables. (For information on flavors and methods, see the section "Flavors".) Also, remember to include a **compile-flavor-methods** form in your program if you have defined flavor methods within it.

### The dw:define-program-framework Macro

Here is the complete definition of the main framework-building tool. It lists and describes in detail all program framework options, including all the program pane options. You may want to skip over this definition to the section "Using Frame-Up", returning to this section for reference as needed.

**dw:define-program-framework** *name* &key *pretty-name* (*command-definer* **nil**) (*command-table* **nil**) (*top-level* **'(dw:default-command-top-level)**) (*command-evaluator* **nil**) (*panes* **'(dw::main :listener)**) *selected-pane query-io-pane terminal-io-pane label-pane* (*configurations* **nil**) (*state-variables* **nil**) (*selectable* **t**) (*select-key* **nil**) (*system-menu* **nil**) (*size-from-pane* **nil**) (*inherit-from* **'(dw::program)**) (*other-defflavor-options* **nil**) *Function*

Defines a program framework, including: a command-defining macro, a flavor named after the program with instance variables you can specify (the program's state variables), a command table for the program, and a list of options specifying the command and screen interfaces for the program.

*name* The name to be given to the program flavor created by **dw:define-program-framework**.

**:pretty-name**

Specifies the user-visible name of the program. This name appears in places like the SELECT key menu and the System menu and as an argument to the Select Activity command. If this option is not supplied, the displayed name is the program-flavor name specified by the *name* argument, but with hyphens removed and initial capitals (for example, "my-program" becomes "My Program"). You can set this option using the Frame-Up Set Program Options command.

The value you supply for this option is not evaluated.

**:command-definer**

Specifies the symbol to be used when defining program commands. In the typical case, this option is supplied with a value of **t** (that is, `:command-definer t`); this results in the creation of a program command-definition macro invoked with the symbol **define**-*program-name*-**command**, where *program-name* is the *name* argument supplied to **dw:define-program-framework**.

The command-definition macro so created has the same syntax as **dw:define-program-command**, but with one exception: You do not have to supply the *program-name*. (See the function **dw:define-program-command**.)

The **:command-definer** option defaults to **nil**, in which case no command-definition macro is created, and you must use **dw:define-program-command**.

The value you supply for this option is not evaluated.

**:command-table**

Specifies a list of two options to the **cp:make-command-table** function: **:inherit-from** and **:kbd-accelerator-p**. **dw:define-program-framework** calls **cp:make-command-table** to create a command table for your program (having as its name the name of your program). You can set these options using the Frame-Up Set Program Options command.

Supplying the name (symbol or string) of a command table to the **:inherit-from** option makes all the commands in that table available during the running of your program. For example, supplying a value of `"global"` or `"user"` results in all the commands in the global or user command table, respectively, being included in your application command table.

If your frame includes an accept-values pane, one of the values to the **:inherit-from** option must be "accept-values-pane".

Supplying a value of **t** to **:kbd-accelerator-p** allows you to specify single-key accelerators for program commands; the default is **nil**. Keyboard accelerators are specified via the command-definition macro **define-*program-name*-command** created by **dw:define-program-framework**; via **dw:define-program-command**; or via **cp:define-command-accelerator**. Keyboard accelerators are also inherited when you use the **:inherit-from** option.

If **:kbd-accelerator-p** is **t**, to enter an unaccelerated command the user must first type a colon or `m-X`. This behavior can be modified via the default command loop function: See the function **dw:default-command-top-level**. See also the **:top-level** option to **dw:define-program-framework**.

The value you supply for this option is not evaluated, but each individual component of the list you give it *is* evaluated, as in

```
:command-table (:inherit-from '("global"))
```

**:top-level** Specifies a list of functions and arguments to be executed when a program instance is created, typically, when the program is selected for the first time. The default behavior is for the program to enter the standard command loop, provided by **dw:default-command-top-level**.

You can take advantage of this option in two ways: 1) to run top-level functions before entering the command loop; 2) to provide your own command loop function or specify modifications or options to the default top-level function. An example of the first is provided by the Flavor Examiner, an application built using **dw:define-program-framework**. The designer of this program wanted a special help message to be displayed the first time the program is entered. This was implemented by the following code:

```
(dw:define-program-framework examiner
 ...
 :top-level (examiner-top-level :prompt examiner-prompt)
 ...)

;;; This top-level function exists to get HELP
;;; text printed out at the start.  Then it just
;;; runs the standard command loop.
(defun examiner-top-level (program &rest options)
 ;; No point in making this a generic function,
 ;; although typically it would be.
 (examiner-help program (dw:get-program-pane 'command) nil)
 (apply #'dw:default-command-top-level program options))
```

Note that **dw:define-program-framework** passes the program instance to the **:top-level** function, in this case **examiner-top-level**. **examiner-top-level** first calls the **examiner-help** function created to generate the top-level help message, then turns control over to the command loop function for the program, **dw:default-command-top-level**. **examiner-prompt** is a function defined elsewhere in the flavor examiner program. It causes an arrow prompt to be used instead of the default prompt.

The purpose of the `&rest options` argument to `examiner-top-level` is to pass through any command loop function options to the command loop function. **:prompt** is one such option to **dw:default-command-top-level**. The `examiner-prompt` value provided to this keyword is responsible for the Flavor Examiner's arrow prompt. (For information on other keywords: See the function **dw:default-command-top-level**.)

Other commonly desired options for the top-level function are the **:form-preferred** or **:command-preferred** dispatch modes. The former is all you need to do to get your command loop to read forms. The latter gives you the same behavior as the default for Lisp Listeners. You supply the **:dispatch-mode** keyword to **dw:default-command-top-level** as follows:

```
(dw:define-program-framework forms-too
  :top-level (dw:default-command-top-level
                :dispatch-mode :form-preferred))
```

For information on facilities available for writing your own command loop function: See the section "Managing Your Program Frame". The value you supply for this option is not evaluated.

**:command-evaluator**

Specifies function called after a command is read. Arguments passed to the called function are the program instance, the command, and any command arguments. At some point before, during, or after the execution of application-specific tasks, the evaluator

function should (**apply** <*command*> <*arguments*>). This only applies when you are using the default top-level command loop function.

The value you supply for this option is not evaluated.

**:panes** Specifies a list of panes to be included in the program frame. Each element of the list is itself a list of the form *(pane-name pane-type options ...)*. Pane types and their options can be specified using the Frame-Up Set Pane Options command. Six types of panes are available:

**:title** Pane for display of the program title (**:pretty-name** is the default).

**:command-menu**
Pane for menu of program commands.

**:display** Pane for display of application-generated output.

**:interactor**
Pane for interactive input/output.

**:listener** Similar to an interactor, but taller. (Use this pane when you want the interaction history to be visible, especially when the history is to include a significant amount of output.)

**:accept-values**
Pane providing the features and services of a **dw:accept-variable-values** menu. (If your frame includes an **:accept-values** pane, supply `"accept-values-pane"` as one of the values with the **:inherit-from** keyword to the **:command-table** option.)

The value you supply for this option is not evaluated.

The appearance and behavior of panes can be modified with a variety of keyword options; not all are appropriate for use with every pane type. Each option is listed below with a description of its purpose and an indication of the pane types for which it is appropriate:

**:default-character-style**
Specifies list of the form *(family face size)* to specify the style of characters displayed in the pane. The default style for **:display** panes is **(:fix :roman :normal)**; for **:title** panes **(:swiss :bold :large)**; and for **:command-menu** panes **(:jess :roman :normal**). (For more information on available styles: See the section "Character Styles".)

This option is applicable to all pane types.

**:height-in-lines**

Specifies integer to fix the height of the pane to a number of text lines. The actual height in pixels depends on the **:default-character-style** for the pane (see above).

This option is applicable to the **:title**, **:display**, **:interactor**, **:listener**, and **:accept-values** pane types. It is settable with the Frame-Up Set Pane Options command.

**:size-from-output**

Boolean option specifying whether a pane is sized according to the space needs of output to that pane; the default is **t** for **:command-menu** and **:accept-values** pane types, **nil** for other pane types.

This option is applicable to the **:title**, **:command-menu**, **:display**, and **:accept-values** pane types. It is settable with the Frame-Up Set Pane Options command.

**:typeout-window**

Boolean option specifying whether a typeout (pull-down) window for ***terminal-io*** appears within the pane; the default is **nil**.

This option is applicable to **:display**, **:interactor**, and **:listener** pane types. It is settable with the Frame-Up Set Pane Options command.

**:automatically-remove-typeout-window**

Boolean option for Display, Interactor, and Listener panes specifying whether automatically to remove any typeout window that may appear. It is settable with the Frame-Up Set Pane Options command. By default the typeout window is automatically removed, that is, the system will generate a "Type any character to refresh this display" message for the window and remove it when a character is typed. If you do not want this action to occur, set this option to **nil** and then be sure to include in your program code to take care of removing the typeout window. See the init option **(flavor:method :deexposed-typeout-action tv:sheet)**.

**:redisplay-string**

Specifies a string written to the pane (starting at top) whenever the pane is redisplayed. This option is mutually exclusive with the **:redisplay-function** option (see below).

**:redisplay-string** is applicable to the **:title** and **:display** pane types. It is settable with the Frame-Up Set Pane Options command.

**:redisplay-function**

Specifies the name of a user-defined function that runs whenever the pane is redisplayed. This option is mutually exclusive with the **:redisplay-string** option (see above).

The redisplay function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream on which to do output. If you write this as a defmethod, you get to access the program state variables.

**:redisplay-function** is applicable to the **:title** and **:display** pane types. It is settable with the Frame-Up Set Pane Options command.

**:redisplay-after-commands**

Boolean option specifying whether output to the pane is to be redisplayed after each command is executed; the default is **t** for **:accept-values** and **:display** panes, **nil** for **:title** panes.

This option is applicable to the **:title**, **:display**, and **:accept-values** pane types. It is settable with the Frame-Up Set Pane Options command.

The following options are applicable only to the **:command-menu** pane type:

**:menu-level**

Specifies a unique identifier for each command menu in the program when more than one command menu is needed. The default value (for a single command menu) is **:top-level**. This is always used in conjunction with the **:menu-level** option to **define**-*program-name*-**command**. You can set this option with the Frame-Up Set Pane Options command by specifying a Menu Identifier after selecting pane type Command-Menu.

**:rows** Specifies a list, each element of which is a list of command names (strings) to be included in the same row. The Menu Geometry choice in the Frame-Up Set Pane Options command allows you to set this option.

**:columns** Specifies a list, each element of which is a list of command names (strings) to be included in the same column. The Menu Geometry choice in the Frame-Up Set Pane Options command allows you to set this option.

**:equalize-column-widths**
Boolean option specifying whether the widths of columns containing command names be equal; the default is **nil** (widths adjust according to size of the output in each column). The Compress item columns choice in the Frame-Up Set Pane Options command allows you to set this option.

**:center-p** Boolean option specifying whether command names are centered (left-right) in the command menu; the default is **nil** (flush left). The Center menu items choice in the Frame-Up Set Pane Options command allows you to set this option.

The following two options are applicable only to the **:display** pane type:

**:flavor** Specifies the pane flavor to use for this pane; the default is **dw:dynamic-window-pane**. You can set this option using the Frame-Up Set Pane Options command.

**:incremental-redisplay**
Boolean option specifying whether redisplayed information is limited to items that have changed since the last redisplay, rather than the entire pane. If **t**, you must write the appropriate redisplay function (see **:redisplay-function** above). You can set this option using the Frame-Up Set Pane Options command.

For information on incremental redisplay: See the section "Displaying Output: Replay, Redisplay, and Formatting". See also the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

The following option is applicable only to the **:accept-values** pane type:

**:accept-values-function**
Specifies a function for creating a **dw:accept-variable-values**-like display; it defaults to an internal one that operates on program state variables. You can set this option using the Frame-Up Set Pane Options command.

The function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream for I/O.

The accept-values display is created by wrapping the body of the function you write in a **dw:accepting-values**

macro: See the function **dw:accepting-values**. The wrapping is done for you by **dw:define-program-framework**. Note that the state is stored only in the program: it is not duplicated in the pane. Your program must "remember" the values and give them back as defaults. You can change the values of these variables elsewhere in the program, as well as in the accept-values pane. The general form of the function you write is

```
(defmethod (my-avv-function program) (stream)
  (setq state-var-1 (accept ... :default s-v-1)
  (setq state-var-2 (accept ... :default s-v-2)
  (setq state-var-3 (accept ... :default s-v-3)
  ...)
```

For an example, see the program avv-pane-test in the file

SYS:EXAMPLES;DEFINE-PROGRAM-FRAMEWORK.LISP .

The **:default-character-style** keyword option is inherited from **dw:dynamic-window** (via **dw:dynamic-window-pane**, on which all program panes are based by default). Many more keyword options to **:panes** exist; most of them, however, are inappropriate for use with panes created via **dw:define-program-framework**. Among keywords that are appropriate, the following are most useful:

**:blinker-p**

Boolean option specifying whether a blinker appears in the pane. This option defaults to **t** for the **:interactor** and **:listener** pane types, **nil** for other pane types. When you define panes in the **dw:define-program-framework** form, you have to make sure that the pane you wish to interact with using accept-values or accepting-values has **:blinker-p t** set.

Example:

```
(edit-pane :DISPLAY :blinker-p t :incremental-redisplay nil)
```

This allows you to

```
(dw:accepting-values (dw:get-window-pane 'edit-pane)) ...)
```

If you do not set **:blinker-p** to **t**, you will get an error that states that the window could not set **:visibility** and that **:visibility** is **nil**. **:more-p**

Boolean option specifying whether *more processing* is enabled. More processing lets the user control scrolling of character output to a window. The default is **t** for the **:display** and **:listener** pane types, **nil** for other pane types.

**:end-of-page-mode**
Specifies what happens when queued output exceeds the space available in the current viewport of the pane. There are three possibilities:

**:scroll** causes the pane to scroll automatically to accommodate the output.

**:truncate** causes scrolling to be the responsibility of the user, who must press the `SCROLL` key to see more output.

**:wrap** causes new output to appear at the top of the pane, rather than at the bottom as in the case of **:scroll** or **:truncate**.

**:scroll-factor**
Specifies the number of lines by which to scroll the pane when the **:end-of-page-mode** is **:scroll**.

**:label** Specifies the string that appears as a label in the lower, left-hand corner of the pane (directly inside the border). The character style used is the default style for the pane. You may only use the **:label** option if not using the **:margin-components** option, described below.

**:margin-components**
Takes a list of options specifying characteristics of pane margins. The default is for a 1-pixel-wide border and a 4-pixel margin between the border and displayed output to the pane. Do not specify the **:margin** for a vertical **dw:margin-scroll-bar** component.

The defaults are implemented by the list `((dw:margin-borders) (dw:margin-white-borders :thickness 4))`. **dw:margin-borders** and **dw:margin-white-borders** are flavors for controlling the margin specifications of dynamic windows. For an overview of these and related facilities: See the section "Using the Window System".

This option is applicable to all pane types.

[End of documentation for **:panes** option to **dw:define-program-framework**.]

**:selected-pane**
Designates pane selected (generally indicated by blinking cursor) when program is activated. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

The value you supply for this option is not evaluated.

**:query-io-pane**
Designates pane to which ***query-io*** is bound when program is ac-

tive. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

The value you supply for this option is not evaluated.

**:terminal-io-pane**

Designates pane to which ***terminal-io*** is bound when program is active. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): the type-out window of the pane with a **:typeout-window** option (see above), a **:listener** pane, a **:display** pane.

The value you supply for this option is not evaluated.

**:label** Designates pane on which program label is displayed if the program does not have a **:title** pane. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

The value you supply for this option is not evaluated.

**:configurations**

Specifies the layout and sizes of panes within the program frame. This option is evaluated. Program frames are built on a more basic type of window known as a *constraint frame*. The *constraint language* used to specify the layout and sizes of panes in a constraint frame is documented elsewhere: See the section "Specifying Panes and Constraints". Frame-Up writes the constraint frame description for you so you do not normally have to learn the constraint language.

In the default configuration, panes are vertically stacked in a single column and in the order specified by the **:panes** option (see above).

The value you supply for this option *is* evaluated.

**:state-variables**

Specifies a list of program variables whose states are preserved between activations of the program. Each variable is itself a list consisting of the variable name and, optionally, a default value, and a presentation type. Note that if you want a state variable to be included in the program frame's accept-values menu, you must include its presentation type. State variables are implemented as writeable instance variables to the program flavor. Example:

```
:state-variables
   ((half-life 2400 integer)
    (log-file #p"local:>log.file" pathname)
    (start-time 2732241600 time:universal-time))
                                  ;default 8/1/86 00:00:00
```

The value you supply for this option is not evaluated.

**:select-key**

Specifies a character for selecting the program via the SELECT key. You can set this option using the Frame-Up Set Program Options command.

The value you supply for this option *is* evaluated.

**:selectable**

Specifies whether or not the program should appear in the list of choices for Select Activity HELP. The default is **t** (the program should appear).

The value you supply for this option *is* evaluated.

**:system-menu**

Boolean option specifying whether the program appears on the System menu. If **t**, the program appears both in the *Programs* column of the top-level menu and in the *Create* second-level menu; the default is **nil**.

The value you supply for this option *is* evaluated.

**:size-from-pane**

Specifies the name of the pane on which to base the size of the whole program frame; the default is **nil**.

The value you supply for this option is not evaluated.

**:help**

Specifies the help message displayed when the HELP key is pressed while the program is selected. The value of this option can be either a string or a function. If it is a string, the string is displayed when the user presses HELP.

If the value of the **:help** option is a function, the function receives three arguments: the program object, the stream to which the help message should be output, and the string that has been typed so far.

The value you supply for this option is not evaluated.

**:inherit-from**

Specifies a list of other program flavors from which this program inherits; the default is **dw::program** (it is not evaluated).

Inheritance affects program **:state-variables** and any explicit local **defmethod**s. This allows one program to be built on top of another. Note that there is no inheritance of panes, configurations, or command tables. You need to exercise the **:command-table** option to specify the one(s) to use.

The value you supply for this option is not evaluated.

**:other-defflavor-options**

Specifies **defflavor** options as a list of keyword-value pairs. The value you supply for this option is not evaluated. The keyword-value pairs are passed through to the **defflavor** form that creates your program flavor when **dw:define-program-framework** is compiled. For information on available options:
See the section "Summary of **defflavor** Options".

See the section "Complete Options for **defflavor**".
Included among **:other-defflavor-options** can also be flavor init options to the program. For example, one option that may be of particular interest is **:superior**. If, for example, you want your program to display on a color console, you can specify `:superior color:color-screen`. Another such option is **:size**.

Note that, since your program has been associated with its own flavor, created by **defflavor**, and may have one or more methods defined for that flavor, you should probably include a **compile-flavor-methods** form in the appropriate place in your program.

For an overview of **dw:define-program-framework** and related facilities: See the section "Defining Your Own Program Framework".

For an example and additional information on the use of certain options to **dw:define-program-framework**, particularly those implementing the command interface: See the section "Managing Your Program Frame". More examples are available in the file SYS:EXAMPLES;DEFINE-PROGRAM-FRAMEWORK.LISP.

## Using Frame-Up

The Frame-Up Layout Designer is an interactive facility for creating the user interface to an application program. It is usually invoked from Zmacs, though also available on `SELECT Q` and through the System Menu.

Frame-Up is the interactive version of **dw:define-program-framework**, a macro for defining a program's window and command interface. Frame-Up lets you configure a *program frame* and specify options for individual *panes* within the frame. (For more information on frames and panes, see the section "Frames".) Other options, for the program as a whole, provide control over the program's command loop.

When you finish configuring the program frame and specifying pane and program options, Frame-Up creates the corresponding **dw:define-program-framework** code. This code is written to an editor buffer where it is available for hand editing. (For information on how to edit the frame configuration, see the above-referenced section on "Frames".) Alternatively, you can go back to Frame-Up, modify the interface, and have the new code written out in place of the old. Note, though, that once you have hand-modified the code, you cannot return to Frame-Up.

### Getting Started

In order to produce code for a framework definition, here is what you do. In a Lisp-mode Zmacs buffer, with the editor cursor at the point where you want the **dw:define-program-framework** macro to be written:

1. Type the extended command m-X Create Program Definition.

2. Type the program name.

Frame-Up enters the name you give in step 2 as the *name* argument to the **dw:define-program-framework** form that it is creating. That macro will create a flavor for your program of the same name.

There are other ways to enter the Frame-Up program, but these are primarily for use *after* you have associated Frame-Up with an editor buffer, as described: You can enter Frame-Up directly by pressing SELECT Q, or by selecting Frame-Up from the System Menu, or by using the command processor Select Activity command.

After you invoke the Frame-Up program in whatever manner, an initial display appears. This includes a default starting configuration for a program frame and a menu of Frame-Up commands. Program- and frame-level commands are listed together on the left of the command menu, pane-level commands on the right. You could start with any of these, but if you are unfamiliar with Frame-Up, we recommend that you start with commands in the first category. ( See the section "Program and Frame Commands in Frame-Up".) If you have entered the Frame-Up program from an editor buffer, then when you exit the program, the resulting **dw:define-program-framework** is written to your buffer. If you first enter the program by pressing SELECT Q or via the System Menu, nothing will be written out: You will need to type the command m-X Insert Program Definition from an editor buffer to write into it the result of running Frame-Up. The section "A Program-Framework Extended Example" contains figures that show what a Frame-Up screen looks like.

### Frame-Up Commands

### Program and Frame Commands

Five Frame-Up Layout Designer commands are included in this category: Set Program Options; Select Configuration; Reset Configuration; Preview; and Done. The following subsections consider each in turn.

### Set Program Options

The program options you can modify using the Set Program Options command are described below. (Where appropriate, references to the corresponding **dw:define-program-framework** options are given.)

**Program name** The name of the program flavor created by **dw:define-program-framework** for your application.

If you invoked Frame-Up from an editor buffer with the Create Program Definition extended command, the default value for this option is the name you supplied to that command.

**Select key** The key to use for selecting your program.

(See the function **dw:define-program-framework**.)

**Name of command-defining macro**

The name given to the macro created by **dw:define-program-framework** and used to define commands for your program.

The default, **t**, causes your program name to be used as part of this name. For example, if the name of your program is `shell-game`, the default command-defining macro will be `define-shell-game-command`.

You use the command macro created for you as you would **dw:define-program-command**. The syntax and keywords are the same, except that you do not have to supply the *program-name* argument, see the function **dw:define-program-command**.

(See the function **dw:define-program-framework**.)

**Read single-character command accelerators**

Boolean option specifying whether your program accepts single-character command accelerators; the default is `No`.

If you enter `Yes` for this option, you have three possible sources of accelerators:

1. Accelerators you inherit when you inherit command tables using the program option discussed below.

2. Standard accelerators you supply to your program. (See the section "Advanced Command Facilities".)

3. Accelerators you define yourself. (See the section "Advanced Command Facilities".)

(See the function **dw:define-program-framework**.)

**Inherit commands from command tables**

The name(s) of command table(s) from which your program inherits commands and, if specified by the above option, command accelerators.

For example, supplying a value of `user` to this option results in all of the commands normally available in a Lisp Listener being available in your program, in addition to program commands you define yourself.

The default for this option — '("colon full command" "standard arguments" "standard scrolling") — enables use of extended (m-x) and colon full commands, standard single-character accelerators like c-U, and standard scroll keys like SCROLL and m-SCROLL. These are enabled only if you specify Yes to the **Read single-character command accelerators** option.

If your frame includes an **:accept-values** pane, supply "accept-values-pane" as one of the values with the **:inherit-from** keyword to the **:command-table** option.

(See the function **dw:define-program-framework**.)

### Select Configuration

The Select Configuration command gives you a choice of two standard configurations for your program frame. The first consists of a command-menu pane and a listener pane; the second consists of a title, command-menu, display, and interactor pane. (For a description of pane types, see the section "Set Pane Options Frame-Up Command".)

You may select a standard configuration and then modify it using one or more of the pane-oriented commands, see the section "Pane Commands in Frame-Up".

### Reset Configuration

The Reset command restores the original program frame. (The original frame is the one displayed when you first enter Frame-Up; it consists of a single display pane.)

### Preview

The Preview command lets you see what the frame you have configured looks like on a full-screen display without having to compile your program. Without this command, to see your program frame you would have to exit Frame-Up, compile the **dw:define-program-framework** definition, and select your program. With it, you can look at the frame directly and, if you don't like the result, continue editing the layout before writing out the interface code.

### Done

The Done command signals the end of the Frame-Up session. What happens when you invoke this command depends on how you entered Frame-Up:

- If you entered Frame-Up from an editor buffer via the Create Program Definition or Edit Program Definition extended editor command, Frame-Up returns you to that buffer and automatically writes out the **dw:define-program-framework** macro corresponding to the interface you configured.

In the case of Edit Program Definition, the new code replaces the code that was already there (**dw:define-program-framework** macro).

- If you entered Frame-Up from the System Menu or via SELECT Q, you are returned to the activity selected prior to entering Frame-Up.

  In this case, the **dw:define-program-framework** code corresponding to your interface is not written automatically to an editor buffer. You must select the buffer you wish the code to be written to and use the extended editor command m-X Insert Program Definition.

## Pane Commands

Five Frame-Up Layout Designer commands are available for manipulating panes: Set Pane Options; Set Pane Name; Split Pane; Swap Panes; and Delete Pane. The following subsections discuss each in turn.

(Note that, after finishing the Frame-Up session, further editing of the code affecting the appearance of program panes and the frame as a whole is possible. For more information, see the section "Frames". In particular, see the section "**:layout** Constraint Frame Specification" and see the section "**:sizes** Constraint Frame Specification".)

## Set Pane Options

Pane options you can modify using the Set Pane Options command include the pane name and type. Other options depend on the pane type. Six types are available:

**accept-values** — Pane providing the features and services of a **dw:accept-variable-values** menu (the kind of menu used to display the pane options themselves).

**display** — Pane for display of application-generated output.

**title** — Pane for display of the program title.

**command-menu** — Pane for menu of program commands.

**interactor** — Pane for interactive input/output.

**listener** — Similar to an interactor, but taller. (Use this pane when you want the interaction history to be visible, especially when this history is to include a significant amount of output.)

The options for the various pane types are listed in the following table.

**Table of Frame-Up Pane Options**

| *Pane Type* | | | *Options* | | | |
|---|---|---|---|---|---|---|
| **Accept-Values** | Accept values function | Redisplay each time around command loop | Height in lines | Set size of pane from contents | | |
| **Display** | Pane flavor | Redisplay each time around command loop | Height in lines | Set size of pane from contents | Typeout window | Four redisplay options* |
| **Title** | | Redisplay each time around command loop | Height in lines | Set size of pane from contents | | Four redisplay options* |
| **Command-Menu** | Menu geometry | Menu identifier | | Center menu items | Compress item columns | |
| **Interactor** | | | Height in lines | | Typeout window | Automatically remove typeout window |
| **Listener** | | | Height in lines | | Typeout window | Automatically remove typeout window |

*The four redisplay options are:
Redisplay output generator
Redisplay string
Redisplay function
Incremental redisplay

Here are the pane options that are settable from the Frame-Up program. Additional options that you can hand-edit into the **dw:define-program-framework** macro are listed in "**:panes** Option to **dw:define-program-framework**". We present only brief descriptions of these options here. Full descriptions are included in the reference section on the **dw:define-program-framework** macro.

**Accept values function**

An option of Accept-Values panes. Specifies a function for creating a **dw:accept-variable-values**-like display. This option maps to the **:accept-values-function** keyword option for **:accept-values** panes. If you include an Accept Values pane in your program frame but do not specify an Accept Values Function, the option defaults to an internal function that uses your program's state variables as the variables in the accept-values display. The state variables are those specified by the **:state-variables** option to **dw:define-program-framework**. ( See the function **dw:define-program-framework**.)

**Pane flavor**

An option for display panes only. The pane flavor to use for this pane; the default is **dw:dynamic-window-pane**. This option maps to the **:flavor** keyword option for **:display** panes, described in **dw:define-program-framework**.

**Menu geometry**

An option for command-menu panes only. Specifies how the menu is to be laid out. You have three choices: you can let Frame-Up come up with a configuration (`Default`) that is in most cases reasonable; you can control the layout yourself by specifying menu `Rows`; or you can control layout by specifying menu `Columns`.

If you select `Rows`, then you are asked if you want to **Specify number of rows or row contents**. If `Number`, then enter a value in the **Number of rows** field that appears. If `Contents`, then enter one or more command names (strings) to be the **Items in row 1**, followed by the entering of one or more strings to be the **Items in row 2**, and so on, until all the rows are specified. This option maps to the **:rows** keyword option for **:command-menu** panes. See the function **dw:define-program-framework**.

If you select `Columns` for the **Menu geometry** option, you proceed in a fashion analogous to that described for `Rows`. This option maps to the **:columns** keyword option for **:command-menu** panes. See the function **dw:define-program-framework**.

If you specify menu rows or columns by their contents, the string used to identify each command must be the same as that specified in the **:menu-accelerator** option to the command definer used for the program. (See the function **dw:define-program-command**.) The command definer is specified by one of the options in the Set Program Options command. See the section "Set Program Options Frame-Up Command".

**Menu identifier**

An option for command-menu panes only. Symbol identifying the command menu to appear in this pane if the program frame includes more than one. If only one command menu is available, choose the default value (`:TOP-LEVEL`) for this option.

This option maps to the **:menu-level** keyword option for **:command-menu** panes, as described in **dw:define-program-framework**

**Center menu items**

An option for command-menu panes only. Boolean option specifying whether command names are centered (left-right) in the command menu. The default is `No`, causing command names to be flush left in the column. This option maps to the **:center-p** keyword option for **:command-menu** panes, as described in **dw:define-program-framework**

**Compress item columns**

An option for command-menu panes only. Boolean option specifying whether columns of command names are compressed on the left side of the pane or spread out over the full horizontal extent of the pane. The default is `Yes` (compressed to the left). This option maps to the **:equalize-column-widths** keyword option for **:command-menu** panes, as described in **dw:define-program-framework**

**Redisplay each time around command loop**

A Boolean option for Accept-Values, Display, and Title panes specifying whether to redisplay the pane after each command is executed. The default is `Yes` for Accept-Values and Display panes, `No` for Title panes. This option maps to the **:redisplay-after-commands** keyword option for program panes, as described in "**:panes** Option to **dw:define-program-framework**"

**Set size of pane from contents**

A Boolean option for Accept-Values, Display, and Title panes specifying whether a pane is sized according to the space needs of output to that pane. The default is `Yes` for Accept-Values panes, `No` for Display and Title panes. This option maps to the **:size-from-output** keyword option for program panes, as described in "**:panes** Option to **dw:define-program-framework**"

**Height in lines**

An option for all types of program panes except Command-Menu panes. Fixes the pane height to the specified number of lines. The default value is `1` for Title panes and `4` for Interactor panes. No default is provided for Listener Panes. This option maps to the **:height-in-lines** keyword option for program panes, as described in "**:panes** Option to **dw:define-program-framework**"

**Redisplay output generator**

An option for Display and Title panes. Specifies one of three possibilities for generating redisplay to the pane: no redisplay generator (`None`); a redisplay string (`String`); or a redisplay function (`Function`).

If you specify `String`, then the **Redisplay string** option appears.

If you specify `Function`, then both the **Redisplay function** and **Incremental redisplay** options appear. See the section "Set Pane Options Frame-Up Command".

**Redisplay string** An option for Display and Title panes. Specifies a string written to the pane (starting at top) whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay function** option.

This option maps to the **:redisplay-string** keyword option for program panes, as described in **dw:define-program-framework**.

**Redisplay function** An option for Display and Title panes. The function that runs whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay string** option. This option maps to the **:redisplay-function** keyword option for program panes, as described in **dw:define-program-framework**.

**Incremental redisplay**

A Boolean option for Display and Title panes, specifying whether redisplayed information is limited to items that have changed since the last redisplay, rather than the entire pane; the default is `No`.

If you specify `Yes`, you must write the appropriate redisplay function. See the section "Displaying Output: Replay, Redisplay, and Formatting". This option maps to the **:incremental-redisplay** keyword option for **:display** panes, as described in **dw:define-program-framework**.

**Typeout window** A Boolean option for Display, Interactor, and Listener panes, specifying whether a typeout (pull-down) window for ***terminal-io*** appears within the pane. The default is `No`. Generally, your program should set this option to `Yes` for the one pane in which you want to receive messages for ***terminal-io***. This option maps to the **:typeout-window** keyword option for program panes, as described in "**:panes** Option to **dw:define-program-framework**"

**Automatically remove typeout window**

A Boolean option for Display, Interactor, and Listener panes, specifying whether automatically to remove any typeout window that may appear. By default, the typeout window is automatically removed, that is, the system generates a "`Type any character to refresh this display`" message for the window and removes it when a character is typed. If you do not want this

action to occur, click No on this option, and then be sure to include code in your program to take care of removing the window. This option maps to the **user::automatically-remove-typeout-window** keyword option for program panes, as described in "**:panes** Option to **dw:define-program-framework**"

### Set Pane Name

The Set Pane Name command lets you change the name of a pane. The arguments to this command are the current name of the pane and the new name.

### Split Pane

The Split Pane command divides the specified pane in half. Arguments to this command are the pane to divide and whether the division is horizontal or vertical.

Splitting a pane horizontally causes the two daughter panes to appear in a column orientation, one on top of the other. Splitting a pane vertically causes the two daughter panes to appear in a row orientation, side-by-side.

### Swap Panes

The Swap Pane command exchanges the position of two panes. The two panes must occur in either the same row or same column.

### Delete Pane

The Delete Pane command deletes a specified pane from the configuration for the program frame.

## Zmacs Commands for Frame-Up

### Create Program Definition

The Create Program Definition command initiates a Frame-Up session from an editor buffer. When the session is terminated (via the Done command to Frame-Up), the **dw:define-program-framework** code corresponding to the configured interface is inserted into the buffer at point.

Create Program Definition is an extended (m-X) Zmacs command. When invoked, it firsts prompts you for the name of the program, then enters Frame-Up.

If you entered Frame-Up via SELECT Q or from the System menu, you must use the Insert Program Definition extended command to write the **dw:define-program-framework** code into an editor buffer.

### Insert Program Definition

Insert Program Definition is an extended (m-X) Zmacs command for writing Frame-Up Layout Designer code into an editor buffer. Use it when you have entered Frame-Up via SELECT Q or from the System menu, rather than through the Create Program Definition extended command.

When you exit from the Frame-Up session (via the Done command), select an editor buffer and use the Insert Program Definition command to write the **dw:define-program-framework** code corresponding to the configured interface. The code is inserted at point.

### Edit Program Definition

You can use the Edit Program Definition extended (m-X) command to re-enter a Frame-Up session and make further modifications to the user interface configuration. This occurs after you have already written into your editor buffer the **dw:define-program-framework** macro corresponding to an earlier session. (The original code may have been written through either the Create Program Definition or Insert Program Definition extended command.) If you have edited in your own changes, they may be lost. The Edit Program Definition command warns you of this.

When you terminate the new Frame-Up session (via the Done command), the code corresponding to the new interface configuration replaces the original code.

### Defining Commands within Your Own Framework

Defining and managing commands within your own program is very similar to managing the Command Processor. The same tasks and issues are involved: defining commands, specifying single-key accelerators, managing command tables, and the like. Most of the information in "Managing the Command Processor" is applicable to program framework command management.

### The Command-Definition Macro

The **dw:define-program-framework** macro sets up a command-definition macro for each program defined by it. The default name for the command-definition macro for a program whose name is *program-name* is **define-***program-name***-command**. You get this default when the keyword **:command-definer** is set to **t**. If you assign some other symbol as the value of **:command-definer**, then that symbol becomes the name of the command-definition macro.

The definition of the **dw:define-program-command** macro serves as a model of command-definition macros created by **dw:define-program-framework**: your program's **define-***program-name***-command** has the same arguments and performs the same operations. The only difference between the two is that your **define-***program-name***-command** does not require an argument that specifies *program-name*, while **dw:define-program-command** does. You should not, in fact, use

**dw:define-program-command** in your program, but instead use **define-*program-name*-command**. As described in the dictionary entry for **dw:define-program-command**, this macro not only allows you to define commands for your program, but also specifies whether they are to be included on a command-menu pane created by **dw:define-program-framework** and other things like whether there are command accelerators.

The macro also ensures that the defined commands are installed in your program's command table. This command table, generated by your command-definition macro, is returned by the function **dw:program-command-table**.

The command definition created by your **define-*program-name*-command** generates two internal methods for the program flavor. (Remember: **dw:define-program-framework** creates a flavor for your program named *program-name*.) One of these internal methods parses the command, and the other one executes it. The methods provide lexical access to the program's state variables, both in the body of the command definition and in the command's argument list, so you can use state variables as arguments.

Here are a couple of examples:

```
(dw:define-program-framework g-t
  :command-definer t
  :command-table (:inherit-from '("colon full command"
                                  "standard arguments"
                                  "standard scrolling")
                 :kbd-accelerator-p 't)
  :panes
  ((pane-1 :display)
   (command-menu-1 :command-menu :menu-level :top-level))
...)

(define-g-t-command (com-clear :menu-accelerator "klear"
                               :keyboard-accelerator #\k) ()
           (send (dw:get-program-pane 'pane-1) :clear-history))

(define-g-t-command (com-show-help-file) ()
   (let ((pane (dw:get-program-pane 'pane-1)))
     (cp:execute-command 'si:com-show-file "v:>elm>help.text"
                         :output-destination (list pane))))
```

### Command Errors

You can use the function **dw:command-error** to specify an error message to be displayed if your command encounters an error in execution. For example, inside a function called by a command to draw a line between two points you could put

```
(when (and (= from-x1 from-x2) (= from-y1 from-y2))
                   (dw:command-error "Length must not be zero"))
```

### Single-Key Accelerators

Commands defined with program frames can have single-keystroke accelerators. To implement them, you need to set the **:keyboard-accelerator-p** option to the **dw:define-program-framework :command-table** keyword to **t**, and you need to specify a **:keyboard-accelerator** in your command definition. This is illustrated in the first example in "The Command-Definition Macro". You can also use **cp:define-command-accelerator** to define accelerated Command Processor commands.

### Menu Commands

To include a mouse-sensitive menu of commands in your program frame, you need to include a **:command-menu** pane in your list of **:panes**, and then all you need to do is include a **:menu-accelerator** keyword and value in your **define-***program-name***-command** macro, as shown in the first example in "The Command-Definition Macro".

### Menu Subcommands

To obtain a sub-menu of a command in a menu pane, you:

1. Use the **define-***program-name***-command** macro to define a command that is to have subcommands. Its **:menu-level** option will specify something other than **:top-level**.

2. Specify a set of subcommands with **define-***program-name***-command** macros, setting each of their **:menu-level** options to the level chosen in step 1.

3. Set up a command-menu pane that has the desired menu level.

4. Use the **dw:define-subcommand-menu-handler** macro to specify the string to be included in the program's command menu, the menu level in which the menu handler is to be included, and the menu level of the subcommands. For an example, see the function **dw:define-subcommand-menu-handler**.

### Getting Your Own Program Interactor to Read Lisp Forms

A Lisp listener is a window running a command loop in its own process and printing out the values. Do not try to make a Lisp listener be a pane of your program. If you do, you will wind up with two processes and the situation will be confused.

The command loop used by a program framework is functionally equivalent to the one in the Lisp window. All you need to do to get your own command loop to read forms is to supply the **:dispatch-mode** keyword to **dw:default-command-top-level**. Example:

```
(dw:define-program-framework forms-too
  :top-level (dw:default-command-top-level :dispatch-mode :form-preferred))
```

### Setting up a Non-Echoing Command Loop

The Command Processor expects an interactive stream, that is, one that does input editing. For this reason, to have a program frame in which the commands are not echoed, you must not only default the echoing of the commands themselves, but you must also arrange not to require an input editing stream. This is most easily done by defining the program to have single-character command accelerators. The program need not define any such accelerators: This is just to avoid the use of the full Command Processor. Note that a program that had any keyboard commands (especially ones with long names), but did not echo, would be very difficult to use.

Here is an extended example of a program with no echoing. Notice the **:echo-stream** option to **dw:default-command-top-level** is set to **ignore**.

```
(dw:define-program-framework no-echo
  :select-key #\2
  :command-definer t
  :command-table (:inherit-from nil
                  :kbd-accelerator-p t)
  :top-level (dw:default-command-top-level :echo-stream ignore)
  :panes ((display :display :redisplay-function 'draw-circles)
          (menu :command-menu))
  :state-variables ((circles nil)
                   ))

(defstruct circle
  center-x
  center-y
  radius)

(defmethod (draw-circles no-echo) (stream)
  (dolist (circle circles)
    (dw:with-output-as-presentation (:object circle :type 'circle :stream stream)
      (graphics:draw-circle (circle-center-x circle) (circle-center-y circle)
                            (circle-radius circle) :stream stream))))

(define-no-echo-command (com-add-circle :menu-accelerator t)
    ((x 'number :default 500)
     (y 'number :default 500)
     (radius 'number :default 50))
   (push (make-circle :center-x x :center-y y :radius radius) circles))

(define-no-echo-command (com-delete-circle )
    ((circle 'circle))
   (setq circles (delete circle circles)))

(define-presentation-to-command-translator delete-this-circle (circle) (circle)
  `(com-delete-circle ,circle))
```

### Accessing Program Frame Objects

The variable **dw:*program-frame*** is bound to the program frame that the current process is operating. The following example was generated by selecting the Frame-Up Layout Designer (which is an example of a program created with **dw:define-program-framework**) and pressing SUSPEND to enter a **break** loop:

```
Command: ,dw:*program-frame* ==>
#<PROGRAM-FRAME Frame-Up 1 3106337 exposed>
```

The function **dw:find-program-window** returns the program frame of a specified program flavor, whether the frame is exposed or not. Optionally, this function creates and initializes an instance of the program if one does not already exist. Using **dw:get-program-pane** is how you access a particular pane of a program frame, rather than the frame as a whole.

### Adding the Help-Program to Your Framework

The Help program defines translators on Mouse Middle for command menu items and command name displays that invoke the formatter on the corresponding documentation records. At present, a record to be used in this way is just a record whose name is "The *Command-name Program* Command" or "The *Item-name Program* Menu Item"; for example, " "The Edit Object Namespace Editor Command" ". Additionally, there is a special Help command which lets you display documentation for commands or overview topics. The set of overview topics is gotten from the links in a record entitled "*Program* Overview Help Topics". Documentation records are created using Symbolics Concordia.

The **dw:help-program** facility automatically attaches documentation to a program framework. It attaches documentation to all menu items, CP commands, and the program name.

To include it in your program framework:

1. In your **dw:define-program-framework**,

   - Include **dw:help-program** in the list of **:inherit-from** flavors.

   - Include "help-program" in the list of **:command-table :inherit-from** command tables.

```
(dw:define-program-framework concordia-bank-account
  :inherit-from (dw:help-program)
  :command-table (:inherit-from '("help-program"
                                  "colon full command"
                                  "standard arguments"
                                  "standard scrolling"
                                  "input editor compatibility")
                  :kbd-accelerator-p 'nil)
        ...)
```

2. For each CP command you want documented, create a documentation topic "The *command-name program-name* Command".

   For each menu item you want documented, create a documentation topic "The *menu-item-name program-name* Menu Item".

```
(dw:define-program-framework concordia-bank-account
   :command-definer define-conc-bank-command
        ...)

(define-conc-bank-command (com-deposit
                            :menu-accelerator t)
        ...)
```

displays these records:

```
Section "The Deposit Concordia Bank Account Command"
Contents
   The Deposit command prompts you for an account to deposit
Contents
Oneliner
   Deposit increases the balance of an account.
Oneliner
Keywords
Keywords
End of "The Deposit Concordia Bank Account Command" record

Section "The Deposit Concordia Bank Account Menu Item"
Contents
   The Deposit menu item prompts for arguments to the Deposit
Contents
Oneliner
   Deposit prompts for arguments from the keyboard
Oneliner
Keywords
Keywords
End of "The Deposit Concordia Bank Account Menu Item" record
```

3. Any presentations of type **dw:program-name**, with a presentation object of the symbol *program-name*, is documented by the topic "The *program-name*". You may want your program's title to display as this presentation type.

```
(dw:define-program-framework concordia-bank-account
 :panes
 ((title-pane :title :height-in-lines 1 :redisplay-after-commands nil
              :redisplay-function 'refresh-title))
       ...)

(defmethod (refresh-title concordia-bank-account) (stream)
  (dw:with-output-as-presentation (:stream stream
                                   :object 'concordia-bank-account
                                   :type 'dw:program-name)
    (send stream :display-centered-string "Concordia Bank Account")))
                 displays this record:

        Section "The Concordia Bank Account"
        Contents
           The Concordia Bank Account is a program which
           provides a simple bank account for your use.
        Contents
        Oneliner
        Oneliner
        Keywords
        Keywords
        End of "The Concordia Bank Account" record
```

4. Make sure your documentation records are loaded while your program is running. The best way to do this is to include the documentation records in your system definition.

5. When running the program frame, click Mouse-Middle over a menu item or the program title to display the documentation.

6. Type `Help` *command-name* to display documentation about a command.

7. Type `Show Documentation` *documentation-topic* to display any loaded documentation.

## A Program-Framework Extended Example

The following extended example demonstrates how to write a simple program using the Frame-Up Layout Designer to produce a program framework and using top-level program framework facilities to define commands for the program.

The program to be written manages a "to do" list. It maintains a list of tasks in a program frame display pane. As the user finishes a task, that task is "crossed off" the list by being overwritten with a gray rectangle. The program includes commands to add new items, as well as to clean up the list by removing some or all the "done" tasks.

Before you enter the Frame-Up program, you need to make some design decisions about the program you're creating:

1. Decide how the program frame should look, that is, how many and what kind of panes you want. For example:

   - Title pane, 1 line high with title "NOTEPAD".
   - Display pane, for putting your to-do items on.
   - Command menu, arbitrarily sized to contain your commands.
   - Interactor pane, for echoing your typein and displaying system messages.

2. Choose a Select Key. Pressing SELECT HELP shows that "O" is unused.

3. Decide what commands the program needs and what they should do:

   - An Add Task command that enters an item on the task list.
   - A Toggle Thing Done command marks an undone task as done or changes the task's state from done to undone.
   - A Delete Task command removes an item from the task list.
   - A Delete Completed Tasks command removes all the crossed out items from the task list.
   - A Clear command empties the task list.

4. Decide how to activate the commands. You have the choices:

   - Type in a command name.
   - Use a single-key accelerator.
   - Click on items (presentations) in various ways.

   Having made all these choices, you are ready to select the editor and start Frame-Up. Press m-X Create Program Definition to bring it up. Figure ! shows the screen as it looks now.

   a. Use the Set Program Options command to set the SELECT key to O.

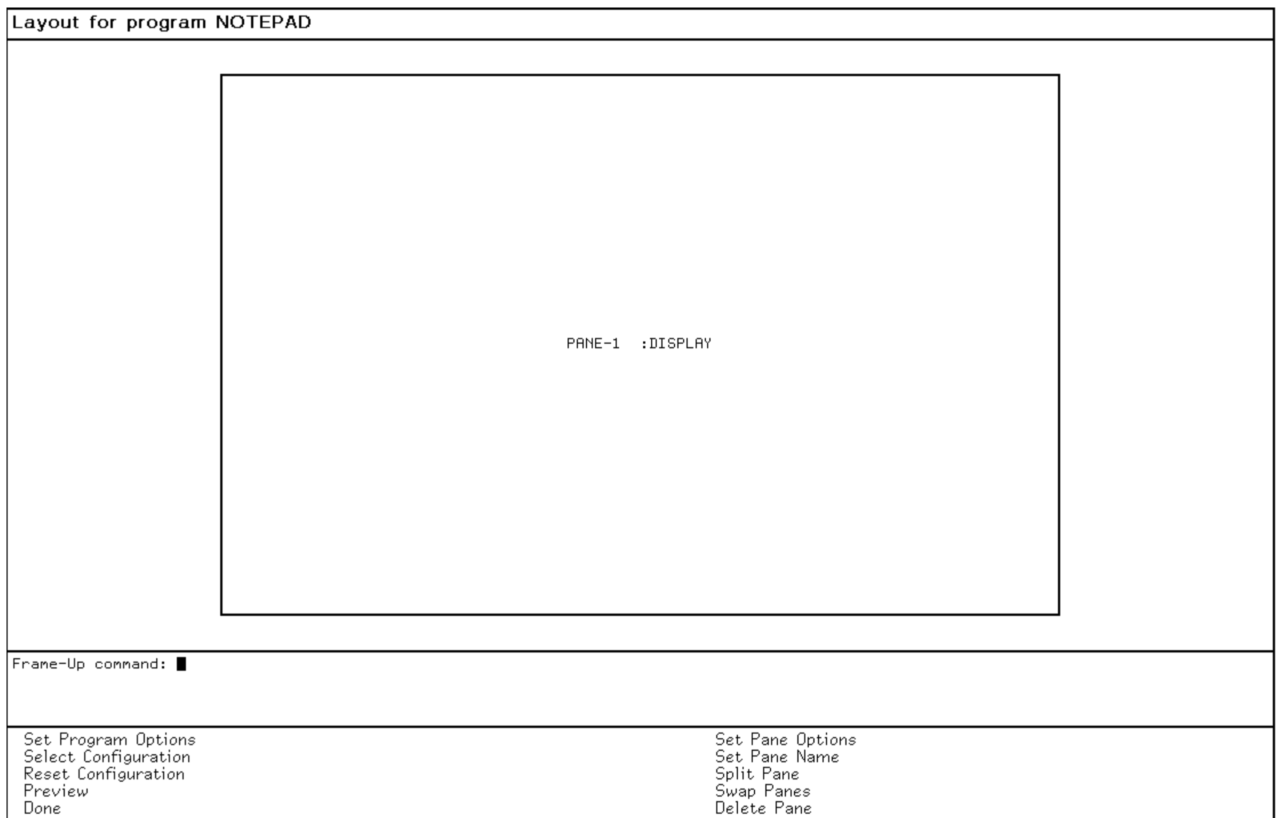



Figure 38. The initial Frame-Up display.

b. Click Right on Select Configuration so that you can choose "title display command-menu interactor".

c. Use the Set Pane Options command to set the following options for the Display pane: `Redisplay output generator` to `Function`, `Incremental Redisplay` to `yes`.

   Figure ! shows the screen as it looks now.

5. Click on the Preview command to see how your program frame now looks. It should resemble Figure !.

6. Type any character to go back into Frame-Up.

7. When you are finished with Frame-Up, click on Done. This puts you back in the editor, ready to edit your program framework definition macro and create your commands. Here is what Frame-Up writes into the editor buffer:

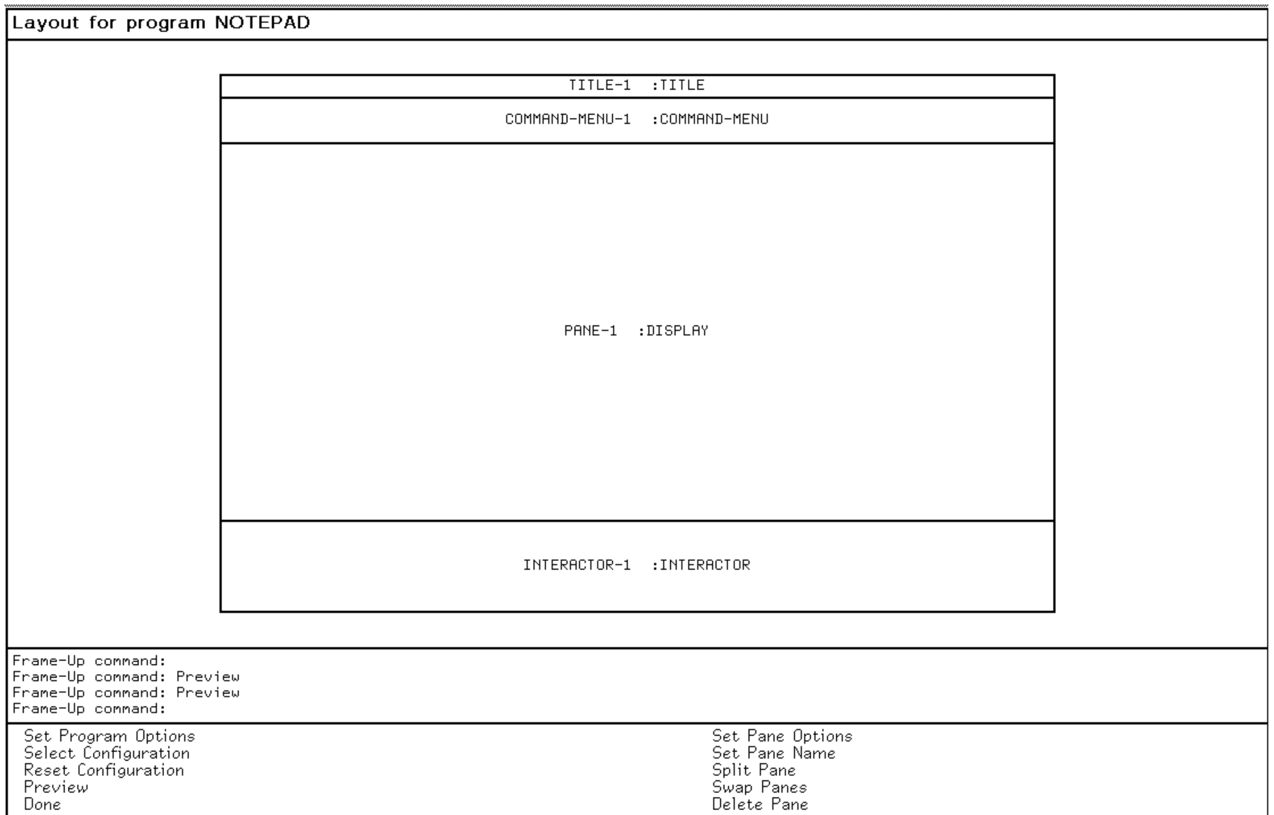



Figure 39. Frame-Up with display options set

```
(DW:DEFINE-PROGRAM-FRAMEWORK NOTEPAD
  :SELECT-KEY
  #\O
  :COMMAND-DEFINER
  T
  :COMMAND-TABLE
  (:INHERIT-FROM '("colon full command" "standard arguments"
                                       "input editor compatibility")
   :KBD-ACCELERATOR-P 'NIL)
  :STATE-VARIABLES
  ((task-list nil))
  :PANES
  ((TITLE-1 :TITLE :HEIGHT-IN-LINES 1 :REDISPLAY-AFTER-COMMANDS NIL)
   (COMMAND-MENU-1 :COMMAND-MENU :MENU-LEVEL :TOP-LEVEL)
   (PANE-1 :DISPLAY :INCREMENTAL-REDISPLAY T :REDISPLAY-FUNCTION 'NIL)
   (INTERACTOR-1 :INTERACTOR :HEIGHT-IN-LINES 4))
  :CONFIGURATIONS
  '((DW::MAIN (:LAYOUT (DW::MAIN :COLUMN TITLE-1 COMMAND-MENU-1
                                            PANE-1 INTERACTOR-1))
     (:SIZES
      (DW::MAIN (TITLE-1 1 :LINES)
```

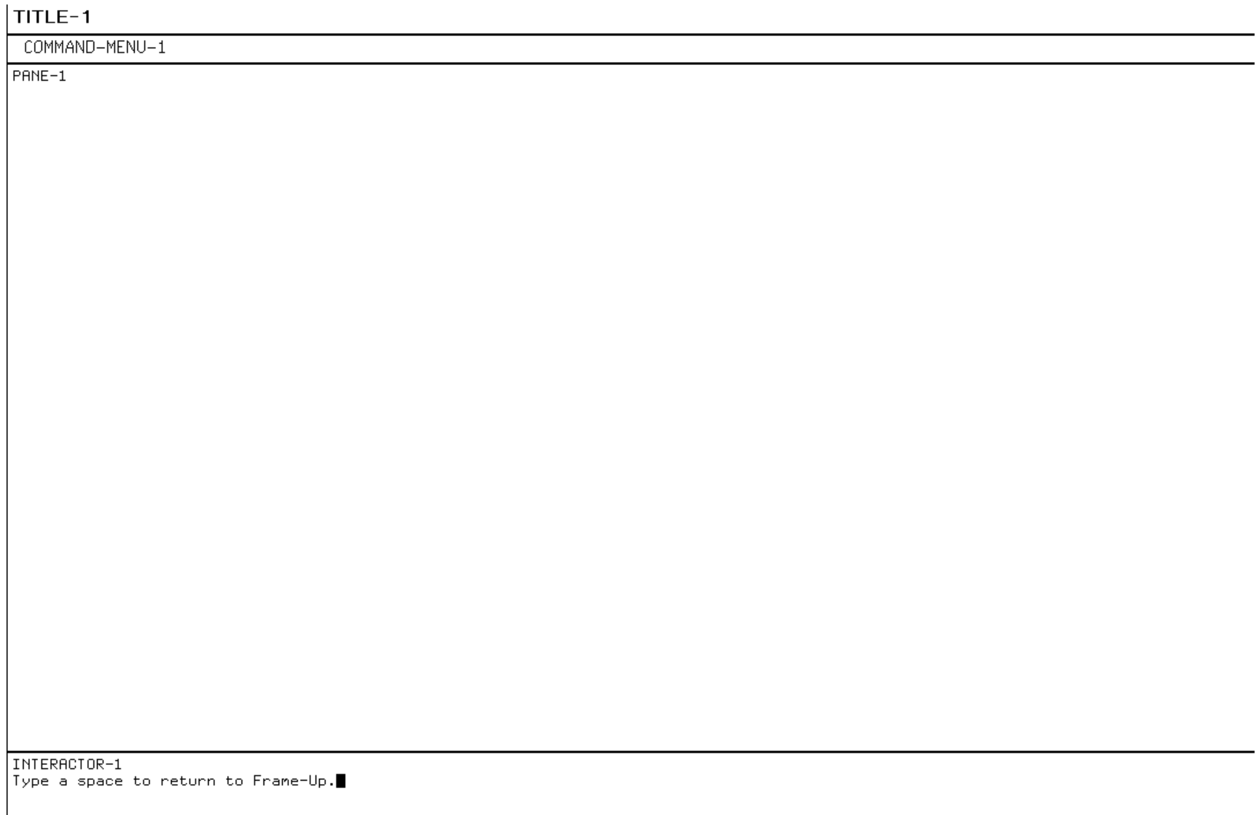



Figure 40. The Notepad Frame, initial appearance

```
(COMMAND-MENU-1 :ASK-WINDOW SELF :SIZE-FOR-PANE COMMAND-MENU-1)
                                                (INTERACTOR-1 4 :LINES)
:THEN (PANE-1 :EVEN))))))
```

At this point, if you want to change a choice, you can go back to Frame-Up by pressing m-X Edit Program Definition, changing items, and clicking on Done to write out a new program framework definition form. Once you have edited the form "by hand," however, you cannot use Frame-Up to change anything. If you do, your hand-edited changes will be overwritten.

8. The first item that needs to be added to the program framework definition is a state variable to represent the list of tasks. Change `:STATE-VARIABLES NIL` to `:STATE-VARIABLES ((task-list nil))`.

9. Create a flavor, a presentation type, and a redisplay function (in this case, a method) for your list items:

```
(defflavor thing-to-do
        ((description) (done?))
        ()
    (:conc-name "THING-")
    :initable-instance-variables
    :readable-instance-variables
    :writable-instance-variables)

(define-presentation-type thing-to-do ()
   :no-deftype t
   :printer ((thing stream)
             (write-string (thing-description thing) stream)))

(defmethod (display-tasks notepad) (stream)
  (formatting-table (stream)
    (loop for thing in task-list
         do
      (dw:with-redisplayable-output (:stream stream
                                     :unique-id thing
                                     :cache-value (thing-done? thing))
        (dw:with-output-as-presentation (:stream stream
                                         :object thing
                                         :type (type-of thing)
                                         :single-box t)
          (formatting-row (stream)
            (formatting-cell (stream)
              (if (thing-done? thing)
                (let ((presentation (present thing 'thing-to-do
                                             :stream stream)))
                  (when presentation
                    (multiple-value-bind (left top right bottom)
                      (dw:box-edges
                        (dw:presentation-displayed-box presentation))
                        (graphics:draw-rectangle
                                    left top right bottom
                                    :opaque nil
                                    :gray-level .1))))
                (present thing 'thing-to-do :stream stream))))))))
```

Part 2 of this manual contains information about how to produce redisplayable output as illustrated by the method above. See the section "Displaying Output: Replay, Redisplay, and Formatting".

Now edit the code specifying PANE-1, changing `:REDISPLAY-FUNCTION` from `NIL` to `'display-tasks`.

10. Here are our command definitions. Including the keyword **:menu-accelerator** and a string to be its value in the command definition causes the string to be

included in the program's command-menu pane. Note that we do not include a menu accelerator for the Toggle Task Done command, because we intend to make that command available only through clicking on a presentation of a task.

```
(define-notepad-command (com-add-task :menu-accelerator "Add Task")
        ((description 'string :default nil :prompt "task description"))
          (push (make-instance 'thing-to-do :description description) task-list))

(define-notepad-command (com-delete-task :menu-accelerator "Delete Task")
        ((thing 'thing-to-do :prompt "Task"))
          (setq task-list (delete thing task-list)))

(define-notepad-command (com-delete-completed-tasks
                           :menu-accelerator "Delete Completed Tasks")
        ()
   (setq task-list
        (delete-if #'thing-done? task-list)))

(define-notepad-command (com-clear :menu-accelerator "Clear")
        ()
  (setq task-list nil))

(define-notepad-command (com-toggle-task-done)
    ((thing 'thing-to-do :default nil :prompt "task"))
  (setf (thing-done? thing) (not (thing-done? thing))))
```

11. Define a presentation-type to command translator that enables the user to toggle the state of a task by clicking on it with the mouse:

```
(define-presentation-to-command-translator toggle-task (thing-to-do
                      :gesture :select :documentation "Toggle")
   (thing)
   `(com-toggle-task-done ,thing))
```

Your program is now ready to use. Remember to enclose each task you enter in quotes.

### Table of Program Framework Facilities

**dw:define-program-framework** *name* &key *pretty-name* (*command-definer* **nil**) (*command-table* **nil**) (*top-level* **'(dw:default-command-top-level)**) (*command-evaluator* **nil**) (*panes* **'(dw::main :listener)**) *selected-pane query-io-pane terminal-io-pane label-pane* (*configurations* **nil**) (*state-variables* **nil**) *(selectable* **t***)* (*select-key* **nil**) (*system-menu* **nil**) (*size-from-pane* **nil**) (*inherit-from* **'(dw::program)**) (*other-defflavor-options* **nil**)
Defines a program framework, including: a command-defining macro, a flavor named after the program with instance variables you can specify (the program's state variables), a command table for the program, and a list of options specifying the command and screen interfaces for the program.

**dw:default-command-top-level** *program* &rest *options* &key (*window-wakeup* **#'dw::default-window-wakeup-handler**) &allow-other-keys
The default command loop function for programs created with **dw:define-program-framework**.

**dw:read-program-command** *program* &rest *options* &key *(:stream* ***query-io****)* *:prompt (:dispatch-mode* **:command-only***) :keyboard-accelerators :environment :window-wakeup :input-wait-handler :intercept-function* &allow-other-keys
Default command reading function for programs created via **dw:define-program-framework**.

**dw:define-program-command** *(name program-name* &rest *options* &key *(keyboard-accelerator* **nil***) (menu-accelerator* **nil***) (menu-level* **'(:top-level)***) (menu-documentation* **t***) menu-documentation-include-defaults provide-output-destination-keyword* &allow-other-keys*) arglist* &body *body*
Defines a Command Processor command named *name* for a program named *programname* created with **dw:define-program-framework** and installs it in the program's command table.

**dw:program-command-table** *program*
Returns the command table used by an instance of a program flavor (created via **dw:define-program-framework**).

**dw:command-error** &optional *format-string* &rest *format-args*
Used inside a **dw:define-program-command** form to signal an error.

**dw:define-subcommand-menu-handler** *menu-name command-table in-menu-level to-menu-level*
Defines a subcommand menu handler for an item in the command-menu pane of a program frame.

**dw:get-program-pane** *name* &key *(:if-does-not-exist* **:error***)*
Returns specified pane in a program frame created with **dw:define-program-framework**.

**dw:find-program-window** *program-name* &rest *make-window-options* &key *(create-p* **t***) (activate-p* **t***) (selected-ok* **t***) reuse-test console superior program-state-variables* &allow-other-keys
Returns the window (frame) of a program (created via **dw:define-program-framework**).

**dw:current-program** &key *window (type* **'dw::program***) (error-p* **t***)*
Returns the current program of the type specified by **:type** given the starting window specified by **:window**.

**dw:find-and-select-program-window** *name* &rest *options*
Returns the window (frame) of a program and selects that window.

**dw:*program-frame***
The program frame associated with the current instance of a program flavor (created via **dw:define-program-framework**).

**define-presentation-to-command-translator** *name (presentation-type* &key *:tester (:gesture* **:select***) :documentation :suppress-highlighting (:menu* **t***) :blank-area) arglist*

&body *body*
Defines a mouse handler that translates from a displayed presentation object into a Command Processor command using that object as input.

**cp:build-command** *command-name* &rest *command-arguments*
Constructs the internal representation of a Command Processor command.

### Creating Graphic Output

Graphic images are data, representing pictures, that can be sent to and received from streams. The Genera graphics output facilities provide an extensive set of operations that allow the user to create and manipulate graphic images for most types of output streams, including windows, files, and hardcopy streams. A basic graphics operation is generic: it produces appropriate results no matter what sort of stream it is called with. Genera also provides advanced graphics operations that operate efficiently on specific types of streams, for example, bitmap streams or raster graphics devices.

The graphics output facilities provide a complete imaging model. All output is done under a general transform that allows for scaling, rotating, and translating. Shapes can be drawn filled or unfilled, with or without an outline of arbitrary thickness. There is a uniform means of specifying texture attributes of the shape drawn, such as colors or stipple patterns.

In general, the mathematical shape of an object is speficied by the function used and its positional arguments. The details of how this shape is imaged are specified by keyword arguments.

The imaging model is compatible with modern graphics imaging standards, in particular PostScript and the X Window System. A graphic image, say for example, the image of a circle, can be defined in several different ways:

- As a discrete logical object, an entity that can be created, selected, deleted, or moved as a single item.

- As a bitmap, that is, an array that describes the pixel values of a rectangular region.

- As a geometrical entity, that is, as a set of points in a coordinate system that can be described mathematically.

- As a topological entity, separating the space it occupies into regions.

Genera graphics output facilities support all of these interpretations. The section, "Basic Graphic Output Facilities" explains the graphics coordinate system and graphics transforms. It also describes the set of basic drawing functions and their options, including pattern filling. The section "Advanced Graphic Output Facilities" describes advanced transformation facilities and graphics drivers.

### Text as Graphics

There are several different interpretations of text as graphical output. The interesting issues arise when the graphics containing the text is scaled or rotated, and when the text is mixed with other graphics.

There are several possible actions that might be taken when a graphics transformation is applied to text:

- Affect only the starting point of the text, keeping the actual letters upright.
- Affect the baseline of the text, slanting it, but keeping the letters upright.
- Actually scale and rotate the individual glyphs along with the rest of the drawing.

Since the various output devices available to the graphics substrate do not implement a consistent set of fonts, one must be careful when mixing text and graphics if the result is to be device- and scale-independent. There are two possibilities:

- Ask the stream the size of the text to be drawn and constrain the other graphics to that size.
- Use the kinds of text graphics drawing that allow scaling to a given size and specify that in accordance with the rest of the drawing.

### Mixing Graphics and Text

When you output several graphical objects one after another, the later objects appear "on top" of those displayed earlier. If you scroll away from and then back to such output, the redrawn display looks like the original because the temporal priority of each component of the drawing has been maintained. This maintanence of priority is done only for graphics output to a window. To keep basic textual output efficient, its ordering relative to any graphics is not preserved. Therefore, if you wish to output text that overlays some graphics and guarantee that it will redisplay properly when the window scrolls, you should use **graphics:draw-string** rather than **write-string**. This restriction does not apply to things that do not in fact overlap, such as is produced by **surrounding-output-with-border**.

The graphics output substrate is generic. In particular, this means that essentially the same program can be made to work on the LGP2/LGP3 and the screen. However, a common use of graphics is mixed in with a stream of textual output. This runs into problems because the character stream output protocol implemented by the hardcopy system is similar but not identical to that implemented by the screen.

These are the important differences that a program may need to be aware of if it is to work generically on the screen and the LGP2/LGP3. All of these differences predate the graphics substrate, so compatibility requirements prevent changing them.

1. The orientation of the initial default coordinate system is different. On a hardcopy stream, <0,0> is the lower-left corner and positive *y* proceeds up the page. On a window, <0,0> is the upper-left corner and positive *y* proceeds down the page.

2. The coordinate system units are incompatible. The screen implements units of **:pixel** and **:character**. The hardcopy system units of **:device**, **:pixel**, **:character**. The initial default unit for graphics on the screen is the same as pixels and for hardcopy the same as device units. Device units on the LGP2/LGP3 are not the same as pixels. There are about four pixels to a device unit, which is approximately one printer's point. The default unit for **:read-cursorpos** on the screen is **:pixel**. For a hardcopy stream, it is **:device**. Therefore, the default units used by cursor addressing and graphics are in fact the same. However, a program which is to work compatibly on both devices must not specify any units in order to get the proper defaults on each device.

3. The origin of the text cursor coordinate system is different. The current cursor position returned by **:read-cursorpos** to a window is relative to the top of the history, the same as graphics coordinate arguments. The **:read-cursorpos** value returned by a hardcopy stream is offset by the page margins, while graphics are not. To get the same coordinate system for the current text position as is used by graphics, use **:read-page-cursorpos**.

4. The interpretation of the current cursor position is different. The position returned for a window is the top of the current output line. For hardcopy streams, it is the position of the current line baseline.

The most realistic strategy for mixing text and graphics is to use the **graphics:with-room-for-graphics** special form. This makes a local graphics coordinate system available, which is oriented as a primary Cartesian quadrant (positive *y* up) and whose *y*=0 axis is positioned at a given position relative to the top of the current character output position. It works for both windows and LGP2/LGP3 streams. The best choice of units to use with graphics operations is a multiple of the stream's character width or line height.

*Example:*

```
(defun histograms (&optional (stream *standard-output*))
  (dotimes (ignore 10)
    (let ((n (random 100)))
      (format stream "~&~3D~10T" n)
      (let ((height (- (send stream :line-height) (send stream :vsp)))
            (width (* 10 (send stream :char-width))))
        (graphics:with-room-for-graphics
                   (stream height :fresh-line nil :move-cursor nil)
          (graphics:draw-rectangle 0 0 width height
                                                     :gray-level .15
                                                     :stream stream)
          (graphics:draw-rectangle 0 0 (* width (/ n 100)) height
                                                     :gray-level .75
                                                     :stream stream)
          (graphics:draw-rectangle 0 0 width height :filled nil
                                                    :stream stream))
        (send stream :increment-cursorpos width 0)
        (format stream " %~%")))))
```

## Basic Graphic Output Facilities

## Coordinate System Facilities

Coordinate System Facilities
- **graphics:with-room-for-graphics**
- **graphics:stream-transform**
- **graphics:with-graphics-translation**
- **graphics:with-graphics-scale**
- **graphics:with-graphics-rotation**
- **graphics:with-graphics-transform**
- **graphics:graphics-translate**
- **graphics:graphics-scale**
- **graphics:graphics-rotate**
- **graphics:graphics-transform**
- **graphics:*identity-transform***
- **graphics:with-graphics-identity-transform**

Keyword Options Affecting the Coordinate System
- **:rotation**
- **:scale**
- **:scale-x**
- **:scale-y**
- **:translation**
- **:transform**

Common to all types of graphics programming is the need to reconcile the coordinate system employed by the programmer and end user with the coordinate sys-

tems built in to particular graphics output devices. A coordinate system is a framework that allows the location of any point on a drawing to be specified, usually in terms of a horizontal and a vertical distance from a point that is defined to be <0, 0>, called the origin.

The initial coordinate system for the graphics functions is the device coordinate system. This system is not the same for all devices, so various facilities are provided for accessing it uniformly. The general graphics transformation facilities can be used to provide any scale that is convenient for a particular operation. Keep in mind that there is no restriction on the type of coordinate arguments. They can be integers, floats, or ratios.

The coordinate system of the LGP2/LGP3 is a single page, with <0,0> at the bottom. The coordinate system of a dynamic window is a plane with <0,0> at the top of the history, and some greater *y* value at the top of the currently visible screenful, and some greater still *y* value at the current cursor position. It is very useful to be able to deal with graphics as a single block interspersed with any text output already on the page or in the output history. Additionally, it is usual to want to deal with graphics within a primary Cartesian quadrant, so that <0,0> is at the lower left. The special form provided for this is **graphics:with-room-for-graphics**. Note that the 0 point of the quadrant is at the bottom of the local coordinate system, not the bottom of the page or screenful.

The initial scale for the LGP2/LGP3 is 1/72 inch, a "PostScript point." (The X Window System uses the printer's point, which is 0.01384 or about 1/72.27 inch.) There is more than one device pixel available within a single point. The initial scale of the screen is a single pixel. In order to draw a figure with a certain physical dimension, either inches/centimeters, or as a fraction of a page or number of lines on the device, the special form **graphics:with-physical-device-scale** is provided.

The *user* coordinate system usually has its origin in the lower left-hand corner of the drawing area. The horizontal reference line is called the *x* axis and its positive direction extends toward the right. The vertical reference is the *y* axis and its positive direction is up. The *x* and *y* coordinates can be specified as arbitrary numbers, either integers or floating-point numbers. When an image in user coordinates is displayed on a window without any scaling, the default unit distance along either axis is approximately 1/90 of an inch. See Figure !. All of the drawing functions in the graphics package expect their arguments to be specified in user coordinates.

A user normally prefers to see drawings displayed as they are drawn — in the user coordinate system. This is not, however, the coordinate system of the default display device, the Genera window, nor is it the system of every graphics hardcopy device. The coordinate system of a Genera window has its origin in the upper left-hand corner of the screen, and the positive *y* direction is down. Its basic units are *pixels*, whose size is device-dependent. In a dynamic Lisp Listener window that has just had its output history cleared, the origin is located at the upper left-hand corner of the prompt-character, as shown in Figure !. The location of this origin is relative to the window in which it appears, not to the screen as a whole. This is

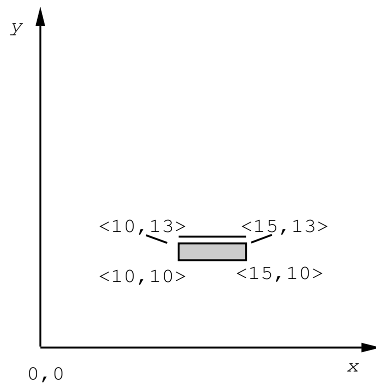



Figure 41. The user coordinate system

true of any window.

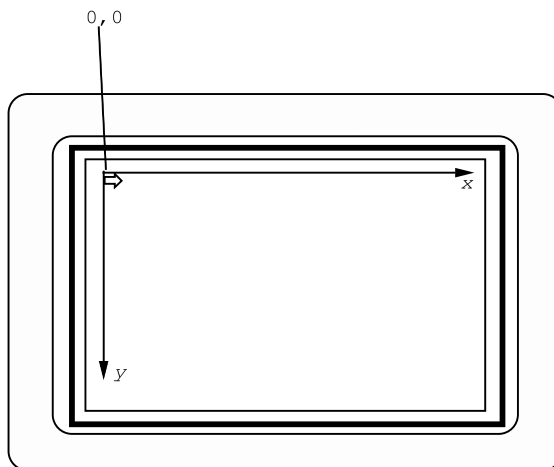



Figure 42. The device coordinate system of a dynamic window

In a dynamic window, scrolling horizontally or vertically moves the position of the origin with respect to the window. For example, the window in Figure ! has been scrolled both horizontally and vertically. Scrolling a dynamic window can be thought of as moving around a screen-sized opening that is over a plane that is unbounded on the right and toward the bottom. In the graphic system, such an opening is called a *viewport*.

In order to present a drawing specified in user coordinates on a display that has a different coordinate system, we need to apply a *coordinate transformation operator*. A coordinate transform operator maps one set of coordinates into another. Such an operator is specified by a mathematical object called a *transformation matrix*, which can specify three types of dimensional changes:

- *Translation*, in which the new coordinates differ from the old ones by some additive constant — the points of a drawing are translated some distance specified by a vector *<delta-x*, *delta-y>*.

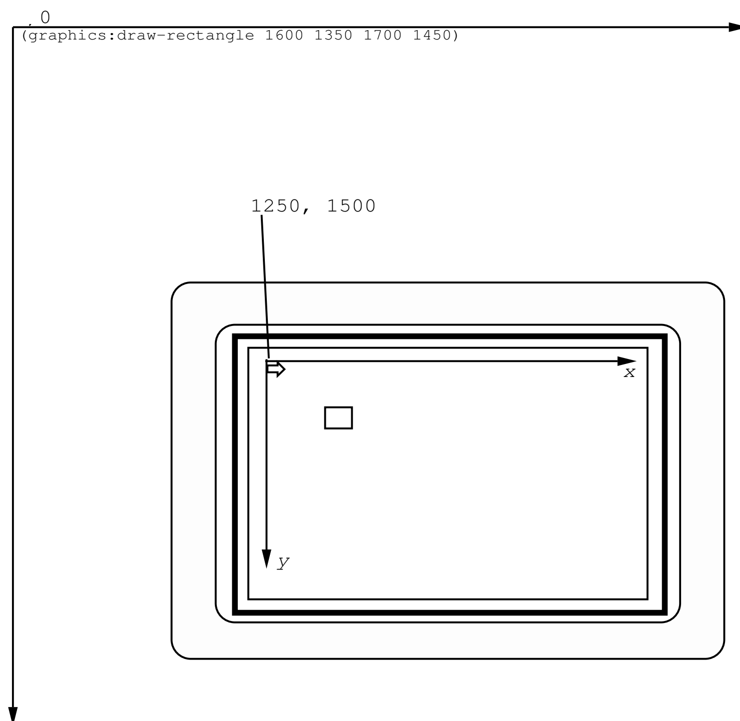



Figure 43. Device coordinate system, window scrolled vertically and horizontally

- *Rotation*, in which the new coordinates are located by rotating the old coordinates a specified angle *theta* about their origin.

- *Scaling*, in which the new coordinates differ from the old ones by some multiplicative constant — the *x* and *y* coordinates are each multiplied by some specified scale factor *scale-x* and *scale-y*. When the *x* and *y* scale factors are the same, the result is magnification or reduction. A negative scale factor reverses the direction of an axis.

There is a special transformation matrix, called the identity transform, which when applied to a set of coordinates makes no change; its result is the original set of coordinates. The essential part of this matrix, expressed as a list, is the value of the variable **graphics:*identity-transform***. The function **graphics:make-identity-transform** creates and returns this list.

There is more information on transformation matrices in another section. See the section "Advanced Transformation Facilities".

Every graphics stream has a transformation matrix associated with it. This is known as the stream's *current transformation matrix* or CTM. The default CTM is the identity transform. The method **graphics:stream-transform** returns the CTM of a stream. The graphics system or the user can change a stream's CTM in order to display a graphic image with the desired size and orientation. A few examples show how this is done.

Figure ! shows a drawing in user coordinates (on the left) and how it appears on a window when displayed without any transformation; that is, when the user coordinate points are mapped to device coordinates with the identity coordinate transform. The window in this case has been scrolled down so that the coordinates of the upper left-hand corner happen to be <0, 10>. The dotted lines indicate the location of the user's *x-y* coordinate frame in device coordinate space. Since we are using the identity mapping, the user coordinates (regular typeface) are the same as the device coordinates (boldface). The figure also shows a mouse cursor and its coordinates. The code to produce the drawing as shown is

```
(graphics:draw-rectangle 10 13 15 10 :filled nil)
(graphics:draw-line 10 15 15 15)
```

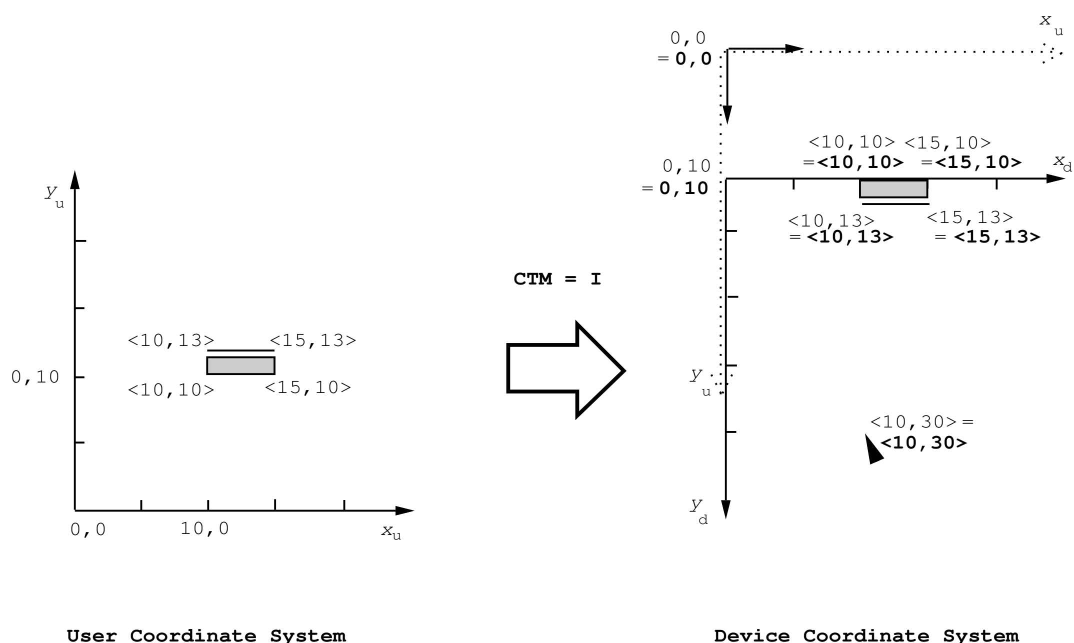



Figure 44. Display of user drawing with no transformation

The macro **graphics:with-room-for-graphics** enables the user to issue a drawing function in terms of user coordinates and have the result displayed on a window in the expected manner. Figure ! illustrates how this is done. **graphics:with-room-for-graphics** changes the window stream's CTM so as to *translate* the user origin downwards with respect to the device origin and then to *scale* the *y* coordinate by a factor of -1 so that the drawing comes out rightside up. The resulting location of the user *x-y* frame is shown in dotted lines. Note the relationship between the coordinates of the two systems. Also note that the window coordinates of the mouse cursor, shown in boldface, are not the same as its user coordinates.

```
(graphics:with-room-for-graphics (t 10)
   (graphics:draw-rectangle 10 13 15 10 :filled nil)
   (graphics:draw-line 10 15 15 15))
```

If you are writing a program that is to display images both on the screen and on

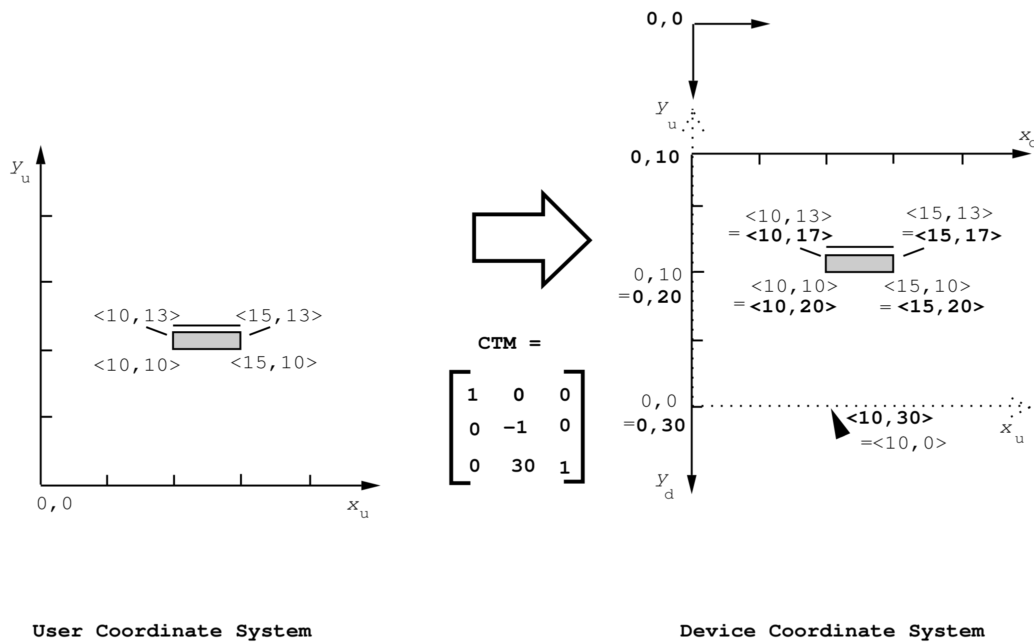



Figure 45. Use of **graphics:with-room-for-graphics**

an LGP2/LGP3, you should employ the user coordinate system and always use **graphics:with-room-for-graphics** when displaying on the screen to have it appear correctly oriented. Similarly, if you want to have your image on the screen offset, rotated, or magnified, you should use one or more of the graphics transformation macros **graphics:with-graphics-translation**, **graphics:with-graphics-rotation**, **graphics:with-graphics-scale**, or **graphics:with-graphics-transform**. Each of these macros has an effect similar to that of **graphics:with-room-for-graphics**, in that each modifies the current stream's CTM so that the specified transformation occurs.

Example:

```
(graphics:with-room-for-graphics (*standard-output* 330)
  (graphics:with-graphics-translation (*standard-output* 150 150)
    (graphics:draw-line 0 -150 0 150)
    (graphics:draw-line -150 0 150 0)
    (graphics:draw-rectangle 10 50 70 20 :filled t :gray-level 1/2)
    (graphics:with-graphics-rotation (*standard-output* (* 1/4 pi))
      (graphics:draw-line 0 -150 0 150)
      (graphics:draw-line -150 0 150 0)
      (graphics:draw-rectangle 10 50 70 20 :filled t :gray-level 1)
      (graphics:with-graphics-scale (*standard-output* 2)
        (graphics:draw-rectangle 10 50 70 20 :filled nil)))))
```

Figure ! shows the result. Note that successively nested transformations are performed *with respect to the coordinate systems resulting from the transformations effected by the outer macros.*

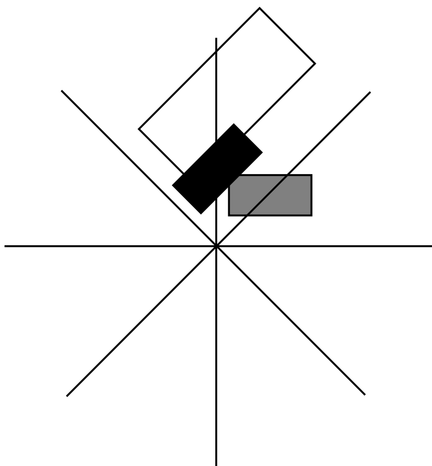


Figure 46. Use of nested graphics transformation macros

The initial scale for the LGP2/LGP3 is 1/72 inch, more or less a printer's point. There is more than one device pixel available within a single point. The initial scale of the screen is a single pixel. In order to draw a figure with a certain physical dimension, either inches/centimeters, or as a fraction of a page or number of lines on the device, the special form **graphics:with-physical-device-scale** is provided. Actually, this is less useful that fractions or multiples of character and line widths, though.

The general graphics transformation facilities can be used to provide any scale that is convenient for a particular operation. There is no restriction on the type of coordinate arguments. They can be integers, floats, or ratios.

All of the graphics *macros* that begin with **with-** affect the CTM of the current stream only within their environment. Once that environment is left behind, the stream's original CTM is restored.

There is a set of equivalent graphics transformation *functions* that begin with **graphics-** instead of **with-**, which do permanently affect the CTM of the stream on which they are called. A call to one of these functions with *stream* as an argument modifies the CTM of *stream* to be the composition of *stream*'s original CTM with the matrix that performs the change specified by the graphics function. These functions are intended to be used inside environments that have previously saved away the stream's CTM so that it can be restored. **graphics:graphics-translate**, **graphics:graphics-scale**, **graphics:graphics-rotate**, and **graphics:graphics-transform** are mainly intended for use with **graphics:draw-path**.

You can use the macro **graphics::with-identity-transform** in the case where you want to use a graphics translation function to draw on the screen without using **graphics:with-room-for-graphics**. Like the other **with-** macros, **graphics::with-identity-transform** preserves the stream's CTM. Unlike these macros, **graphics::with-identity-transform** ignores the previous value of the CTM.

**Keyword Options Affecting the Coordinate System**

Each of the graphics drawing functions affords a set of options for transforming the image the function produces. The transformations available are the same as those provided by the coordinate system transformation functions and macros: rotation, uniform scaling, translation, horizontal or vertical scaling, or general transformation.

These options to the graphics drawing functions affect the local coordinate system for the duration of the function; that is, they affect only the output of the functions with which they are used. The other coordinate system facilities make more lasting changes to the coordinate system of the output stream: See the section "Coordinate System Facilities".

There are six transformation keywords: **:rotation**, **:scale**, **:scale-x**, **:scale-y**, **:translation**, and **:transform**. When two or more of these are used in the same drawing function, the options are applied in the order given here, *not in the order you supply the keywords*. However, as a matter of style, we recommend that you only use one keyword at a time. More than one might confuse others who subsequently read your code.

In cases where you wish to apply more than one coordinate system change to the same drawing operation, use the facilities described in "Coordinate System Facilities". Doing so will make your intentions more obvious. Also, it gives you control over the order in which the coordinate system changes are applied. Another possibility is to use the **:transform** option.

Be especially careful when using any of these options within coordinate system macros such as **graphics:with-room-for-graphics**. For example, if you specify a negative value for **:scale-y** within this form, your image will not be drawn within the space provided by the macro, but will extend below it and be overwritten by the prompt.

### Drawing Functions

Functions for Drawing Objects

- **graphics:draw-arrow**
- **graphics:draw-bezier-curve**
- **graphics:draw-cubic-spline**
- **graphics:draw-conic-section**
- **graphics:draw-circle**
- **graphics:draw-ellipse**
- **graphics:draw-line**
- **graphics:draw-lines**
- **graphics:draw-point**
- **graphics:draw-polygon**
- **graphics:draw-regular-polygon**
- **graphics:draw-rectangle**
- **graphics:draw-triangle**
- **graphics:draw-glyph**
- **graphics:draw-image**

**graphics:draw-string**
**graphics:draw-string-image**

Functions for Drawing Paths
**graphics:draw-path**
**graphics:drawing-path**
**graphics:draw-bezier-curve-to**
**graphics:draw-circular-arc-to**
**graphics:draw-conic-section-to**
**graphics:draw-line-to**
**graphics:close-path**
**graphics:set-current-position**
**graphics:current-position**
**graphics:graphics-origin-to-current-position**

Clipping Functions and the Mask Option
**graphics:with-clipping-path**
**graphics:with-clipping-from-output**

The drawing functions offer straightforward means of drawing strings, points, arrows, lines, and a variety of closed plane figures. Many of these drawing functions have options in common. It is by means of these options that you can control the painting qualities — opacity, gray-levels, patterns — and outline characteristics of your figures. Before discussing the options, we describe some of the graphics objects that may not be well known to all users. After describing the options, we present a summary of the different types of drawing functions.

### Graphics Objects

A *Bezier curve* is a type of cubic parametric curve for which the endpoints are specified, while the tangents at the curve's endpoints are specified indirectly by two other points that are generally not on the curve. You can get an idea of how the control points affect the shape of a Bezier curve by looking at Figure !. Refer to any standard graphics textbook, such as *Fundamentals of Interactive Computer Graphics* by Foley and Van Dam (Addison-Wesley 1982), for the mathematical details of cubic parametric curves.

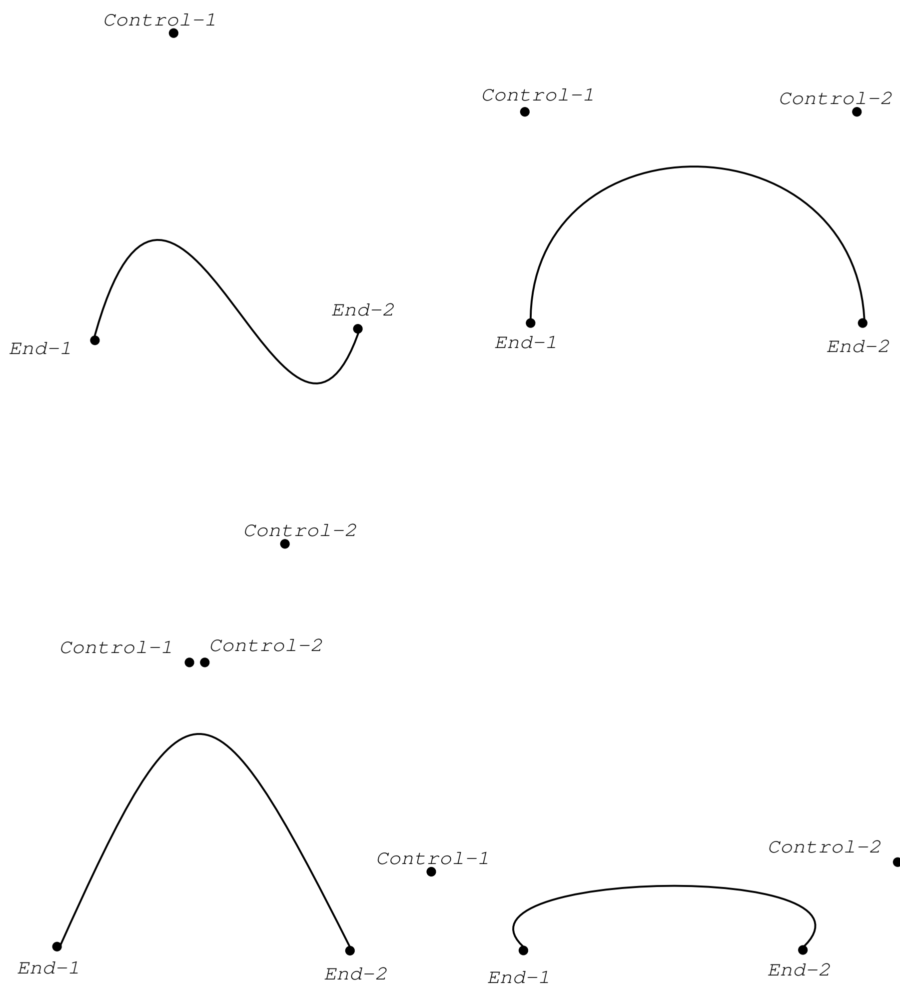



Figure 47.  Bezier curves and their control points

A *cubic spline* is another type of cubic parametric curve. For this curve, a set of points, including the endpoints, specifies the shape: the curve is constrained to pass through all the points such that the curve is "smooth"— that is, there is no sudden change in the slope of curve at any point.

Figure ! contains several examples of cubic splines. Compare Figure 47.

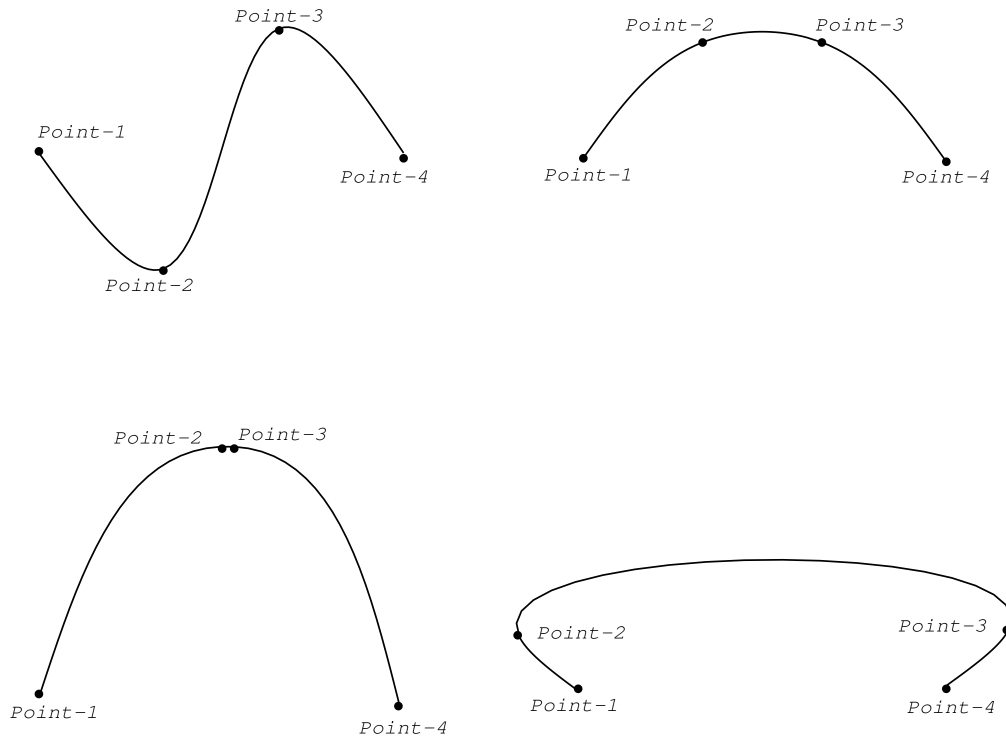



Figure 48. Cubic splines and their control points

The **graphics:draw-cubic-spline** function has several options, which are described completely in its dictionary entry. Here is a table that summarizes these options:

| | |
|---|---|
| The default option | If the user supplies no keyword arguments, but only a list of points, cubic splines like the ones in Figure 48 are the result. The slopes of the curves are constrained to change continuously everywhere. |

The clamped option

The keyword **:clamped**, accompanied by specifications for the starting and ending slopes, allows the user to draw a special kind of cubic curve, called a Hermite curve. The slopes at the endpoints of the curves are constrained to be those specified.

Figure ! shows a family of these cubics. The tangent vector at the right end of each curve is fixed at -90 degrees (*dx* = 0, *dy* = -1); at the left the angle is the value printed below the curve. All tangents were specified using unit *dx* and *dy* values. The curves in the figure are all clamped at both ends, but this need not be the case: you can specify that only the start or the end should be clamped.

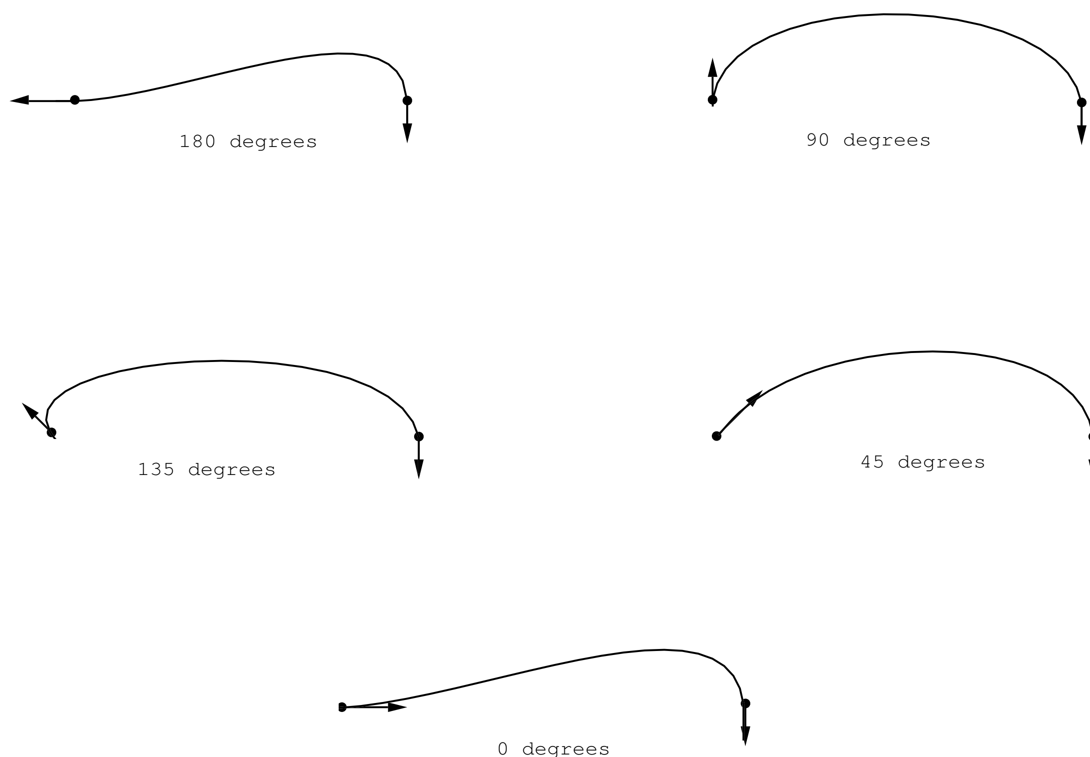



Figure 49. Hermite cubic splines

The cyclic or anti-cyclic option

This option is intended to be used for drawing closed curves that either close to form a smooth curve (cyclic) or that close in a cusp (anti-cyclic). The **:start-relaxation** and **:end-relaxation** must be specified identically for this option: either both **:cyclic** or both **:anti-cyclic**. Also, the first point and the last point in the list of control points must be identical.

A *glyph* is a single character from a specified font such as a timesroman (tr12) letter A or a northwest arrow from the mouse font. The **graphics:draw-glyph** function is primarily intended for drawing characters like the latter, that is, special symbols as opposed to regular characters for which **graphics:draw-string** would be more appropriate.

An *image* is a graphics image contained in a bit array. You can create such an array using, for example, **tv:with-output-to-bitmap**. The function **graphics:draw-image** allows you to send an image or part of an image to a stream.

A *string image* can be thought of as character outline shapes being transformed. In other words, you can think of a string image as being the result of putting a **graphics:draw-string** function inside a **tv:with-output-to-bitmap** form. **graphics:draw-string-image** allows you to draw a string image scaled or rotated as a unit, something that you cannot do with **graphics:draw-string**. See the section "Graphics Objects".

There are several different interpretations of text as graphical output. The interesting issues arise when the graphics containing the text is scaled or rotated, and when the text is mixed with other graphics.

There are several possible actions when applying a graphics transformation to text.

- Affect only the starting point of the text, keeping the actual letters upright.
- Affect the baseline of the text, slanting it, but keeping the letters upright.
- Actually scale and rotate the individual glyphs along with the rest of the drawing.

Since the various output devices available to the graphics substrate do not implement a consistent set of fonts, one must be careful when mixing text and graphics if it is to be device and scale independent. There are two possibilities:

- Ask the stream the size of the text to be drawn and constrain the other graphics to that size. For example, using the function **dw:continuation-output-size**.
- Use the kinds of text graphics drawing that allow scaling to a given size, that is, **graphics:draw-string-image**, and specify that in accordance with the rest of the drawing.

### Scan Conversion

The screen is a *raster* device. This means that it is drawn on by means of an array of single units called *pixels*, each of which represents the contents of a single cell of bitmap memory. The most important thing to keep in mind when trying to understand scan conversion is that these pixels are actually little squares that have finite extent: they are a single device unit on a side. Mathematical shapes are made up of points, which have no size. The business of scan conversion is to

take the infinitely thin outline of a mathematical shape and determine which pixels should be affected in order to draw it.

Generally, a pixel is affected by drawing a shape when it is inside that shape. Recalling that pixels are little squares of finite dimension, you can see that most of the time there will be a number of pixels that are only partially within the shape to be drawn. It is important that there by an exact decision procedure to determine which of these pixels to draw.

The definition used by windows is as follows:

> A pixel is inside a shape, and hence affected when drawing that shape, if the center of the pixel is inside the shape. If the center of the pixel lies exactly on the boundary of the shape, it is considered inside if the inside of the shape is immediately to the right of the center point (increasing $x$ direction). If the center of the pixel lies exactly on a horizontal boundary, it is considered inside if the inside of the shape is immediately below (increasing $y$ direction).
>
> An unfilled shape is drawn by generating new outlines consisting of those points which are within 1/2 the thickness (normal distance) from the outline curve of the corresponding filled shape, and filling in the new outlines by applying the definition above.

This definition is compatible with that used by the X Window System. Even though X windows does not deal with fractional coordinates and the Genera graphics substrate does, there is no problem in generalizing the definition to accomodate that case. It is important to note that the decision point used for insideness checking is offset from the point used for addressing the pixel by half a device unit in both the $x$ and $y$ directions.

It is worth mentioning that for pixel devices that can store or display a single pixel cell in more than one color or graytone, one usually attempts to draw a given pixel proportionately lighter or darker depending on the fraction of the pixel within the shape. This is known as *anti-aliasing*. The Genera graphics output substrate for windows does not support anti-aliasing, because the black-and-white screen is still the normal output device for Genera applications.

The resolution limitations of a raster device make scan conversion a hard problem, for which there are no easy universal solutions. For an application in which exactness is important, it is critical that the same means be applied for all shapes without exception. The definition used by windows is borrowed from the X Window System, extended in the most obvious way to allow for non-integral coordinates. Even though this definition is standard, it is worth considering some of the motivations.

When two shapes share a common edge, it is important that only one of them own any pixel. Mathematically speaking, given two figures whose intersection is of measure zero, the intersections of their rasterizations should be of measure zero. Fig-

ure ! illustrates this. The pixels along the diagonal belong to the lower figure. When the decision point within the pixel (the center) lies to one side of the line or the other there is no issue. When the boundary passes through the decision point, which side the inside of the figure is on is used to decide.

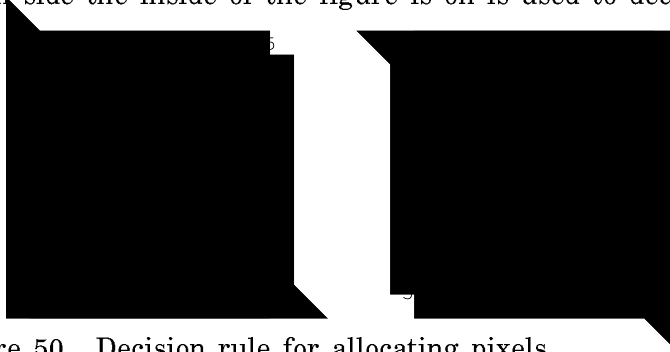
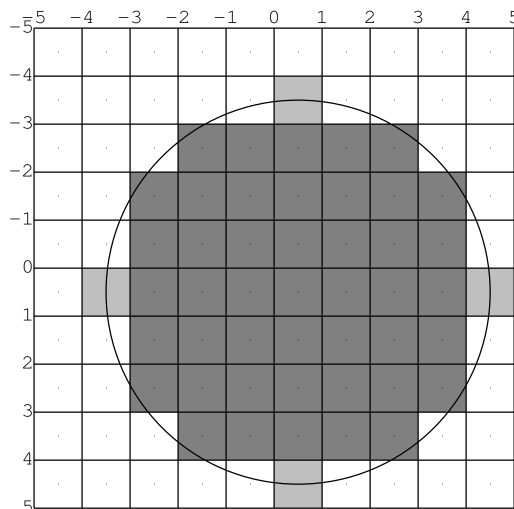


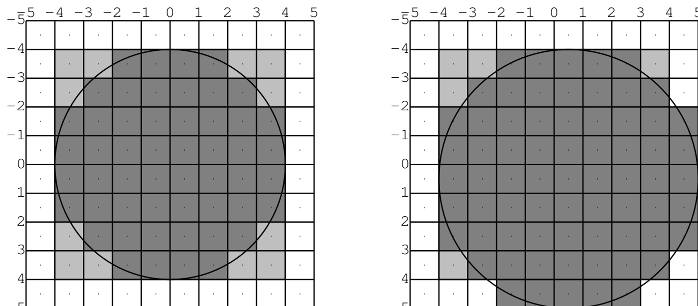
Figure 50. Decision rule for allocating pixels

The reason for choosing the decision point half a pixel offset from the address point is to reduce the number of common figures that invoke the boundary condition rule. This leads to more symmetrical results. For instance, in the figure below, we see a circle drawn when the decision point is the same as the address point. The four lighter points are indeterminate: it is not clear whether they are inside or outside. Since we desire to have each boundary case determined according to which side has the figure on it, and since we must apply the same rule uniformly for all figures, we have no choice but to pick only two of the four points, leading to a lopsided figure, which is clearly undesirable.

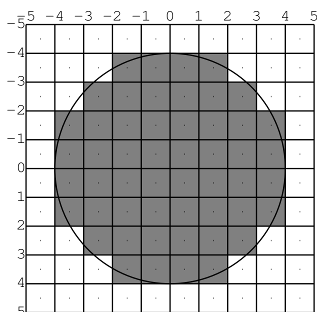



If we had instead chosen to take all four points, as the **:draw-filled-in-circle** method does, we would have a nice symmetrical figure. This figure is symmetrical about a whole pixel, however, so it is one pixel wider than it ought to be. The problem with this can be seen clearly if we attempt to draw a rectangle and circle overlaid.

```
(defun shape (r)
  (graphics:draw-circle 0 0 r)
  (graphics:draw-rectangle (- r) (- r) (+ r) (+ r) :alu :flip))
```

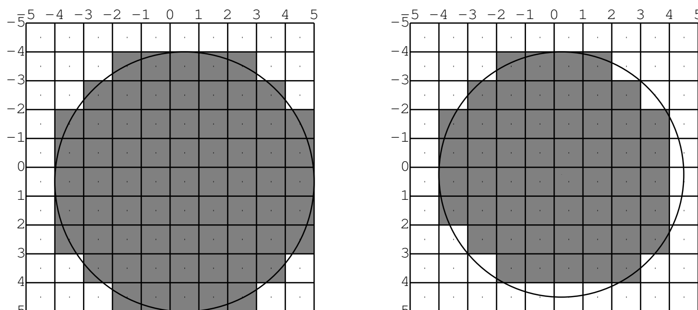



For this reason, we choose to have the decision point at the center of the circle. This draws circles that look like the figure below.




The problem with these circles is that they look rather more like stop-signs than the ones the old **:draw-circle** produced. This is due to the eye picking up on the straight lines before it picks up on the rotational symmetry. We sometimes prefer something that is indeed centered around a whole pixel. Once we understand the rules for scan conversion, it is easy to see how to draw this figure. We want a circle that is of radius $r$+1/2 (since we have one more pixel in the middle to account for) and whose center is at <1/2,1/2> (the point in the middle of the square about which we want to be symmetrical). Since the graphics functions are not restricted to integer coordinates, this is easy to do.

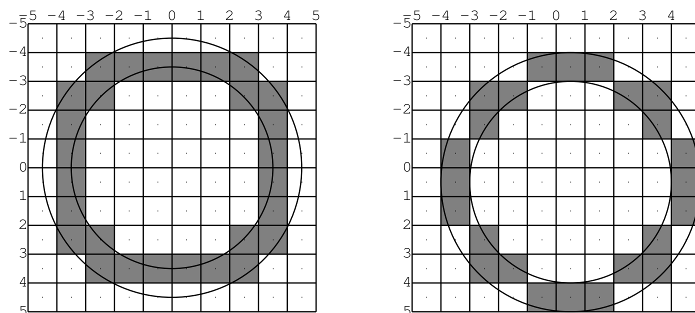
Since this is a rather common case, we provide a special drawing mode to enable this. This is **:coordinate-mode :center**. You can draw such circles either way.



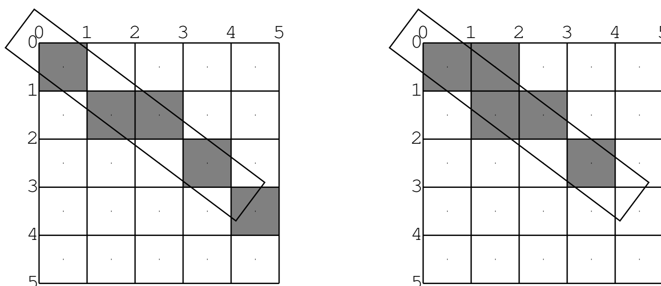
Despite the well-controlled symmetry of the above shapes, there is really no way to guarantee that all circles look round when scan converted. For instance, one whose coordinates are offset by one quarter unit does not look very round. For this reason, it is often desirable to make sure that coordinates and radii are integers if they come from some potentially inexact computation and you want the figure to look round.

Since this is also a rather common case, you can disable the exact conversion of fractional shapes with **:coordinate-mode :integer**.

Unfilled circles follow exactly the same rules as filled circles. An unfilled circle is essentially a circular ring of the desired thickness with the circle in question lying in the middle of the ring.

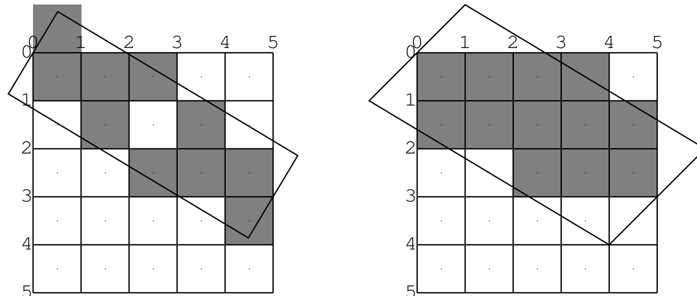



We distinguish lines of thickness 0, which are drawn by a special fast integer slope algorithm, from those of thickness 1, which are exactly drawn as tilted rectangles. For the majority of cases, these produce results that are similar enough for most purposes.

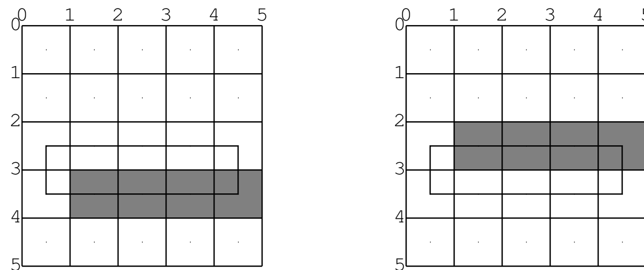



For thick lines, we can draw the exact tilted fractional rectangle, or we can round the coordinates of the rectangle so it becomes a polygon with integral coordinates and draw that. The latter case is faster since it can make full use of the triangle microcode. For the majority of cases, these two methods produce results that are usually similar enough.

The decision about which side of the figure to take when a boundary line passes through the decision point is made arbitrarily. We have chosen to be compatible with the X Window System definition, although this is not necessarily the most convenient. The main problem with this is illustrated by the case of a horizontal line. Our definition chooses to draw the rectangular slice above the coordinates, since those pixels are the ones whose centers have the figure immediately above




them. It would be more convenient if we could recognize that a line from <1,3> to <4,3> is the same as the primitive device rectangle of height 1 and width 4 at <1,3>.




We have attempted to choose a primitive imaging definition that is convenient for lower level control. We have then tried to add higher level facilities on top of it that capture some of the more common convenient interfaces. These modes are not the only possible modes that could be added. For instance, we could choose a model which did not attempt to tile correctly and always selected pixels that had any of the figure at all in them. Fortunately, the framework of the **graphics:** functions permits adding this modularly at a later time if the need can be shown and the details worked out.

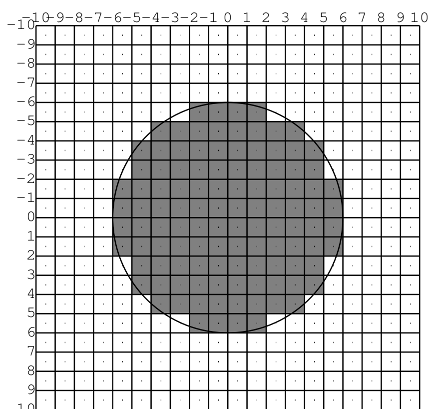
We have chosen to make our definition compatible with the X Window System to permit future migration to a graphics system based on that system at the lowest level. Since X is designed for a healthy combination of generality and efficiency, it is hoped that this would lead to superior performance and generality, while remaining essentially compatible.

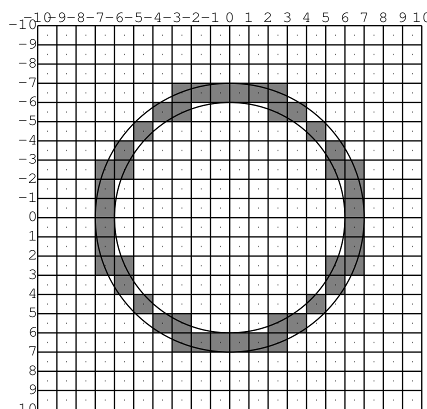
It would also be possible to define all operations in terms of a primitive path filling operation. This would take a shape as a set of continuous segments, such as lines, conic sections, and cubics. It would then scan convert the area within the segments, using standard techniques. This is essentially the primitive operation defined for the PostScript imaging system. This would require much less special-case code, but would not be as well optimized. Such a system could benefit, however, from the aid of a graphics coprocessor or special assistance microcode.

### Outlining with Thin Lines

Drawing a shape with **:thickness 0** is not the same as outlining it. For instance, take a circle of radius 6 and consider it and its unfilled counterpart. These two shapes overlap in a few pixels because of the overlap of the mathematical shapes.

Keeping in mind the scan conversion model, it is not hard to generate an outline for this shape. You just draw an unfilled circle of radius 6.5 and thickness 1 with the same center. This shape is a ring with inner radius 6 and outer radius 7, which is outlines the filled circle.





Other simple shapes can be treated similarly. For more complicated shapes, there is no simple representation of the outline based on the filled or unfilled shape. A wholly new mode just for outlining is required. This has not yet been implemented. There are, then, some shapes for which there is no convenient way to generate a closely surrounding and similarly shaped mouse-sensitivity box.

**Drawing Function Options**

The options available for drawing graphics objects consist of those for specifying:

- The type of line used for the object or its outline
- The type of pattern filling the interior of closed objects
- How the drawing is to interact with objects that it might be drawn on top of
- The target stream
- Whether the object is to be a dynamic-windows presentation
- A masking pattern for the object

Another set of options specify transformations to be performed on the object. These are described in another section. See the section "Coordinate System Facilities".

**Line Options**

You can specify the following characteristics of a line:

Line thickness

The **:thickness** option specifies the line thickness in an integral number of device line-width units (for windows, pixels). A value of 0 specifies the minimum line thickness for the output device used. This option only takes effect when you are drawing line objects (lines, arrows, curves) or *unfilled* closed objects (circles, rectangles, and the like). The default is device-dependent: for the screen, it is 1; for the LGP2/LGP3, it is 0. Note that when thickness is greater than 1, the coordinates specifying the location of the object refer to the center points of the line ends. If you want a closed object with a thick outline to fit within a specified space, remember to subtract out the line thickness from the dimensions you specify.

When thickness is specified to be greater than one, two additional options become available:

Line joint shape

The **:line-joint-shape** option specifies how the joints between line segments of closed, unfilled figures, or sequences of line segments are to be drawn. The default, **:miter**, produces the commonly expected result. **:round** produces the same effect as drawing the figure with a sequence of thick lines with their end shapes specified to be **:round**. Note that this produces roundness on the outside of figures but not on the inside. **:bevel** produces a result like **:miter**, except that the corners of the object are trimmed: Each trimming line is perpendicular to the bisector of the angle formed by intersecting lines of the figure and is located halfway between the line centers and the corner they form. **:none** produces figures with gaps at the corners if line thickness is greater than one.

Thickness scaling

Ordinarily, when you scale up an object that has lines in it, the thickness of the lines is scaled up. You can change this, however, by setting the **:scale-thickness** option to **nil**.

Dashed lines

The lines that make up objects are, by default, solid. You can specify dashed lines using the Boolean option **:dashed**. Having chosen the dashed option, you can also specify four characteristics of the dashes:

Line dash pattern

With the keyword **:dash-pattern** you can specify in a vector the number of pixels to

be on and off. (Whether a pixel that is on is white or black depends on whether the image is displayed in reverse video.) For example, the vector #(5 2 2 2) produces a dash-dot-dash-dot pattern. The exact number of pixels drawn depends on whether and how the dash pattern is scaled. The first number in the dash pattern vector always specifies the number of pixels in the solid — that is, "on" — portion of the dash; if you want the dash pattern to start with "off" pixels, use the **:initial-dash-phase** keyword described below.

Scaled dashes

Ordinarily, when a line, or object containing lines, is scaled, the dash pattern is *not* scaled: Lines are drawn longer or shorter, but the number of pixels in the solid and open parts remains constant. You can change this by specifying **:scale-dashes** to be **t**, which causes the number of pixels to be multiplied by the current scale factor. Note that this scale factor is the combination of any such factor specified by the **:scale** keyword, by **graphics:with-graphics-scale**, by **graphics:with-graphics-transform**, or the like.

Initial dash phase

The **:initial-dash-phase** keyword specifies a number of pixels to be skipped over (left off) from the beginning of a dashed line. The default value is 0: The dashed line starts off with solid or "on" pixels. Keep in mind that when you are drawing lines within transforming macros such as **graphics:with-room-for-graphics**, the beginning of the line may differ from what you expect.

Drawing partial dashes

When the **:draw-partial-dashes** option is true (**t**, the default), the on and off pattern of the dashes is repeated as many times as is necessary to complete the line: if the pattern does not come out even with the line's end, a partial pattern is drawn. This could have the effect, for example, of "leaving out" the corners of a polygon. The alternative is to specify **:draw-partial-dashes nil**. Compare the following two forms:

```
(graphics:with-room-for-graphics (t 200)
(graphics:draw-regular-polygon 40 40 100 40 5
:filled nil :dashed t :draw-partial-dashes nil))

(graphics:with-room-for-graphics (t 200)
(graphics:draw-regular-polygon 40 40 100 40 5
:filled nil :dashed t :draw-partial-dashes t))
```

The exact behavior of the **:draw-partial-dashes nil** option is rather complicated and depends upon the dash pattern. Experimentation is probably the best way to obtain the results you want.

### Pattern Options

There are several options for specifying how the interiors of graphics images are to be drawn.

Filling

The **:filled** keyword specifies when **t** (the default) that figures are to be drawn filled — that is, the region within the drawing path is to be filled in — and when **nil** that figures are to be drawn "stroked" — that is, just the drawing path itself is to be drawn, with the specified width. (Whether this is black or white depends on whether the sense of drawing on the graphics output device is normal or inverted.)

Selecting a gray-level

The **:gray-level** keyword allows you to specify a system-defined gray level in a range between solid white (0) and solid black (1), to be drawn within the figure (when the drawing sense is normal — when the drawing sense is inverted, white is 1 and black 0). This option allows a more complete range of possibilities for gray levels than does the next one. Stippling
The **:stipple** keyword allows you to specify a stipple pattern to be drawn within the figure. A stipple pattern is a two-dimensional array of one-bit values, 0 or 1 for each pixel. On a color screen, ones are drawing in one color and zeros in a different color. On a monochrome screen one "color" is white and the other black. A stipple pattern always aligns with the coordinate origin used.

You can use one of a series of standard background gray patterns of the form **tv:n%-gray** with this option, but for this use, **:gray-level** is preferred. Likewise, you can specify a predefined stipple array, a tile, or a color, but, again, if you want one of these you should probably be specifying **:stipple**, **:tile**, or **:color**.

You can create your own bitmap pattern for use with **:stipple** by drawing an image inside a **tv:with-output-to-bitmap** macro, or you can use one of the standard bitmap patterns supplied by the system. See the option **:stipple**. Tiling
The **:tile** keyword allows you to specify a colored-stipple pattern — that is, a *tile* pattern — within a figure. A tile pattern is a two-dimensional array of $n$-bit values, which specify colors: each of the $n$ bits controls a color. To tile a figure means to perform a cell-to-cell transfer from the tile pattern to the figure. Like stipple patterns, a tile pattern aligns with the coordinate origin used. Tiling and stippling are similar on one-bit-per-pixel devices that do not have special alu functions. See the option **:tile**.

Coloring

The **:color** option allows you to specify a color within a figure. If the output device used supports color, the specified color will be used; if not, the object will be filled with a gray-level pattern that approximates the intensity of the color specified. See the option **:color**.

Pattern filling

The **:pattern** keyword allows you to specify any bitmap pattern to be drawn within the figure. This allows you to, in effect, paint some image on the interior of the figure. According to how you specifiy **:pattern**, you can cause the way the pattern is produced to be conditionally device dependent.

Lastly, you can specify a flavor instance as the pattern. The methods of the flavor of the instance specify how the drawing is to be done.

See the option **:pattern**.

The pattern options **:gray-level**, **:stipple**, **:tile**, **:color**, and **:pattern** interact with each other: If you specify more than one, the effect is the result of "anding" the on pixels resulting from each, in the general case.

The options **:filled** and **:thickness** are mutually exclusive; you cannot have a figure that is filled and that has some non-zero thickness. If you specify a non-zero thickness together with any of the other pattern options, the results are combined: The thick line is drawn with the specified gray-level, or color.

### Drawing Mode Options

The keyword **:alu** specifies the drawing mode for drawing any kind of graphics object. That is, it specifies how the drawing of an image is to affect other images that may already be present in the target location. The effect of **:alu** is nullified by setting the keyword **:opaque** to **t**: when **:opaque** is **t**, the pixels in the target area are cleared (turned off) before the image is drawn. In this case, the only alu option that makes sense is **:draw**, since there will not be any pixels in the target to erase or flip. Use the **:opaque** option when you want to draw one image on top

of another and not have the two images interact, or when you want to draw one image over another and not have the first show through. Use the **:alu** modes with **:opaque nil** when you want the images to combine with one another.

The possible values of **:alu** are **:draw**, **:erase**, and **:flip**.

**:draw** does the obvious thing: Pixels in the image being drawn are turned on, regardless of what the pixel values may be in the target area, that is, in the place where the image is being drawn. **:erase** is a bit more complicated. Pixels that are included in the image being drawn (ones specified to be "on" by **:filled :pattern**, **:stipple**, **:tile**, or **:gray-level**) are turned off in the target area. For example, if you draw a filled, 100% black rectangle in **:draw** mode and then draw a patterned circle over it in **:erase** mode, the result is a circular pattern in white inside the black rectangle. If you draw that patterned, erased circle over a white area, nothing appears.

The **:flip** keyword allows you to draw in an exclusive-or (XOR) mode: If a pixel in the image being drawn is on and the corresponding pixel in the target area is off, the pixel is drawn (turned on); if the image pixel is on and the target pixel is on, the pixel is "un"-drawn (turned off). Pixels that are not on in the image being drawn are not affected. The result is that the drawn image is rendered normally when drawn over a white area and rendered in reverse over a solid black area. (If the screen you are drawing on is in reverse video mode, this description is still true with the words "black" and "white" interchanged.)

### Other Options

The **:stream** option for drawing functions specifies a stream for the drawing function. It defaults to ***standard-output***.

The **:return-presentation** option specifies whether the drawing function should return the graphic image it creates as a presentation object. The default is **nil**; no type and object of the presentation is returned. Keep in mind however, that whether or not the image is returned as a presentation, it is recorded in the output history unless you specifically take steps to prevent this. **:return-presentation t** is for presenting a single graphics object: If you want a presentation to comprise a collection of images, use **graphics:with-output-as-graphics-presentation**. For more information on graphics images as presentation objects: See the section "Other Basic Facilities for Graphic Output".

The **:mask** option is presented in another section: See the section "Clipping Functions and the Mask Option".

### Functions for Drawing Objects

The graphics output facility has functions for drawing the following types of objects:

| | | |
|---|---|---|
| arrows | triangles* | Bezier curves |

| | | |
|---|---|---|
| circles* | polygons* | cubic splines |
| ellipses* | regular polygons* | conic sections |
| lines | rectangles* | glyphs |
| points | strings | images |
| | | string images |

* *closed figures*

All of these functions have options that are specified by keywords. See the section "Drawing Function Options". There is another collection of graphics functions for drawing fillable outlines of arbitrary shape. See the section "Path-Drawing Functions". Any of the objects drawn with one of these can be returned as a graphics presentation. See the section "Other Basic Facilities for Graphic Output". The objects listed in the rightmost column above, are described in detail in another section: See the section "Graphics Objects".

The functions for drawing closed figures, marked with an asterisk above, have options for filling: **:filled**, **:pattern**, **:gray-level**, **:stipple**, **:tile**, and **:color**.

Circles and ellipses can be drawn either clockwise or counterclockwise, which is necessary because of the way the winding rule works (see the section "Path-Drawing Functions"). The complete figure can be drawn, or you can specify a beginning and ending angle for drawing an arc. Also, you can draw circular or elliptical rings by specifying inner radii. The ellipse drawing function has an additional option, **:join-to-path** that lets you specify that the ellipse or elliptical arc be joined to a path properly (that is, with no gap).

Polygons, whose successive sides are specified by a list of their endpoints, can be convex, in which case a more efficient drawing algorithm is used to draw them. Regular polygons are specified by the endpoints of a line segment that is to be one of the congruent sides. A keyword, **:handedness**, allows you to specify whether the polygon is drawn to the left of the given segment or to the right.

There is a function for drawing a single line and another for drawing a connected series of lines. The latter has a **:join-to-path** option similar to that of **graphics:draw-ellipse**. In addition, it has an option, **:closed**, that lets you specify all but the last segment of a closed polygon; the **graphics:draw-lines** function with **:closed t** adds the closing line segment for you.

The function for drawing a string allows you to specify the character style, the position of the string in relation to the placement point you give the function, and a vector indicating the direction in which the string is to be drawn. This vector can be thought of as a line along which the characters of the string are to be placed. You can also specify whether the characters of the string are to be stretched out or squeezed together. Note that you *cannot* specify any graphics translations of the actual string image; the translation options of the keywords of the **graphics:draw-string** function apply only to the string position you specify, not to the characters in the string. Use the **graphics:draw-string-image** function to perform transformations on the string itself.

**Path-Drawing Functions**

You can draw an arbitrary closed path using the graphics functions for drawing ordinary objects — just connect the ending point of one object with the beginning point of the next. This is, in fact, how you create a path-function argument for the path-drawing function.

The path-drawing function **graphics:draw-path** and the macro **graphics:drawing-path** allow you to do two things with closed paths that you could not do otherwise: make the closed path fillable and make it usable for clipping. See the section "Clipping Functions and the Mask Option". The path-drawing function **graphics:draw-path** has an option, **:winding-rule**, that allows you to specify how the region outlined by the path is to be filled. **graphics:drawing-path** allows you to specify a path by sequentially calling several drawing functions, and it collects and returns any values produced by the functions called.

There is an object called the *graphics cursor* that is of use in drawing paths. The graphics cursor, which is completely unrelated to the window cursor or mouse position, is simply an *x-y* coordinate pair that you can use as a positional reference. You set its value with **graphics:set-current-position** and use (refer to) it with **graphics:current-position**. You can perform a local transformation to make it the current origin for graphics transformations with **graphics:graphics-origin-to-current-position**. Four drawing functions make use of the graphics cursor: **graphics:draw-bezier-curve-to**, **graphics:draw-circular-arc-to**, **graphics:draw-line-to**, and **graphics:close-path**. The first three functions draw their figure starting at the current position of the graphics cursor, and then move that cursor to the endpoint of the drawn figure.

The **graphics:set-current-position** function not only fixes the location of the graphics cursor, but also marks the start of a "path segment." All the lines or curves drawn subsequent to a **graphics:set-current-position** belong to the "current path segment" until another **graphics:set-current-position** is issued. **graphics:close-path** finishes a closed path by connecting the current graphics cursor position to the beginning of the path segment, with a straight line.

**graphics:graphics-origin-to-current-position** is useful for specifying transformations with respect to the current position of the graphics cursor. For an example of its use: See the function **graphics:draw-path**. The referenced section also presents examples of how to construct path functions and presents an example of the effect of the **:winding-rule** option.

### Clipping Functions and the Mask Option

There are three methods you can use to restrict the output of graphical objects to be within a limited region:

**graphics:with-clipping-path** clips the output produced by forms contained within the macro so that it lies within the specified path. The path is created by means of graphics path-drawing functions.

**graphics:with-clipping-from-output** clips the output produced by forms within the macro so that it lies within a specified figure. The figure is created by one of the graphics functions for drawing two-dimensional figures such as circles and rectangles.

The **:mask** option allows you to specify a bitmap to serve as a mask through which graphical output is drawn.

## Other Basic Facilities for Graphic Output

Other Basic Facilities for Graphic Output
**graphics:with-output-as-graphics-presentation**
**graphics:replacing-graphics-presentation**
**graphics:erase-graphics-presentation**
**graphics:erase-rectangle**
**graphics:2pi**

### Graphics as Presentations

Just as a piece of text displayed on a dynamic window can be a presentation, so can any graphic output. To "present" a figure as a presentation, either you include the form that generates it inside a **graphics:with-output-as-graphics-presentation** form or you use the **:return-presentation** keyword of the drawing function. **graphics:with-output-as-graphics-presentation** is the graphical equivalent of the **dw:with-output-as-presentation** form for text.

Once you have presented a figure as a presentation, you can **accept** the figure as input just as any other presentation; the system automatically highlights the figure in the proper input context. Also, once you have a figure that is a presentation, you can use **graphics:replacing-graphics-presentation** to produce the effect of animation by successively replacing a presentation at incrementally changing locations, and you can use **graphics:erase-graphics-presentation** both to delete the presentation and to erase the figure.

### Erasing versus Deleting versus Painting White

There are three different ways of getting rid of some graphics that you have drawn, each of which has its place. It is important to understand the difference and to know when a particular one is right for you.

1. Erasing, as with the **graphics:erase-rectangle** function, or the **:clear-region**, **:clear-rest-of-window**, or **:clear-history** messages to a window, means clearing a portion of the output history, removing anything that is in it. This is the fastest way to get rid of something that you have output. However, it is important to keep in mind that things are only removed from the history in single units. If you erase a region that intersects a graphical object, that object will be removed from the output history, even if it is not wholly contained in the region. For reasons of efficiency, only the area you requested to clear will be erased right away. Only when you scroll away from an area and back again will you notice that a partially erased graphic has been wholly removed from the history.

2. Deleting, as with **graphics:erase-graphics-presentation**, means removing something from the output history and redisplaying everything that overlaps

it. Because proper temporary ordering of output is remembered as part of the history, when you delete something, it is possible to completely reconstruct what would appear on the screen as if you had never drawn it.

3. Painting in white means adding a new graphic to the output history on top of what is already there that obscures what is underneath it. If the shape in question does not completely cover something underneath, then the combined shape will display properly even when scrolled back.

### Graphics and the Stream Output

When an erasing, hyphen, or delete type of operation, such as **:clear-rest-of-window** or **:clear-between-cursorposes**, encounters a piece of graphics, it removes the piece from the history. It does not compute the corresponding several graphics that would actually result from erasing pieces of the whole.

If you forget to move the cursor past some graphics that you draw in a window, the **:clear-rest-of-line** that happens after the command loop returns to command level and does a **terpri** will erase part of it. It may look mostly okay right then (only a little bit has been nicked out), but it will be gone completely when you scroll back.

*Examples: Wrong:*

```
(dw:with-own-coordinates ()
  (graphics:draw-circle 100 200 100))
```

*Better:*

```
(dw:with-own-coordinates ()
  (graphics:draw-circle 100 200 100)
  (send *standard-output* :set-cursorpos 0 310))
```

*Generally better still:*

```
(graphics:with-room-for-graphics ()
  (graphics:draw-circle 100 100 100))
```

### Handy Constant

The constant **graphics:2pi** is supplied for use in generating the angle arguments to functions such as **graphics:draw-circle**. It is a double-precision floating-point approximation of the circumference-to-radius ratio.

## Choosing the Best Graphic Output Technique for Your Application

There are a number of things to be considered when choosing how to use the graphics substrate.

- If you already have an application using the **:draw-** messages, it will continue to work. Especially if you have had to go to lots of trouble to determine the exact pixel locations and offsets of various figures, you may not have any motivation for changing it. If you want your drawing to work compatibly on the LGP2/LGP3 and any future similar device which is supported, then you must

use the new **graphics:draw-** facilities. You must also keep in mind the coordinate system issues. See the section "Coordinate System Facilities".

Note that the LGP2/LGP3 has high enough resolution that exact pixel choices are usually unimportant, and that Symbolics has no control over the speed of the Adobe code that runs in it, so the remaining decision points covered here apply to the screen only.

- If you are just doing a simple drawing, such as a pie chart, you probably do not need to think about any further refinement.

- If it is important that figures tile correctly, you should set the value of the keyword **:coordinate-mode** to **:exact**, or better yet do not use this keyword, since it defaults to **:exact**. Similarly, if you require that shapes with fractional coordinates be drawn properly rather than rounded to integer shapes, set **:coordinate-mode** to **:exact**.

- If you want small circles which appear symmetrical about a single pixel dot and do not need to carefully align them with other shapes, you can use **:coordinate-mode :center**.

- If you do need to align round circles with other shapes, you are probably best off adding in the 1/2-pixel offsets yourself so that you will add them exactly where needed.

- If speed is more important than tiling accuracy in sketching a figure, drawing with **:thickness 0** is the faster way.

- If speed is of moderate importance in drawing a filled figure or one with thick lines, and exactness is of little importance, then **:coordinate-mode :integer** will always use special integer drawing methods, which are faster. Keep in mind that it is better to do this by means of a single **graphics:with-coordinate-mode** around all the output than to do it in each drawing function call, which requires more argument decoding.

- If speed is of paramount importance, you will save time in some cases by using the old **:draw-** messages. Of course, the application will have to take responsibility of conversion to integers, or errors will result, and no scaling or output to the LGP2/LGP3 is available. As this is only a small gain over the fastest case of the **graphics:draw-** functions, it is also possible that even this technique will not meet your needs adequately.

### Graphics Output Performance

There are many considerations affecting the performance of a graphics output program. Ways of drawing that do not differ substantially in their specification (and hence in what they look like), can differ significantly in how they perform.

Some of the factors affecting speed of output are:

- The modularity level at which your program interfaces.
  - The **:draw-rectangle** method on a window generates an error at a rather embarrassing time when passed floating point or ratio coordinates. The **graphics:draw-rectangle** function has no such restriction. Even when passed integer coordinates, this function has to verify that they are integers. When drawing lots of rectangles, this moderately small amount of time can still be noticeable.
- The algorithm used for drawing.
  - A **:thickness** parameter of 0 is interpreted by the graphics substrate to mean not a line that is too thin to ever be seen, but the thinnest line which does indeed display visibly. This interpretation is compatible with PostScript and the X Window System. Additionally, as with X windows, a thin line (one of thickness 0) is allowed to be drawn using a faster algorithm which does not obey the complete requirements for raster scan conversion. A line of thickness 1 device unit, which is approximately as thick, must be scan converted exactly, which entails a much slower algorithm.

    For more details on this, see the section "Scan Conversion".
  - A circle whose center and radius are integers or integers plus 1/2 can be imaged without using any fractional arithmetic. Circles that have fractional parameters need a slower, more general algorithm.
- How the algorithm is implemented.
  - Because of compatibility considerations, it was not possible to change any of the microcoded drawing routines or to add new ones. As a result, some drawing methods are in macrocode and other have greater microcode assistance.

## Advanced Graphic Output Facilities

## Advanced Transformation Facilities

Advanced Transformation Facilities

**graphics:transform-point**
**graphics:untransform-point**
**graphics:transform-window-points**
**graphics:untransform-window-points**
**graphics:compose-transforms**

**graphics::stream-compose-transforms**
**graphics:decompose-transform**
**graphics::build-transform**
**graphics:transform-distance**
**graphics:untransform-distance**
**graphics:invert-transform**

## The Transformation Matrix

Arbitrary transformation of coordinates is effected by multiplication of coordinate vectors by a transformation matrix. For coordinates in two-dimensional space, a homogeneous vector <*x y* 1> is used, and the 3 x 3 transformation matrix is premultiplied by the point vector to produce the transformed coordinates of the point.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} = \begin{bmatrix} ax + cy + e & bx + dy + f & 1 \end{bmatrix}$$

of which the elements in the third row are constant. In the Genera graphics substrate, such a transformation matrix is stored in list format, that is, as the list (*a b c d e f*). The six elements of the matrix effectively control the transformation as follows:

- **Scaling** in the *x* and *y* dimensions is controlled by elements *a* and *d*, respectively. In the absence of rotation, these are the respective scale factors, and values of `1` for these elements result in no scaling.

- **Translation** in the *x* and *y* dimensions is controlled by elements *e* and *f*, respectively. Values of `0` for these elements result in no translation.

- **Rotation** about the origin is controlled by elements *a*, *b*, *c*, and *d*. In the absence of scaling, counterclockwise rotation by an angle *theta* is effected by *a* = cos *theta*, *b* = sin *theta*, *c* = -sin *theta*, and *d* = cos *theta*. A value of `0` for *b* and *c* results in no rotation (*theta* = 0).

The graphics data structure *transform* is a list of these six elements, in order *a*, *b*, *c*, *d*, *e*, *f*.

To further understand the relationship between the scaling and rotation elements and how various transforms interact, consider the following example:

Here is a transformation matrix for a ninety-degree counterclockwise rotation:

$$\mathbf{R} = \begin{bmatrix} \cos(\pi/2)=0 & \sin(\pi/2)=1 & 0 \\ \sin(\pi/2)=-1 & \cos(\pi/2)=0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And here is one for a scaling by a factor of 1.5 in the $x$ direction and a factor of 2 in the $y$ direction:

$$\mathbf{S} = \begin{bmatrix} \text{scale-x=1.5} & 0 & 0 \\ 0 & \text{scale-y=2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Application of **R** transforms the *coordinate system* so that a point $p$ at location <10,10> in the original coordinate system, when redisplayed in the transformed coordinate system, appears at location <-10,10> *with respect to the unchanged coordinates.* Similarly, **S** transforms the same point so that it appears at <15,20>.

Figure ! shows these transformations. The figure shows the transformation of a line so that the changes in orientation are apparent. The original coordinate-system is shown with dotted lines and the transformed coordinate-system in solid lines.
Once a transformation has occurred, subsequent transformations occur *with respect to the transformed coordinates,* as Figure ! illustrates.

The upper right coordinate system shows the location of point $p$ resulting from the scaling done in Figure 51 , followed by a ninety-degree counterclockwise rotation. Note that the new coordinates of the point, <-20,15>, still with reference to the original coordinate system, are the result of *first rotating* point <10,10> into point <-10,10> and *then scaling up* to <-20,15>. (The order of transformation of the original coordinate system, shown by dotted axes, is the opposite: first scaling to the coordinates shown by dashed axes, then rotating to the coordinates shown by solid axes.) The lower right coordinate system shows the result of taking the transforms in the opposite order. Note that the results are not the same as the first case.

Graphics transformations are not, in general, commutative. The only cases that are order-independent are (1) successive translations, (2) successive rotations, (3) successive scalings, and (4) rotations and scalings *if the x and y scale factors are equal*.

The coordinate transform matrices of Figure 52 are computed by forming the products of those in Figure 51, in the order shown. In general, transform matrices are composed by multiplying an original transform by the new transform; that is, the new transform is on the left. This is how **graphics:compose-transforms** works: Its first argument is replaced with the product of its second argument times its first argument. The composed matrix effects the same transformation as first performing new-transform and then performing transform.

**graphics::stream-compose-transforms** is similar to **graphics:compose-transforms**, but always uses the CTM of its stream argument as the original transform to be composed with a given new transform. Note that you should never use a transform as the first argument to **graphics:compose-transforms** if you need to keep that transform unaltered. If you need the composition, make a copy of your original and compose with that.

You can specify your own transforms by creating a list of specified elements, or you can construct one using the **graphics:build-graphics-transform** function. For

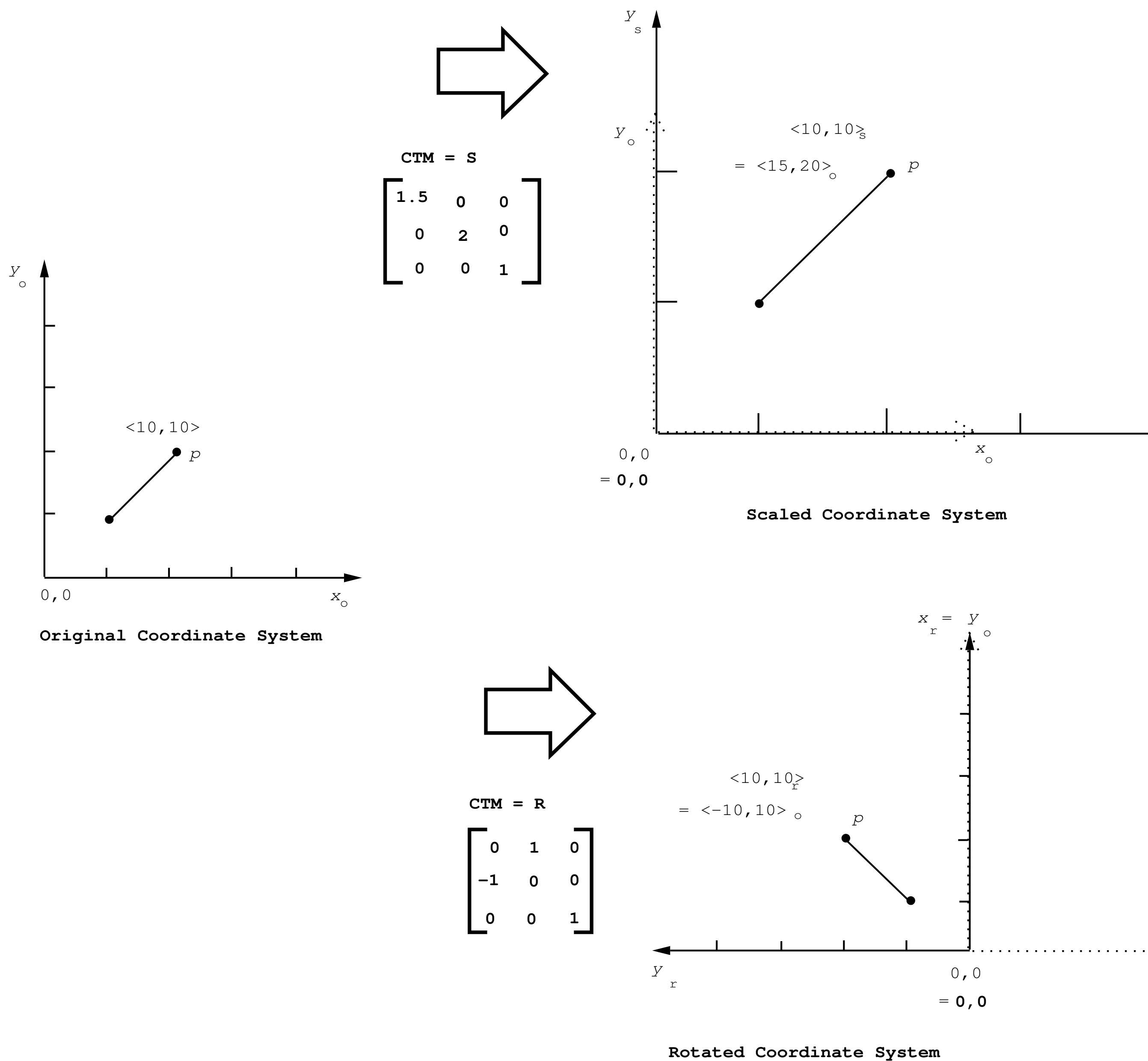



Figure 51. Changes effected by single coordinate transforms

example:

```
(graphics:build-graphics-transform :rotation (* 1/2 pi)
:scale-x 2 :scale-y 3 :translation '(150 90))
```

This form returns a list that is, in simplified form, '(0 2 -3 0 150 90), which corresponds to the CTM:

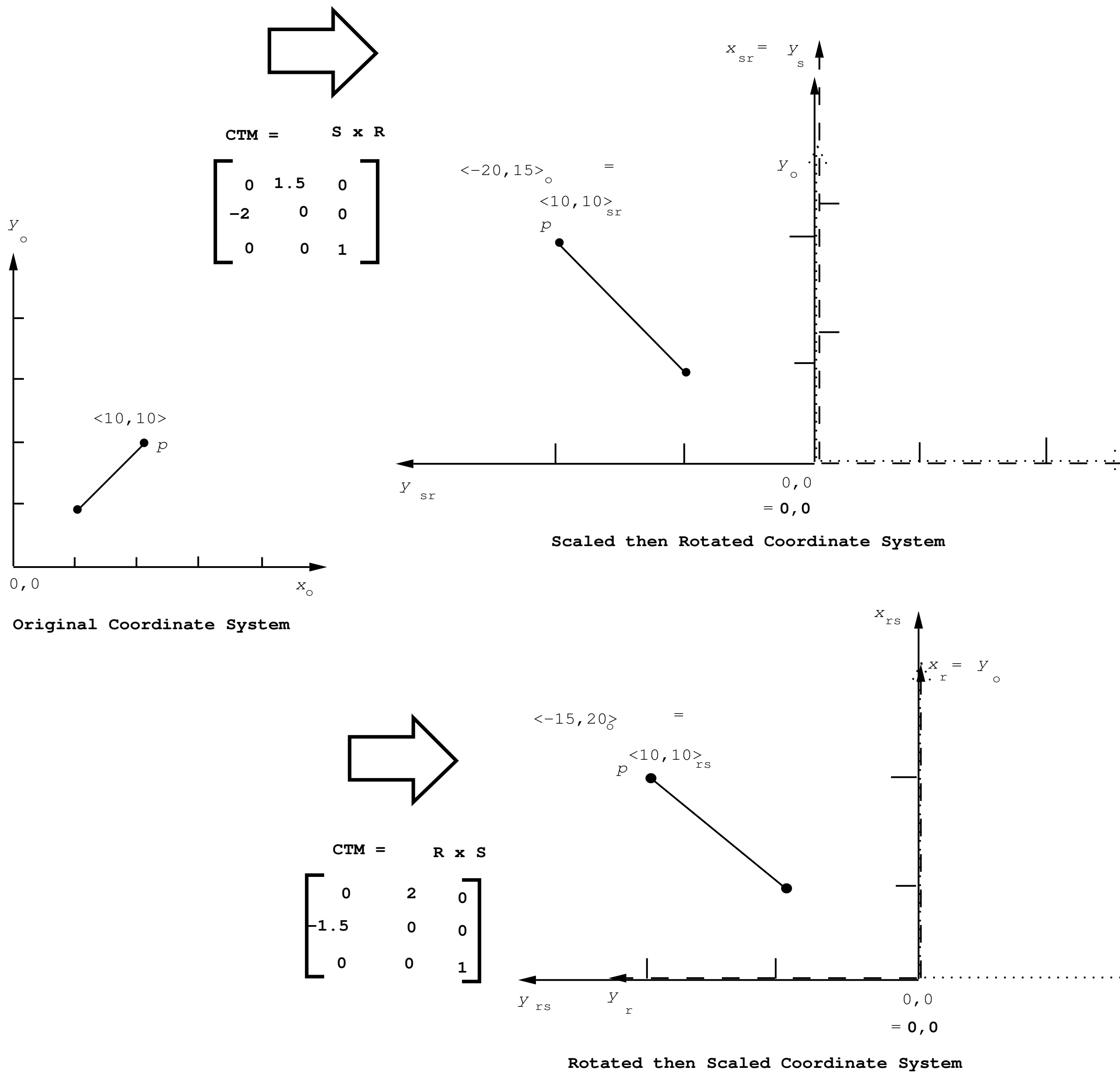



Figure 52. Changes effected by sequential coordinate transforms

$$\underset{\mathbf{S}}{\begin{bmatrix} \text{scale-x=2} & 0 & 0 \\ 0 & \text{scale-y=3} & 0 \\ 0 & 0 & 1 \end{bmatrix}} * \underset{\mathbf{R}}{\begin{bmatrix} \cos(\pi/2)=0 & \sin(\pi/2)=1 & 0 \\ \sin(\pi/2)=-1 & \cos(\pi/2)=0 & 0 \\ 0 & 0 & 1 \end{bmatrix}} * \underset{\mathbf{T}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ e=150 & f=90 & 1 \end{bmatrix}} = \begin{bmatrix} 0 & 2 & 0 \\ -3 & 0 & 0 \\ 150 & 90 & 1 \end{bmatrix}$$

The resulting transformation is a result of composing coordinate system transforms for scaling, rotation, and then translation, in that order. The effect of the resulting transform on the coordinates of any given point is first to scale the coordinates by $x$ and $y$ factors of 2 and 3, then rotate by 90 degrees (1/2 pi radians) counterclockwise, and finally to translate the point's coordinates (by +150 $x$ units and +90 $y$ units). You can verify this by using the list above as the argument to **graphics:decompose-transform**, which will return 1.570796 (radians), 2, 3, 150, and 90.

Nesting graphics **with-graphics-xxx** macros within one another causes the CTM of the given stream to be composed with each succeeding inner transform, so that the coordinate system is changed, in order, from the outside in — the innermost transform is the last one performed on the coordinate system. See figure 45.

The function **graphics:invert-transform** returns a transform that is the multiplicative inverse of its argument; when the inverse transform is composed with the original transform, the result is the identity transform. If a point $p$ has been transformed by matrix **T**, applying the inverse of **T** returns $p$ to its original coordinates.

### Transforming Points in Window Coordinates

If you wish to use the mouse cursor to position graphics images drawn by functions within transformation macros, you must transform the device coordinates obtained from the mouse, by **dw:tracking-mouse** for example, into user coordinates before you use them in drawing commands. For example, in the following code, **graphics:transform-window-points** adjusts the $x$-$y$ coordinates returned by the mouse so that they are transformed in same way as the other points drawn inside **graphics:with-room-for-graphics**. For better results, press the REFRESH key before running this example.

```
(graphics:with-room-for-graphics (*standard-output* 200)
  (dw:tracking-mouse ()
    (:mouse-motion (x y)
     (graphics:transform-window-points *standard-output* x y)
     (graphics:draw-rectangle x y (+ x 10) (+ y 10)))
    (:mouse-click (button x y)
     (return (graphics:draw-rectangle 10 50 70 20)))))
```

**graphics:transform-window-points** performs the same function within any other environment that causes offsets between the stream and its associated window, such as one caused by **dw:with-own-coordinates** or **dw:in-sub-window**.

To obtain the resulting coordinates of any given point, given the application of a specified transform, you can use the function **graphics:transform-point**. Given a point that has been transformed, you can use **graphics:untransform-point** to find its original coordinates. The function **graphics:stream-transform-point** does the same thing as **graphics:transform-point** but uses as its transform the CTM of the specified stream. **graphics:stream-untransform-point** is likewise analogous to **graphics:untransform-point**.

The functions **graphics:transform-distance** and **graphics:untransform-distance** are analogous to **graphics:transform-point** and **graphics:untransform-point**, but they do not perform any translation; they just scale and rotate the given coordinates according to the transform specified. These functions are useful for converting vector data (for example, the unit vector) into user coordinates.

### Graphics Drivers

The graphics drivers are low-level routines each of which draws a rectangular slice of the figure or path for its associated shape — circle, triangle, and so forth — using a specified slice function to do the actual drawing. The routines scan-convert the shapes, that is, they convert the coordinates of the pixels that lie on the shape into a two-dimensional raster grid. The set of available drivers includes

**graphics:draw-line-driver**
**graphics:draw-triangle-driver**
**graphics:draw-ellipse-driver**
**graphics:draw-unfilled-ellipse-driver**
**graphics:draw-unfilled-circle-driver**
**graphics:draw-circular-ring-driver**

You can use these routines to write device-specific drivers for your own graphics output devices.

### Texturing

There are three layered protocols for implementing one's own instances to be passed as **:pattern**.

All pattern instances should be built upon **graphics:basic-pattern**, which has only the required method, **graphics:pattern-call-with-drawing-parameters**.

This protocol is device-independent, in that the same generic function is called by all graphical output devices. If the pattern wishes to gain further device-dependent control, it should implement **graphics:pattern-call-with-drawing-parameters** so as to call its *function* argument with **:pattern self**. That is, it should pass in an instance for the pattern once more. This is most easily done by including the flavor **graphics:device-pattern**, which is built on **graphics:basic-pattern** and implements **graphics:pattern-call-with-drawing-parameters** in just this way.

The protocol(s) that the instance must implement then depend on the desired output devices, currently the LGP2 (and other postscript devices) and the screen (and other raster devices).

For the LGP2, the generic function **lgp:pattern-output-postscript-code** will be invoked. If the pattern is being used at this level unconditionally, it is best to get this by including the flavor **lgp:postscript-device-pattern**, which is built on **graphics:device-pattern**, and has this as a required method.

For the screen, the generic function **graphics:pattern-compute-raster-source-pattern** is invoked. If the pattern is being used at this level unconditionally, it is best to get this by including the flavor **graphics:raster-device-pattern**, which is built on **graphics:device-pattern**, and has this as a required method.

If even more control is desired of a raster device like the screen, the **graphics:pattern-compute-raster-source-pattern** implementation can return an instance (such as **self**) for *updated-source*. The instance is then invoked for the **graphics:pattern-draw-raster-slice** generic function. If the pattern is being used at this level, it is best to get this by including the flavor **graphics:raster-slice-device-pattern**, which is built on **graphics:raster-device-pattern** and has this as a required method.

Note that it is perfectly legitimate to include both **graphics:raster-slice-device-pattern** and **lgp:postscript-device-pattern** in the same flavor, as these require two separate protocol implementations, which can both be accommodated.

### Other Advanced Facilities for Graphic Output

These are the advanced facilities for graphic output:

**graphics:compute-cubic-spline**
**graphics:draw-circular-arc-to-compute-points**
**graphics:map-points**

**graphics:make-raster-array-with-correct-width**
**tv:with-output-to-bitmap**
**tv:with-output-to-bitmap-stream**

**graphics:write-encoded-graphics-as-characters**
**graphics:read-encoded-graphics-as-characters**
**graphics:binary-encode-graphics-to-array**
**graphics:binary-decode-graphics-from-array-into-function**
**graphics:with-graphics-subroutine**

The first three are primitive operations that compute sets of points or call a function to operate on a set of points.

The next three are useful for creating raster arrays or bitmaps for use as stipple patterns or masks.

The graphics encoding/decoding functions are useful for saving the output to a graphics stream in a compacted form. For example, the illustrations in Concordia have been created by the Graphic Editor and encoded in this manner.

### Table of Graphics Facilities

Since the names of the graphics functions are, for the most part, self-explanatory, they are simply listed here without description.

**graphics:2pi**

**graphics:angle-between-angles-p**

**graphics:basic-pattern**

**graphics:binary-decode-graphics-from-array-into-function**

**graphics:binary-encode-graphics-to-array**

**graphics:build-graphics-transform**

**graphics:build-multiple-point-transform**

**graphics:building-graphics-transform**

**graphics:close-path**

**graphics:compose-transforms**

**graphics:compute-cubic-spline**

**graphics:compute-cubic-spline-points**

**graphics:current-position**

**graphics:decompose-transform**

**graphics:defstipple**

**graphics:device-pattern**

**graphics:draw-arrow**

**graphics:draw-bezier-curve**

**graphics:draw-bezier-curve-to**

**graphics:draw-circle**

**graphics:draw-circle-driver**

**graphics:draw-circular-arc-through-point-to**

**graphics:draw-circular-arc-to**

**graphics:draw-circular-arc-to-compute-points**

**graphics:draw-circular-ring-driver**

**graphics:draw-conic-section**

**graphics:draw-conic-section-to**

**graphics:draw-cubic-spline**

**graphics:draw-ellipse**

**graphics:draw-ellipse-driver**

**graphics:draw-elliptical-ring-driver**

**graphics:draw-glyph**

**graphics:draw-image**

**graphics:draw-line**

**graphics:draw-line-driver**

**graphics:draw-line-to**

**graphics:draw-lines**

**graphics:draw-oval**

**graphics:draw-path**

**graphics:draw-pattern**

**graphics:draw-point**

**graphics:draw-polygon**

**graphics:draw-rectangle**

**graphics:draw-regular-polygon**

**graphics:draw-string**

**graphics:draw-string-image**

**graphics:draw-triangle**

**graphics:draw-triangle-driver**

**graphics:draw-unfilled-circle-driver**

**graphics:draw-unfilled-ellipse-driver**

**graphics:drawing-path**

**graphics:erase-graphics-presentation**

**graphics:erase-rectangle**

**graphics:graphics-origin-to-current-position**
**graphics:graphics-rotate**

**graphics:graphics-scale**

**graphics:graphics-stream-p**

**graphics:graphics-transform**

**graphics:graphics-translate**

**graphics:gray-level-stipple**

**graphics:*identity-transform***

**graphics:invert-transform**

**graphics:line-intersection**

**graphics:make-contrasting-pattern**

**graphics:make-device-conditional-pattern**

**graphics:make-graphics-transform**

**graphics:make-identity-transform**

**graphics:make-raster-array-with-correct-width**

**graphics:make-simple-pattern**

**graphics:make-two-color-stipple**

**graphics:map-points**

**graphics:pattern-call-with-drawing-parameters**

**graphics:pattern-compute-raster-source-pattern**

**graphics:pattern-draw-raster-slice**

**lgp:pattern-output-postscript-code**

**graphics:*pattern-stipple-arrays***

**lgp:postscript-device-pattern**

**graphics:raster-graphics-mixin**

**graphics:read-encoded-graphics-as-characters**

**graphics:replacing-graphics-presentation**

**graphics:saving-graphics-transform**

**scale-float**

**graphics:sector-wide-p**

**graphics:set-current-position**

**graphics:standard-graphics-mixin**

**graphics:*stipple-arrays***

**graphics:stream-transform**

**graphics:stream-transform-point**

**graphics:stream-untransform-point**

**graphics:transform-distance**

**graphics:transform-point**

**graphics:untransform-distance**

**graphics:untransform-point**

**graphics:untransform-window-points**

**graphics:with-clipping-from-output**

**graphics:with-clipping-mask**

**graphics:with-clipping-path**

**graphics:with-coordinate-mode**

**graphics:with-drawing-state**

**graphics:with-graphics-identity-transform**

**graphics:with-graphics-rotation**

**graphics:with-graphics-scale**

**graphics:with-graphics-subroutine**

**graphics:with-graphics-transform**

**graphics:with-graphics-translation**

**graphics:with-output-as-graphics-presentation**

**graphics:with-physical-device-scale**

**graphics:with-room-for-graphics**

**graphics:write-encoded-graphics-as-characters**

## Augmenting the Top-Level Tools: Extending User Interface Features

## Defining Your Own Presentation Types

### Introduction: More Presentation-Type Concepts

In an earlier chapter (see the section "Using Presentation Types") we introduced presentation types and explained how to use the extensive set of predefined types available with Genera. In this chapter, we describe presentation types in greater detail so that you can write your own.

### Why Define Your Own Presentation Types

### Presentation Types for Command Arguments

A straightforward case in which you need to define a presentation type is when you simply want to make a command — either a CP command or an application program command — out of an existing function. Commands require that the presentation types of their arguments be specified, and often the function you wish to

make a command of takes an argument whose type is not predefined. So, you must define an appropriate presentation type.

Here is an example. To define a CP command to draw a standard-size rectangle with a specified stipple pattern, we need a presentation type for stipple patterns. We can define such a thing as follows:

```
(define-presentation-type pattern-stipple ()
   :expander 'graphics:stipple-array
   :parser ((stream &key original-type)
            (dw:complete-from-sequence
              graphics:*pattern-stipple-arrays* stream
              :type original-type
              :name-key #'graphics:stipple-array-name))
   :printer ((stipple stream)
             (write-string
               (graphics:stipple-array-name stipple) stream)))
```

The CP command is then easy:

```
(cp:define-command (com-draw-rectangle :command-table "user")
    ((stipple 'pattern-stipple))
   (graphics:with-room-for-graphics (t 50)
     (graphics:draw-rectangle 0 20 40 0 :stipple stipple)))
```

### A Hierarchy of Presentation Types for Application Programs

When you are defining a program framework for an application, you often need to define a number of presentation types for objects in your program that a user will want to manipulate. In a graphics program, these might be graphics objects, as well as aspects of graphics objects — lines, rectangles, circles, as well as line-thickness, scale factor, and fill pattern. In some cases, these types are to be arranged in a hierarchy, so your presentation-type definitions need to specify sub- and supertype relationships, in addition to specifying how to recognize an object of the type, how to print one, and how to display information about one.

For example, consider a simple graphic editor that uses the presentation types shown in Figure !. Defined properly, an object of one of these types will be mouse-sensitive in appropriate contexts. For example, in the context of a Fill Closed Figure command, circles and polygons — including rectangles and triangles — should be sensitive, but lines and text should not. Additionally, when the user enters a Draw Figure command and asks for help, the system should display a message such as "Specify a figure, such as a line, a text-string, or a closed figure," or "A line, text-string, circle, rectangle, or triangle."

In summary, you define your own presentation types to:

- Control mouse sensitivity of objects within a set of given input contexts. This entails establishing the placement of objects within a type hierarchy.

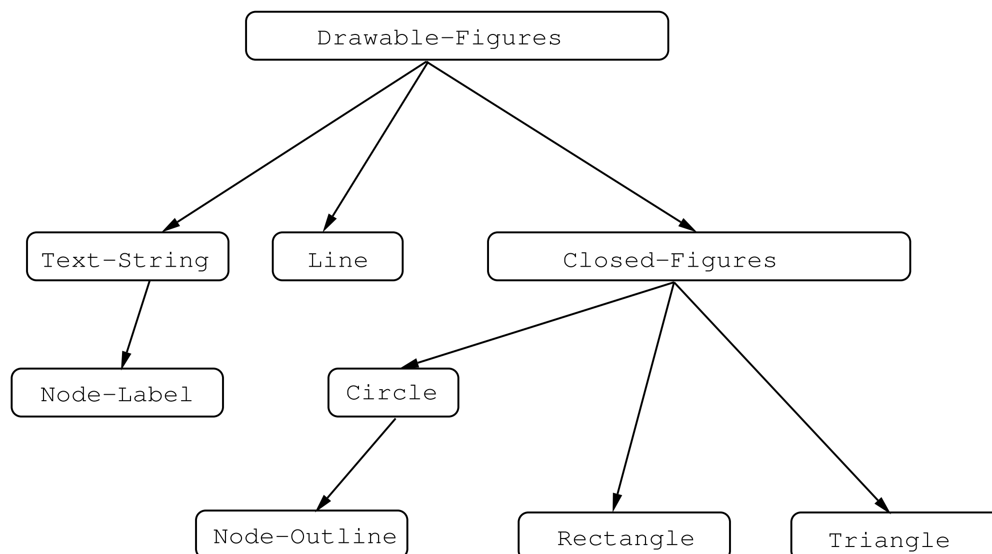



Figure 53. A collection of hierarchically related presentation types for a simple graphic editor

- Differentiate between objects that have similar syntax and implementations, but different meanings.

- Tailor the appearance of and help facilities for objects of different types.

A presentation type has associated with it a collection of specifications describing:

- Membership in the type.

- How to parse input when accepting, including how to prompt the user and how to use the input history for the type, if it has any.

- How to display output when presenting an object of the type.

- How the user can control the viewspecs of the presentation, that is, how alternative ways of displaying the presention can be selected.

The **define-presentation-type** macro sets up all these specifications for you when you use it to define your own presentation type. In order to use the defining macro, you need to understand the purpose of each major part of a presentation type description and the syntax for specifying it. The following sections explain each of these in turn. First, however, you need to know how presentation types can inherit various aspects from one another, because this inheritance determines exactly what your definitions must specify.

## Type Inheritance

As we mentioned in "Using Presentation Types", the types in the Genera presentation system constitute a partial ordering — that is, a hierarchy defined by the subset relationship. In a manner analogous to that of the flavor system, presentation types inherit attributes from their superiors in the type hierarchy. At the top of the hierarchy is the type **t**, which is a supertype of all other presentation types. As explained in the "Dictionary of Predefined Presentation Types", the type **t** is used for specifying "any type" when you are defining a mouse handler but, because of its universality, it is too general to have useful inheritable aspects; it has no parser.

The next highest type in the hierarchy, **sys:expression**, does have a printer, a parser, and a history. All data types not explicitly defined as presentation types inherit these three aspects from **sys:expression**. Thus **sys:expression** represents ordinary Lisp objects without application-specific semantics. The parser is essentially the Lisp **read** function, with the **:escape** option chosen heuristically. One subtype of **sys:expression** is **sys:form**, the presentation type for Lisp forms to be evaluated. In the Lisp read-eval-print loop, input is accepted with presentation type **sys:form** and output is presented with presentation type **sys:expression**. (Normally the Lisp listener accepts commands as well as forms, so the above description is not completely accurate for Lisp listeners.)

Many other defined presentation types inherit one or more of these features as well.

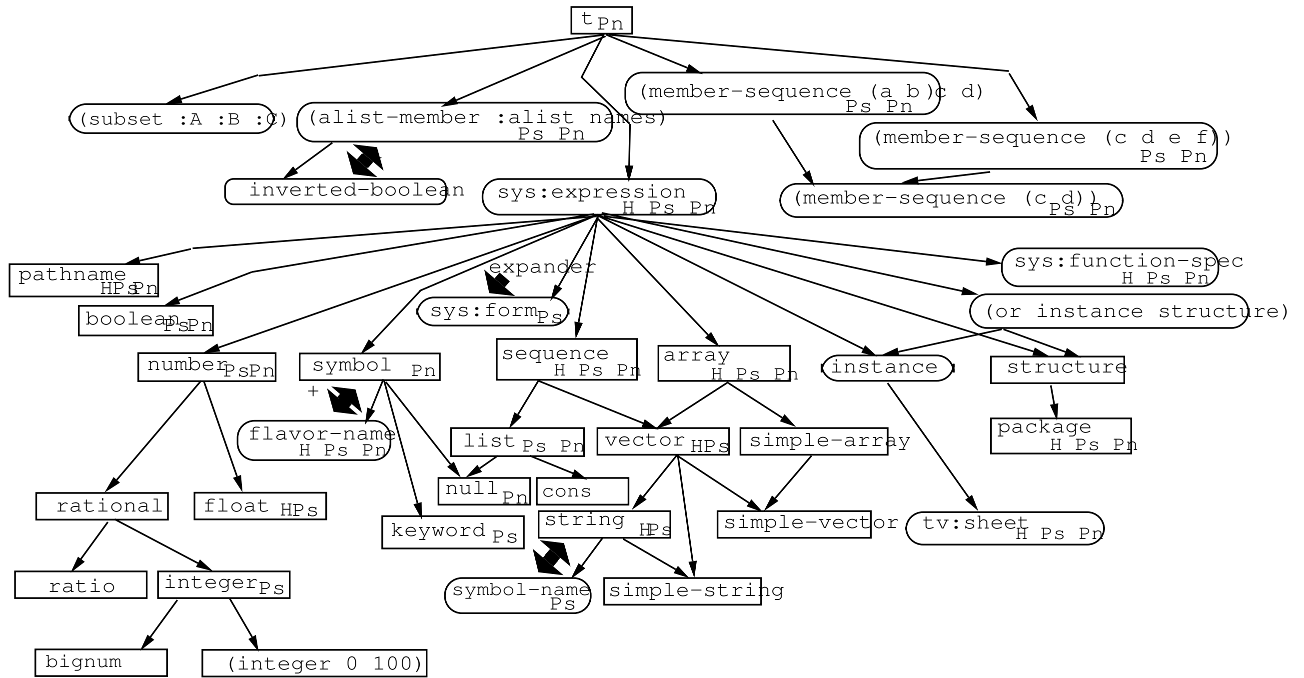
Figure ! illustrates several features of the presentation type system. **Note well**: The relationships shown in this figure are *representative*, not necessarily actual. The internal implementation of predefined presentation types is subject to change.

In the figure:

- The presentation type relationships form a lattice, not a tree. Note, for example, that the types **vector**, **list**, **simple-string**, and others are subtypes of more than one supertype.

- The boxes with square corners contain presentation types that are also Common Lisp types. The boxes with rounded corners contain types that are not Common Lisp types. Note that there are derivative presentation types (shown enclosed in double parentheses) in both categories. The letters H, Ps, and Pn indicate types that have their own histories, parsers, and printers. (We show these here only for instruction purposes, however. The actual assignments of these type attributes are subject to change and should not be counted upon.) For example, **sys:expression**, **sequence**, **array**, **package**, **sys:flavor-name**, and others have their own type histories. Types that do not have their own histories, and are subtypes of **sys:expression**, inherit a type history from that, but pruned. For example, in the context of accepting an object of type **symbol**, the system generates a list of possibilities by finding in the list of all **sys:expression**s those objects for which **symbolp** is true.

  In the same manner, types that do not have their own parsers or printers inherit those from their superiors. For example, **sys:form** inherits its printer from




```
* abbreviation-for (alist-member :alist '(("Yes" . nil)("No" . t)))
+ abbreviation-for (and symbol (satisfies flavor-name-p))
```

Figure 54. Presentation type subset/superset representations

**sys:expression** and all the subtypes of **number** inherit their printers from that type. Notice, though, that **float** and **integer** have their own parsers.

- The large arrows point out types that illustrate the use of the keywords **:expander** (one-way arrow) and **:abbreviation-for** (two-way arrow) in their type definitions. For example, the definition of **symbol-name** includes `:abbreviation-for 'string` and that for **sys:form** includes `:expander 'expression`. This will be discussed in greater detail next.

Besides understanding inheritance, you also need to know something about how the system goes about matching potential input with the current context. For example, given a command that requires an object of a particular presentation type, say, a **net:local-host**, how does the system decide whether some object is a member of that type? This knowledge is necessary so that you can define a presentation type such that the matching process will be as efficient as possible.

## Input Context Matching

In order to provide automatic highlighting and mouse sensitivity of displayed objects, the presentation system matches the presentation type of an object with the presentation required by the current *presentation input context*. The presentation input context establishes the *to-presentation-type*. (The presentation type of a displayed object is the *from-presentation-type*.) When the *to* and *from* types match, the object usually becomes mouse-sensitive and is highlighted. It is important to note that the meaning of "match" here refers to type/subtype relationships, not to equality.

A presentation input context is established by a call to **accept** or by a higher-level function that incorporates such a call. The macros **dw:with-presentation-input-context** and **dw:with-presentation-input-editor-context** also establish it. The mechanism for effecting mouse sensitivity is the translating mouse handler. A handler defined for a presentation type translates *from* that type, or any subtype of that type, *to* something else, possibly another type, possibly a command or action. Mouse handlers and the way they work are documented in the chapter "Programming the Mouse: Writing Mouse Handlers".

Here is a simple example of input context type matching. Suppose we call `(accept 'symbol)`. This establishes the input context in which anything whose type is a subtype of the presentation type **symbol**, for example, **sys:flavor-name** or **keyword**, is recognized to qualify as input to the call. Thus, if **:pattern** has previously been presented as a keyword, it will be highlighted and sensitive in the **symbol** input context.

Here is a more complicated case of type matching. Suppose we have defined a presentation type **integer-name**, using **:expander 'integer**. We also have a mouse handler that translates from the presentation type **number** to the presentation type **version-number**, which we have defined using **:expander 'file-identifier**. The translating handler, on the *from* side, applies to any presentation that is a subtype of the *from* presentation type. In this case, it applies to **integer-name**, since that is a subtype of **integer**, which is a subtype of **number**. On the *to* side, the handler applies in any context for which the *to-presentation-type* is a subtype of the context type. In this case, it applies in the context of **file-identifier**, since **version-number** is a subtype of that. So, whenever the input context type is **file-identifier**, any object of the type **integer-name** is mouse-sensitive. This is illustrated in the accompanying figure.
Note that, as a special case, when the presentation type **t** is specified as the *to-presentation-type*, it means "all contexts" — that is, any presentation type is acceptable.

Input context matching is explained in greater detail elsewhere. See the section "Presentation Type Matching for Mouse Handlers".

## The Recursive Behavior of Accept

Often a presentation type can best be parsed by means of reference to some other, usually more general, presentation type. For this kind of type, it is perfectly ac-

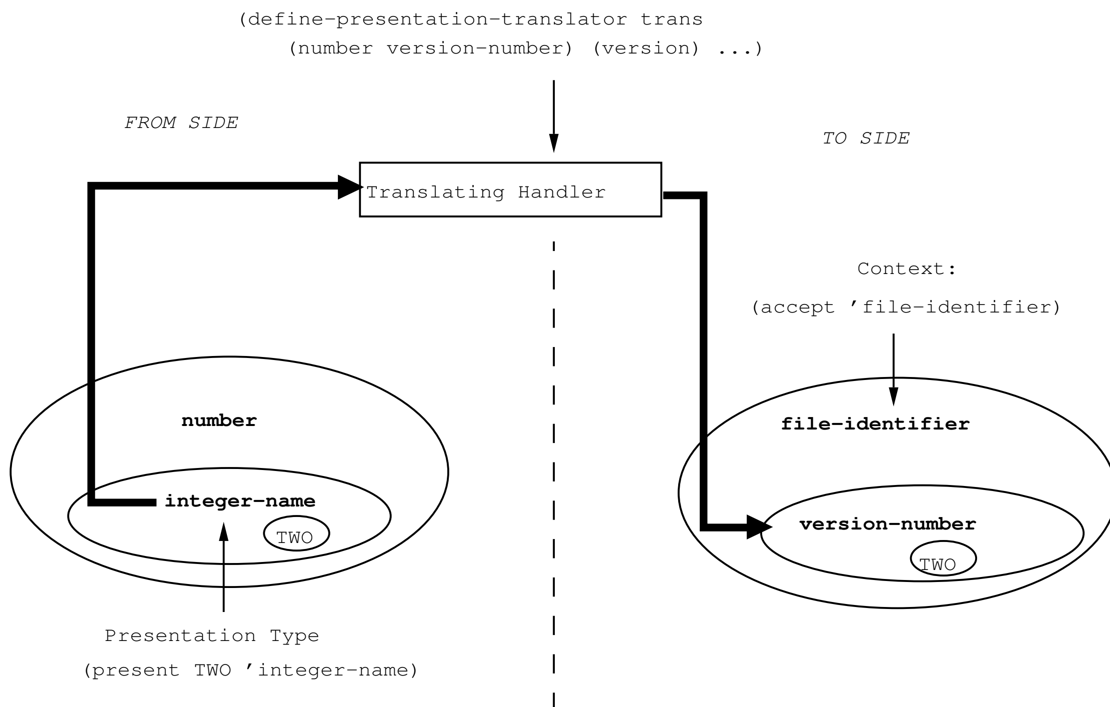



Figure 55. Matching betweem input context and a presentation type by a translating handler

ceptable to write a parser that makes a call to **accept**. Here is an example.

```
(define-presentation-type ones-complement-integer ()
   :history t
   :printer ((object stream)
             (prin1 (if (minusp object)
                        (- (lognot object))
                        object)
                    stream))
   :parser ((stream &key default original-type)
            (let ((twos-complement
                     (accept 'integer :stream stream
                                      :prompt "Enter an integer"
                                      :default (and default
                                                     (if (minusp default)
                                                         (- (lognot default))
                                                         default))
                                      :original-type original-type)))
              (if (minusp twos-complement)
                  (lognot (- twos-complement))
                  twos-complement))))
```

Now, when you or the user want a ones-complement integer as input, you call **accept** or an equivalent:

```
(accept 'ones-complement-integer)
```

Here **accept** is being called recursively: First an integer is accepted, and then the ones-complement integer generated from the result is accepted. There is no limit to this kind of nesting of **accept** calls.

Another situation in which **accept** is called recursively is the case of compound objects, for example, **character-style**: **accept** is called recursively on each item of family, face, and size.

The recursive use of **accept** is important to recognize because of the implications it has for the stream, prompt, and default arguments of the presentation type's parser.

Try making the definition above and running the **accept** form several times, noticing the behavior of the prompt and the default. You will see that they are awkward. After we introduce the defining macro for presentation types, we will discuss writing parsers, and how to avoid this kind of awkwardness, in the section "Writing a Parser That Recursively Calls Accept".

## The define-presentation-type Macro

The **define-presentation-type** macro is the tool for defining your own presentation types. It allows you to define many different sorts of functions to be associated with your new type. These become part of the presentation-type substrate and are run by it when appropriate. Additionally, it allows you to specify data arguments and presentation arguments that can be used in just the same manner as those of Common Lisp types and other Symbolics predefined presentation types. The data and presentation arguments you specify are lexically available as variables in the bodies of those presentation-type functions that are supposed to depend on them.

Always keep in mind, when you are defining a new presentation type, that a substantial number of predefined types are available, one of which might suit your needs. Or, there may be a combination or specialization of existing types that you can make use of using the **:expander** or **:abbreviation-for** keyword arguments.

The keyword options of the **define-presentation-type** macro are summarized in the following table, which lists the purpose of each option and auxiliary functions that are typically used within the form that is its value.

| *Keyword* | *Description* | *Auxiliary Functions* |
|---|---|---|
| **:printer** | Display an object of this type. | **format**, **princ** **write-string**, **present**, **present-to-string**, **dw:with-output-as-presentation** |
| **:parser** | Recognize keyboard input as a member of the type. | **accept**, **dw:complete-input**, **dw:complete-from-sequence**, **dw:completing-from-suggestions** |
| **:viewspec-choice** | Set up accepting-values-like structures for displaying this type of object in several ways. | **dw::presentation-type-keyword-options-into-cvv** |
| **:describer** | Describe this presentation type. | **dw:describe-presentation-type**, **format**, **formatting-textual-list**, **write-string**, **princ**, **dw::presentation-string-pluralize** |
| **:default-preprocessor** | Coerce default to conform to type's data or presentation arguments. | **dw::common-sequence-default-preprocessor** |
| **:highlighting-box-function** | Highlight objects of this type in a special way. | Generic graphics primitives |
| **:choose-displayer** **:menu-displayer** **:accept-values-displayer** | Show the choices that can be made for objects of this type. | **dw:accept-values-choose-from-sequence** |
| **:presentation-subtypep** | Use data arguments to determine type relationship between subtypes of this type. | Lisp functions, such as **eq**, **eql**. |

| | | |
|---|---|---|
| **:typep** | Use data arguments to determine if some object is of this type. | Lisp functions, such as **eq**, **eql** |

The last two options in the table require explanation. If you define a presentation type that has data arguments that restrict membership in the type, you often need to specify how these arguments are to be used. For example, suppose you define a presentation type **bins** with a data argument of *maximum-size*. You want a bin to be accepted as a member of **((bins 8))** if it has a size of 8 or less. In this case, you must include a **:presentation-subtypep** function in your presentation type definition to specify that the type **((bins 6))** is a subtype of **((bins 8))**. Similarly, you should include a **:typep** function in your presentation type definition to specify how to test whether an object is a bin of an acceptable size.

Of the remaining keyword options to **define-presentation-type**, **:expander** and **:abbreviation-for** are discussed in an earlier section. See the section "**:expander** and **:abbreviation-for**". The other options are summarized in the following table:

| *Keyword* | *Description* |
|---|---|
| **:description** | A string that describes the type. Used if there is no **:describer**. |
| **:no-deftype** | A Boolean, which when **t**, specifies that the data type has been defined elsewhere, so define *only* the presentation type rather than altering the definition of the data type. |
| **:history** | A Boolean, which, when **t**, specifies that this type is to have its own history (instead of inheriting one). |
| **:presentation-type-arguments** | A list of the data arguments that are presentation types themselves. |
| **:data-arguments-are-disjoint** | Indicates that the data-arguments partition the set into disjoint sets. This can be specified, for example, in cases where `(fn a)` and `(fn b)` denote disjoint sets if and only if `(not (eql a b))`. |
| **:do-compiler-warnings** | A function for checking that presentation-type arguments in *data-arglist* are the correct type. Not needed if **:presentation-type-arguments** is used. |

Other keywords mentioned in the argument list of **define-presentation-type** are exclusively for internal system use.

**define-presentation-type** *:no-deftypetype-name* (*data-arglist* . *pr-arglist*) &key *parser printer viewspec-choices description describer no-deftype disallow-atomic-type* (*history* **nil** *history-supplied-p*) *expander abbreviation-for choose-displayer accept-values-displayer menu-displayer default-preprocessor history-postprocessor highlighting-box-function presentation-type-arguments presentation-subtypep key-generator key-function do-compiler-warnings map-over-subtypes map-over-supertypes map-over-supertypes-and-subtypes typep with-cache-key data-arguments-are-disjoint* *Function*

Defines a new presentation type. Note that you cannot use **:no-deftype** and **:abbreviation-for** together.

*type-name* The name for the new type.

*data-arglist*

Arguments describing an object of this type; *data-arglist* may be any permissible **defun**-style argument list.

Data arguments are used to determine the sensitivity of an object in any given input context established by **accept**, and the applicability of defined mouse handlers. They also participate in determining the subtype and supertype relationships of the type. (For more information and examples, see the section "Predefined Presentation Types".)

*pr-arglist* Keyword arguments that affect the accepting or presenting of an object of this type; such keywords are handled in the body of the presentation type's **:parser**, or **:printer** respectively (see below).

Unlike data arguments, presentation arguments are not relevant to determining mouse sensitivity or subtype and supertype relationships. (For more information and examples, see the section "Predefined Presentation Types".)

(Certain predefined keywords are meta-presentation arguments. They can be used when calling any type and are understood directly by **accept** or **present**, rather than used by the type's parser or printer. At present, such arguments are limited to **:description**. Meta-presentation arguments are not available to type-methods unless explicitly listed in *pr-arglist*. (For more information, see the section "The Presentation Type System: an Overview".)

All of the functions that can be defined as values for options take the following two arguments.

**:original-type** Specifies the presentation type originally supplied in the call to **accept**.

**:type** Specifies the presentation type from which the function being specified is inherited.

**:parser** Specifies a function for parsing a presentation object of the defined type. This is what **accept** calls for inputting objects entered as a series of characters.

Arguments passed to the parser function include the input stream and a set of optional keywords. Of these, the arguments that you intend for the parser to use must be declared in the argument list for the parser function, after an **&key**. The parser keyword options, in addition to **:original-type** and **:type** are:

**:default** Value is supplied by **accept**. You only need to use this for merging; actual defaulting is handled at a higher level.

**:initially-display-possibilities**

Boolean option specifying whether to display the objects

that could be used as input in the current context; the default is **nil**. If **t**, the possibilities are presented before the input prompt appears.

**:default-supplied**
True if the **:default** argument is valid.

**:default-type**
The presentation type of the default.

The **:parser** function that gets invoked is found via the presentation-type inheritance mechanism — it will not necessarily be *this* parser. The parser that is found may have called **accept** recursively on a less specific type and passed along the original type via the **:original-type** option. As was mentioned above, any of the **accept** keyword arguments in the following list, can be passed through and used in your parser, if you declare them in the parser's argument list: **:type**, **:original-type**, **:initially-display-possibilities**, **:default-supplied**, and **:default-type**. For example, if you want to use the default specified by **accept**, you can use:

```
(define-presentation-type foobar ()
   :parser ((stream &key default) ...))
```

Additionally, there is one other source of arguments: those declared in the type's *data-arglist* and *pr-arglist*. These are available lexically in the body of the parser function, and are not explicitly declared in the argument list to the parser function.

The syntax for the parser function is as follows:

:parser ((stream &key *parser keywords*) *body*)

The parser should return one required value, the object parsed, and possibly one additional value, the presentation type of the object parsed. The latter defaults to the type for the parser. Do not return this second value unless you really intend for it to be used.

**:printer** Specifies a function for printing a presentation object of the defined type. This is what **present** calls for outputting objects.

Arguments passed to the printer function include the object, the output stream, and a set of optional keywords. Of these, those arguments that the printer uses must be declared in the argument list for the printer function. The printer keyword options, in addition to **:original-type** and **:type**, are:

**:acceptably**
Specifies when **t** to print the presentation in such way that it can be parsed by **accept** as the specified presentation type. Other possibilities include **nil** and **:very**. The latter is for use with **:for-context-type**.**:for-context-type**
Specifies the context in which the presentation is to be

presented with **:acceptably :very**. The most often used value is `'((cp:command-or-form :dispatch-mode :form-preferred)))`, which causes presentations of the type **cp:command** to be printed with a leading colon.

Additionally, there is one other source of arguments: those declared in the type's *data-arglist* and *pr-arglist*. These are available lexically in the body of the parser function, and are not explicitly declared in the argument list to the printer function.

The syntax for the printer function is as follows:

:printer ((object stream &key *<parser keywords>*) *body*)

**:viewspec-choices**

Specifies a form that returns a list of locatives, presentation types, and prompts to slots in the presentation type. This provides the ability to do in-place modification of presentation printing.

The keyword option for this form, in addition to **:original-type** and **:type**, is:

**:presentation** The particular presentation being re-viewed (having its viewspecs changed).

Example:

```
(defflavor employee ((first-name)
                     (last-name)
                     (status))
           ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

(define-presentation-type employee (()
                ;; keywords for different printed representations
                &key (format :last-name-first)
                     (include-status nil))
```

```
:no-deftype t
:printer ((employee stream)
          (ecase format
            (:last-name-first
              (format stream "~A, ~A"
                      (employee-last-name employee)
                      (employee-first-name employee)))
            (:first-name-first
              (format stream "~A ~A"
                      (employee-first-name employee)
                      (employee-last-name employee)))
            (:last-name-only
              (write-string
                      (employee-last-name employee) stream)))
          (when include-status
            (format stream " (~(~A~))"
                      (employee-status employee))))
:viewspec-choices ((&key type)
                   ;; a necessary internal function
                   (dw::presentation-type-keyword-options-into-cvv
                     type
                       ;; Choice 1:  keyword, pres type (member),
                       ;; selected choice (optional), and prompt
                     '((:format (member :last-name-first
                                        :first-name-first
                                        :last-name-only)
                         :last-name-first "Format of name")
                       ;; Choice 2:  keyword, pres type (boolean),
                       ;; selected choice (optional), and prompt
                       (:include-status boolean nil "Include status")))))

(present (make-instance 'employee :last-name "Jones"
                        :first-name "Fred" :status :retired))
```

Compile the two definitions; then evaluate the **present** function. You can either click s-sh-Middle on the presentation to invoke the Edit Viewspecs mouse handler or click Right on the presentation to get a menu of options, one of which is "Edit viewspecs". Clicking s-sh-Middle or selecting "Edit viewspecs" brings up a **dw:accept-variable-values** menu. Using this, you can specify how the presentation is displayed.

With the **:viewspec-choices** option, you give your users the ability to modify at runtime any displayed presentations of the defined type. To provide the same capability with respect to arbitrary program output, you can use **dw:with-replayable-output**. See the function **dw:with-replayable-output**.

In addition to **:original-type** and **:type**, the keyword arguments for this option are:

**:description**
Specifies a string describing the presentation type, for example, "an

integer". This string is used by **accept** to prompt for an object of this type.

This option and the **:describer** option are mutually exclusive. If neither option is supplied, a description is created based on inheritance from another type; if type is **t**, it will be the string "anything".

Do not confuse this option with the **:description** meta-presentation argument. This option supplies the default for the meta-presentation argument. See the section "The Presentation Type System: an Overview".

**:describer**

Specifies a function that outputs a description of the presentation type. This is used by **accept** to prompt for an object of this type. The describer function is generally used only for complex presentation types, such as compound and aggregate types.

Arguments passed to the describer function include the output stream and a set of optional keywords. The describer function keyword, in addition to **:original-type** and **:type**, is:

**:plural-count**

Boolean option specifying whether the type description is pluralized.

The syntax for the describer function is:

:describer ((stream &key *<describer keywords>*) *body*)

This option and the **:description** option are mutually exclusive. If neither option is supplied, a description is created based on inheritance from another type; if that type is **t**, this description is the string "anything".

**:no-deftype**

Boolean option specifying whether this definition only defines a presentation type and not also a new data type. The default (**nil**) results in the generation of a **deftype**.

`:no-deftype t` must be supplied if a **deftype** is provided elsewhere for the symbol used as the *type-name* argument in the presentation type definition. This also applies to presentation types being defined for flavors and structures previously defined by **defflavor** and **defstruct**, respectively. For more information: See the section "Using User-Extendable Data Types as Presentation Types".

**:disallow-atomic-type**

This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:history** Boolean option specifying whether a separate history is created for this presentation type. The default is **nil**, meaning that the history will be found via inheritance.

**:expander**

Specifies a form that is invoked to generate the "expansion" of the presentation type, for example, (or *pres-type1 pres-type2*). Expansions allow for presentation types to inherit presentation functions (that is, parsers, printers, describers) from other presentation types.

If you do not specify an **:expander**, then you must either specify the **:abbreviation-for** option or supply a parser and printer. If you do specify an expander, you can still supply the presentation type with its own parser or printer, and just inherit the functions not supplied; however, you may not specify the **:abbreviation-for** option.

**Note**: This should not depend on outside influences to determine its expansion. This includes global variables. It is generally better to pass in the outside information as data arguments. If this is not practical, however, you may use the form (**dw:prepare-for-type-change** '<*type-name*>) before changing the variable, and (**dw:finish-type-redefinition**) after the change is complete. This will allow SemantiCue to maintain its handler tables properly.

**:abbreviation-for**

Specifies the form that is invoked to generate another presentation type for which this presentation type serves as an abbreviation. The form lets you define a new presentation type by combining, or in other ways qualifying, existing presentation types, for example, (**and** *pres-type* (**satisfies** *a-predicate*)). Example:

```
(defvar *star-list* '(("Vega" :vega)
                      ("Altair" :altair)))
(dw:define-presentation-type star ()
  :abbreviation-for `(alist-member :alist ,*star-list*))
```

Better ways to write the defining code for the type are:

```
(dw:define-presentation-type star ((&key star-list))
  :abbreviation-for `(alist-member :alist ,star-list))
(accept `(star :star-list ,*star-list*))
```

or

```
(dw:define-presentation-type star ()
  :parser ((stream &key original-type)
           (dw:complete-from-sequence
            *star-list* stream
            :type original-type
            :name-key #'first
            :value-key
              #'tv:menu-execute-no-side-effects)))
(accept 'star)
```

**:choose-displayer**

Specifies a form that does output showing the choice or choices that can be made for a presentation of this type in a menu or accept-values context. This output is in place of the default value, which is displayed if you do not define a **:choose-displayer**, and is useful in cases when you want a sequence or enumeration of choices displayed. There are no other keyword arguments for this function besides **:original-type** and **:type**.

Use the internal function **dw:accept-values-choose-from-sequence** to write this form. The following example is extracted from the definition for the **alist-member** presentation type. The full definition is included in the file sys:dynamic-windows;sequence-types.lisp.

Example:

```
(define-presentation-type alist-member ((&key alist)
        &key (convert-spaces-to-dashes nil))
   :choose-displayer ((stream object query-identifier
                        &key original-type)
                       (accept-values-choose-from-sequence
                        stream alist object query-identifier
                        :type original-type
                        :key #'tv:menu-execute-no-side-effects))
   ...)
```

**:accept-values-displayer**

Specifies a form that does output showing the choice or choices that can be made for a presentation of this type in a accept-values context. This output is in place of the default value normally used, and is useful in cases when you want a sequence or enumeration of choices displayed. The **:accept-values-displayer** form should return five values: object, presentation-type, presentation, x, and y. The only keyword argument for this form, besides **:original-type** and **:type** is:

**:provide-default**

Boolean option specifying whether a default should be provided.

Use the internal macro **dw:standard-accept-values-displayer** to write this form. Here is an example, which first defines the presentation type, and then shows its use. The defined type does not have a **:parser**, so it inherits the integer parser, and it does not allow spelled-out numbers to be typed in. In the example, **dw:present-editable-choice** is an internal function **flet**'ed by the **dw:standard-accept-values-displayer** macro.

```
(define-presentation-type spelled-integer
                          (() &key (count 3) (language :english))
  :abbreviation-for 'integer
  :describer
  ((stream &key plural-count)
   (unless plural-count
     (write-string (if
                     (find (char
                             (string language) 0)
                           "AEIOU") "an " "a ")
                   stream))
   (write-string (string-capitalize language) stream)
   (write-string " number" stream)
   (when plural-count
     (write-string "s" stream)))
  :printer
  ((n stream &key acceptably)
   (let ((*print-base* (if (or acceptably (minusp n) (> n count))
                           10
                           language)))
     (prin1 n stream)))
  :accept-values-displayer
  ((stream object query-identifier &key
           original-type provide-default)
   (dw::standard-accept-values-displayer
     (stream object query-identifier provide-default)
;; First a row of spelled out integers, selectable with the mouse
     (let ((choices (loop for n from 0 to count collect n)))
       (dw:accept-values-choose-from-sequence
         stream choices object query-identifier
         :type original-type))
     (write-string "  " stream)
;; Position the cursor here if keyboard entry is used
;; dw::x & dw::y were bound by dw::standard-accept-values-displayer
     (multiple-value-setq (dw::x dw::y)
                          (send stream :read-cursorpos))
;; Now a field into which any integer can be entered via keyboard
     (let ((presentation
             (dw::present-editable-choice object
                                          'integer
                                          (and
                                            (integerp object)
                                            (≤ 0 object count)))))
;; Return values describing the displayed choices
       (values object original-type presentation dw::x dw::y)))))
```

```
(dw:accepting-values ()
  (list (accept 'spelled-integer :default 2)
        (accept '((spelled-integer) :count 10 :language :roman))))
```

**:default-preprocessor**

Allows preprocessing (coercion) of the default before the user sees it. This is useful when you want to change the object gotten from the presentation history so that it conforms to the data or presentation args. For example, the default-preprocessor for pathname looks like

```
(define-presentation-type pathname
     (() &key (default-version :newest default-version-p)
     (default-type nil default-type-p)
     (default-name nil default-name-p)
     dont-merge-default (direction :read)
     (format :normal))
  :default-preprocessor
  ((default)
  (when default-version-p
   (setq default (send default :new-version default-version)))
  (when default-type-p
   (setq default (send default :new-type default-type)))
  (when default-name-p
   (setq default (send default :new-name default-name)))
  default) ....
```

The only keyword argument, besides **:original-type** and **:type** is:

**:default-type**

The presentation type of the default.

If the object cannot be coerced properly, the default preprocessor returns **nil**. This is useful when the system is mapping over elements of the history trying to find a default. If the default preprocessor returns two values, the second one should be a presentation-type, which should be at least as specific as the **:original-type** argument.

The syntax for the default-preprocessor function is as follows:

:default-preprocessor (object &key type original-type default-type) *body)*

**:highlighting-box-function**

Specifies a non-standard way of highlighting objects of a particular presentation type, for example, by drawing circles around them. The keyword arguments :x and :y are the upper left corner of the presentation's bounding box. All drawing to *stream* should be done using the generic graphics primitives, and only *outline* drawing should be done. (In other words, be sure to specify **:filled nil**.) The outline

drawn by the highlighting box function is cached in the presentation so that the function does not have to run each time the presentation should be highlighted. (Note this, because it means that if you recompile the presentation type, existing presentations of this type that have already been highlighted will not get the new definition.)

The syntax for the highlighting function is as follows:

:highlighting-box-function ((stream &key presentation x y type original-type) *body*)

Example:

```
(define-presentation-type circle-pathname ()
  :abbreviation-for 'pathname
  :highlighting-box-function
  ((stream &key presentation)
  (multiple-value-bind (left top right bottom)
     (dw:box-edges
      (dw:presentation-displayed-box presentation))
   (let* ((width (- right left))
          (height (- bottom top))
          (max-dimension (max width height))
          (center-x (floor (+ left (/ width 2))))
          (center-y (floor (+ top (/ height 2))))
          (diameter (+ max-dimension 10))
          (radius (floor (/ diameter 2))))
    (graphics:draw-circle center-x center-y radius
                          :stream stream
                          :filled nil)))))
```

`(present (fs:user-homedir) 'circle-pathname)` and then move the mouse over it.

**:presentation-type-arguments**

Specifies a list of type arguments, appearing in the presentation type's *data-arglist*, that are themselves presentation types.

**define-presentation-type** uses this list in writing the appropriate **:do-compiler-warnings** option to the macro if this option is not supplied explicitly. Also, it is used for caching: omitting it can make the caches much larger and less effective.

**:presentation-subtypep**

Specifies a comparison function for the case where two type names are the same but the data arguments are different.

The function receives two arguments, both lists. The first is a list of the type-name and data arguments of this presentation type; the second is a list of the type-name and data arguments of the putative supertype, that is, of the presentation type with which this one is being compared.

**dw:presentation-subtypep** determines the applicability of mouse handlers to displayed presentations in a given input context. By writing the comparison function yourself, you can control the mouse sensitivity of presentations of the defined type relative to available mouse handlers.

Because you likely want to use arguments in the *data-arglist* for writing the comparison function, you should use the default value (**nil**) for the **:data-arguments-are-disjoint** option to **define-presentation-type**.

**:key-generator**
This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:key-function**
This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:do-compiler-warnings**
Specifies a function for checking that presentation-type arguments appearing in the *data-arglist* and available at compile time are of the correct type. If you specify such arguments in the **:presentation-type-arguments** option, you do not need to write a **:do-compiler-warnings** function to handle these.

**:map-over-supertypes**
This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:map-over-subtypes**
This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:map-over-supertypes-and-subtypes**
This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:typep** Specifies a function that determines whether a given object is of the type specified by the data arguments in the presentation type. It takes one argument, the object.

The **:typep** function is used to determine, for example, whether a displayed integer presentation in an input context established by `(accept '((integer 1 10)))` can be used as input, that is, whether the displayed integer is in fact between 1 and 10. In the general case, the **:typep** function must consider all of the positional and keyword data arguments to a presentation type in determining if the presentation object at hand is of the type sought. The data arguments are made lexically available to the **:typep** function when it

is invoked. (The presentation arguments are not available.)

**:with-cache-key**

This keyword argument is for the use of internal system functions only. Do not use it in application code.

**:data-arguments-are-disjoint**

Boolean option specifying whether the arguments included in the *data-arglist* are to be used as keys for determining the equivalence class of the presentation type. The default is **nil**. If **:data-arguments-are-disjoint** is **nil**, then the presentation type name and the data-arguments are not considered in subtype relationships unless you define a **:presentation-subtypep** function.

If you use the **:presentation-subtypep** option to **define-presentation-type** for writing the comparison function controlling subtype relationships, then use the default value, **nil**, for **:data-arguments-are-disjoint**. Also use the default if the data arguments to this type are not appropriate for comparison by **eql**.

For information on writing parsers for presentation types, including examples, see the section "Defining Your Own Presentation Types". For more examples, see the file sys:dynamic-windows;presentation-types.lisp.

### Miscellaneous Presentation Facilities

Perhaps the most important among the various other presentation facilities is **dw:presentation-subtypep**. This function tests to see if one type can be regarded as a subtype of another. Subtype considerations are key for determining the availability of presentation objects for input in a given context, and the applicability of mouse handlers to such objects.

In general, when the input context is for a supertype, all subtypes to that supertype are acceptable as input. Similarly, if a mouse handler is defined for the supertype, it is also active for all the subtypes. In both cases, the reverse is not true; that is, when a subtype is specified, a supertype is not acceptable.

In concrete terms, when you are accepting a **number**, any kind of number — **integer**, **ratio**, etc. — will do; when you are looking for an **integer**, any kind of integer will do, but not any kind of number. **dw:presentation-subtypep** and equivalent internal functions are the basis of such determinations.

The remaining facilities in this subcategory are for taking apart presentations and manipulating presentation-type arguments. They are:

**dw:presentation-type-p**
**dw:presentation-object**
**dw:presentation-type**
**dw:presentation-equal**
**dw:describe-presentation-type**

**dw:check-presentation-type-argument**
**dw:with-presentation-type-arguments**
**dw:with-type-decoded**
**dw:presentation-type-name**
**dw:presentation-type-default**

### Using User-Extendable Data Types as Presentation Types

All user-extendable data types (flavors and structures) are also presentation types. These types inherit from the **sys:expression** presentation type. Unless you want to define more restrictive subtypes of the structured types using data arguments, there is no particular advantage to defining your own presentation types for them, unless perhaps to supply a parser or a printer.

You will, however, need to define a print-self method. All presentation types used as CP arguments must be presentable in a way that **accept** can handle, since this is the way that CP unparsing works. The default printer for flavors, using the #<> syntax, even with **princ**, is not parsable by **read**. So, if you are using flavor names as presentation types for CP arguments, even if the commands are only echoed by a program frame and not to be entered from the keyboard, you must supply a **sys:print-self** method to handle **princ**.

If you want to introduce data arguments to form subtypes of, for example, a flavor, the best thing to do is to define a new presentation type for that subtype. In other words, do not try to use data arguments and **:presentation-subtypep** to form subtypes of structures, but define a separate type for each kind of subtype.

If you do define a non-separate presentation type for an extended type, you must remember to include the **:no-deftype t** option to **define-presentation-type**.

If you do not supply a **:printer** for your new presentation type, the default printer that it inherits will do **princ**. If you do not supply a **:parser**, you will only be able to type in a structure, and only using #**s** syntax. This is presumably not very desirable — if you really want to type in structures or instances, you need to define the parser.

### Writing a Parser

### General Approach to Parser Writing

It is a presentation type's parser function that determines which sequences of characters are potentially acceptable as members of the type. This is because **accept** sets up the input context. The parser does not always do the whole job, however: If you specify data arguments to restrict type membership, the **:typep** function you supply is the final arbiter of inclusion.

In addition to its function of membership determination, it is the parser's responsibility to provide appropriate prompting, defaults, completion, and the like. Here are the important requirements of a parser:

The parser and printer must work as a team. A parser must be able parse back into the original object the printed representation the printer produces (at least for presentation types that are going to be used in command lines).

A parser can process input at several levels. You need to specify what errors to signal when invalid input is received (see the **sys:parse-error** function and the errors **dw:input-not-of-required-type** and **dw:object-parsed-not-of-type**).

The lowest-level approach is to read individual characters until a valid input is recognized.

The next level is to call **dw:read-standard-token** to get an entire token, and then look it up in a database or otherwise map from the token to an object. This is the lowest-level method that any normal parser would use. You can use **dw:with-accept-blip-chars** to define the set of "trigger" characters, which, when typed, will cause **dw:read-standard-token** to return (it returns in the token all the characters up to, but not including, the blip character). It is the parser's responsibility to read out embedded delimiter characters (for example, the dots separating family.face.size in a character style), but the parser must leave the final delimiter that terminated the whole parsing process for the caller to deal with (for example, the space that separates fields in a command line).

```
:parser ((stream &key original-type)
         (let ((token (dw:read-standard-token stream)))
           (second (or (assoc token *names-to-objects-mapping-alist*)
                       (error 'dw:input-not-of-required-type
                              :string token
                              :type original-type)))
```

The next level is to call one of the completion utilities: **dw:complete-from-sequence**, **dw:complete-input**, or **dw:completing-from-suggestions** and **dw:suggest**.

Mouse sensitivity is automatically provided for your parser when you make use of any of the presentation system input functions, all of which are based on **accept**. You can think of **accept** as having been defined by:

```
(defun accept (type &key stream etc...)
  (let ((parser-function (lookup-presentation-type-parser type)))
    (dw:with-presentation-input-context (type)
                                        (blip)
        (funcall parser-function stream)
      (t (dw:presentation-blip-object blip)))))
```

Where **dw:with-presentation-input-context** establishes the mouse sensitivity context and the parser function is called to do the actual input.

Here are a few general suggestions:

- Avoid the necessity for writing your own parser by making use of those already provided for the predefined presentation types. That is, use **:abbreviation-for** and **:expander** whenever possible. An exception to this rule, however, is the next suggestion.

- When the type you are defining is a member of a list that changes dynamically, write a parser for it using **dw:complete-from-sequence** rather than defining the type as an **:abbreviation-for** using **member** or **alist-member**. If the list is a constant, define the type using those predefined types.

- If you are using an extended type, such as a flavor or a structure, and you want to use data arguments to further restrict type membership, define a presentation type for such a restriction. See the section "Using User-Extendable Data Types as Presentation Types". Remember that you can use existing presentation types inside of recursive calls to **accept** to provide the input context you want. See the section "Writing a Parser That Recursively Calls Accept". Make sure your parser only returns one value: The object it accepts as a member of the type. In rare cases, you can return two values: The second value must be a presentation type, usually a subtype of the parser presentation type. See the section "Returning Values from a Parser".

**Writing a Parser That Recursively Calls Accept**

The key points to remember when writing recursive calls to **accept** are:

- Be careful to read from the correct stream.

- Modify as necessary any default, and pass it on. Include

- ° **:default-type**

  ° **:default-supplied** - the value supplied here gets passed on as the argument to **:provide-default** in the recursive call to **accept**.

  Any prompt in the recursive call is output in parentheses after the main prompt.

- Pass on the value of **:original-type** if appropriate.

- Pass on the value of **:initially-display-possibilities**.

You should pass on the original type whenever the inner call to **accept** is reading a more general argument and filtering it. You should not pass it on when the inner call reads a piece of a more complex object.

*Examples:*

```
(define-presentation-type system-version-number
     ((&optional for-system))
  :expander '(integer 1)
  :parser ((stream &key original-type)
            (accept '(integer 1) :stream stream
                      :original-type original-type
                      :default
                      (when for-system
                       (sct:get-system-version for-system)))))
(define-presentation-type system-patch ()
  :printer ((object stream)
             (present (first object) 'sct:system :stream stream)
             (write-char #\space stream)
             (present (second object)
                       'patch-version :stream stream))
  :parser ((stream)
            (let ((system
                   (accept '(sct:system :patchable-only t)
                             :stream stream
                             :prompt "system"
                             :additional-blip-chars '(#\space))))
             (unless (dw:compare-char-for-accept
                       (dw:read-char-for-accept stream) #\space)
              (sys:parse-error "Must terminate with space"))
             (list system (accept 'patch-version
                                  :stream stream
                                  :prompt "version"))))
```

```
  :description "a system patch version")
(define-presentation-type patch-version ((&optional for-system))
  :printer ((object stream)
            (apply #'format stream "~D.~D" object))
  :parser ((stream &key initially-display-possibilities)
           (let ((major (accept `(system-version-number ,for-system)
                                :stream stream
                                :prompt "major"
                                :initially-display-possibilities
                                initially-display-possibilities
                                :additional-blip-chars
                                '(#\. #\space))))
            (unless (member (dw:read-char-for-accept stream)
                            '(#\. #\space)
                            :test #'dw:compare-char-for-accept)
             (sys:parse-error "Must terminate with space or period"))
            (list major (accept '(integer 1)
                                :stream stream
                                :prompt "minor"
                                :initially-display-possibilities
                                initially-display-possibilities)))))

(present (accept 'system-patch) 'system-patch)
```

If we did *not* pass down the original type in **system-version-number**, then random numbers would be sensitive (even though a parser might afterwards reject them). If we *did* pass the original type down from **patch-version** to **system-version-number**, we would be confusing looking for a list with looking for one of the elements of that list and patch version numbers would not be sensitive at all.

### Writing a Parser That Calls accept Several Times

The delimiter is left in the stream after **accept** returns. You must read it out between successive calls to **accept**. You should leave the final delimiter in the stream, so that your parser properly obeys the protocol.

The delimiter is used by programs like the Command Processor to determine whether a command argument field was terminated by a space to move to the next field, or by a newline to finish the whole command. Leaving the delimiter in the stream causes all nested levels of **accept** to properly terminate and return to the next higher level in the case of the activation (return) character.

If you want to have more field delimiters in effect, use the **:additional-blip-chars** option to **accept**. *Examples:*

```
(define-presentation-type system-patch ()
  :printer ((object stream)
            (apply #'format stream "~A ~D.~D" object))
  :parser ((stream &key default)
           (let ((system (accept '(sct:system :patchable-only t)
                                 :stream stream :default (first default)
                                 :additional-blip-chars '(#\space)
                                 :prompt nil)))
            (unless (dw:compare-char-for-accept (dw:read-char-for-accept stream)
                                                #\space)
             (sys:parse-ferror "You must terminate system name with space"))
            (let ((major (accept '(integer 1)
                                 :stream stream
                                 :default (second default)
                                 :additional-blip-chars '(#\.)
                                 :prompt nil)))
             (unless (dw:compare-char-for-accept (dw:read-char-for-accept stream)
                                                 #\.)
              (sys:parse-ferror "You must terminate major version with period"))
             (let ((minor (accept '(integer 1)
                                  :stream stream
                                  :default (second default)
                                  :prompt nil)))
              (list system major minor))))))
(define-presentation-type system-patch-with-prompts ()
  :printer ((object stream)
            (apply #'format stream "~A ~D.~D" object))
  :parser ((stream &key default)
           (let ((system (accept '((sct:system :patchable-only t))
                                 :stream stream :default (first default)
                                 :additional-blip-chars '(#\space)
                                 :prompt "system")))
            (unless (dw:compare-char-for-accept (dw:read-char-for-accept stream)
                                                #\space)
             (sys:parse-ferror "You must terminate system name with space"))
            (let ((major (accept '(integer 1)
                                 :stream stream
                                 :default (second default)
                                 :additional-blip-chars '(#\.)
                                 :prompt "major.minor")))
             (unless (dw:compare-char-for-accept (dw:read-char-for-accept stream)
                                                 #\.)
              (sys:parse-ferror "You must terminate major version with period"))
             (let ((minor (accept '(integer 1)
                                  :stream stream
                                  :default (second default)
                                  :prompt nil)))
              (list system major minor))))))
```

### Parsing Objects for Which There is No Character Representation

The **accept** function is called with a character stream. It takes care of setting up the right input context for use of the mouse, but it expects a parser function to be available to feed the character stream. For this reason:

- Give the presentation type a parser function that either ignores or signals a parsing error when something is typed on the keyboard. This makes the type available even for CP command arguments.

- Use **dw:with-presentation-input-context** to do the input without using the input editor. In this case, the presentation type cannot be used as a CP command argument. You will also need to extract the presentation value from its associated blip.

- Use the lower level mouse-oriented **dw:tracking-mouse** facility instead. In this case, you must keep in mind that you will see more presentations than just the ones that match, and must select for yourself.

*Examples:*

```
(define-presentation-type circle ()
   :printer ((object stream)
             (if (graphics:graphics-stream-p stream)
                 (graphics:with-room-for-graphics (stream)
                   (graphics:draw-circle 150 20 10 :stream stream))
                 (princ object stream)))
   :parser ((stream)
            (dw:read-char-for-accept stream)
            (sys:parse-ferror
                         "You must click on the desired circle.")))

(present 'foo 'circle)

(accept 'circle)
```

or

```
(defun pick-a-circle (&optional (stream *standard-input*))
(dw:with-presentation-input-context ('circle :stream stream) (blip)
       (let ((char (sys:read-character stream :presentation-context t)))
         (error "Don't know what to do with ~C" char))
     (circle (dw:presentation-blip-object blip))))

(present 'foo 'circle)

(pick-a-circle)
```

or

```
(defun pick-a-circle (&optional (stream *standard-input*))
  (dw:tracking-mouse (stream :whostate "Pick circle")
    (:presentation (presentation)
     (unless
       (loop as presentation = presentation then
                             (dw:presentation-superior presentation)
             while presentation
             doing
         (when (dw:presentation-subtypep
                         (dw:presentation-type presentation) 'circle)
           (send stream :set-highlighted-presentation presentation
                                                  "L: Use this circle.")
           (return t)))
       (send stream :set-highlighted-presentation nil)))
    (:presentation-click (presentation click)
     (unless (eql click #\mouse-l)
       (signal 'sys:abort))
     (loop as presentation = presentation then
                           (dw:presentation-superior presentation)
           while presentation
           doing
       (when (dw:presentation-subtypep
                        (dw:presentation-type presentation) 'circle)
         (return-from pick-a-circle
                        (dw:presentation-object presentation)))))))

(present 'foo 'circle)

(pick-a-circle)
```

## Returning Values from a Parser

The parser returns two values, the object parsed and its type. The second value is optional, meaning that if you do not return a second value, it defaults to the type for the parser. However, if you do return a second value, that is used. Be careful about accidentally returning values.

*Wrong:*

```
(define-presentation-type like-expression ()
   :parser ((stream)
            (read-from-string
                               (dw:read-standard-token stream))))
```

*Right:*

```
(define-presentation-type like-expression ()
   :parser ((stream)
            (values
              (read-from-string
                          (dw:read-standard-token stream)))))
```

### Writing a Printer

Writing a printer for a presentation type is straightforward, but there are a couple of things to note. The first is that the printer and parser must work as a team. A printer must be able to produce a printed representation of an object that the parser can parse back into the original object (at least for presentation types that are going to be used in command lines).

If you are printing your object as a graphic display, you must allow for the fact that **present** may be called with your presentation type on streams that do not support graphical output. Also, if you are outputting in a typescript (as in a Lisp window, as opposed to your own program framework), the cursor can be anywhere and should properly be factored into your output. For these reasons:

- Check whether the stream you are using does support graphics

- Or, do not put the graphics in the **:printer** function itself, but rather use an explicit **dw:with-output-as-presentation**.

*Examples:*

```
(define-presentation-type circle ()
   :printer ((object stream)
             (if (graphics:graphics-stream-p stream)
                 (graphics:with-room-for-graphics (stream)
                   (graphics:draw-circle 150 20 10 :stream stream))
                 (princ object stream))))

(present 'foo 'circle)
```

or

```
(define-presentation-type circle ()
   :printer ((object stream)
             (princ object stream)))

(defun draw-as-circle (object stream x y)
  (dw:with-output-as-presentation
                    (:object object :stream stream :type 'circle)
    (graphics:draw-circle x y 10 :stream stream)))

(dw:with-own-coordinates ()
  (draw-as-circle 'foo *standard-output* 150 20)
  (send *standard-output* :set-cursorpos 0 50))
```

The functions most often used by printers to produce output are **princ**, **format**, **write-to-string**, **present**, **present-to-string**, and **dw:with-output-as-presentation**. When you use one of the first three without enclosing it in **dw:with-output-as-presentation**, the object is presented as an object of type **sys:expression**. This is not, in general, what you want; in most cases you should use one of the latter three facilities, all of which are based upon **present**.

You can think of **present** as having been defined by:

```
(defun present (object type &key stream etc...)
  (dw:with-output-as-presentation (:object object
                                   :type type
                                   :stream stream)
    (funcall (lookup-presentation-type-printer type) object stream)))
```

The use of **present** in a printer, like that of **accept** in a parser, is recursive. Do not present an object of a specific type as a less specific one. Instead, call a subroutine from both printers (or one could have a keyword argument to allow doing this).

### Table of Facilities for Defining Presentation Types

**define-presentation-type** *:no-deftypetype-name* (*data-arglist* . *pr-arglist*) &key *parser printer viewspec-choices description describer no-deftype disallow-atomic-type* (*history* **nil** *history-supplied-p*) *expander abbreviation-for choose-displayer accept-values-displayer menu-displayer default-preprocessor history-postprocessor highlighting-box-function presentation-type-arguments presentation-subtypep key-generator key-function do-compiler-warnings map-over-subtypes map-over-supertypes map-over-supertypes-and-subtypes typep with-cache-key data-arguments-are-disjoint*
Defines a new presentation type.

**dw:with-presentation-input-context** (*presentation-type* &rest *options*) (&optional (*blip-var* **'dw::.blip.**)) *non-blip-form* &body *blip-cases*
Binds local environment to the input context of a specified presentation type.

**dw:read-standard-token** *stream*
Parses string as delimited by activation and blip characters established by **dw:with-accept-activation-chars** and **dw:with-accept-blip-chars**, respectively.

**dw:with-accept-blip-chars** (*additional-characters* &key *override*) &body *body*
Binds local environment to establish additional characters to be used as delimiters of input blips. The characters are additional only if a previous, higher-level call to this macro in a nested structure has established an existing set of delimiters; no predefined set exists.

**dw:complete-from-sequence** *sequence stream* &key *type* (*name-key* **#'string**) (*value-key* **#'identity**) (*delimiters* **dw::*standard-completion-delimiters***) (*allow-any-input* **nil**) (*enable-forced-return* **nil**) (*initially-display-possibilities* **nil**) (*partial-completers* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter* )
Provides input completion from a sequence of possible completions for input to **accept**. Returned values are the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string.

**dw:complete-input** *stream function* &key (*allow-any-input* **nil**) *enable-forced-return partial-completers* (*type* **nil**) *parser* (*compress-choices* **20**) (*compression-delimiter* ) (*help-offers-possibilities* **t**) (*initially-display-possibilities* **nil**) (*complete-activates* **nil**) (*documenter* **nil**) (*document* **(not (null dw::documenter))**)
Provides input completion for input to **accept**.

**dw:completing-from-suggestions** (*stream* &key (*allow-any-input* **t**) (*delimiters* **dw::*standard-completion-delimiters***) (*enable-forced-return* **nil**) (*partial-completers* **nil**) (*type* **nil**) (*parser* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter* **nil**) (*initially-display-possibilities* **nil**)) &body *body*
Binds local environment to build a completion table for input to **accept**. Three values are returned.

**dw:suggest** *completion-string object*
Adds an element to a completion table being constructed inside a **dw:completing-from-suggestions** macro.

**dw:with-presentation-input-context** (*presentation-type* &rest *options*) (&optional (*blip-var* **'dw::.blip.**)) *non-blip-form* &body *blip-cases*
Binds local environment to the input context of a specified presentation type.

**dw:presentation-type-p** *type*
Returns the presentation type descriptor if its argument is a presentation type, **nil** otherwise.

**dw:presentation-object** *presentation*
Returns the Lisp object represented by a presentation.

**dw:presentation-type** *presentation*
Returns the presentation type of a presentation.

**dw:presentation-equal** *presentation-1 presentation-2*
Determines whether two presentations are "equal", that is, whether they are presenting the same object in the same manner.

**dw:describe-presentation-type** *type* &optional (*stream* ***standard-output***) *plural-count*
Outputs the description of a presentation type provided by the type's definition (**define-presentation-type** macro).

**dw:check-presentation-type-argument** *type-arg* &key (*evaluated* **t**) (*function* **compiler:default-warning-function**) (*definition-type* **compiler:default-warning-definition-type**)
Checks an argument that is expected to be a presentation type for validity.

**dw:with-presentation-type-arguments** (*type-name type*) &body *body*
Binds local environment such that the arguments in a presentation-type specification are lexically available within the body of the macro.

**dw:with-type-decoded** (*type-name-var* &optional *data-args-var presentation-args-var*) *type* &body *body*
Binds local environment such that the type-name and, optionally, arguments in a presentation-type specification are bound to variables lexically available within the body of the macro.

**dw:presentation-type-name** *type*
Returns the name of the presentation type from a presentation-type specification.

**dw:presentation-type-default** *presentation-type*
Returns the current default — the object at the top of the type history — for a presentation type, if the type supports a history; otherwise, it returns **nil**.

## Programming the Mouse: Writing Mouse Handlers

This chapter explains what translating mouse handlers are, what they do, and how to write one.

The first section explains the principles and terminology of translating mouse handlers. The next section then introduces the facilities provided by SemantiCue for writing and using them. This section includes the complete definitions of **define-presentation-translator**, **define-presentation-to-command-translator**, and **define-presentation-action**. The third section presents a collection of suggestions and examples to help you write mouse handlers, and the last section discusses how to investigate and debug presentation problems.

## Mouse Handler Concepts

A translating mouse handler translates an output presentation into an input presentation. When a program is accepting input, the user can use the mouse to supply that input by pointing at a presentation previously output on a window and clicking a mouse gesture. The input is the object previously presented, or some function of that object. For information on low-level mouse handling functions, see the section "Mouse Input".

## Mouse Sensitivity

Mouse sensitivity causes immediate context-sensitive help to be displayed when Genera is accepting input. As the user moves the mouse around the screen, the mouse documentation line describes what would happen if a mouse button were clicked with the mouse at its current position. Any relevant presentation is highlighted by drawing a box around it.

Mouse sensitivity is a function of the current input context, the location of the mouse, and the chord of modifier keys being pressed.

- Input context: a presentation type describing the type of input currently being accepted.

- Mouse location: the mouse is pointing either at a presentation or at a blank area of the screen.

- Modifier keys: control, meta, super, hyper, and shift. These expand the space of available gestures beyond what is available from just three mouse buttons.

## Mouse Handlers

All aspects of mouse sensitivity and mouse input are controlled by mouse handlers. A mouse handler specifies the conditions under which it is applicable, a description to be displayed in the mouse documentation line, and what to do when the handler is invoked by clicking a mouse button. The "relevant" presentation to be highlighted is a presentation that has at least one applicable handler that could be invoked by clicking a mouse button with the mouse at its current location and the modifier keys in their current state. If there is no applicable handler, there is no mouse-sensitivity highlighting.

Each mouse handler has two associated presentation types, its *from-presentation-type* and *to-presentation-type*, which are the primary definition of its applicability. The basic idea is that a mouse handler translates an output presentation into an input presentation. Thus a handler is applicable if the previously-output presentation at which the mouse is pointing matches *from-presentation-type* and the input context matches *to-presentation-type*. Each mouse handler is attached to a particular mouse gesture, which is a combination of a mouse button and a set of modifier keys. Clicking the mouse button while holding down the modifier keys invokes the handler.

*From-presentation-type* is sometimes called *displayed-presentation-type* and *to-presentation-type* is sometimes called *context-presentation-type*.

A mouse handler is either a translator or an action. A translator produces an input presentation, consisting of an object, a presentation type, and some options, to satisfy the program accepting input. The result of a translator might be returned from **accept**, or might be absorbed by a parser and provide part of the input. An input presentation is not actually represented as an object. Instead, a translator's body returns multiple values. The object is the first value. The presentation type is the second value; it defaults to *to-presentation-type* if the body returns only one value. Remaining values after the second are alternating keywords and values for options.

An action does not actually produce any input. Instead, it performs some side effect that will help the user choose the desired input with a second gesture. Examples of actions include popping up a menu of translations and actions, expanding a subdirectory name in a directory listing to show the files contained in the subdirectory, and changing the viewspecs of a presentation to show more or less detail.

## Presentation Type Matching for Mouse Handlers

A mouse handler's *from-presentation-type* matches a presentation at which the mouse is pointing if that presentation's presentation type is a subtype of *from-presentation-type*. Thus a presentation inherits mouse handlers from supertypes of the presentation type. A mouse handler with a *from-presentation-type* of **t** is applicable to all presentations, since **t** is a supertype of every type.

Mouse handler matching also depends on the presented object, in two ways.

First, *from-presentation-type* can be *reduced* to another presentation type that is not a subtype, combined with a predicate that tests the presented object. For example, a handler whose *from-presentation-type* is **(and integer (satisfies oddp))** tentatively matches a presentation of type **integer**, even though **integer** is not a subtype of **(and integer (satisfies oddp))**. The handler is only applicable if the presented integer satisfies the **oddp** predicate. Another example is a handler whose *from-presentation-type* is **(sequence pathname)**, with the mouse pointing at a presentation of type **vector**. **vector** is not a subtype of **(sequence pathname)**, however **(sequence pathname)** reduces to **vector** and a predicate that tests that each element of the vector is a pathname.

Second, certain presentation types are standins for a more specific presentation type determined by the type of the object presented. Handler matching and mouse sensitivity use the more specific type. These presentation types are **sys:expression**, **sys:form**, and **sys:code-fragment**. For instance, values printed by the Lisp Listener are **sys:expression** presentations, but for purposes of mouse sensitivity, handlers whose *from-presentation-type* is a supertype of the actual type of the value are considered, in addition to handlers whose *from-presentation-type* is a supertype of **sys:expression**. For example, a handler whose *from-presentation-type* is **integer** matches a presentation whose type is **sys:expression** if the object presented is of type **integer** or a subtype of **integer**.

A mouse handler's *to-presentation-type* matches the input context if *to-presentation-type* is a subtype of the input context presentation type. In other words, if the handler is a translator, the input presentation that the handler produces must be a member of the presentation type that the program doing input is expecting. A mouse handler with a *to-presentation-type* of **nil** is applicable to all input contexts, since **nil** is a subtype of every type. For historical reasons, **t** as a *to-presentation-type* is a special case and is treated as a synonym for **nil**.

Translator matching also depends on the object returned by the handler. The context type can be *reduced* to another presentation type that is not a subtype, combined with a predicate that tests the object returned by the translator. For example, consider a translator whose *to-presentation-type* is **cp:command** and an input context of **(cp:command :command-table "Global")**. The translator could return any command, but the input context only accepts commands in the global command table. **cp:command** is a supertype of **(cp:command :command-table "Global")**, not a subtype of it; however, the input context type reduces to **cp:command** along with a predicate that tests whether the command is available in the global command table. Thus the translator tentatively matches and its body is executed. If the object returned by the translator satisfies the predicate, the translator is applicable and contributes to mouse sensitivity. If not, the translator is ignored.

Note that, because of type reduction to a type and a predicate, mouse handler matching is not simply

```
(and (dw:presentation-subtypep
        (dw:presentation-type presentation)
        from-presentation-type)
      (dw:presentation-subtypep
       to-presentation-type
        (dw::presentation-input-context-presentation-type
         context)))
```

You can think of **dw:presentation-subtypep** as a first approximation to the mouse handler type matching test, however the actual test is less restrictive.

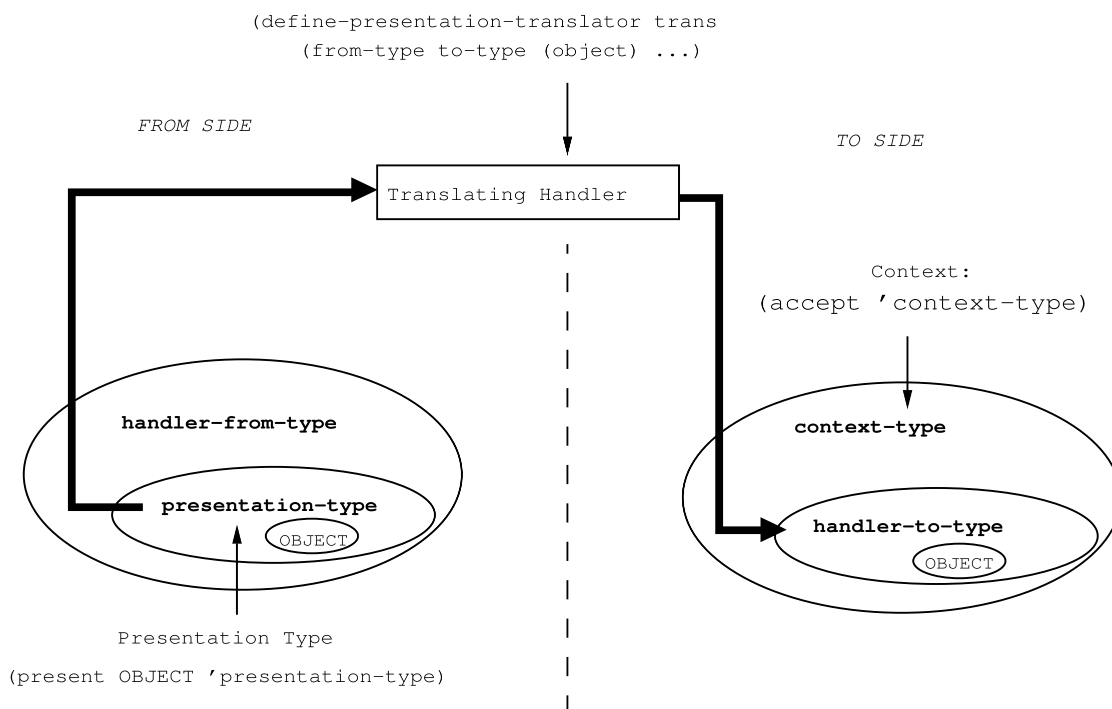



Figure 56. Mouse handler applicability

## Nested Input Contexts

The input context is not simply a presentation type. Input contexts can be nested, so several different context presentation types can be available to match mouse handler *to-presentation-types*. One level of input context is established by calling **accept**, or by calling a higher-level function that in turn calls **accept**. The macros **dw:with-presentation-input-context** and **dw:with-presentation-input-editor-context** also establish a level of input context.

One source of input context nesting is compound objects. For example, when reading the argument to the Show File command, the input context contains **pathname** nested inside of **(sequence pathname)**. Acceptable keyboard input is a sequence of pathnames separated by commas. A mouse handler that translates to a **(sequence pathname)** supplies the entire argument to the command, and the com-

mand processor moves on to the next argument (the keywords). A mouse handler that translates to a **pathname** is also applicable. It supplies just one element of the sequence being built up, and the command processor awaits additional input for this argument, or entry of a Space or Return to terminate the argument.

A second source of input context nesting is dynamic nesting of program levels, such as when a presentation type's parser calls **accept**. For example, the complete set of nested input contexts when reading the argument to the Show File command, from the inside out, are

```
((dw::in-band-menu) :name "Menu of completions")
pathname
si:input-editor
((sequence pathname))
si:input-editor
```

The **pathname** and **(sequence pathname)** contexts are associated with accepting an object. Mouse handlers for these contexts involve files and pathnames. The **si:input-editor** contexts are associated with accepting a sequence of characters that can be edited and parsed. Mouse handlers for **si:input-editor** context involve marking and yanking text. The **dw::in-band-menu** context is associated with pathname completion; a mouse handler for this context pops up a menu of possible completions. Pathname acceptance, completion, and input editing are three different levels of program.

Another example is the nested input contexts established by **(accept 'cons)**. From the inside out, these are

```
(and ((sys:expression)) (satisfies listp))
si:input-editor
cons
si:input-editor
```

The **and** context is present because the parser for **cons** (inherited from **list**) calls the general expression parser with a restriction that the result must be a list.

When there are multiple nested contexts, mouse sensitivity considers only the innermost context that has any applicable mouse handlers for the currently pressed chord of modifier keys. Contexts nested outside that one contribute only to the list of other available modifier chords displayed in the second line of the mouse documentation line.

### Nested Presentations

There can be more than one presentation at the mouse location, since presentations can overlap on the screen. Normally, when two presentations overlap one is nested inside the other. One cause of nesting is presentations of compound objects. For example, a presentation of a list encloses presentations of its elements, and a presentation of a formatted table encloses presentations of its content cells. Another cause of nesting is presentations that consist of other presentations. For example, the printer for a **sys:function-spec** presentation calls **prin1** recursively, which produces a **sys:expression** presentation. The **sys:expression** is nested inside the

**sys:function-spec**; unlike the case of a list, both presentations occupy the same region of the window. A third cause of nesting is raw-text presentations; nested inside that **sys:expression** presentation is a presentation of the string of characters displayed as the printed representation of the expression. The priority for mouse selection of graphics presentations, by the way, is the same as the temporal priority used to determine how to refresh overlapping graphic objects.

When there is more than one candidate presentation at the mouse location, SemantiCue must decide which presentation is the sensitive one. It starts with the innermost presentation at the mouse location and works outwards through levels of nesting until a sensitive presentation is discovered. This is the innermost presentation that has any applicable mouse handlers, in any of the nested input contexts, for the currently pressed chord of modifier keys. Next SemantiCue checks for the case of nested presentations that occupy exactly the same region of the window, as in the **sys:function-spec** example above. If the sensitive presentation is nested in this way, and its containing presentation is sensitive in the same context, or in a context nested inside that one, the containing presentation becomes the sensitive presentation and the search continues. Continuing the search in this way ensures that a more specific presentation is sensitive, for example, **sys:function-spec** rather than **sys:expression**. Note that nested input contexts are searched first, before nested presentations.

The above algorithm is slightly modified to try to choose a more appropriate sensitive presentation: mouse handlers with certain options do not receive equal consideration when the sensitive presentation is being chosen.

**:context-independent t**

Context-independent mouse handlers do not affect the choice of sensitive presentation and sensitive context, unless there are no applicable context-*de*pendent mouse handlers. In that case, the innermost presentation sensitive to any context-independent mouse handlers for the currently pressed chord of modifier keys becomes the sensitive presentation. Use this option for handlers that should be available in all contexts, with a *to-presentation-type* of **t**. The presentation debugging menu (on **#\s-Mouse-right**) is a good example.

**:suppress-highlighting t**

This handler does not affect the choice of sensitive presentation. Use this option for handlers that do not apply to any particular presentation: the system menu (on **#\sh-Mouse-right**) and the window operations menu (on **#\m-sh-Mouse-right**) are good examples. The *from-presentation-type* is almost always **t** when you use this option.

**:suppress-highlighting :defines-menu**

This handler interacts with the value of **:context-independent** and also depends on the value of **:defines-menu**. It is used to prevent the highlighting of some presentations under a certain set of circumstances. The complete description of how this

works is included in the documentation of the **:suppress-highlighting** option for the various translator-definition macros. (See the function **define-presentation-translator**.)

The raw text characters underlying textual (as opposed to graphical) presentations have presentation type **dw:raw-text**. The object is a list whose first element is a string and whose second element is the position of a character in that string.

You can write mouse handlers that apply to blank areas of the window, where there are no presentations. Use **dw:no-type** as the *from-presentation-type* and specify the **:blank-area t** option. There is no highlighting when such a mouse handler is applicable, since there is no presentation to highlight.

### Mouse Gestures

Each mouse handler is attached to a particular mouse gesture, which is a combination of a mouse button and a set of modifier keys. Mouse gestures are named by keyword symbols. The use of mouse gestures, rather than mouse characters such as **#\m-c-sh-Mouse-Middle**, provides a level of indirection that makes it possible to customize the user interface by changing the mapping between mouse gestures and mouse characters. It also allows the possibility of adaptation of tablets and single-button mice.

There are two kinds of mouse gesture name, logical and physical. A physical gesture name looks like **:meta-control-shift-middle**. It describes which keys and button are pressed to create the gesture. A logical gesture name looks like **:edit-definition**. It describes the type of operation invoked by the gesture. The advantage of using logical gesture names is that it encourages a more consistent user interface by ensuring that similar operations in different contexts are on the same gesture. This makes the system easier to learn and explore. Most programs should use logical gesture names in preference to physical gesture names. The physical names are sometimes appropriate when the desired user interface is defined in terms of actual buttons rather than in terms of consistency with the rest of the system.

The special mouse gesture name **nil** is used in handlers that are not directly invokable by mouse gesture. Such a handler can only be invoked from a menu. The **:menu** option should be used to specify which menu; the default is the standard click-right menu.

The special mouse gesture name **t** means that the handler is available on every gesture.

The following logical mouse gesture names are defined in Genera. You can add your own, using **(setf (dw:mouse-char-for-gesture** *symbol***)** *mouse-char***)**. The mapping of these logical gesture names to mouse characters follows the following conventions.

The left button is generally used for selection. The right button is generally used for menus. The shift key is used to simulate the existence of six buttons instead of three and does not have any significance of its own. Gestures with no modifiers other than shift are used for the most common operations specific to a particular application program. The meta key provides additional operations, often for a larger presentation. Meta-shift is for commands that apply to the whole window. The control key is for text marking and yanking commands. Control and meta together are for debugging-related commands. The super key is for commands related to presentations themselves, rather than to the objects presented. The hyper key is reserved for customer use.

### Actions Versus Translations

*Translations* return values, but *actions* do not — they cause *side effects*. Normally, the purpose of the side effect should be to aid in getting some more input. An action, for example, can add a menu of possibilities.

Actions are almost always defined for the **t** context, since they are not returning values for anything. Actions must always have **:documentation**.

### The Facilities

This section introduces the facilities for writing mouse handlers and controlling their application: macros for writing the handlers, input context and input blip facilities, and functions for using mouse gestures.

### Mouse Handler Facilities

A large number of predefined mouse handlers are already included in SemantiCue. Clicking Right on a displayed presentation in a Dynamic Lisp Listener throws up a menu of handlers applicable to the presentation object.

You define your own, application-specific handlers using these definition macros and their adjuncts:

**define-presentation-translator**
**define-presentation-to-command-translator**
**define-presentation-action**
**dw:handler-applies-in-limited-context-p**
**dw:presentation-subtypep**
**dw:delete-presentation-mouse-handler**

### The define-presentation-translator Macro

| *Logical Gesture Name* | *Physical Gesture* | *Typical Use* |
|---|---|---|
| :select | Left | Select the presented object |
| :describe | Middle | Describe the presented object |
| :select-and-edit | Middle | Edit an editable field |
| :menu | Right | Pop up a menu |
| :alternate-select | Shift-Left | Select in a modified way |
| :select-and-activate | Shift-Left | |
| :inspect | Shift-Middle | |
| :delete | Shift-Middle | |
| :remove | Shift-Middle | |
| :system-menu | Shift-Right | Pop up the system menu |
| :hold-and-mark-region | Control-Left | Highlight some text |
| :yank-word | Control-Middle | Copy text into input editor |
| :marking-and-yanking-menu | Control-Right | Pop up a menu of text operations |
| :mark-word | Control-Shift-Middle | Highlight some text |
| :edit-definition | Meta-Left | |
| :edit-function | Meta-Left | |
| :evaluate-form | Meta-Middle | |
| :disassemble | Meta-Middle | |
| :window-operation-menu | Meta-Shift-Right | Pop up a menu of window operations |
| :set-breakpoint | Control-Meta-Left | Breakpoint if instruction executed |
| :clear-breakpoint | Control-Meta-Middle | |
| :set-complex-breakpoint | Control-Meta-Right | |
| :modify | Control-Meta-Right | Change contents of presented location |
| :monitor-location | Control-Meta-Shift-Left | Breakpoint if location modified |
| :unmonitor-location | Control-Meta-Shift-Middle | |
| :select-object | Super-Left | Forcibly select the presented object |
| :describe-presentation | Super-Middle | |
| :presentation-debugging-menu | Super-Right | |
| :reprint-differently | Super-Shift-Left | Change viewspecs heuristically |
| :edit-viewspecs | Super-Shift-Middle | Change viewspecs via menu |

**define-presentation-translator** *name* (*from-presentation-type to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose*) *arglist* &body *body* *Function*

Defines a mouse handler that translates a displayed presentation into an input object. Typically, the "translation" is a matter of extracting a nested object, for example, a host object from a pathname object.

*name* The name of the handler.

*from-presentation-type*
The type of the displayed presentation.

*to-presentation-type*
The presentation type of the returned object.

**:tester** Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument — the current presentation object — and a subset of the keywords:

| | |
|---|---|
| **:presentation** | The presentation at the mouse location that matches *from-presentation-type* |
| **:presentation-type** | The presentation type of that presentation |
| **:input-context** | The input context that matches *to-presentation-type* |
| **:gesture** | The mouse gesture that could invoke the handler |
| **:mouse-char** | The corresponding mouse character |
| **:window** | The window containing the presentation |
| **:handler** | The handler itself |

These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are also documented under *arglist* in the handler documentation; they are also documented separately ( see the function **define-presentation-action**.).

Note: inefficient testers can degrade the performance of your program. Tester functions must be capable of rapid execution. Also, do not use the body of your handler as an implicit tester if it does a large amount of consing or in other ways consumes resources; this will similarly affect program performance. For more information, see the section "Some Efficiency Caveats for Mouse Handlers".

For functions used in **:tester**s, see the function **dw:handler-applies-in-limited-context-p**. See the function **dw:presentation-subtypep**.

**:gesture** Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. See the section "Mouse Gestures". The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures.

**:documentation**

Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

The argument to :documentation can be a list of the form ((object) . body), where object is bound to the presentation-object of the presentation to which the mouse points, and body is a form referencing this object.

**:suppress-highlighting**

Suppresses the contribution of a mouse handler to mouse-sensitivity highlighting, in circumstances that depend on the value of the option. A mouse handler whose highlighting is suppressed will not affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. The choice of sensitive presentation and context is determined by other applicable mouse handlers. Once that choice has been made, if the mouse handler whose highlighting is suppressed is applicable for the chosen presentation, context, and modifier keys, then this handler is available too.

When the **:context-independent t** and **:suppress-highlighting** options are used together, **:suppress-highlighting** only controls whether or not the sensitive presentation is highlighted. When context-dependent mouse handlers are applicable for the currently pressed modifier keys, context-independent mouse handlers never affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. If only context-independent mouse handlers are applicable for the currently pressed modifier keys, the innermost presentation with any applicable handlers is sensitive, and it is highlighted unless all of those handlers have their highlighting suppressed.

Note that **:suppress-highlighting** is not used with a **:blank-area t** mouse handler. When a blank area is mouse-sensitive, there is never any highlighting, since there is no presentation to be highlighted.

The possible values of the **:suppress-highlighting** option are:

**nil** Highlighting is never suppressed. This is the default.

**t** Highlighting is always suppressed. Use this option for handlers that do not apply to any particular presentation; the system menu and the window operations menu are good examples. The *from-presentation-type* is generally **t** when you use this option.

**:defines-menu** Use this only with a mouse handler that has the **:defines-menu** *name-of-menu* option and does not have the **:context-independent t** option. Highlighting is suppressed unless there are no applicable mouse handlers for the currently pressed modifier keys whose highlighting is not suppressed. In that case, **:suppress-highlighting :defines-menu** handlers are considered. If any of these define menus containing "interesting" items, the "interesting" menu for the innermost presentation and context is used and that presentation is highlighted. If not, the "uninteresting" menu for the innermost presentation and context is used, but nothing is highlighted. A menu item is "uninteresting" if it has the **:defines-menu** or **:suppress-highlighting** option with a non-null value.

Use **:suppress-highlighting :defines-menu** to prevent a menu from interfering with the choice of the appropriate sensitive presentation and input context, which will be based on applicable handlers on mouse gestures that use the same chord of modifier keys but other buttons. The standard menu on mouse-right is a good example.

**:within-menu** Highlighting is not suppressed when the mouse handler is directly available on a gesture. The only effect of this option is to make this handler "uninteresting" when it appears in a menu defined by a **:suppress-highlighting :defines-menu** handler. This prevents this handler from making an uninteresting presentation be highlighted by a menu. Use this for very general mouse handlers, for example, "Edit Viewspecs". The heading of a directory listing is sensitive for "R: Menu", but the menu that is popped up

contains only "Edit Viewspecs" and some other menus. Thus **:suppress-highlighting :within-menu** is used on "Edit Viewspecs" to prevent the directory listing from being highlighted, which would be distracting. The directory listing is still highlighted for "Edit Viewspecs" when the super and shift modifier keys are pressed.

**:menu** Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu. **nil** means do not include the handler in any menu.

You can define you own handler menu with **define-presentation-action**: See the function **define-presentation-action**.

**:context-independent**

Boolean option specifying whether handler behavior (that is, applicability to displayed presentations) is the same for all contexts in a nested-context structure (**accept** being called recursively); the default is **nil**.

This option is supplied with **t**, for example, if the handler's *to-presentation-type* is **t** (any context), and its contract is to print additional information about a particular presentation (that is, only the output matters).

**:context-independent** should be set to **t** only if the *to-presentation-type* is **t** and when the **:do-not-compose** option has been set to **t**. If either of these does not hold, the compiler will issue a warning if you try to set **:context-independent** to **t**.

For more information on context matching and related handler issues, see the section "How Mouse Handlers Are Found".

**:priority** Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on two factors: the presentation type of the presentation and the current input context.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one for that gesture is determined by handler precedence or priority. The system automatically assigns priorities according to the matching factors as follows: the priority is incremented by 4 when the presentation type matches; and by 2 when the context type matches. (Note: in the case of priority assignment, "matches" means "having the identical presentation-type name" — contrary to the meaning of this term in the general case, where it connotes hierarchical type relationship.)

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (`Q:>rel-7>sys>doc>uims>ui-dict2.sar`) and a **sys:expression** presentation (`#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"`).

Two handlers defined on the **:select** gesture are applicable to both presentations in top-level command context. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

| | Pathname Presentation `Q:>rel-7>sys>doc>...` | Expression Presentation `#P"Q:>rel-7>sys>doc>...` |
|---|---|---|
| Show File | 4 | 0 |
| Quoted Expression | n/a | 5.5 |

It was the system programmer's intent that the quoted expression handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the value of the **:priority** option was made `1.5`. This is added to the system-generated priority of `4` in the bottom right cell of the table for a total score of `5.5`, enough to give this handler precedence even over another handler for **sys:expression** that would have priority 4.

**:exclude-other-handlers**

Boolean option, used with `:gesture t` handlers, specifying whether to exclude non-**t** handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item". If the menu item includes the **:no-select** keyword, the translator does not apply. But, if

`:exclude-other-handlers t` were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (`Mouse-R`) gesture.

**:exclude-other-handlers** provides a way of saying "this translator implements the entire contract for the presentation it matches".

**:blank-area**

Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil**.

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the *from-presentation-type* positional argument to the handler.

**:do-not-compose**

Boolean option specifying when **t** that the value of *body* is not to be computed to determine if the handler satisfies the current input context. The default is **nil**: the body is computed.

To see the need for this option, consider the default behavior. For example, if 1) you have a translating mouse handler that returns integer objects; 2) the mouse cursor is currently over the handler's *from-presentation-type* ; 3) any shift keys modifying the mouse gesture the handler is on are pressed; and 4) the current input context is for integers, the default system behavior would be to determine what the body of the handler returns. If it returns anything other than a single value of **nil**, then the handler is applicable; this fact is indicated in the mouse documentation line and the presentation is highlighted (if it's not already).

Now, if the input context in this situation was for odd integers, rather than for any integer — that is, `(accept '((and integer ((satisfies oddp)))))` — by default this handler would still be run to see if it returns an *odd* integer, that is, that the returned object will satisfy the input context requirements. Only if this is the case will the handler be available. This is the motivation for the default behavior.

However, some translating handlers have side effects, for example, popping up a menu or asking a question. It is unlikely that you want such events occurring merely when a user of your program waves the mouse over a presentation. You want this behavior suppressed until the user actually clicks on the presentation. `:do-not-compose t` is how you express this intent.

As a general rule, avoid defining translators that have side effects. One way of doing this is by defining side-effecting handlers explicitly, with **define-presentation-action**. Another way is to make the

translator produce a command and put the side effects in the body of the command.

*arglist* The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body.

**:input-context**
The current presentation-input context.

**:presentation**
The presentation instance that the mouse cursor is over.

**:handler** The handler object of which the body is a part.

**:mouse-char**
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

**:window** The window object in which the current presentation occurs.

**:x** The x-coordinate of the mouse cursor when the mouse was clicked.

**:y** The y-coordinate of the mouse cursor when the mouse was clicked.

**:gesture** The mouse gesture (symbolic name) this handler is on.

The parameter list can specify only those keywords that are explicitly used, for example, `(object &key window x y)`.

The *body* of your handler must return at least one value, the object. Optionally, it can return the presentation type of its result, which defaults to *to-presentation-type* if only one value is returned. Also optionally, it can also return keyword-value pairs that you define. In this case, you must return the presentation type of the object as well. The object is the first item returned, its presentation type the second; these are followed by the keyword-value pairs. If the desired object is **nil**, you must return two values, since a single value of **nil** means the mouse handler does not apply.

One predefined keyword is available, **:activate**. Supplied with **nil**, the activation of input entered via this handler is suppressed, with **t** it's promoted. The following example is taken from the system code:

```
(define-presentation-translator command-name-to-command
   (cp:command-name cp:command)
   (command-name)
  (values
    `(,command-name) 'cp:command :activate nil))
```

This translator allows commands displayed as command-name presentations — for example, in the display generated when you press HELP after entering the first word of a command to the command processor prompt — to be used as command object input. Because :activate nil is provided, the command is not executed immediately after clicking on its name; the user must press RETURN to activate the command. This allows the opportunity to enter arguments.

The values returned by the translator will be used to construct a presentation blip. You do not make the blip; the handler takes care of this automatically. Any keywords the translator returns are included in the options field of the blip. Options can be extracted from blips with the **dw:presentation-blip-options** function. For an overview of this and related functions, see the section "Presentation Input Blip Facilities".

For an overview of **define-presentation-translator** and related facilities, see the section "Programming the Mouse: Writing Mouse Handlers". For information on handler lookup and performance issues, see the section "How Mouse Handlers Are Found".

Here is an example that defines a translating handler to extract the version number, an **integer** object, from a **pathname** presentation. Users have the options of typing in a version number to the input prompt or clicking on a **pathname** presentation that included a version number.

```
(define-presentation-translator pathname-version
    (pathname integer  ;From pathname to integer
     :documentation "Return file version number"
     :gesture :middle
     ;; Only works for pathnames with numeric versions
     :tester ((path) (integerp (pathname-version path))))
    (path)
 (pathname-version path))

(present #P"KOALA:>KJones>foo.lisp.17")
(accept 'integer)
```

After compiling this translator, try doing a Show Directory listing, then evaluate (accept 'integer). In this input context, move the mouse cursor over one of the pathnames and notice that the top mouse documentation line now says Mouse-M: Return file version number; Mouse-R: Menu. Clicking Middle enters the file version number as an integer object.

### The define-presentation-to-command-translator Macro

The **define-presentation-to-command-translator** macro creates handlers for performing a single kind of translation: from presentations to Command Processor commands.

**define-presentation-to-command-translator** *name* (*presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose*) *arglist* &body *body* *Function*

Defines a mouse handler that translates from a displayed presentation to a list whose first element is a command function name and whose remaining elements are argument values.

*name* The name of the handler. Usually, you give the handler the same name as the Command Processor command.

*presentation-type*
The type of the displayed presentation for which the handler is intended.

**:tester** Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument — the current presentation object — and a subset of the keywords:

| | |
|---|---|
| **:presentation** | The presentation at the mouse location that matches *from-presentation-type* |
| **:presentation-type** | The presentation type of that presentation |
| **:input-context** | The input context that matches *to-presentation-type* |
| **:gesture** | The mouse gesture that could invoke the handler |
| **:mouse-char** | The corresponding mouse character |
| **:window** | The window containing the presentation |
| **:handler** | The handler itself |

These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are also documented under *arglist* in the handler documentation; they are also documented separately ( see the function **define-presentation-action**.).

Note: inefficient testers can degrade the performance of your program. Tester functions must be capable of rapid execution. Also, do not use the body of your handler as an implicit tester if it does a

large amount of consing or in other ways consumes resources; this will similarly affect program performance. For more information, see the section "Some Efficiency Caveats for Mouse Handlers".

For functions used in **:tester**s, see the function **dw:handler-applies-in-limited-context-p**. See the function **dw:presentation-subtypep**.

**:gesture** Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. See the section "Mouse Gestures". The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures.

**:documentation**

Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

The argument to :documentation can be a list of the form ((object) . body), where object is bound to the presentation-object of the presentation to which the mouse points, and body is a form referencing this object.

**:suppress-highlighting**

Suppresses the contribution of a mouse handler to mouse-sensitivity highlighting, in circumstances that depend on the value of the option. A mouse handler whose highlighting is suppressed will not affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. The choice of sensitive presentation and context is determined by other applicable mouse handlers. Once that choice has been made, if the mouse handler whose highlighting is suppressed is applicable for the chosen presentation, context, and modifier keys, then this handler is available too.

When the **:context-independent t** and **:suppress-highlighting** options are used together, **:suppress-highlighting** only controls whether or not the sensitive presentation is highlighted. When context-dependent mouse handlers are applicable for the currently pressed modifier keys, context-independent mouse handlers never affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. If only con-

text-independent mouse handlers are applicable for the currently pressed modifier keys, the innermost presentation with any applicable handlers is sensitive, and it is highlighted unless all of those handlers have their highlighting suppressed.

Note that **:suppress-highlighting** is not used with a **:blank-area t** mouse handler. When a blank area is mouse-sensitive, there is never any highlighting, since there is no presentation to be highlighted.

The possible values of the **:suppress-highlighting** option are:

**nil**
: Highlighting is never suppressed. This is the default.

**t**
: Highlighting is always suppressed. Use this option for handlers that do not apply to any particular presentation; the system menu and the window operations menu are good examples. The *from-presentation-type* is generally **t** when you use this option.

**:defines-menu**
: Use this only with a mouse handler that has the **:defines-menu** *name-of-menu* option and does not have the **:context-independent t** option. Highlighting is suppressed unless there are no applicable mouse handlers for the currently pressed modifier keys whose highlighting is not suppressed. In that case, **:suppress-highlighting :defines-menu** handlers are considered. If any of these define menus containing "interesting" items, the "interesting" menu for the innermost presentation and context is used and that presentation is highlighted. If not, the "uninteresting" menu for the innermost presentation and context is used, but nothing is highlighted. A menu item is "uninteresting" if it has the **:defines-menu** or **:suppress-highlighting** option with a non-null value.

  Use **:suppress-highlighting :defines-menu** to prevent a menu from interfering with the choice of the appropriate sensitive presentation and input context, which will be based on applicable handlers on mouse gestures that use the same chord of modifier keys but other buttons. The standard menu on mouse-right is a good example.

**:within-menu**
: Highlighting is not suppressed when the mouse handler is directly available on a gesture. The

only effect of this option is to make this handler "uninteresting" when it appears in a menu defined by a **:suppress-highlighting :defines-menu** handler. This prevents this handler from making an uninteresting presentation be highlighted by a menu. Use this for very general mouse handlers, for example, "Edit Viewspecs". The heading of a directory listing is sensitive for "R: Menu", but the menu that is popped up contains only "Edit Viewspecs" and some other menus. Thus **:suppress-highlighting :within-menu** is used on "Edit Viewspecs" to prevent the directory listing from being highlighted, which would be distracting. The directory listing is still highlighted for "Edit Viewspecs" when the super and shift modifier keys are pressed.

**:menu** Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu. **nil** means do not include the handler in any menu.

You can define you own handler menu with **define-presentation-action**: See the function **define-presentation-action**.

**:priority** Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on two factors: the presentation type of the presentation and the current input context.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one for that gesture is determined by handler precedence or priority. The system automatically assigns priorities according to the matching factors as follows: the priority is incremented by 4 when the presentation type matches; and by 2 when the context type matches. (Note: in the case of priority assignment, "matches" means "having the identical presentation-type name" — contrary to the meaning of this term in the general case, where it connotes hierarchical type relationship.)

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (Q:>rel-7>sys>doc>uims>ui-dict2.sar) and a **sys:expression** presentation (#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest").

Two handlers defined on the **:select** gesture are applicable to both presentations in top-level command context. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

| | Pathname Presentation Q:>rel-7>sys>doc>... | Expression Presentation #P"Q:>rel-7>sys>doc>... |
|---|---|---|
| Show File | 4 | 0 |
| Quoted Expression | n/a | 5.5 |

It was the system programmer's intent that the quoted expression handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the value of the **:priority** option was made 1.5. This is added to the system-generated priority of 4 in the bottom right cell of the table for a total score of 5.5, enough to give this handler precedence even over another handler for **sys:expression** that would have priority 4.

**:exclude-other-handlers**

Boolean option, used with :gesture t handlers, specifying whether to exclude non-**t** handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item". If the menu item includes the **:no-select** keyword, the translator does not apply. But, if :exclude-other-handlers t were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (Mouse-R) gesture.

**:exclude-other-handlers** provides a way of saying "this translator implements the entire contract for the presentation it matches".

**:blank-area**
Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil**.

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the *from-presentation-type* positional argument to the handler.

*arglist* The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body.

**:input-context**
The current presentation-input context.

**:presentation**
The presentation instance that the mouse cursor is over.

**:handler** The handler object of which the body is a part.

**:mouse-char**
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

**:window** The window object in which the current presentation occurs.

**:x** The x-coordinate of the mouse cursor when the mouse was clicked.

**:y** The y-coordinate of the mouse cursor when the mouse was clicked.

**:gesture** The mouse gesture (symbolic name) this handler is on.

The parameter list can specify only those keywords that are explicitly used, for example, `(object &key window x y)`.

The *body* of your translator must return at least one value, the list of command name and argument values. Optionally, it can also return keyword-value pairs that you define. In this case, you must return the presentation type of the object as

well. The object is the first item returned, its presentation type the second; these are followed by the keyword-value pairs. Here is an example that translates from a blank area of the screen to a command that draws a circle (providing that you have defined a command com-add-circle — See the section "Incremental Redisplay of Graphics".)

```
(define-presentation-to-command-translator add-circle-here
   (dw:no-type :documentation "Add a circle here.")
   (ignore &key x y)
  `(com-add-circle ,x ,y))
```

One predefined keyword is available, **:activate**. Supplied with **nil**, the activation of input entered via this handler is suppressed, with **t** it's promoted. For an example: See the function **define-presentation-translator**.

The values returned by the translator will be used to construct a presentation blip. You do not make the blip; the handler takes care of this automatically. Any keywords the translator returns are included in the options field of the blip. Options can be extracted from blips with the **dw:presentation-blip-options** function. For an overview of this and related functions: See the section "Presentation Input Blip Facilities".

The following example is taken from the system source. It defines the Delete File presentation-to-command translator:

```
(define-presentation-to-command-translator si:com-delete-file
   (fs:pathname
     :gesture nil)
   (path)
  `(si:com-delete-file ,(list path)))
```

Note the use of the backquoted form `'(si:com-delete-file ,(list path))` in the body of this translator. This is the recommended way of interfacing to Command Processor commands from presentation-to-command-translators. Note also that the **:gesture** option to the translator is **nil**. This means that the translator is not available on any gesture, but only in the click right menu available for all presentations.

### The define-presentation-action Macro

*Side-effecting mouse handlers*, the kind you create with **define-presentation-action**, are run while your program is waiting for input, but do not themselves supply input. Rather, they run code outside the main control loop of your program to accomplish some action that is useful relative to the presentation which activates them.

A common use for side-effecting handlers is to display additional information about some presentation. For example, if your program is providing graphic presentations of several key variables, it may be the case that to select one of the variables to use as input, your user will require more information about the variables than can be included in the graphic representations. A side-effecting mouse handler could be used at this point to provide a display of all pertinent information about each of the available objects.

A major use made of side-effecting handlers by SemantiCue is to display menus of other handlers. The standard click-right menu for presentations, which shows handlers available in the current input context for the presentation at hand, is implemented in this fashion. Such handlers are created by specifying the **:defines-menu** option to **define-presentation-action**.

**define-presentation-action** *name* (*from-presentation-type* *to-presentation-type* &key *tester* (*gesture* **:select**) *documentation* *suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority* *exclude-other-handlers* *blank-area* *defines-menu*) *arglist* &body *body* *Function*

Defines a side-effecting mouse handler for performing actions on a displayed presentation that are independent of the main body and command loop of an application.

*name* The name of the handler.

*from-presentation-type*
The type of the displayed presentation.

*to-presentation-type*
A presentation type. This argument establishes the input context in which the handler is active. The value usually supplied is **t**, meaning that the handler is potentially available in any input context.

**:tester** Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument — the current presentation object — and a subset of the keywords:

| | |
|---|---|
| **:presentation** | The presentation at the mouse location that matches *from-presentation-type* |
| **:presentation-type** | The presentation type of that presentation |
| **:input-context** | The input context that matches *to-presentation-type* |
| **:gesture** | The mouse gesture that could invoke the handler |
| **:mouse-char** | The corresponding mouse character |
| **:window** | The window containing the presentation |
| **:handler** | The handler itself |

These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are also documented under *arglist* in the handler documentation; they are also document-

ed separately: See the function **define-presentation-action**. Note: inefficient testers can degrade the performance of your program. Tester functions must be capable of rapid execution. For more information: See the section "Some Efficiency Caveats for Mouse Handlers".

For functions used in **:tester**s: See the function **dw:handler-applies-in-limited-context-p**.

See the function **dw:presentation-subtypep**.

**:gesture** Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. See the section "Mouse Gestures". The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures.

**:documentation**

Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

The argument to :documentation can be a list of the form ((object) . body), where object is bound to the presentation-object of the presentation to which the mouse points, and body is a form referencing this object.

**:suppress-highlighting**

Suppresses the contribution of a mouse handler to mouse-sensitivity highlighting, in circumstances that depend on the value of the option. A mouse handler whose highlighting is suppressed will not affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. The choice of sensitive presentation and context is determined by other applicable mouse handlers. Once that choice has been made, if the mouse handler whose highlighting is suppressed is applicable for the chosen presentation, context, and modifier keys, then this handler is available too.

When the **:context-independent t** and **:suppress-highlighting** options are used together, **:suppress-highlighting** only controls whether or not the sensitive presentation is highlighted. When con-

text-dependent mouse handlers are applicable for the currently pressed modifier keys, context-independent mouse handlers never affect the choice of which of the nested presentations is sensitive, nor the choice of which of the nested contexts is sensitive. If only context-independent mouse handlers are applicable for the currently pressed modifier keys, the innermost presentation with any applicable handlers is sensitive, and it is highlighted unless all of those handlers have their highlighting suppressed.

Note that **:suppress-highlighting** is not used with a **:blank-area t** mouse handler. When a blank area is mouse-sensitive, there is never any highlighting, since there is no presentation to be highlighted.

The possible values of the **:suppress-highlighting** option are:

**nil** Highlighting is never suppressed. This is the default.

**t** Highlighting is always suppressed. Use this option for handlers that do not apply to any particular presentation; the system menu and the window operations menu are good examples. The *from-presentation-type* is generally **t** when you use this option.

**:defines-menu** Use this only with a mouse handler that has the **:defines-menu** *name-of-menu* option and does not have the **:context-independent t** option. Highlighting is suppressed unless there are no applicable mouse handlers for the currently pressed modifier keys whose highlighting is not suppressed. In that case, **:suppress-highlighting :defines-menu** handlers are considered. If any of these define menus containing "interesting" items, the "interesting" menu for the innermost presentation and context is used and that presentation is highlighted. If not, the "uninteresting" menu for the innermost presentation and context is used, but nothing is highlighted. A menu item is "uninteresting" if it has the **:defines-menu** or **:suppress-highlighting** option with a non-null value.

Use **:suppress-highlighting :defines-menu** to prevent a menu from interfering with the choice of the appropriate sensitive presentation and input context, which will be based on applicable handlers on mouse gestures that use the same chord of modifier keys but other buttons. The

standard menu on mouse-right is a good example.

**:within-menu** Highlighting is not suppressed when the mouse handler is directly available on a gesture. The only effect of this option is to make this handler "uninteresting" when it appears in a menu defined by a **:suppress-highlighting :defines-menu** handler. This prevents this handler from making an uninteresting presentation be highlighted by a menu. Use this for very general mouse handlers, for example, "Edit Viewspecs". The heading of a directory listing is sensitive for "R: Menu", but the menu that is popped up contains only "Edit Viewspecs" and some other menus. Thus **:suppress-highlighting :within-menu** is used on "Edit Viewspecs" to prevent the directory listing from being highlighted, which would be distracting. The directory listing is still highlighted for "Edit Viewspecs" when the super and shift modifier keys are pressed.

**:menu** Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu. **nil** means do not include the handler in any menu.

You can define you own handler menu with **define-presentation-action**: See the function **define-presentation-action**.

**:context-independent**
Boolean option specifying whether handler behavior (that is, applicability to displayed presentations) is the same for all contexts in a nested-context structure (**accept** being called recursively); the default is **nil**.

This option is supplied with **t**, for example, if the handler's *to-presentation-type* is **t** (any context), and its contract is to print additional information about a particular presentation (that is, only the output matters).

**:context-independent** should be set to **t** only if the *to-presentation-type* is **t**. If this does not hold, the compiler will issue a warning if you try to set **:context-independent** to **t**.

For more information on context matching and related handler issues: See the section "How Mouse Handlers Are Found".

**:priority** Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on two factors: the presentation type of the presentation and the current input context.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one for that gesture is determined by handler precedence or priority. The system automatically assigns priorities according to the matching factors as follows: the priority is incremented by 4 when the presentation type matches; and by 2 when the context type matches. (Note: in the case of priority assignment, "matches" means "having the identical presentation-type name" — contrary to the meaning of this term in the general case, where it connotes hierarchical type relationship.)

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (`Q:>rel-7>sys>doc>uims>ui-dict2.sar`) and a **sys:expression** presentation (`#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"`).

Two handlers defined on the **:select** gesture are applicable to both presentations in top-level command context. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

| | Pathname Presentation `Q:>rel-7>sys>doc>...` | Expression Presentation `#P"Q:>rel-7>sys>doc>...` |
|---|---|---|
| Show File | 4 | 0 |
| Quoted Expression | n/a | 5.5 |

It was the system programmer's intent that the quoted expression handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the

value of the **:priority** option was made `1.5`. This is added to the system-generated priority of `4` in the bottom right cell of the table for a total score of `5.5`, enough to give this handler precedence even over another handler for **sys:expression** that would have priority 4.

**:exclude-other-handlers**

Boolean option, used with `:gesture t` handlers, specifying whether to exclude non-**t** handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item". If the menu item includes the **:no-select** keyword, the translator does not apply. But, if `:exclude-other-handlers t` were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (`Mouse-R`) gesture.

**:exclude-other-handlers** provides a way of saying "this translator implements the entire contract for the presentation it matches".

**:blank-area**

Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil**.

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the *from-presentation-type* positional argument to the handler.

**:defines-menu**

Specifies the handler menu that this handler invokes. That is, when this option is specified, it means that this handler is to produce a menu of other handlers that apply to the presentation at hand. Other handlers are included in this menu by specifying their **:menu** options with the menu named by **:defines-menu**.

The following example is for the `Presentation debugging menu`, available on s-Mouse-R for all presentations, in all input contexts (both the *from-* and *to-presentation-type*s are **t**):

```
(define-presentation-action presentation-debugging-menu
   (t t
      :documentation "Presentation debugging menu"
      :gesture :presentation-debugging-menu
      :menu (t :style (nil :italic nil))
      :defines-menu :presentation-debugging
      :context-independent t
      :blank-area t)
   (ignore &rest args)
  (return-from presentation-debugging-menu
    (apply #'dw:call-presentation-menu
           :presentation-debugging args)))
```

Note the body: except for the keyword identifying the menu, **:presentation-debugging**, this is the same for all side-effecting handlers that generate handler menus. The function creating the menu is **dw:call-presentation-menu**. Use it exactly as shown in the example.

*arglist* The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body.

**:input-context**
The current presentation-input context.

**:presentation**
The presentation instance that the mouse cursor is over.

**:handler** The handler object of which the body is a part.

**:mouse-char**
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

**:window** The window object in which the current presentation occurs.

**:x** The x-coordinate of the mouse cursor when the mouse was clicked.

**:y** The y-coordinate of the mouse cursor when the mouse was clicked.

**:gesture** The mouse gesture (symbolic name) this handler is on.

The parameter list can specify only those keywords that are explicitly used, for example, `(object &key window x y)`.

For an overview of **define-presentation-action** and related facilities: See the section "Mouse Handler Facilities". For information on handler lookup and performance issues: See the section "How Mouse Handlers Are Found".

### Other Mouse-Handler Facilities

**dw:handler-applies-in-limited-context-p** and **dw:presentation-subtypep** are related facilities used in **:tester** functions defined for translators. They restrict handler applicability to a specified input context. The latter can be used to restrict applicability to a displayed presentation type also. For more information, see the section "User-defined Data Types as Presentation Types".

Other facilities concerned with mouse handlers include **dw:delete-presentation-mouse-handler**, which eliminates a handler from your world.

### Presentation Input Context Facilities

Facilities for manipulating presentation input contexts are listed below:

**dw:with-presentation-input-context**
**dw:clear-presentation-input-context**
**dw:presentation-input-context-option**
**dw:with-presentation-input-editor-context**
**dw:*presentation-input-context***

The primary facility in this subcategory of presentation substrate tools is the first listed, **dw:with-presentation-input-context**. This macro can be used to establish an input context just as **accept** establishes a context. In a sense, its relationship to **accept** is analogous to that of **dw:with-output-as-presentation** to **present**. (See the section "Using Presentation Types for Output".) It just provides the input context; you have to do your own input/parsing. The other facilities in this group provide additional help in manipulating the input context.

### Presentation Input Blip Facilities

A *presentation input blip* is created by a translating mouse handler when a user clicks on a displayed presentation with the gesture appropriate for that handler. Conceptually, the blip represents how the user clicked on a sensitive presentation: it encodes the object, its presentation type, and the gesture used. The facilities for managing these are:

**dw:echo-presentation-blip**
**dw:presentation-blip-object**

**dw:presentation-blip-options**
**dw:presentation-blip-presentation-type**
**dw:presentation-blip-mouse-char**
**dw:presentation-blip-typep**
**dw:presentation-blip-p**
**dw:presentation-blip-case**
**dw:presentation-blip-ecase**

Do not confuse presentation blips with ordinary mouse blips. The former are generated by translating handlers in presentation input contexts established by **accept** or **dw:with-presentation-input-context**. Mouse blips, on the other hand, are generated by clicking the mouse in non-presentation input contexts, for example, that established by `(send *terminal-io* :any-tyi)`. Do not mix presentation and non-presentation input contexts in your applications. (For more information on mouse blips, see the section "Mouse Blips".)

The Presentation Input Blip Facilities are used within the blip clauses of a **dw:with-presentation-input-context** macro to manipulate input blips. The functions in this subcategory extract certain fields of the blip or test them in some way.

### Mouse Gesture Interface Facilities

The mouse gesture interface facilities are ancillary to the mouse handlers. They provide the interface between mouse gestures, the symbolic names for mouse clicks and the mouse characters to which they correspond. They are:

**dw:mouse-char-gesture**
**dw:mouse-char-gestures**
**dw:mouse-char-for-gesture**

With these facilities, you can use predefined mouse gestures in your code where the symbolic names are required, or define and use new ones. Gestures are required, in particular, for defining mouse handlers. Handlers are always defined on some gesture.

For other information about mouse characters and mouse character functions, see the section "Mouse Characters".

### Suggestions and Examples

This sections contains some general suggestions about how to write efficient mouse handlers and some examples of how to accomplish specific tasks using mouse handling.

### Some Efficiency Caveats for Mouse Handlers

Following are some caveats for making your mouse handlers efficient:

- Make handlers as specific as possible.

  Use the most specific types appropriate as your handler's *from-presentation-type* and *to-presentation-type*. Doing so will respectively restrict the number of presentations to which the handler potentially applies and the variety of input contexts in which it is potentially available.

  In particular, avoid handlers for **t** as a *from* or *to* presentation type and **sys:expression** as a *from* presentation type. These apply in a wide variety of contexts, and the effect is cumulative; the more there are, the slower everything becomes. If you do define such handlers, pay particular attention to their efficiency. This also applies to translators from and to subtypes of **sys:expression**. See the section "Using User-Extendable Data Types as Presentation Types".

- Keep presentation-type **:expander** and **:abbreviation-for** forms simple.

  These forms are evaluated a large number of times. They should avoid both consing and excessive computation. It is best if they are simple backquoted forms, as the system knows how to turn such consing into stack-consing, resulting in more speed and less work for the garbage collector.

  Also, avoid large type expansions. An **:expander** or **:abbreviation-for** clause with a large expansion, especially inside an **or**, results in much extra searching and possibly increased memory requirements for the handler lookup tables. Carried to an extreme, this could make all handler lookups slow owing to excessive paging. If needed, use a more general type and a **satisfies** clause.

- Keep **:tester** forms fast.

  Bodies of translators can be slow so long as the **:tester** form returns **nil** in the cases where the body would be slow.

- Keep translators fast.

  Expensive computations are best done as commands, rather than as translators. Translators run when you move the mouse; commands do not run until you ask for them.

- If a slow translation is necessary, use `:do-not-compose t`.

  If you feel a slow operation must be done as a translator, use `:do-not-compose t`. This suppresses SemantiCue's evaluation of the result. Because it also suppresses any contextual checking of the result, use it sparingly.

- Avoid interpreted **satisfies** clauses.

Write an auxiliary function and use that instead. **satisfies** clauses are run during mouse handling; running them interpreted creates a needless slowdown.

For some related information and examples, see the section "User-defined Data Types as Presentation Types". To get an idea of how many handlers are being considered when you move the mouse, use the Show Handlers All Presentations command in the Presentation Inspector (see the section "Presentation Inspector".). If your handler shows up in this list in a context where it should not be involved, you may need to make its *from/to* presentation types more specific. Note: Due to optimizations, not all handlers listed by the presentation inspector take time during mouse-sensitivity computations. However, most of them do.

### Handlers on the Same Mouse Chord

Handlers on the same mouse chord should go together. The system stops searching nested contexts and nested presentations when it finds any match for this chord because it cannot know which button the user will press. Note that an applicable mouse handler (other than a **:context-independent** or **:suppress-highlighting** handler) on any of the three mouse buttons will stop the search through nested presentations and input contexts. A mouse handler on another button will not be available if it applies to an outer context or an outer presentation. This means that you should choose the mouse handlers that go on the set of gestures associated with a particular chord of modifier keys so that the handlers are a related family, all involving the same context and presentation type. For example, all the handlers for gestures with just the control key pressed involving operations on raw-text presentations and input-editor context. This example shows what can go wrong if you do not follow this guideline.

```
(define-presentation-translator example
  ((sequence pathname) pathname
  :gesture :middle
  :documentation "my example")
  (object)
 (send (first object)
       :new-type "example"))

(present '(#p"foo") '(sequence pathname))
(present '(#p"foo" #p"bar") '(sequence pathname))
:Show File
```

While the Show File command is awaiting its argument, move the mouse over the **(sequence pathname)** presentation "`S:foo and S:bar`". Note that no handler is available on Middle while the mouse is pointing at one of the file names; this is because the applicable handler on Left terminates the mouse sensitivity search at the **pathname** presentation, without ever considering the **(sequence pathname)** presentation. Now try pointing the mouse over the earlier **(sequence pathname)** presentation containing just one pathname. Since the **pathname** presentation occupies the same area of the screen as the **(sequence pathname)** presentation containing it, the latter is always sensitive and the handler on Left is not available.

To avoid such anomalies as this, be careful that the three applicable handlers for a given chord of modifier keys go together. This also makes for an easier to grasp user interface.

### Writing a Translator From a Blank Area

When you are writing an interactive graphics routine, you will probably encounter the need to have commands available when the mouse is not over any object. To do this, you write a *translator* from the blank area.

The presentation type of the blank area is **dw:no-type**. You probably want the **:x** and **:y** arguments to the translator.

*Examples*:

```
(define-presentation-to-command-translator add-circle-here
   (dw:no-type :documentation "Add a circle here.")
   (ignore &key x y)
  `(com-add-circle ,x ,y))
```

### Doing Typein or Typeout From an Action

Actions are usually run inside the input editor. The input editor does its redisplay by sending the stream messages like **:string-out** and **:delete-char**. For efficiency, while the input editor is running, output is not recorded. So, when actions are run, output recording is disabled. If you want your output to remain in the history, you must temporarily reenable it. Similarly, if you want to do input, you must establish a properly recursive input editor context, by using **si:with-ie-typein**.

The system cannot automatically establish contexts or enable history for you, because it does not know in advance that you will do input or output. Besides that, establishing such things flashes the input editor and draws blank lines which would be undesirable if they were not needed. For these reasons, you must write code to do these things when and where you want them.

*Examples*:

```
(define-presentation-type string-with-more (() &key (show-more nil))
  :printer ((object stream)
            (write-string (first object) stream)
            (when show-more
             (write-string " -- " stream)
             (write-string (second object) stream))))

(present '("little" "somewhat more discursive, prolix and redundant")
         'string-with-more)
```

```
(dw:define-presentation-action show-more
  (string-with-more t
                      :gesture :middle
                      :documentation "Print all of this.")
  (object &key window)
 (si:with-ie-typeout-if-appropriate (window)
  (present object '((string-with-more) :show-more t) :stream window)))
```

Here is a better version that does not do typeout directly, but affects the old presentation:

```
(dw:define-presentation-action show-more
                              (string-with-more t
                                   :gesture :middle
                                   :documentation "Print all of this.")
   (ignore &key window presentation)
  (send window :represent-presentation presentation
                                   '((string-with-more) :show-more t)))
```

**Making Your Own Click-Right Menu**

You can define your own handler for a **:menu** gesture, from the presentation type in question to whatever context type you want it to be active in, putting **:defines-menu** *your-menu-name* in the handler definition. Use **dw:call-presentation-menu** to call the handlers for your menu, supplying *your-menu-name* so it knows which handlers to run. If necessary, give it a **:priority** high enough to ensure it takes precedence over the system's default mouse handler for the **:menu** gesture.

You can define the handlers that you want to be available via a new menu with **:menu** *your-menu-name*.

Here is how to define your own custom menu.

```
(define-presentation-action your-menu-name
  (your-presentation-type your-context-type
   :documentation "Menu"
   :gesture :menu
   :suppress-highlighting :defines-menu
   :menu nil
   :defines-menu your-menu-name)
  (ignore &rest args)
  (return-from your-menu-name
    (apply #'dw:call-presentation-menu 'your-menu-name args)))
```

**Refining Sensitivity**

There are several ways to refine the applicability of a mouse handler, and hence to refine the choice of what presentation is mouse sensitive in a given context and what operations can be performed on it. The most important is the **:tester** option. The tester is called if the handler's *from-presentation-type* matches the presentation's type and object and the handler's *to-presentation-type* matches the input context. You can implement any desired applicability condition by making *from-presentation-type* and *to-presentation-type* more general and using the tester to compute the actual applicability condition. Be careful; the more handlers that have to be tested, the slower mouse sensitivity will respond when the user moves the mouse. The tester returns *true* if the handler should be applicable, or *false* (**nil**) if the handler should be ignored. The tester receives the presented object as an argument. The following additional keyword arguments are available if needed:

**:presentation** The presentation at the mouse location that matches *from-presentation-type*

**:presentation-type** The presentation type of that presentation

**:input-context** The input context that matches *to-presentation-type*

**:gesture** The mouse gesture that could invoke the handler

**:mouse-char** The corresponding mouse character

**:window** The window containing the presentation

**:handler** The handler itself

Unless the mouse handler is an action, rather than a translator, or specifies the **:do-not-compose** option, a handler that appears to be applicable after calling the tester is invoked and the values it returns are checked. The object returned must satisfy any predicate derived from the input context. In addition, if the first and second values (object and presentation type) are both **nil**, the handler is ignored. It is better practice to use a **:tester** instead of relying on this.

Finally, if a mouse handler defines a menu (it has the **:defines-menu** option), it is ignored if the menu would be empty, in other words, if none of the mouse handlers in that menu is applicable.

A common use of testers is to limit the inheritance of a mouse handler by presentation types other than the ones specifically named. To disable the mechanism that reduces *from-presentation-type* to another presentation and a predicate, the tester can call **dw:presentation-subtypep** of its **:presentation-type** argument and *from-presentation-type*. To disable applicability to subtypes of *from-presentation-type*, the tester can check **dw:presentation-type-name** of its **:presentation-type** argument. Similarly, a handler can call **dw:handler-applies-in-limited-context-p** with its **:input-context** argument and *to-presentation-type*. This returns *true* only if *to-presentation-type* and the context type are the same presentation type, or one is an **:abbreviation-for** the other.

### Resolving Conflicts Among Mouse Handlers

When more than one mouse handler is applicable for the same physical mouse gesture, Semanticue chooses one handler and *shadows* the others. There are several ways to control this, so that the handler that is made available is the one you want.

The **:priority** option of a mouse handler specifies a number. The handler with the highest priority is chosen. If two handlers have the same priority, which one is chosen is unpredictable. Note that priorities are only compared for handlers that are applicable to the same presentation and the same input context. Priorities cannot be used to resolve conflicts among handlers for different nested presentations or (more commonly) different nested input contexts. See the function **define-presentation-translator**.

You can make the **:tester** options of the conflicting mouse handlers complementary, so that one handler knows when the other handler should be available and turns itself off by returning **nil** from its tester. While this practice can be considered unmodular if the two handlers are not from the same program, often the conflicting handlers are closely related and the use of complementary testers is the best way to achieve the desired behavior.

One of the conflicting mouse handlers can be moved to a different gesture. More commonly, one of the conflicting mouse handlers can be moved to a menu. If it is available on both a menu and a gesture, as is common, the user can simply select it from the menu when it is not available via the gesture.

### Editor Mouse Commands

You can use **zwei:define-presentation-to-editor-command-translator** to define editor commands. The list of a function name and argument values that you return calls an editor command function rather than a CP command function. The function need not be defined with **zwei:defcom**. It should return **nil** if the typeout window should be flushed or non-**nil** if the typeout window should be left alone.

Here is how you use it to write a mouse command in the editor:

```
(defun show-length-of-plist (symbol)
  (zwei:typein-line "~D" (length (symbol-plist symbol))))

(zwei:define-presentation-to-editor-command-translator
                                    show-length-of-plist
                                      (symbol "Plist length"
                                          zwei:*standard-comtab*
                                          :gesture :super-middle)
                                      (symbol)
  `(show-length-of-plist ,symbol))
```

### Rubberbanding

It is convenient to draw a line or other graphic figure by clicking a mouse button at the start of the figure and then moving the mouse to pull or drag the end of

the figure to its final position. You can see how the final figure will appear as you are drawing. This is called *rubber banding*. There are two things to remember when you write code to do this with the mouse:

- Rubberbanding is accomplished by continually erasing the figure at its previous position while redrawing it at new positions.

- Disable output recording during the rubberbanding, for speed.

This example requires the user to click once at the start of the line and once again at the end.

```
(defun input-a-line ()
  (multiple-value-bind (start-x start-y)
      (dw:tracking-mouse (t :whostate "Pick starting point"
                            :who-line-documentation-string
                                  "Put start of line here.")
        (:mouse-click (click x y)
         (unless (eql click #\mouse-l)
           (signal 'sys:abort))
         (return (values x y))))
    (let ((old-x nil) (old-y nil))
      (dw:with-output-recording-disabled ()
        (dw:tracking-mouse (t :whostate "Pick end point"
                              :who-line-documentation-string
                              "Put other end of line here.")
          (:mouse-motion (x y)
           (when (and old-x old-y)
             (graphics:draw-line start-x start-y old-x old-y
                                                   :alu :flip))
           (graphics:draw-line start-x start-y x y :alu :flip)
           (setq old-x x old-y y))
          (:mouse-click (click x y)
           (unless (eql click #\mouse-l)
             (signal 'sys:abort))
           (when (and old-x old-y)
             (graphics:draw-line start-x start-y old-x old-y
                                                  :alu :flip))
           (return (values start-x start-y x y))))))))
```

This example is similar, but uses **:mouse-motion-hold**, so the user only clicks once to start the line. The end is the point at which the mouse button is released.

```
(defun input-a-line ()
  (multiple-value-bind (start-x start-y)
      (dw:tracking-mouse (t :whostate "Pick starting point"
                            :who-line-documentation-string
  "Put start of line here and hold.")
        (:mouse-click (click x y)
         (unless (eql click #\mouse-l)
           (signal 'sys:abort))
         (return (values x y))))
    (let ((old-x nil) (old-y nil))
      (dw:with-output-recording-disabled ()
        (dw:tracking-mouse (t :whostate "Pick end point"
                              :who-line-documentation-string
                                "Put other end of line here.")
          (:mouse-motion-hold (x y)
           (when (and old-x old-y)
             (graphics:draw-line start-x start-y old-x old-y
                                                  :alu :flip))
           (graphics:draw-line start-x start-y x y :alu :flip)
           (setq old-x x old-y y))
          (:release-mouse ()
           (when (and old-x old-y)
             (graphics:draw-line start-x start-y old-x old-y
                                                  :alu :flip))
           (return (values start-x start-y old-x old-y))))))))
```

In order to leave the lines drawn on the screen, it is easiest to simply draw them again outside the with-output-recording-disabled.

```
(defun draw-some-lines ()
  (loop
    (multiple-value-bind (x1 y1 x2 y2)
        (input-a-line)
      (graphics:draw-line x1 y1 x2 y2))))
```

## Exploring Presentation Types and Presentations

This section documents several system facilities for acquiring information on the presentation types defined in your world, and on displayed presentations. Information about defined presentation types is provided by three mouse-click commands and two Command Processor commands, Show Presentation Type and Show Handlers for Types. Information about displayed presentations is provided by facilities available on the Presentation Debugging menu, the Presentation Inspector in particular.

## Mouse-Click Facilities for Looking at Handlers

Three mouse clicks provide access to several facilities to inspect and debug mouse handlers and the mouse-sensitivity of presentations on the screen.

s-Left — When expecting a **sys:expression** or **sys:form** (as in a Lisp Listener), returns the object in a presentation. This differs from simply clicking Left in that it treats the object as if it were presented as a **sys:expression**, even if the presentation's type is not a subtype of **sys:expression**. Thus, you have access to the object even when clicking left would not make it available to you.

s-Middle — When expecting a **sys:form** (as in a Lisp Listener), calls **describe** on the presentation. This is particularly useful for seeing the object and the presentation type associated with a particular presentation.

s-Right — Presentation Debugging Menu. Provides access to the following facilities:

- Does a **describe** of the presentation (in **sys:form** context only).
- Does Edit Handler for the presentation (in **((cp:command :command-table "GLOBAL"))** context only).
- Presentation Inspector. See the section "Presentation Inspector".

## CP Commands to Show Presentation Types and Handlers

Show Presentation Type *type keywords*

Shows the argument list, supertypes, and subtypes of a presentation type.

*type* — A presentation type.

*keywords* — :For Lookup, :Include Predicate, :More Processing, :Output Destination

:For Lookup — {Yes No} Whether to list the types examined during mouse handler lookup. Listed supertypes for lookup are examined when *type* is the *to-presentation-type* in a handler definition; listed subtypes for lookup are examined when *type* is the *from-presentation-type* in a handler definition.

:Include Predicate — {Yes No} Whether to show, if applicable, the type reduction step of *type* to a supertype and a predicate. For example, the **symbol** presentation type is reducible to the supertype **sys:expression** and the predicate **symbolp**.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Show Handlers for Types *context-type presentation-type keywords*

Lists the mouse handlers applicable in a specified input context to a specified object type and presentation type.

*context-type* The input context (*to-presentation-type* in the handler definition).

*presentation-type* The presentation type of the presentation to which the mouse is pointing (the *from-presentation-type* in the handler definition).

*keywords* :More Processing, :Output Destination

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

### Presentation Inspector

The Presentation Inspector is an option on the Presentation debugging menu, itself an option on the standard click-right menu available for all displayed presen-

tations. It allows you to explore the presentation you call it on from a variety of aspects.

### Using the Presentation Inspector

The Presentation Inspector exists primarily to allow you to answer one of two questions: "Why *wasn't* a particular handler available?" or "Why *was* a particular handler available?" In addition, it can help answer general inquiries, like "What input contexts are active?" or "What sub-presentations does this presentation contain?"

### Invoking the Presentation Inspector

To use the Presentation Inspector, first get the presentation you want to know about on the screen, and get into the input context you want to know about. For example, if you are debugging a translator that gives version numbers (**integer**) from filenames (**pathname**), you would want to get a **pathname** on the screen, and get into an **integer** context, perhaps by doing:

```
(define-presentation-translator pathname-version
    (pathname integer  ;From pathname to integer
     ;; Only works for pathnames with numeric versions
     :tester ((path) (integerp (pathname-version path))))
    (path)
 (pathname-version path))

(present #P"KOALA:>KJones>foo.lisp.17")
(accept 'integer)
```

Then click s-Right on the pathname KOALA:>KJones>foo.lisp.17, to get the Presentation Debugging Menu, and choose the Presentation Inspector.

### The Presentation Inspector's Frame

The Presentation Inspector's frame is divided into three parts. From top to bottom these are: a title pane, a listener pane, and a command menu.

The title pane at the top tells you what presentation the Inspector is currently examining, and what input contexts are active. The presentation and the input contexts are presentations, and can be used as input for commands.

The listener pane in the middle accepts Presentation Inspector commands in addition to the usual global commands. See the section "Presentation Inspector Commands". The listener pane does not accept Lisp forms, but if you wish to enter Lisp forms, you can enter a Lisp breakpoint by pressing SUSPEND.

The command menu at the bottom of the frame provides easy access to the simple forms of the various Presentation Inspector commands, and serves as a reminder if you prefer to enter commands from the keyboard.

**Important Caveat:**

- The Presentation Inspector covers most of the screen. If you wish to reexamine the original output, you can switch to the window with the presentation being debugged and then switch back. Use FUNCTION S to switch back and forth.

- However, if you switch to another window without exiting from the Inspector and then switch back to the program you are debugging, the result can be confusing:

  - Your program is still in the Presentation Inspector, trying to read from the Presentation Inspector's frame. There is no visual indication that this is happening, but the program does not accept input from its usual window.

  - If this happens, you should select the Presentation Inspector, using [Select] in the System menu.

  - You can then use the Exit command to return to your program, or resume using the Presentation Inspector.

**Strategy for Using the Presentation Inspector**

This section assumes that you know how to define Presentation Types and understand their structure and context.

To learn about Presentation Types:

See the function **define-presentation-type**.
See the section "Presentation Input Context Facilities".
See the function **dw:with-presentation-input-context**.

The first thing to do after invoking the Presentation Inspector is to examine the title pane. Do you find any surprises there? Often, the reason for unexpected behavior is that you are not really in the input context you expected, or the object or presentation type of the presentation is not as expected.

If you are examining nested presentations, you might also find that the presentation selected is not the one you wish to examine. If so, you can click Left on the Show Presentation Hierarchy command to pick the presentation you wish to work with. If you want to see the hierarchy in graph form, specify :Format Graph. You can also click on the Show Presentation Hierarchy command in the command menu. See the section "Show Presentation Hierarchy Presentation Inspector Command".

The next step is to get an overview of the situation with the Show Handlers All Presentations command. See the section "**Show Handlers All Presentations** Presentation Inspector Command". This produces a long display that indicates whether your handler was ever considered, and if so, what the result of that consideration was.

If the handler was considered, the next step in the investigation is to click Left on your handler in the display from Show Handlers All Presentations, to get the Show Handler Applicability command. See the section "**Show Handler Applicability** Presentation Inspector Command". This will show, for the current presentation, and for each of the nested contexts, either that the handler succeeded, and what it returned, or the reason it did not succeed. You might need to use the Show Presentation Hierarchy command in conjunction with this command to choose the appropriate presentation for investigation.

Alternatively, you can limit the investigation to a single context by using the Show Handler Context Applicability command. See the section "**Show Handler Context Applicability** Presentation Inspector Command".

If the handler was not considered, the presentation types for the handler did not match any combination of the presentation type and the various levels of context. The Show Presentation Type command might help you understand why the types do not match. See the section "Show Presentation Type Command". Using this command gives only a partial answer, as it does not allow you to specify data arguments, which can affect the expansion of the type.

In trying to determine why a handler was never considered, your best tool is careful logic.

The various types involved are matched as follows:

- The translator's context type (that is, the type of the result) must be a subtype of a type in the presentation input context (that is, the translator must translate into the type of object that is currently being prompted for).

- The presentation's presentation type must be a subtype of the handler's *From* type.

Certain presentation types are standins for a more specific presentation type determined by the type of the object presented. Handler matching and mouse sensitivity use the more specific type. These presentation types are **sys:expression**, **sys:form**, and **sys:code-fragment**. For instance, values printed by the Lisp Listener are **sys:expression** presentations, but for purposes of mouse sensitivity, handlers whose *from-presentation-type* is a supertype of the actual type of the value are considered, in addition to handlers whose *from-presentation-type* is a supertype of **sys:expression**. For example, a handler whose *from-presentation-type* is **integer** matches a presentation whose type is **sys:expression** if the object presented is of type **integer** or a subtype of **integer**.

Actually the system is more flexible and powerful than just using subtypes. For example, consider a context that is accepting **((cp:command :command-table "GLOBAL"))**, and a translator that produces a **cp:command**. A **cp:command** is not a subtype of a **((cp:command :command-table "GLOBAL"))**. (Remember, a type with no data arguments is always a supertype of the same type-name with data arguments). However, the type **((cp:command :command-table "GLOBAL"))** can be *reduced* to **cp:command** and a test for the command actually being in the global command table. Thus a translator producing a **cp:command** will tentatively

match a context looking for a **((cp:command :command-table "GLOBAL"))**, with a predicate applied to the result of the translator to see if it satisfies **command-in-global-command-table-p**. This is referred to in the display from the Show Handlers All Presentations and Show Handlers This Presentation commands as "the predicate derived from the context".

Similarly, the handler's *From* type might undergo a similar reduction. For example, a handler from **((and integer ((satisfies oddp))))** will match a presentation of an **integer**, and yield an **oddp** predicate. The handler will match, but it will not succeed unless the presented object satisfies the **oddp** predicate. This is referred to in the display from the Show Handlers All Presentations and Show Handlers This Presentation commands as "the predicate derived from handler's presentation and object types".

### Presentation Inspector Commands

These commands are shown in the command menu pane at the bottom of the frame, and can be selected from there. Also, many of the commands can be selected by various mouse clicks on the output of other commands, as indicated by the Mouse Documentation line. Some of the commands have additional options available only from the keyboard.

These commands are arranged in five groups. From left to right on the pane:

*General* - Commands relating to the general operation of the Presentation Inspector:

"**Exit** Presentation Inspector Command"
"**Help** Presentation Inspector Command"

*Overview* - Commands giving you a general overview of what handlers were considered and the result.

"**Show Handlers All Presentations** Presentation Inspector Command"
"**Show Handlers This Presentation** Presentation Inspector Command"

*Handlers* - Commands to investigate individual handlers.

"**Show Handler Applicability** Presentation Inspector Command"
"**Show Handler Context Applicability** Presentation Inspector Command"
"**Describe Handler** Presentation Inspector Command"

*Environment* - Commands to show what is being inspected.

"Show Presentation Hierarchy Presentation Inspector Command"
"**Show Input Context** Presentation Inspector Command"

*Controls* - Commands to control the operation of the Presentation Inspector

"**Set Presentation** Presentation Inspector Command"

Other useful commands - Commands relating to handlers and presentation types, but not part of the Presentation Inspector. (These are available from the Presentation Inspector).

"Show Presentation Type Command"
"Show Handlers For Types Command"

**Help** Presentation Inspector Command

Help

Displays a one-screen help message explaining the basic operation of the Presentation Inspector.

**Exit** Presentation Inspector Command

Exit

Exits the Presentation Inspector and removes its frame from the screen.

**Show Presentation Type** Command

Show Presentation Type *type keywords*

Shows the argument list, supertypes, and subtypes of a presentation type.

*type* A presentation type.

*keywords* :For Lookup, :Include Predicate, :More Processing, :Output Destination

:For Lookup {Yes No} Whether to list the types examined during mouse handler lookup. Listed supertypes for lookup are examined when *type* is the *to-presentation-type* in a handler definition; listed subtypes for lookup are examined when *type* is the *from-presentation-type* in a handler definition.

:Include Predicate
{Yes No} Whether to show, if applicable, the type reduction step of *type* to a supertype and a predicate. For example, the **symbol** presentation type is reducible to the supertype **sys:expression** and the predicate **symbolp**.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window}

Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

**Show Handlers All Presentations** Presentation Inspector Command

Show Handlers All Presentations *keywords*

Show all the handlers that were considered for any presentation — starting with the lowest one pointed to by the mouse, up to the top of the hierarchy — and whether they were successful.

*keywords* :Show Context, :Show Presentation

:Show Context {Yes, No} Show the context that each handler succeeded for. The default is No. The mentioned default is Yes.

:Show Presentation
{Yes, No} Show the presentation for which each handler succeeded. The default is No. The mentioned default is Yes.

The individual mouse handlers shown by this command are mouse sensitive and can be used in the Show Handler Applicability command.

Here is some sample output (trimmed for conciseness):

```
Presentation Inspector command: Show Handlers All Presentations
Searching the presentation and context hierarchies, for handlers
from presentation Text "THIS-I...".
to context SI:INPUT-EDITOR.
```

**--- Handlers appearing on mouse buttons ---**

```
     Left   : (priority 5.5) QUOTED-EXPRESSION: 'THIS-IS-A-SYMBOL
     Middle : (priority 4) DESCRIBE: (DESCRIBE 'THIS-IS-A-SYMBOL)
     Right  : (priority 0) MENU: Menu
  sh-Right  : (priority 0) SYSTEM-MENU: System menu
   c-Left   : (priority 6) CLICK-AND-HOLD-MARK-REGION: Mark a region
   c-Middle : (priority 6) YANK-WORD: THIS-IS-A-SYMBOL
   c-Right  : (priority 2) MARKING-AND-YANKING-MENU: Marking and yanking menu
c-sh-Middle : (priority 6) MARK-WORD: Mark this word
```

**--- Handlers appearing in the :PRESENTATION-DEBUGGING menu ---**

```
DESCRIBE-PRESENTATION: (DESCRIBE '#<DISPLAYED-PRESENTATION 467471015>)
EDIT-PRESENTATION-HANDLER: Edit handler for this presentation
INVOKE-PRESENTATION-INSPECTOR: Presentation Inspector
```

**--- Handlers appearing in the standard Right menu ---**

```
QUOTED-EXPRESSION: 'THIS-IS-A-SYMBOL
DESCRIBE: (DESCRIBE 'THIS-IS-A-SYMBOL)
SYSTEM-MENU: System menu
MARKING-AND-YANKING-MENU: Marking and yanking menu
```

**--- Handlers not appearing on mouse buttons because they have no gesture ---**

```
YANK-FROM-KILL-RING: Yank top of kill ring
EDIT-PRESENTATION-HANDLER: Edit handler for this presentation
INVOKE-PRESENTATION-INSPECTOR: Presentation Inspector
```

**--- Handlers whose test functions failed ---**

```
DBG:MONITOR-LOCATION: Monitor this location
```

**--- Handlers for empty menus ---**

```
MY-FUNNY-NUMBER-MENU:  Menu of Numerology commands
```

**--- Handlers who failed predicates derived from their object and presentation types ---**

```
FLAVOR::EDIT-FLAVOR: Edit flavor definition
```

Notes:

`from presentation Text "THIS-I...". to context SI:INPUT-EDITOR-BUFFER.`

Indicates that the search starts from the text presentation (the bold "**T**" indicates exactly where in the text) and searches the presentation hierarchy upward from there. The context is searched outward from **si:input-editor-buffer**.

Each heading indicates how a handler did or did not succeed. For the categories relating to handlers appearing on mouse gestures, the physical mouse gesture it appears on is listed, together with the priority. (See the function **define-presentation-translator**.) This information is provided both for handlers that appear on mouse buttons, and, under separate heading, for handlers which do not appear because they were shadowed by other handlers.

**Handlers appearing on mouse buttons**
The handlers that actually appear on the mouse buttons.

**Handlers which have been shadowed by other handlers**
The handlers that would have appeared on the mouse buttons, but are shadowed by other handlers with higher (or same) priority.

**Handlers appearing in the :PRESENTATION-DEBUGGING menu**
Shows successful handlers that were defined with the **:menu :presentation-debugging** option to **define-presentation-translator**.

**Handlers which do not appear on mouse buttons**
Handlers that can appear in menus. Their appearance here implies they are successful, but if they do not appear in a menu, they will be unavailable.

**Handlers not appearing on mouse buttons because they have no gesture**
Handlers defined with **:gesture nil** and are available from their menu.

**Successful handlers not on any mouse buttons or menus**
Handlers that are unavailable because they are not on any mouse button or in any menu.

**Handlers whose test functions failed**
Refers to handlers whose **:tester** function returned **nil**. In the example above, the symbol was not presented with the **:location** option to **present**, so **dbg:monitor-location**'s tester returned **nil** since there is no location to monitor.

**Handlers for empty menus**
Unsuccessful handlers. A menu handler defined with the **:defines-menu :numerology** option in **define-presentation-translator** appears under this heading if no handlers defined with **:menu :numerology** are successful.

**Handlers which did not return a value**
Handlers that returned **nil** with no second value (that is, no type). Returning **nil** from a translator (but not an action, whose values are not used), indicates that the translator declines to handle the situation.

**Handlers who failed predicates derived from their object and presentation types**
Handlers with a *From* type that was reduced to a supertype and predicate, and the predicate returned **nil** when called on the presentation object. An example would be a handler defined on **((and integer ((satisfies oddp))))**, with a presentation of an even integer.

**Handlers who failed the predicate derived from the context**

Includes any handler that returned an object that did not satisfy a predicate derived from the context type used in the match. For instance, if a command loop is accepting **((cp:command :command-table "GLOBAL"))** and the handler returns a command that is not in the global command-table, this type is reduced to the **cp:command** type. The "command-in-global-command-table-p" predicate that is the result of this handler fails.

**Show Handlers This Presentation** Presentation Inspector Command

Show Handlers This Presentation *keywords*

Shows the handlers that were considered for a single presentation (the current presentation). This is just like the Show Handlers All Presentations command, except it limits the report to handlers for the current presentation. (The current presentation is selected with the Set Presentation command or by choosing a presentation from the display of the Show Presentation Hierarchy command).

An alternative to using this command is to use the :Show Presentation option to Show Handlers All Presentation.

*keywords* :Show Context, :Show Presentation

:Show Context {Yes, No} Show the context that each handler succeeded for. The default is No. The mentioned default is Yes.

:Show Presentation

{Yes, No} Show the presentation that each handler succeeded for. The default is No. The mentioned default is Yes.

For details of the output display, see the section "**Show Handlers All Presentations** Presentation Inspector Command".

**Show Handler Applicability** Presentation Inspector Command

Show Handler Applicability *Handler-name keyword*

Shows what contexts this handler does and does not apply to for the current presentation, and why. (To set the current presentation, see the section "Show Presentation Hierarchy Presentation Inspector Command".)

*Handler-name* The name of a handler to be investigated.

*keyword* :Detailed

:Detailed {Yes No} If Yes, show additional detail. The default is No, unless :Detailed is mentioned.

For each context in the context hierarchy (see the title pane), shows whether the handler is applicable, and if not, why. A couple of examples will help you decode the format:

```
Show Handler Applicability si:com-show-file
For context SI:INPUT-EDITOR:
 Handler SI:COM-SHOW-FILE PATHNAME (presentation) → CP:COMMAND
  The handler failed because it failed to match the context and
  displayed presentation types.
 Handler SI:COM-SHOW-FILE PATHNAME (object), EXPRESSION (presentation) → CP:COMMAND
  The handler failed because it failed to match the context presentation type.
For context ((CP:COMMAND-OR-FORM :COMMAND-TABLE
  #<COMMAND-TABLE User 100214200> :DISPATCH-MODE ...)):
 Handler SI:COM-SHOW-FILE PATHNAME (presentation) → CP:COMMAND
  The handler failed because it failed to match the displayed presentation type.
 Handler SI:COM-SHOW-FILE PATHNAME (object), EXPRESSION (presentation) → CP:COMMAND
  The handler succeeded and returned values:(SI:COM-SHOW-FILE (#P"Y:>rwk>mail.text.1"))

For context SI:INPUT-EDITOR:
 Handler SI:COM-SHOW-FILE PATHNAME (presentation) → CP:COMMAND
  The handler failed because it failed to match the context and displayed
  presentation types.
 Handler SI:COM-SHOW-FILE PATHNAME (object), EXPRESSION (presentation) → CP:COMMAND
  The handler failed because it failed to match the context presentation type.
For context <null context>:
 Handler SI:COM-SHOW-FILE PATHNAME (presentation) → CP:COMMAND
  The handler failed because it failed to match the context and displayed
  presentation types.
 Handler SI:COM-SHOW-FILE PATHNAME (object), EXPRESSION (presentation) → CP:COMMAND
  The handler failed because it failed to match the context and displayed
  presentation types.
```

Notes:

`Handler SI:COM-SHOW-FILE PATHNAME (presentation) → CP:COMMAND`
Means the handler named **si:com-show-file**, with the From type of **pathname**, (looking at the presentation's presentation type), translating to the type **cp:command**.

`Handler SI:COM-SHOW-FILE PATHNAME (object), EXPRESSION (presentation) → CP:COMMAND`
Means the handler named **si:com-show-file**, with the From type of **pathname**, (looking at the the type of the object for things presented as **sys:expression**), translating to the type **cp:command**.

The presentation in question here was a **pathname**, **#p"Y:>RWK>mail.text.1"**, presented as a **sys:expression**. (Perhaps it was just printed with **print** by the command loop in a Lisp Listener). This shows that the translator never matches an **si:input-editor-buffer** context, but it does match a command context when presented as a **sys:expression**.

**Show Handler Context Applicability** Presentation Inspector Command

Show Handler Context Applicability *handler-name context level*

Show why a handler does and does not apply to the current presentation and a specified context, and why. (To set the current presentation, see the section "Show Presentation Hierarchy Presentation Inspector Command".) For details on the format of the display, see the section "**Show Handler Applicability** Presentation Inspector Command".

*Handler-name* The name of a handler to be investigated.

*Context level* A level of context to explain the handler's applicability for. This can be specified by choosing it from the title pane with the mouse.

**Describe Handler** Presentation Inspector Command

Describe Handler *Handler-name*

Calls **describe** on the internal representation of a handler.

*Handler-name* The name of a handler to describe.

A presentation mouse handler is implemented in two pieces: a **dw::presentation-mouse-handler** and **dw::presentation-mouse-handler-functions**. The latter holds the name of handler and all of the functions. The former is separate so it can be entered in the tables under varying combinations of types.

This also lets you see such options as **:context-independent**, **:menu**, and **:defines-menu**.

Show Presentation Hierarchy Presentation Inspector Command

Show Presentation Hierarchy *keyword*

Show the hierarchy of presentations. The output is mouse-sensitive; if you want to select a different presentation as the current presentation for Show Handlers This Presentation, Show Handler Applicability, and Show Handler Context Applicability, click Left on the appropriate presentation.

*keyword* :Format

:Format {Text, Graph}. Text is the default. Chooses textual or graphic display of the hierarchy.

**Show Input Context** Presentation Inspector Command

Show Input Context

Shows the input context in more detail than is shown in the title pane.

The display looks like this:

```
Current presentation input context:
  PRESENTATION-TYPE:         ((OR ZWEI:PRESENTATION-COMMAND SI:INPUT-EDITOR-BUFFER))
  THROW-P:                   T
  OPTIONS:                   NIL
Superior (inherited) context:
  PRESENTATION-TYPE:         ((AND STRING
                                   ((SATISFIES NOT-NULL-STRING-P))))
  THROW-P:                   T
  OPTIONS:                   (:INHERIT T)
```

presentation-type is the type being accepted.

throw-p indicates whether the context is expecting the object to be returned to it, or the value thrown to the context packaged up in a **presentation-blip**.

**Set Presentation** Presentation Inspector Command

Set Presentation *presentation*

Select a presentation as the current presentation for the Show Handlers This Presentation, Show Handler Applicability, and Show Handler Context Applicability commands. The Title pane is updated to reflect the new current presentation.

You do not normally enter this command directly. Rather, you normally get it by clicking left on a presentation, as is shown by Show Presentation Hierarchy. See the section "Show Presentation Hierarchy Presentation Inspector Command".

*presentation* The presentation to make current.

**Table of Facilities for Writing Mouse Handlers**

**define-presentation-translator** *name* (*from-presentation-type to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose*) *arglist* &body *body*
Defines a mouse handler that translates from a displayed presentation object of a certain type to a returned presentation object of a different type.

**define-presentation-to-command-translator** *name* (*presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose*) *arglist* &body *body*
Defines a mouse handler that translates from a displayed presentation object into a Command Processor command using that object as input.

**define-presentation-action** *name* (*from-presentation-type to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area defines-menu*) *arglist* &body *body*
Defines a side-effecting mouse handler for performing actions on a displayed presentation object that are independent of the main body and command loop of an application.

**dw:handler-applies-in-limited-context-p** *context limiting-context-type*
This function is intended for use in the **:tester** forms of mouse handlers.

**dw:presentation-subtypep** *subtype supertype*
This function is the presentation system equivalent of the Common Lisp function **subtypep**.

**dw:delete-presentation-mouse-handler** *name*
Removes an already defined presentation mouse handler.

**dw:mouse-char-gesture** *mouse-char*
Returns the standard gesture associated with a mouse character.

**dw:mouse-char-gestures** *mouse-char*
Returns a list of gestures associated with a mouse character.

**dw:mouse-char-for-gesture** *gesture*
Returns the mouse character associated with a gesture.

**dw:with-presentation-input-context** (*presentation-type* &rest *options*) (&optional (*blip-var* **'dw::.blip.**)) *non-blip-form* &body *blip-cases*
Binds local environment to the input context of a specified presentation type.

**dw:clear-presentation-input-context**
Clears the current input context.

**dw:presentation-input-context-option** *presentation-input-context indicator*
Extracts the value of the specified option from an input context. The input context options are supplied in the *options* clause to **dw:with-presentation-input-context**.

**dw:with-presentation-input-editor-context** (*stream presentation-type . options*) (&optional (*blip-var* **'dw::.blip.**) *start-loc-var*) *non-blip-form* &body *blip-cases*

Establishes an input context around a call to the input editor to read keyboard input from the user.

**dw:*presentation-input-context***
Bound to the current presentation input context.

**dw:echo-presentation-blip** *stream blip* &optional (*start-bp* **(send stream :read-location)**) *for-context-type*
Echoes a presentation blip from the input buffer.

**dw:presentation-blip-object** *presentation-blip*
Returns the presentation object from a presentation blip.

**dw:presentation-blip-options** *presentation-blip*
Returns the options field (a list of keyword-value pairs) of a presentation blip.

**dw:presentation-blip-presentation-type** *presentation-blip*
Returns the presentation type from a presentation blip.

**dw:presentation-blip-mouse-char** *presentation-blip*
Returns the mouse character from a presentation blip.

**dw:presentation-blip-typep** *blip type*
Determines whether the presentation type of a presentation blip is of a specified type. (The comparison is based on **dw:presentation-subtypep**).

**dw:presentation-blip-p** *blip*
Determines whether a blip is a presentation blip.

**dw:presentation-blip-case** *blip* &body *clauses*
Dispatches to clauses based on the presentation-type field of a presentation blip.

**dw:presentation-blip-ecase** *blip* &body *clauses*
Dispatches to clauses based on the presentation-type field of a presentation blip.

**dw:describe-presentation-type** *type* &optional (*stream* ***standard-output***) *plural-count*
Outputs the description of a presentation type provided by the type's definition (**define-presentation-type** macro).

### Displaying Output: Replay, Redisplay, and Formatting

This chapter explains how to produce output that can be redisplayed and reformatted. It also describes how you can write your own formatting output macros. (To find out how to use basic facilities to produce formatted output, see the section "Presenting Formatted Output".) *Replayable presentations* are ones that can be re-run, in place, and displayed in a new format. You, the programmer, specify the redisplay options, called "viewspec choices." At runtime, the user of your program can click on the replayable presentation and call up a menu listing the viewspec choices. After the user exits the menu, the presentation is erased and redisplayed according to the choices made.

To see an example of a replayable presentation, invoke the Show Processes command in a Lisp Listener or **break** window. Now, with the mouse cursor anywhere in the displayed listing, click s-sh-Middle. This brings up a menu entitled "Output parameters" listing the viewspec choices. Try changing the selected choice from **None** to any of the others, then click on **Done**, and watch what happens.

*Incremental redisplay* of program output is implemented by another set of interrelated facilities. Output intended for redisplay is saved in an *output cache*. With the redisplay facilities, you can cache an output display and compare it against the old cached output of the same display to check for changes. If changed, the cache is updated and the objects are redisplayed; if not, both the cache and the original display remain unaltered.

### How Redisplay Works

Redisplay is organized around a hierarchy of output caches. Each cache represents a portion of the screen, divided into nested rectangles according to the hierarchy. The contents of the cache are validated when redisplay output is performed. If they are valid, that is, if the same output is being done now as last time, the screen area can be saved and reused rather than having to actually be redrawn onto the screen.

In order to validate the cache, two pieces of information are necessary. First, an association between the output being done by the program and a particular cache. This is a user-supplied datum, the unique-id of the cache. It is given by the **:unique-id** option to **dw:with-redisplayable-output**. Second, a means of determining whether this particular cache is valid. This is the **:cache-value** option. Normally, you would supply both options. The unique-id would be some data structure associated with the corresponding part of output. The cache value would be something in that data structure that changes whenever the output changes.

It is valid to give the **:unique-id** and not the **:cache-value**. This is done to identify a superior in the hierarchy. By this means, the inferiors essentially get a more complex **:unique-id** when they are matched for output. (In other words, it is like using a telephone area code.) The cache without a cache value is never valid. Its inferiors always have to be checked.

It is also valid to give the **:cache-value** and not the **:unique-id**. In this case, unique ids are just assigned sequentially. So, if output associated with the same thing is done in the same order each time, it isn't necessary to invent new unique ids for each piece. This is especially true in the case of inferiors of a cache with a unique id and no cache value of its own. In this case, the superior marks the particular data structure, whose components can change individually, and the inferiors are always in the same order and properly identified by their superior and the order in which they are output.

A **:unique-id** need not be unique across the entire redisplay, only among the inferiors of a given output cache; that is, among all possible (current and additional) uses you make of **dw:with-redisplayable-output** that are dynamically (not lexically) within another.

To make your incremental redisplay maximally efficient, you should attempt to give as many caches with **:cache-value** as possible. For instance, if you have a deeply nested tree, it is better to be able to know when whole branches have not changed than to have to recurse to every single leaf and check it. So, if you are maintaining a modification tick in the leaves, it is better to also maintain one in

their superiors and propagate the modification up when things change. While the simpler approach works, it requires redisplay to do more work than is necessary.

Redisplay is organized around the movement of boxes. The boxes contain the output that is cached. Cursor motion, including output of newlines or **terpri**, need not be contained within any particular **dw:with-redisplayable-output**. The cursor motion only determines the placement of the actual text and graphics that make up the display. A few restrictions apply, though:

- You cannot use **fresh-line**, because its meaning might be ambiguous between the two redisplay passes.

- Instead of outputting a conditional newline at the start of some output, you should do an unconditional newline at the end. Furthermore, it is almost always better to do any unconditional newline at the end rather than the beginning of the contents of a **dw:with-redisplayable-output**.

- If you are writing a function that is meant to be called from Lisp top level, with output starting on a blank line, you should put a **fresh-line** before any redisplay:

Here are some examples of wrong and right ways to do redisplay.

*Wrong:*

```
(defun redisplay-1 ()
  (let* ((list '(1 2 3))
         (displayer (dw:redisplayer ()
                      (dolist (thing list)
                        (dw:with-redisplayable-output (:unique-id thing
                                                       :cache-value t)
                          (format t "~&~D" thing))))))
    (dw:do-redisplay displayer)
    (sleep 1)
    (setq list (nconc list (ncons 4)))
    (dw:do-redisplay displayer)))
```

*Right:*

```
(defun redisplay-2 ()
  (fresh-line)
  (let* ((list (copy-list '(1 2 3)))
         (displayer (dw:redisplayer ()
                      (dolist (thing list)
                        (dw:with-redisplayable-output (:unique-id thing
                                                       :cache-value t)
                          (format t "~D~%" thing))))))
    (dw:do-redisplay displayer)
    (sleep 1)
    (setq list (nconc list (ncons 4)))
    (dw:do-redisplay displayer)))
```

If the state of the display is changing asynchronously, so that it might change be-

tween passes of redisplay, it can be snapshotted during the first pass of redisplay by using a variable for **:cache-value**. When **dw:with-redisplayable-output** sees a variable in that position, it creates a closure over it during the first pass of redisplay, and uses that value for the display body of the macro on both passes.

### Creating Replayable Output

These are the macros you can use to produce replayable or resortable output:

**dw:with-output-to-presentation-recording-string**
**dw:with-replayable-output**
**dw:with-resortable-output**

The first, **dw:with-output-to-presentation-recording-string**, is the presentation-system equivalent of the Common Lisp macro **with-output-to-string**. It works similarly, but you can subsequently output the string as a presentation or collection of presentations, instead of just as a string.

**dw:with-replayable-output** and **dw:with-resortable-output** are closely related, the latter being a special case of the former. **dw:with-replayable-output** lets you present all of the output generated in the body of the macro as a single presentation. This presentation is "replayable"; that is, it can be input as a whole, internally re-arranged in some fashion, and presented again as the same object.

**dw:with-resortable-output** takes a sequence and a sorting predicate, and constructs a **dw:with-replayable-output** macro to implement the sorting function. Users can click on the presented sequence and have it redisplayed in a different order.

The following example is a function for presenting a resortable display of network servers. It is implemented similarly to the Show Processes command.

```
(defun format-servers (&optional (sort-by :none))
  (fresh-line)
  (dw:with-resortable-output
    ((servers sort-by :copy-of neti:*servers*)
     (:none #'ignore)
     (:protocol (lambda (s-1 s-2)
                  (string<
                    (string (neti:server-protocol-name s-1))
                    (string (neti:server-protocol-name s-2)))))
     (:medium (lambda (s-1 s-2)
                (string<
                  (string (neti:server-medium-type s-1))
                  (string (neti:server-medium-type s-2)))))
     (:arguments (lambda (s-1 s-2)
                   (< (neti:server-number-of-arguments s-1)
                      (neti:server-number-of-arguments s-2)))))
    ()
    (formatting-table ()
      (formatting-column-headings ()
        (with-character-face (:italic)
          (with-underlining ()
            (formatting-cell ()
              (write-string "protocol"))
            (formatting-cell ()
              (write-string "medium"))
            (formatting-cell ()
              (write-string "no. of arguments")))))
      (loop for server in servers do
        (formatting-row ()
          (formatting-cell ()
            (format t "~a"
                    (neti:server-protocol-name server)))
          (formatting-cell ()
            (format t "~a"
                    (neti:server-medium-type server)))
          (formatting-cell (*standard-output* :align :right)
            (format t "~a"
                    (neti:server-number-of-arguments server)))))))))
```

### Redisplaying with dw:accepting-values Forms

The following examples illustrate how to use **dw:with-redisplayable-output** and several table formatting macros within **dw:accepting-values** to obtain complex formatting.

```
(defun enter-matrix (m n)
  (fresh-line)
  (let ((matrix (make-array (list m n))))
    (dw:accepting-values ()
      (formatting-table ()
        (formatting-column-headings ()
          (formatting-cell ()
            (dw:with-redisplayable-output
                   (:unique-id (list nil nil) :id-test #'equal
                                              :cache-value t)
              (format t "\\")))
          (dotimes (j n)
            (formatting-cell ()
              (dw:with-redisplayable-output
                   (:unique-id (list nil j) :id-test #'equal
                                            :cache-value t)
                (format t "~D" j)))))
        (dotimes (i m)
          (dw:with-redisplayable-output (:unique-id i)
            (formatting-row ()
              (formatting-cell ()
                (dw:with-redisplayable-output (:unique-id (list i nil)
                                               :id-test #'equal
                                               :cache-value t)
                  (format t "~D" i)))
              (dotimes (j n)
                (formatting-cell ()
                  (setf (aref matrix i j)
                        (accept 'sys:expression
                                :default (aref matrix i j)
                                :provide-default t
                                :query-identifier (list i j)
                                :prompt nil :prompt-mode :raw)))))))))
    matrix))

(define-presentation-type choice-box-subset
                         ((&key keyword-alist) &key (possible-keywords
                                (map 'list #'car keyword-alist)))
   :expander `((subset . ,possible-keywords))
   :choose-displayer ((stream object query-identifier)
                      (choice-box-subset-choose-displayer
                                    stream object
                                    keyword-alist possible-keywords
                                    query-identifier)))
```

```
(defun choice-box-subset-choose-displayer (stream value keyword-alist
                                          possible-keywords
                                          query-identifier
                                          &key (box-size
                                            (send stream :baseline)))
  (flet ((do-dependencies (new old)
           (let ((dependencies (or (cddr (assoc new keyword-alist))
                                       '(nil t nil nil))))
             (macrolet ((dep-set (accessor)
                          `(let ((tem (,accessor dependencies)))
                             (if (eq tem 't) possible-keywords tem))))
               (cond ((member new old)
                      ;; Removing
                      (dolist (also (dep-set third))
                        (setq old (adjoin also old)))
                      (dolist (also (dep-set fourth))
                        (setq old (delete also old)))
                      (setq old (delete new old)))
                     (t
                      ;; Adding
                      (dolist (also (dep-set first))
                        (setq old (adjoin also old)))
                      (dolist (also (dep-set second))
                        (setq old (delete also old)))
                      (setq old (adjoin new old))))))
           old))
    (let ((choices (dw::make-accept-values-choices
                     :query-identifier query-identifier
                     :sequence keyword-alist
                     :select-action #'do-dependencies)))
      (dw:with-output-as-presentation
                         (:type 'dw::accept-values-choices-display
                          :object choices
                          :stream stream)
        (loop for (keyword) in keyword-alist do
          (formatting-cell (stream :align-x :center :align-y :top)
            (when (member keyword possible-keywords)
              (let ((on-p (member keyword value)))
                (dw:with-redisplayable-output (:stream stream
                                               :unique-id keyword
                                               :cache-value on-p)
                  (dw:with-output-as-presentation
                                   (:stream stream
                                    :single-box t
                                    :type 'dw::accept-values-choice-display
                                    :object (dw::make-accept-values-choice
                                             :choices choices
```

```
                              :choice keyword
                              :value keyword))
(send stream :increment-cursorpos 1 1)
                              ;Thickness 2 goes 1 back.
(graphics:with-room-for-graphics (stream box-size
                                 :fresh-line nil
                                 :move-cursor nil)
  (graphics:with-drawing-state (stream :thickness 2)
    (graphics:draw-rectangle 0 0 box-size box-size
                        :filled nil :stream stream)
    (when on-p
      (graphics:draw-line 0 0 box-size box-size
                                     :stream stream)
      (graphics:draw-line 0 box-size box-size 0
                                     :stream stream))
    ))))))))))))
```

```
(defun almost-compatible-multiple-choose (item-name item-list keyword-alist
                                          &key (own-window nil)
                                               (near-mode '(:mouse)))
  (let ((state (loop for (item name choices) in item-list
                     collect (list* item
                                    (loop for choice in choices
                                          when (and (consp choice)
                                                    (second choice))
                                            collect (first choice))))))
    (dw:accepting-values (t :own-window own-window :near-mode near-mode)
      (formatting-table ()
        (dw:with-redisplayable-output (:unique-id 'headings :cache-value t)
          (formatting-column-headings ()
            (format-cell item-name #'princ)
            (loop for (keyword name) in keyword-alist do
              (format-cell name #'princ))))
        (loop for (item name choices) in item-list
              for state in state
              do
          (formatting-row ()
            (setf (cdr state)
                  (accept `((choice-box-subset
                              :keyword-alist ,keyword-alist)
                              :possible-keywords ,(map 'list
                                                       #'(lambda (x)
                                                         (if (consp x)
                                                             (car x) x))
                                                       choices))
                            :query-identifier item
                            :prompt #'(lambda (stream ignore)
                                        (formatting-cell (stream)
                                          (princ name stream)))
                            :prompt-mode :raw
                            :newline-after-query nil
                            :default (cdr state)
                            :provide-default t))))))
    state))

(defun mc-test (&optional (new-p t) &rest other-options)
  (apply (if new-p #'almost-compatible-multiple-choose #'tv:multiple-choose)
         "Buffer"
         '((1 "Buffer 1" ((:save t) (:kill nil)))
           (2 "Buffer 2" ((:save nil) (:kill t))))
         '((:save "Save") (:kill "Kill"))
        other-options))
```

**Snapshotting Variables**

To understand the need for variable snapshotting, consider the following programs.

```
(map 'list #'funcall
     (loop for i from 1 to 10 collect #'(lambda () i)))
=> (11 11 11 11 11 11 11 11 11 11)
(map 'list #'funcall
     (let ((continuations nil))
       (dotimes (i 10)
         (push #'(lambda () i) continuations))
       (nreverse continuations)))
=> (10 10 10 10 10 10 10 10 10 10)
(map 'list #'funcall
     (let ((continuations nil))
       (dolist (x '(a b c d))
         (push #'(lambda () x) continuations))
       (nreverse continuations)))
=> (D D D D)
(map 'list #'funcall
     (map 'list #'(lambda (x) #'(lambda () x))
          '(a b c d)))
=> (A B C D)
```

In all but the last case, the program does not do as expected. A strict interpretation of the definition of **dolist** and **dotimes** requires that there by a single iteration variable that is set each time around the loop to the next value. The semantics of the complete version of **loop**, with multiple variables of iteration, makes this even more imperative.

Unfortunately, this collecting of continuations is just how the formatted output macros accomplish their layout, as the following examples show.

```
(defun with-snapshotting (l)
  (terpri)
  (formatting-item-list ()
    (dolist (x l)
      (formatting-cell ()
        (princ x)))))

(with-snapshotting '(a b c d e f g h i j))

=> A B C
   D E F
   G H I
   J
```

```
(defun without-snapshotting (l)
  (terpri)
  (formatting-item-list ()
    (dolist (x l)
      (formatting-cell (t :dont-snapshot-variables (x))
        (princ x)))))

(without-snapshotting '(a b c d e f g h i j))

=> J J J
   J J J
   J J J
   J
```

So, in order to make the most common usage of the the simplest iteration macros work with the formatted output macros, variables used freely inside those macros are placed in a separate lexical contour, that is, *snapshotted*, so that they are not shared.

However, this snapshotting can cause problems for legitimate uses of shared lexical variables, which the macros cannot distinguish from variables of iteration or other things requiring snapshots. For example:

```
(defun show-some-hash-elements (table)
  (terpri)
  (let ((items-output nil))
    (formatting-table ()
      (maphash #'(lambda (key item)
                   (when (oddp key)
                     (pushnew item items-output)
                     (formatting-row ()
                       (formatting-cell ()
                         (princ key))
                       (formatting-cell ()
                         (princ item)))))
               table))
    (sort items-output #'<)))

(show-some-hash-elements
     (make-hash-table :initial-contents '(1 2 2 3 3 4 4 5 5 6)))
5 6
3 4
1 2
=> NIL
```

This program gets a compiler warning because it is writing to a snapshotted variable, and it does not work. In order to get it to function correctly, it must be rewritten to inhibit the snapshotting of the **dw::items-output** variable, as follows:

```
(defun show-some-hash-elements (table)
  (terpri)
  (let ((items-output nil))
    (formatting-table (t :dont-snapshot-variables (items-output))
      (maphash #'(lambda (key item)
                   (when (oddp key)
                     (pushnew item items-output)
                     (formatting-row ()
                       (formatting-cell ()
                         (princ key))
                       (formatting-cell ()
                         (princ item)))))
               table))
    (sort items-output #'<)))

(show-some-hash-elements
      (make-hash-table :initial-contents '(1 2 2 3 3 4 4 5 5 6)))
5 6
3 4
1 2
=> (2 4 6)
```

**Doing Incremental Redisplay**

Here is the set of inter-related facilities is provided for creating redisplayable output and doing incremental redisplay:

> **dw:redisplayable-present**
> **dw:redisplayable-format**
> **dw:independently-redisplayable-format**
> **dw:with-redisplayable-output**
> **dw:redisplayer**
> **dw:do-redisplay**

Redisplayable output is similar to ordinary output in the actual display; it differs in that, in addition to being output to a window for display, the output value is also stored in an *output cache* uniquely identified with that display. When the window is redisplayed, the cached output value is first compared to output values previously cached and, if different, the cache is updated with the new value for display. This has efficiency advantages compared with non-cached output: the display is not changed saving time and reducing flicker.

*Incremental redisplay* refers to the redisplay of individual pieces of the output to a window, rather than redisplaying the window as a whole. It works in the manner described above, except that each redisplayed piece of the window output is associated with its own output cache.

The first four facilities are for creating redisplayable output. **dw:redisplayable-present** is used like **present** but creates a redisplayable presentation. Similarly, **dw:redisplayable-format** works as **format** does, but generates redisplayable output. **dw:independently-redisplayable-format** is like the previous function, except that each argument in the format-control string gets cached separately, hence its usefulness for incremental redisplay. Finally, the macro **dw:with-redisplayable-output** lets you make any output-producing code produce redisplayable output. When doing incremental redisplay with tables or graphs, you put a **dw:with-redisplayable-output** *outside* the **formatting-row** with only a **:unique-id**, and a **dw:with-redisplayable-output** *inside* the **formatting-cell**s with only a **:cache-value**.

How you do redisplay once you have functions producing redisplayable output depends on whether you are taking advantage of **dw:define-program-framework**. If you are, making a program pane use your redisplay function is simply a matter of supplying that function via the **:redisplay-function** keyword. If, additionally, incremental redisplay is what you want, you should specify so with the **:incremental-redisplay** keyword.

If you are doing redisplay outside of **dw:define-program-framework**, you need to create a redisplay object that you can pass to **dw:do-redisplay**, which, as its name says, does the redisplay. Creating a redisplay object is the job of **dw:redisplayer**; use this macro to enclose your redisplay function.

Here is an example that demonstrates the use of **dw:redisplayer**, **dw:do-redisplay**, and **dw:redisplayable-present**. The function waits for a character to be input before redisplaying:

```
(defvar *l*)

(defun redisplay-test-1 ()
  (fresh-line)
  (setq *l* (copy-list '("Old Top" "Old Middle" "Old Bottom")))
  (let ((displayer (dw:redisplayer ()
                     (dolist (thing *l*)
                       (dw:redisplayable-present thing 'string
                                                 :unique-id thing)
                       (terpri)))))
    (dw:do-redisplay displayer)
    (read-char)
    (push "New Top" *l*)
    (dw:do-redisplay displayer)
    (read-char)
    (pop (cddr *l*))
    (dw:do-redisplay displayer)
    (read-char)
    displayer))
```

Here are more elaborate examples, both of which require the following auxiliary internal function, which sets up ***L*** and runs **dw:do-redisplay**. **dw:do-redisplay** uses, in the examples, **dw:redisplayer** to look at ***L*** to decide what output to do.

```
(defun redisplay-test-2-internal (displayer stream)
  (fresh-line)
  (setq *l* (loop for (symbol . value) in '((*a* . a)
                                            (*bb* . b)
                                            (*c* . cc))
                  do (set symbol value)
                  collect symbol))
  (dw:do-redisplay displayer stream)
  (read-char)
  (set '*@* 'ddd) (push '*@* (cddr *l*))
  (dw:do-redisplay displayer stream)
  (read-char)
  (set '*c* 'not-c)
  (dw:do-redisplay displayer stream)
  (read-char)
  (pop (cdr *l*))
  (dw:do-redisplay displayer stream)
  (read-char)
  displayer)
```

This example illustrates the use of **dw:with-redisplayable-output** and **dw:redisplayable-format**:

```
(defun redisplay-test-2 (&optional (stream *standard-output*))
  (redisplay-test-2-internal
    (dw:redisplayer ()
      (dolist (thing *l*)
        (dw:with-redisplayable-output (:unique-id thing)
          (dw:redisplayable-format t "~S: " thing)
          (dw:redisplayable-present (eval thing) 'expression))
        (terpri)))
    stream))
```

Try moving the mouse over the results of the preceding example. Note that both the symbols and their values are sensitive.

This illustrates the use of **dw:independently-redisplayable-format**:

```
(defun redisplay-test-2a (&optional (stream *standard-output*))
  (redisplay-test-2-internal
    (dw:redisplayer ()
      (dolist (thing *l*)
        (dw:with-redisplayable-output (:unique-id thing)
          (dw:independently-redisplayable-format t "~S: ~S~%"
                                                 thing (eval thing)))))
    stream))
```

Try moving the mouse over the results of the preceding example. Only the last value is mouse-sensitive.

Here is an example that shows how you can use redisplay functions to display non-questions in **dw:accepting-values** output:

```
(defun another-test ()
  (let ((n 5)
        (m 6))
    (dw:accepting-values ()
      (setq n (accept '((mod 7)) :prompt "The number" :default n))
      (setq m (mod (1+ n) 7))
      (dw:with-redisplayable-output (:cache-value m :unique-id 'm)
        (format t " (The next number is ~D.)~%" m))
      m)))
```

### Incremental Redisplay of Nested Structures

Here is an example you can follow.

```
(dw:define-program-framework callers
 :command-definer t
 :command-table (:inherit-from nil)
 :panes
 ((display :display
           :redisplay-function 'display-callers
           :incremental-redisplay t
           :end-of-page-mode :truncate
           :margin-components
           `dw:((margin-ragged-borders )
                (margin-scroll-bar )
                (margin-scroll-bar :margin :bottom)
                (margin-white-borders :thickness 2)))
 (interactor :interactor)
 (menu :command-menu))
 :state-variables
 ((display-format :text ((member :text :graph)))
 (root-nodes nil)))

(defmethod (display-callers callers) (stream)
  (ecase display-format
    (:text
      (dolist (top root-nodes)
        (callers-node-display-self-and-inferiors top stream)))
    (:graph
      (format-graph-from-root root-nodes
                              #'callers-node-display-in-box
                              #'callers-node-inferior-nodes
                              :stream stream
                              :root-is-sequence t
                              :orientation :horizontal
                              :border nil))))
```

```
(defflavor callers-node
        (caller callee calls-how
         (inferiors-visible nil) inferior-nodes)
        ()
  (:writable-instance-variables inferiors-visible)
  (:constructor make-callers-node (caller callee calls-how)))

(defmethod (sys:print-self callers-node) (stream ignore slash)
 (if slash
    (sys:printing-random-object (self stream :typep)
     (princ self stream))
    (present caller 'sys:function-spec :stream stream)))

(define-presentation-type callers-node ()
  :no-deftype t
  ;; printer from print-self above via sys:expression.
  :parser
  ((stream)
  (dw:read-char-for-accept stream)
  (sys:parse-error "You can only enter the node with the mouse.")))

(defmethod (callers-node-inferior-nodes callers-node) ()
 (when inferiors-visible
  (unless (variable-boundp inferior-nodes)
   (setq inferior-nodes (make-callers-nodes caller)))
  inferior-nodes))

(defmethod (callers-node-display-in-box callers-node) (stream)
 (dw:with-redisplayable-output
    (:unique-id self
    :cache-value t      ;Always draws the same
    :stream stream)
  (surrounding-output-with-border (stream)
   (present self 'callers-node :stream stream))))
```

```
(defmethod (callers-node-display-self-and-inferiors callers-node)
              (stream &optional (depth 0))
 (dw:with-redisplayable-output
    (:unique-id self :cache-value inferiors-visible :stream stream)
  (send stream :increment-cursorpos
         (* depth 2) 0 :character)
  (dw:with-output-as-presentation
     (:object self :type 'callers-node
               :stream stream
               :allow-sensitive-inferiors nil)  ;Looks nicer
   (present self 'callers-node :stream stream)
   (format stream
             (cadr (assoc calls-how si:*who-calls-how-alist*)) callee))
  (terpri stream)
  (dolist (inferior (callers-node-inferior-nodes self))
   (callers-node-display-self-and-inferiors inferior stream (1+ depth)))))

(defun make-callers-nodes (callee)
 (let ((result nil))
  (si:map-over-callers callee
                            #'(lambda (caller calls-how)
                                 (push
                                  (make-callers-node caller callee calls-how)
                                  result)))
  result))

(compile-flavor-methods callers-node)

(define-callers-command (com-clear-all-callers :menu-accelerator t)
                                 ()
 (setq root-nodes nil)
 (send dw:*program-frame* :redisplay-pane 'display t))

(define-callers-command
 (com-add-callers :menu-accelerator t)
 ((callers '((sequence sys:function-spec)) :confirm t))
 (setq root-nodes (append root-nodes
                                  (loop for caller in callers
                                         nconc (make-callers-nodes caller)))))
```

```
(define-callers-command
 (com-set-display-format :menu-accelerator t
                          :menu-documentation-include-defaults t)
 ((format '((member :text :graph))
          :default (if (eq display-format :text) :graph :text)))
 (unless (eq format display-format)
  (setq display-format format)
  ;; Could rely on incremental redisplay, but why make it work extra hard?
  (send dw:*program-frame* :redisplay-pane 'display t)))

(define-callers-command
 (com-toggle-callers-visibility)
 ((callers-node 'callers-node))
 (setf (callers-node-inferiors-visible callers-node)
       (not (callers-node-inferiors-visible callers-node))))

(define-presentation-to-command-translator
 com-toggle-callers-visibility
  (callers-node) (callers-node)
 `(com-toggle-callers-visibility ,callers-node))

(compile-flavor-methods callers)
```

### Incremental Redisplay of Tables

The table formatting facilities are specially written to cooperate with redisplay, so there is nothing special about doing this. Generally, to display a set of similar structures, you put a **dw:with-redisplayable-output** *outside* the **formatting-row** with only a **:unique-id**, and a **dw:with-redisplayable-output** *inside* the **formatting-cell**s with only a **:cache-value**. If you can cheaply tell whether any attribute of the object has changed (for example, using a tick) a cache-value outside may improve performance.

*Example:*

```
(defun redisplay-processes (&optional (processes sys:all-processes))
 (formatting-table ()
  (dw:with-redisplayable-output (:unique-id 'headings :cache-value t)
   (formatting-column-headings (t :underline-p t)
    (with-character-face (:italic)
     (formatting-cell () "Process Name")
     (formatting-cell () "State")
     (formatting-cell () "Priority")
     (formatting-cell () "Quantum")
     (formatting-cell (t :align :right) " %")
```

```
   (formatting-cell () "Idle"))))
(dolist (process processes)
 (dw:with-redisplayable-output (:unique-id process)
  (formatting-row ()
   (formatting-cell ()
    (let ((name (process-name process)))
     (dw:with-redisplayable-output (:cache-value name)
      (present process))))
   (formatting-cell ()
    (let ((whostate (cp::process-whostate-or-special process)))
     (dw:with-redisplayable-output (:cache-value whostate)
      (write-string whostate))))
   (formatting-cell ()
    (let ((priorities (list (si:process-priority process)
                            (si:process-base-priority process))))
     (dw:with-redisplayable-output
        (:cache-value priorities :cache-test #'equal)
      (prin1 (first priorities))
      (let ((diff (reduce #'- priorities)))
       (unless (zerop diff)
        (write-string (format nil "~@D" diff)))))))
   (formatting-cell ()
    (let ((quanta (list
                    (tv:peek-process-quantum-remaining process)
                    (process-quantum process))))
     (dw:with-redisplayable-output
        (:cache-value quanta :cache-test #'equal)
      (format t "~4D/~D" (first quanta) (second quanta)))))
   (formatting-cell (t :align :right)
    (let ((%tage (send process :percent-utilization)))
     (dw:with-redisplayable-output
        (:cache-value %tage
        :cache-test #'(lambda (x y)
                          ;; Equal as a percentage.
                          (= (round (* x 100))
                             (round (* y 100)))))
```

```
        (format t "~1,1,4$%" %tage))))
      (formatting-cell ()
       (let ((idle (send process :idle-time)))
        (dw:with-redisplayable-output
           (:cache-value idle
           :cache-test #'(lambda (x y)
                             (flet
                              ((round-idle (x)
                                (cond
                                 ((null x) nil)
                                 ((< x 60) x)
                                 ((< x 3600) (* 60 (floor x 60)))
                                 (t (* 3600 (floor x 3600))))))
                              (eql (round-idle x) (round-idle y)))))
         (present idle 'cp::process-idle-time)))))))))

(defun redisplaying-processes ()
  (fresh-line)
  (let ((displayer (dw:redisplayer ()
                     (redisplay-processes))))
    (loop
      (dw:do-redisplay displayer)
      (sleep 1))))
```

Note, from the above example, that **redisplay-processes** can be called from outside of redisplay. The **dw:with-redisplayable-output** macro is harmless outside of **dw:redisplayer**. Also note that the redisplay context is dynamically scoped, not lexically. Finally, note the use of variables as **:cache-values**, since the state of a process can change between passes.

## Incremental Redisplay of Graphs

**format-graph-from-root** supplies a box around each node of a graph by default. Normally, when doing redisplay, you want the output cache to contain both the box and its contents. So, you have to arrange for the **dw:with-redisplayable-output** to include both.

When doing graph redisplay in a program framework, it is usually desirable to give the pane **:end-of-page-mode :truncate**, as this allows drawing the whole graph without moving the viewport each time redisplay happens. In this case, the window should also be given a scroll bar.

Here is the non-redisplay version of a graph-formatting routine:

```
(defun flavor-components (flavor-name)
  (flavor::flavor-local-components
    (flavor:find-flavor flavor-name)))
```

```
(defun present-flavor (flavor-name &optional (stream *standard-output*))
  (present flavor-name 'flavor:flavor-name :stream stream))

(cp:define-command (com-show-flavor-tree :command-table "Global")
    ((root-flavor-name 'flavor:flavor-name))
  (fresh-line)
  (format-graph-from-root root-flavor-name #'present-flavor #'flavor-components))
```

Here is the code needed to make the graph redisplay:

```
(dw:define-program-framework flavor-grapher
  :panes ((command-menu :command-menu)
          (graph :display
                 :redisplay-function 'redisplay-graph
                 :typeout-window t
                 :end-of-page-mode :truncate
                 :incremental-redisplay t
                 :margin-components dw::*listener-margin-components*)
          (interactor :interactor))
  :command-definer t
  :state-variables ((root-node nil)
                    (node-printer #'present-flavor)
                    (node-inferior-producer #'flavor-components)))
(defflavor grapher-node
        (object
         printer
         inferior-producer
         inferiors
         (inferiors-visible nil))
        ()
  (:readable-instance-variables object printer)
  (:writable-instance-variables inferiors-visible)
  (:constructor make-grapher-node (object printer inferior-producer)))
(define-presentation-type grapher-node ()
   :no-deftype t
   :printer ((node stream)
             (funcall (grapher-node-printer node)
                      (grapher-node-object node) stream)))
(defmethod (grapher-node-inferior-nodes grapher-node) ()
  (when inferiors-visible
    (if (variable-boundp inferiors)
        inferiors
        (setq inferiors (map 'list
                             #'(lambda (object)
                                 (make-grapher-node object
                                        printer inferior-producer))
```

```
                              (funcall inferior-producer object))))))
(defmethod (grapher-node-draw grapher-node) (stream)
  (dw:with-redisplayable-output (:stream stream
                                 :unique-id self
                                 :cache-value inferiors-visible)
    (dw:with-output-as-presentation (:stream stream
                                     :type 'grapher-node
                                     :object self
                                     :single-box t
                                     :allow-sensitive-inferiors nil)
      (surrounding-output-with-border (stream)
        (present self 'grapher-node :stream stream)))))
(defmethod (redisplay-graph flavor-grapher) (stream)
  (when root-node
    (format-graph-from-root root-node #'grapher-node-draw
                                      #'grapher-node-inferior-nodes
                                               :border nil
                                               :stream stream)))

(define-flavor-grapher-command (com-set-root-flavor-name
                                                :menu-accelerator t)
                               ((flavor-name 'sys:flavor-name
                                                :confirm t))
  (setq root-node (make-grapher-node flavor-name
                              node-printer node-inferior-producer)))
(define-flavor-grapher-command (com-toggle-inferior-visibility )
                               ((node 'grapher-node))
  (setf (grapher-node-inferiors-visible node)
                       (not (grapher-node-inferiors-visible node))))
(define-presentation-to-command-translator toggle-this-node
                                                (grapher-node) (node)
  `(com-toggle-inferior-visibility ,node))

(compile-flavor-methods grapher-node flavor-grapher)
```

### Incremental Redisplay of Graphics

Redisplay is oriented around rectangles. Some graphics may have shapes which overlap in their bounding rectangles, even though they don't overlap in what they draw. The current implementation of incremental redisplay will not always function properly in this cases. Other than that, there is nothing special about graphical output as opposed to text output.

*Example:*

```
(dw:define-program-framework circles
 :command-definer t
 :command-table (:inherit-from nil
                      :kbd-accelerator-p t)
 :top-level (dw:default-command-top-level :echo-stream ignore)
 :panes ((display :display
                    :redisplay-function 'draw-circles
                    :incremental-redisplay t)
          (menu :command-menu))
 :state-variables ((circles nil)))
(defstruct circle
  center-x
  center-y
  radius)
(defmethod (draw-circles circles) (stream)
 (dolist (circle circles)
  (dw:with-redisplayable-output
     (:unique-id circle :stream stream :cache-value t)
   (dw:with-output-as-presentation
      (:object circle :type 'circle :stream stream)
    (graphics:draw-circle
     (circle-center-x circle) (circle-center-y circle)
     (circle-radius circle) :stream stream)))))
(define-circles-command (com-delete-circle )
    ((circle 'circle))
   (setq circles (delete circle circles)))

(define-presentation-to-command-translator
 delete-this-circle (circle) (circle)
 `(com-delete-circle ,circle))
(define-circles-command (com-add-circle )
   ((x 'number :default 100)
   (y 'number :default 100)
   (radius 'number :default 50))
  (push (make-circle :center-x x
                       :center-y y
                       :radius radius)
        circles))
(define-presentation-to-command-translator add-circle-here
  (dw:no-type :documentation "Add a circle here.")
  (ignore &key x y)
 `(com-add-circle ,x ,y))

(define-circles-command (com-delete-all-circles :menu-accelerator t)
   ()
  (setq circles nil))

(compile-flavor-methods circles)
```

### Writing Formatted Output Macros

Given a continuation (usually a closure) and a stream, **dw:continuation-output-size** tells you how much room, in spaces or pixels, the continuation will require on the stream. This is useful, for example, for making windows no larger than necessary to accommodate formatted displays. The reference documentation for this facility includes an example ( see the function **dw:continuation-output-size**).

**dw:named-value-snapshot-continuation** is a macro that makes separate bindings for free variables referenced in its body; that is, it "snapshots" the free variables at the time the closure is constructed. This provides lexical separation between variables in the inner loops of a formatting function and variables with the same names in the outer loops. The reference documentation for **dw:named-value-snapshot-continuation** contains additional details on when and how to use this facility, including examples. See the function **dw:named-value-snapshot-continuation**.

### Table of Replay and Redisplay Facilities

**dw:displayed-presentation-set-highlighting** *displayed-presentation window* &optional (*highlighting-mode* **:underline**)
Highlights a displayed presentation.

**dw:displayed-presentation-clear-highlighting** *displayed-presentation window* &optional (*highlighting-mode* **:underline**)
Eliminates highlighting of a displayed presentation.

**dw:redisplayable-present** *object* &optional *presentation-type* &key (*stream* ***standard-output***) (*unique-id* **nil**) &allow-other-keys
Presents an object redisplayably.

**dw:redisplayable-format** *stream format-string* &rest *format-args*
Outputs a formatted string redisplayably.

**dw:independently-redisplayable-format** *stream format-string* &rest *format-args*
Outputs a formatted string such that each format argument is independently redisplayable.

**dw:with-redisplayable-output** (&key *stream cache-value unique-id* (*cache-test* **#'eql**) *copy-cache-value* (*id-test* **#'eql**) ) &body *body*

Introduces a caching point for incremental redisplay.

**dw:redisplayer** (&optional *stream*) &body *body*
Creates a redisplay object out of its *body* which can be used to do incremental redisplay on *stream*.

**dw:do-redisplay** *redisplay-piece* &optional (*stream* ***standard-output***) &key *full-set-cursorpos truncate-p once-only save-cursor-position*
Causes incremental redisplay from a redisplay object (created by **dw:redisplayer**.

**dw:with-output-to-presentation-recording-string** (*stream*) &body *body*
Binds local environment to output to a string that records presentations resulting from calls to **present** and **dw:with-output-as-presentation**.

**dw:with-replayable-output** (&rest *parameters*) &body *body*
Binds the local environment such that all of the output generated by *body* becomes a single, replayable presentation.

**dw:with-resortable-output** ((*list key* &key *copy-of*) &rest *sort-clauses*) *other-parameters* &body *body*
Binds the local environment such that all of the output generated by *body* becomes a single, resortable presentation.

**dw:continuation-output-size** *continuation stream* &optional (*unit* **:pixel**)
Determines the amount of space a specified continuation would require for output on a specified stream.

**dw:named-value-snapshot-continuation** *name var-list* &body *body*
Generates a lexical closure of its *body*, except that it snapshots the current values of lexical variables used free within *body*.

### Managing Your Program Frame

In an earlier chapter, "Defining Your Own Program Framework", we introduced the macro for creating an application program framework and the Layout Designer, a tool to use to automatically write such a macro. In the present chapter, we present the facilities and techniques you can use to modify and extend your program framework beyond what you can accomplish using the Layout Designer by itself.

The topics covered in this chapter fall into the following areas: the top-level loop and redisplay function, the commands and command menus, and the window layout. Before discussing these topics, however, we look at the various ways you can arrange for the user to invoke your application program.

### Invoking an Application Program

The most often used and most straightforward way to invoke an application program is to use the SELECT key and the select character you define for the program in your **dw:define-program-framework** form. Even if you do not assign a character, you can still use the CP command Select Activity to select a program, if you have not set the **:selectable** option to **nil**.

If you do not want an application program to be selectable, then you do set the **dw:define-program-framework :selectable** option to **nil**. There are two ways to invoke a program that is not selectable. One way is to call **dw:make-program** to make an instance of the given program flavor and to apply **dw::run-program-top-level** to the result. Another, way is to use **allocate-resource** to allocate a program frame of the type of your program, call the frame with **tv:window-call**, and then apply **dw:program-frame-top-level**.

The following sample program illustrates the first method. It also illustrates how to write a redisplay function for a program's top-level loop, in this case called **simple-menu-top-level**. Note the use of **dw:with-redisplayable-output dw:do-**

**redisplay**, and **dw:read-program-command**.

```
(dw:define-program-framework simple-menu
  :selectable nil
  :command-definer t
  :top-level (simple-menu-top-level)
  :command-table
    (:kbd-accelerator-p t
     :inherit-from '("standard scrolling"
                     "standard arguments"))
  :state-variables
    ((item-list nil)
     (selected-item nil)
     (displayer)
     (stream)))
(define-presentation-type simple-menu-item ((&key alist))
  :printer ((item stream)
            (write-string (string item) stream)))

(defmethod (redisplay-choices simple-menu) (stream)
 (let ((presentation-type `(simple-menu-item :alist ,item-list)))
  (dw:formatting-item-list (stream :row-wise nil)
   (loop for item in item-list
         do
     (dw:formatting-cell (stream)
      (dw:with-redisplayable-output
         (:stream stream
          :unique-id item
          :cache-value (eql item selected-item))
       (with-character-face
        ((if (eql item selected-item) :bold :roman)
        stream)
        (present item presentation-type :stream stream
                   :single-box t
                   :allow-sensitive-inferiors nil
```

```
                    :allow-sensitive-raw-text nil))))))))
(defmethod (simple-menu-top-level simple-menu) (real-stream)
 (setf stream real-stream
       displayer (dw:redisplayer (stream)
                    (redisplay-choices self stream)))
 (tv:with-blinker-visibility (stream nil)
  (catch 'return-item
   (loop
    (dw:do-redisplay displayer stream)
    (multiple-value-bind (command arguments)
       (dw:read-program-command self
                                       :echo-stream #'ignore
                                       :notification nil)
     (when command
      (apply command arguments)))))))
(define-simple-menu-command (com-choose-item)
   ((item 'simple-menu-item))
  (setf selected-item item))

(dw:define-presentation-to-command-translator choose-item
  (simple-menu-item :gesture :select
                         :documentation "Choose this item")
  (item)
 `(com-choose-item ,item))
(define-simple-menu-command
 (com-exit :keyboard-accelerator (#\end)) ()
  (throw 'return-item selected-item))

(defun simple-menu-choose
        (item-list &optional (stream *query-io*))
 (fresh-line stream)
 (let ((program (dw:make-program 'simple-menu)))
  (setf (simple-menu-item-list program) item-list)
  (dw::run-program-top-level program stream)))
```

This next example illustrates the second method and additionally demonstrates how to create a pop-up window that does not have its own process. To do this, you need to create the program without its own process and run its top level in the user's process. To do this, use the **:process nil** option to the **dw:program-frame** resource. Use **dw:program-frame-top-level** to call the program. Give **:selectable nil** if the program cannot also be run by itself. This also presents an example of how to write a display function. Note the use of **catch** and **throw** for the redisplay function **display-buttons**. Also note the use of **tv:window-call** and **dw:program-frame-top-level**.

```
(dw:define-program-framework buttons
  :command-definer t
  :selectable nil
  :command-table (:inherit-from ()
                              :kbd-accelerator-p 't)
  :top-level (dw:default-command-top-level
               :echo-stream ignore)
  :panes ((buttons :display
                   :size-from-output t
                   :redisplay-after-commands t :typeout-window t
                   :incremental-redisplay t
                   :redisplay-function 'display-buttons)
          (command-menu :command-menu :menu-level :top-level))
  :size-from-pane buttons
  :configurations '((dw::main
                      (:layout
                        (dw::main :column buttons command-menu))
                      (:sizes
                        (dw::main
                          (buttons :ask-window self
                                   :size-for-pane buttons)
                          (command-menu :ask-window self
                                        :size-for-pane command-menu)
                          :then))))
  :state-variables ((on-buttons nil)))
(define-presentation-type button ()
   :expander '((integer 1 10)))

(defmethod (display-buttons buttons) (stream)
 (formatting-item-list (stream :n-rows 2)
  (loop for button from 1 to 10 do
    (formatting-cell (stream)
     (let ((on-p (member button on-buttons)))
      (dw:with-redisplayable-output
         (:stream stream
          :unique-id button
          :cache-value on-p)
       (dw:with-output-as-presentation
          (:stream stream :object button :type 'button
                   :single-box t
                   :allow-sensitive-inferiors nil)
        (surrounding-output-with-border
         (stream :shape :circle :thickness 2
                 :filled on-p)
         (present button 'button :stream stream)))))))))
```

```
(define-buttons-command (com-turn-off-all-buttons
                                 :menu-accelerator t)
                               ()
 (setq on-buttons nil))
(define-buttons-command (com-toggle-button ) ((button 'button))
  (setq on-buttons (if (member button on-buttons)
                       (remove button on-buttons)
                       (adjoin button on-buttons))))

(define-presentation-to-command-translator toggle-button
  (button) (button)
 `(com-toggle-button ,button))

(define-buttons-command (com-done :menu-accelerator t
                                    :keyboard-accelerator #\End) ()
 (throw 'done on-buttons))
(defun button-choose (&optional (on-buttons '(1 3)))
  (catch 'done
    (using-resource (frame dw:program-frame 'buttons
                           (tv:mouse-default-superior)
                           :temporary-p t :process nil)
      ;; When sizing at first, need maximum space.
      (setf (buttons-on-buttons (send frame :program)) nil)
      (let ((max-width 0)
            (total-height 0))
        (dolist (pane-name '(buttons command-menu))
          (let ((pane (send frame :get-pane pane-name)))
            (multiple-value-bind (width height)
                (send frame :inside-size-for-pane pane pane-name)
              (incf width
                    (- (send pane :width) (send pane :inside-width)))
              (incf height
                    (- (send pane :height) (send pane :inside-height)))
              (maxf max-width width)
              (incf total-height height))))
        (send frame :set-inside-size max-width total-height))
      (setf (buttons-on-buttons (send frame :program)) on-buttons)
      (tv:window-call (frame :deactivate)
        (dw:program-frame-top-level frame))
      (buttons-on-buttons (send frame :program)))))
```

### The Top-Level Loop

The most often used top-level loop form is the default that is supplied for the **:top-level** option to **dw:define-program-framework**. This is the function **dw:default-command-top-level**. The next most frequent practice is to write a top-level loop

that makes some very simple modification to **dw:default-command-top-level**, perhaps just calling it with different options.

### Modifying the Default Top-Level Function

**Making the default in your listener pane** ***form-preferred*****:**

```
(dw:define-program-framework something
                :top-level (dw:default-command-top-level
                             :dispatch-mode :form-preferred)
  .
  .
  .)
```

**Changing the prompt:**

```
(define-program-framework foo
  :top-level (my-top-level)
        .
        .)

(defun my-top-level (program)
  (let ((prompt #'si:arrow-prompt))
    (dw:default-command-top-level program :prompt prompt)))
```

**Combining your own form(s) with the default:**

This example comes from the flavor examiner program. It simply runs the **examiner-help** program first before applying the **dw:default-command-top-level** function to the program and its options.

```
(defun examiner-top-level (program &rest options)
  (examiner-help program (dw:get-program-pane 'command) nil)
  (apply #'dw:default-command-top-level program options))
```

In general, your top-level function is written in this manner:

```
(dw:define-program-framework something
  :top-level (my-own-top-level)
  .
  .
  .)

(defun my-own-top-level (program)
                                ;Notice that the program
                                ;instance is passed as the first arg.
 ...                            ;Insert own code here.
 (dw:default-command-top-level program
                                   :dispatch-mode :form-preferred))
```

### Writing a Non-Echoing Command Loop

The command processor expects an interactive stream, that is one that does input editing. So if you do not want your command loop to echo commands, not only must you default the echoing of the commands themselves, you must also arrange not to require an input editing stream. This is most easily done by defining the program to have single-character command accelerators. The program need not define any such accelerators, this is just to avoid the use of the full command processor. Note that a program that had any keyboard commands (especially ones with long names) but did not echo would be very difficult to use. Note the **:echo-stream ignore** option to **dw:default-command-top-level**.

```
(dw:define-program-framework no-echo
  :command-definer t
  :command-table (:inherit-from nil
                  :kbd-accelerator-p t)
  :top-level (dw:default-command-top-level :echo-stream ignore)
  :panes ((display :display :redisplay-function 'draw-circles))
  :state-variables ((circles nil)))

(defstruct circle
  center-x
  center-y
  radius)
(defmethod (draw-circles no-echo) (stream)
  (dolist (circle circles)
    (dw:with-output-as-presentation
        (:object circle :type 'circle :stream stream)
      (graphics:draw-circle
        (circle-center-x circle)
        (circle-center-y circle)
        (circle-radius circle)
        :stream stream))))

(define-no-echo-command (com-add-circle)
                                  ((x 'number :default 100)
                                  (y 'number :default 100)
                                  (radius 'number :default 50))
 (push (make-circle
          :center-x x
          :center-y y
          :radius radius)
        circles))
(define-no-echo-command (com-delete-circle )
                                  ((circle 'circle))
 (setq circles (delete circle circles)))
```

```
(define-presentation-to-command-translator
  delete-this-circle (circle) (circle)
  `(com-delete-circle ,circle))

(compile-flavor-methods no-echo)
```

**Implementing a Timeout At Command Level**

Timeouts with actions should be performed in the command loop. When reading a command, an event should trigger a return to the loop to do this check. It is best not to have the trigger do the action itself, since the context is not as controlled. If you have an accelerated command loop, you can use the **:timeout** option to **cp:read-accelerated-command**. Alternatively, you can insert your own wakeup blips and process them yourself.

Example using **:timeout**:

```
(dw:define-program-framework timeouts-1
 :command-definer t
 :top-level (timeouts-1-top-level)
 :command-table (:inherit-from
                       '("colon full command" "standard arguments"
                        "input editor compatibility")
                       :kbd-accelerator-p 't)
 :panes ((pane-1 :display :height-in-lines 1
                   :incremental-redisplay nil
                   :redisplay-function 'show-time
                   :redisplay-after-commands t)
          (pane-2 :listener))
 :configurations '((dw::main
                     (:layout (dw::main :column pane-1 pane-2))
                     (:sizes (dw::main (pane-1 1 :lines)
                                         :then (pane-2 :even))))))
(defmethod (show-time timeouts-1) (stream)
  (format stream "The time is ~\\datime\\."))

(defmethod (timeouts-1-top-level timeouts-1) (&rest args)
 (apply #'dw:default-command-top-level self
        :timeout (* 60 10)     ;ten seconds
        :unknown-accelerator-is-command t
        args))
```

Example using **:window-wakeup**:

```
(dw:define-program-framework timeouts-2
 :command-definer t
 :top-level (timeouts-2-top-level)
 :command-table (:inherit-from '("user"))
 :panes ((pane-1 :display
                     :height-in-lines 1 :incremental-redisplay nil
                     :redisplay-function 'show-time
                     :redisplay-after-commands t)
           (pane-2 :listener))
 :configurations '((dw::main
                       (:layout (dw::main :column pane-1 pane-2))
                       (:sizes (dw::main (pane-1 1 :lines)
                                      :then (pane-2 :even))))))
(defmethod (show-time timeouts-2) (stream)
  (format stream "The time is ~\\datime\\."))

(define-presentation-type timeout-wakeup ()
   :expander 'dw::window-wakeup)

(defmethod (timeouts-2-top-level timeouts-2) (&rest args)
 (labels ((set-next-timeout ()
             ;; We could use the repeat option, but this way
             ;;when the program is killed
             ;; you don't have it lying around any more.
             (si:add-timer-queue-entry
              (+ (time:get-universal-time) 10.) nil
              "timeout"
              #'(lambda (stream)
                  (send stream :force-kbd-input
                         (dw::make-presentation-blip
                          :presentation-type 'timeout-wakeup)
                         t))    ;Don't hang
              (si:follow-syn-stream *standard-input*)))
           (wakeup (blip)
            (dw:presentation-blip-case blip
             (timeout-wakeup
              (set-next-timeout)
              (throw 'dw::return-from-read-command
               (values nil blip :wakeup)))
             (otherwise
              (dw::default-window-wakeup-handler blip)))))
  (set-next-timeout)
  (apply #'dw:default-command-top-level self
          :window-wakeup #'wakeup args)))
```

## Handling Asynchronous Window System Events

There are a number of generic functions for which you can define methods in your program which are called whenever something changes. These are:

**dw:after-program-frame-activation-handler**
**dw:after-program-frame-selection-handler**
**dw:before-program-frame-deactivation-handler**
**dw:before-program-frame-deexpose-handler**
**dw:before-program-frame-kill-handler**

The contexts in which these functions are called is unpredictable, though, so the methods should only do innocuous things like setting state flags. Additionally, wakeup blips are inserted into the program's I/O buffer when most events happen.

Example using methods:

```
(dw:define-program-framework complement-when-selected
  :select-key #\circle)

(defmethod
 (dw:after-program-frame-selection-handler
  complement-when-selected) (frame)
 (send (send frame :screen) :set-bow-mode nil))

(defmethod
 (dw:before-program-frame-deexpose-handler
  complement-when-selected) (frame)
 (send (send frame :screen) :set-bow-mode t))
```

Example using wakeups:

```
(dw:define-program-framework count-refreshes
  :top-level (count-refreshes-top-level)
  :state-variables ((refresh-count 0)))

;;; Start up and do Function Refresh
(defmethod
 (count-refreshes-top-level count-refreshes) (&rest args)
 (flet ((prompt (stream ignore)
          (format stream "Command (~D refreshes): " refresh-count))
         (wakeup (blip)
          (dw:presentation-blip-case blip
           (dw::window-wakeup-refresh
            (incf refresh-count)))
          (dw::default-window-wakeup-handler blip)))
   (apply #'dw:default-command-top-level self
           :prompt #'prompt
           :window-wakeup #'wakeup arg)))
```

**Commands and Command Menus**

**How Command Menus Work**

A command menu is a table. Conceptually, the items in the table are command verbs, that is, command names that are translated into commands. In the simplest cases, where all items come from the **:menu-accelerator** option to **dw:define-program-command**, this description is quite close to how command menus are implemented. In actuality, however, the implementation is done in a way that allows for more flexibility.

Menu items are presentations of the type **dw::command-menu-item**. Their data arguments specify **:menu-level** and **:command-table**. A **:menu-level** is a partition of command menu items, which normally corresponds to a pane in a program framework. The **:menu-level** option to the **:command-menu** pane type causes that corresponding subset to be displayed. Normally, all the items in a command menu will have the same **:menu-level**, and there will not be any other presentations, but this is not required.

The object in a command menu item presentation is a string. This string is unique within its menu level and command table. Or more exactly, the presentation stands for all command menu handlers which have that string. Normally, the presentation on the screen consists of the characters in that string, but this is not required. (For instance, see the calculator example in the file `sys:examples;define-program-framework.lisp`.)

The system defines a translator from **dw::command-menu-item** to commands. This translator looks up a command menu handler which matches the command-table, menu-level, and object string. Handlers can be inherited via the command table's **:inherit-from** option (this applies both to this lookup process and to the displaying done by command menu program panes). Conceptually, it is reasonable to think of a command menu handler as a special kind of translator from a command menu item to a command.

Use **dw:define-command-menu-handler** to define command menu handlers. Normally, your command menu handler will call **dw:standard-command-menu-handler**, which takes a command name and arguments as passed to the command form of **dw:define-command-menu-handler**, and does the standard actions for two mouse gestures.

The **dw:define-command-menu-handler** generated by the **:menu-accelerator** option to **dw:define-program-command** looks more or less like this:

```
(dw:define-command-menu-handler (,menu-accelerator-option
                                  ,program-name ,menu-level-option)
                                 (:gesture (:left :right)
                                  :documentation
                                  ,menu-documentation-option)
  (&rest args)
  (apply #'dw:standard-command-menu-handler ',command-name
         :command-table ',program-name args))
```

When the command form body is running, **dw:*command-menu-test-phase*** will be bound to **t** during the **:tester** phase and **:documentation** during the documentation phase. The body can throw to **dw:command-menu-test-phase** with a command

(list of command name and arguments) or string (in the documentation case). Note that if your body pops up a menu or reads from the keyboard to get arguments, it must look at this flag to prevent doing so except when the user really clicked.

Set up a program framework using this form:

```
(dw:define-program-framework command-menu-test
  :command-definer t
  :panes ((listener :listener)
          (command-menu :command-menu)))
```

Here is a command without any arguments: note that mouse-left and mouse-right effect the same result.

```
(define-command-menu-test-command
  (com-command-without-arguments :menu-accelerator "No args")
  ()
  (format t "No args~%"))
```

Here is a command with arguments: mouse-left defaults them and mouse-right gives a menu.

```
(define-command-menu-test-command
  (com-command-with-args :menu-accelerator "Some args")
  ((integer 'integer :default 69)
   (string 'string :default "foobar"))
  (format t "~D ~A~%" integer string))
```

Here is a command with **:confirm** arguments, mouse-left reads from the keyboard.

```
(define-command-menu-test-command
  (com-command-with-confirm :menu-accelerator "Confirm arg")
  ((file 'pathname :confirm t :prompt "File to delete"))
  (format t "~A~%" file))
```

Here is a command with the normal actions on mouse-left and mouse-right, and a new action on middle.

```
(define-command-menu-test-command
 (com-command-with-middle :menu-accelerator "Middle too")
 ((verbose 'boolean :default nil :prompt "Verbose"))
 (format t "The answer is ")
 (present verbose 'boolean)
 (terpri))

(dw:define-command-menu-handler
 ("Middle too" command-menu-test :top-level
 :gesture :middle)
 ()
 `(com-command-with-middle t))
```

Another way of doing the same thing:

```
(dw:define-command-menu-handler
 ("Middle two" command-menu-test :top-level
 :gesture (:left :middle :right))
 (&rest args &key gesture &allow-other-keys)
 (if (eq gesture :middle)
    `(com-command-with-middle t)
    (apply #'dw:standard-command-menu-handler 'com-command-with-middle args)))
```

Here is a command with just an enumeration argument. It makes mouse-right give a pop-up menu instead of an accept-values menu.

```
(define-presentation-type greek-letter ()
   :abbreviation-for `((member alpha beta gamma)))

(define-command-menu-test-command (com-command-with-enumeration )
                                  ((choice 'greek-letter :default 'alpha))
  (format t "The letter is ")
  (present choice 'greek-letter)
  (terpri))

(dw:define-command-menu-handler ("Choose" command-menu-test :top-level
                                 :gesture (:left :right))
                                (&key gesture)
  (if (eq gesture :left)
      `(com-command-with-enumeration )
    (case dw:*command-menu-test-phase*
      ((t) `(com-command-with-enumeration ))
      ((:documentation) (throw 'dw:command-menu-test-phase "Choose"))
      (otherwise
        (let ((choice (dw:menu-choose-from-set '(alpha beta gamma)
                                               'greek-letter
                                               :default 'alpha
                                               :prompt "Choose one")))
          (when choice
            `(com-command-with-enumeration ,choice)))))))
```

### Using Single-Character Accelerators

### Single-Character Command Accelerators Together with Ordinary Commands

The single-character commands used by the input editor are not the same as command accelerators. They are part of the small editor program that lets you do typein. Put the single-character commands you want at top-level on non-alphabetic characters and then arrange for alphabetic typein to invoke the input editor for a command. Otherwise, users must type : or m-X before an extended command. Here is an example:

```
(dw:define-program-framework commands-and-accelerators
 :command-definer t
 :top-level
 (dw:default-command-top-level :unknown-accelerator-is-command :alpha)
 :command-table
 (:inherit-from '("colon full command" "standard arguments"
                  "input editor compatibility")
 :kbd-accelerator-p t))

(define-commands-and-accelerators-command
 (com-show-time :keyboard-accelerator #\c-T)
 (&key (verbose 'boolean
                  :default nil :mentioned-default t))
 (if verbose
    (time:print-current-date-and-holidays)
    (format t "The time is ~\\datime\\~%")))
```

Note that users can type `c-T` or `Show Time :Verbose` to the command loop.

**Single-Character Command Accelerators and Lisp Forms**

```
(dw:define-program-framework forms-and-accelerators
 :command-definer t
 :top-level
 (dw:default-command-top-level :dispatch-mode :command-preferred
  :unknown-accelerator-is-command :alpha)
 :command-table
 (:inherit-from '("colon full command" "standard arguments"
                  "input editor compatibility" "user")
 :kbd-accelerator-p t))

(cp:define-command-accelerator
 show-herald forms-and-accelerators (#\c-H) () ()
 `(si:com-show-herald))
```

Note also that command table inheritance was used both to get accelerators and to get real CP commands.

### Sharing State among Program Commands

The best place to put state that is shared among various commands in the program is in the **:state-variables** of the program. These will be lexically apparent in the body of the commands, since these are methods on the program flavor. You can also use **:command-evaluator** to bind some special variables, but this is more awkward. The state variables will not be lexically apparent to mouse translators. To find them at this point, **dw:current-program** can be used. For example:

```
(dw:define-program-framework shapes-sequence
  :command-definer t
  :command-table (:inherit-from ())
  :panes ((display :display :typeout-window t
                   :redisplay-after-commands t
                   :incremental-redisplay nil
                   :redisplay-function 'show-shape)
          (pane-2 :interactor :height-in-lines 4))
  :configurations '((dw::main
                      (:layout
                        (dw::main :column display pane-2))
                      (:sizes
                        (dw::main (pane-2 4 :lines)
                                  :then (display :even)))))
  :state-variables ((shape :diamond)))
(define-presentation-type shape ()
   :expander 'keyword)

(defmethod (show-shape shapes-sequence) (stream)
  (dw:with-output-as-presentation
      (:stream stream
       :object shape
       :type 'shape
       :single-box t
       :allow-sensitive-inferiors nil)
    (surrounding-output-with-border (stream
                                     :shape shape
                                     :thickness 2)
      (present shape 'shape :stream stream))))
(define-shapes-sequence-command (com-set-shape )
                                ((new-shape 'shape))
  (setq shape new-shape))

(define-presentation-to-command-translator
  new-shape (shape)
  (shape &key window)
   (ignore shape)
  (let ((program
          (dw:current-program
            :window window
            :type 'shapes-sequence
            :error-p nil)))
    (when program
      `(com-set-shape ,(case (shapes-sequence-shape program)
                         (:diamond :oval)
                         (:oval :rectangle)
                         (otherwise :diamond))))))
```

Remember that state variables and methods are inherited, but configurations and panes are not. The methods defining commands are inherited, but actual command inheritance is handled by the **:command-table** option separately.

### Maintaining State with Accept-Values Panes

New values for queries in accept-values panes are not stored separately as they are for normal uses of **dw:accepting-values**. The **:default** given to **accept** by such panes is returned except at the time right after a new value has been entered. This is to allow the program to modify the state itself using other commands and other means besides the accept-values pane. If the accept-values pane saved newly entered values, this would be overridden each time. For these reasons, you should keep all of the state of your program in state-variables. Supply each as the **:default** and store it right back. For example:

```
(dw:define-program-framework avv-shapes-sequence
  :command-definer t
  :command-table (:inherit-from '("accept-values-pane"))
  :panes ((display :display
                   :typeout-window t
                   :redisplay-after-commands t
                   :incremental-redisplay nil
                   :redisplay-function 'show-shape)
          (accept-values :accept-values
                         :accept-values-function 'change-state)
          (pane-2 :interactor :height-in-lines 4))
  :configurations '((dw::main (:layout
                                (dw::main
                                  :column accept-values display pane-2))
                              (:sizes
                                (dw::main
                                  (accept-values 2 :lines)
                                  (pane-2 4 :lines)
                                  :then (display :even)))))
  :state-variables ((shape :diamond)
                    (thickness 2)))(define-presentation-type shape ()
   :expander 'keyword)
```

```
(defmethod (show-shape avv-shapes-sequence) (stream)
  (dw:with-output-as-presentation
      (:stream stream
       :object shape
       :type 'shape
       :single-box t
       :allow-sensitive-inferiors nil)
    (surrounding-output-with-border (stream
                                     :shape shape
                                     :thickness thickness)
      (present shape 'shape :stream stream))))
(define-avv-shapes-sequence-command
  (com-set-shape ) ((new-shape 'shape))
  (setq shape new-shape))

(defmethod (change-state avv-shapes-sequence) (stream)
  (setq shape (accept 'shape
                      :stream stream
                      :default shape
                      :prompt "Shape")
        thickness (accept 'integer
                          :stream stream
                          :default thickness
                          :prompt "Line thickness")))

;;; Note that using this translator also updates the displayed AVV state.
(define-presentation-to-command-translator new-shape (shape) (shape)
  `(com-set-shape ,(case shape
                     (:diamond :oval)
                     (:oval :rectangle)
                     (otherwise :diamond))))
```

## Incorporating Accept-Values Keyboard Commands Into Programs

If you have one large accept-values pane and you want the keyboard commands from accepting-values to work in your program too, include the command table named **accept-values-pane-with-keyboard-commands** in place of **accept-values-pane**. For example:

```
(dw:define-program-framework avv-with-keyboard
  :command-definer t
  :command-table
  (:inherit-from '("accept-values-pane-with-keyboard-commands")
   :kbd-accelerator-p 't)
  :panes ((accept-values
            :accept-values :typeout-window t
            :accept-values-function 'modify-state
            :size-from-output t :redisplay-after-commands t)
          (pane-1 :interactor))
  :configurations '((dw::main
                      (:layout (dw::main :column accept-values pane-1))
                      (:sizes
                        (dw::main (accept-values
                                    :ask-window self
                                    :size-for-pane accept-values)
                                  :then (pane-1 :even)))))
  :state-variables
  ((state (loop for i from 1 to 10 collect (list i t :a)))))
```

Note that we obey the rule of storing back into state variables what we get and passing it as the default, but not directly. Also note care needed to construct query identifiers.

```
(defmethod (modify-state avv-with-keyboard) (stream)
  (setq state (loop for (index boolean choice) in state
                  do
              (dw:with-redisplayable-output
                  (:stream stream
                   :unique-id index
                   :cache-value t)
                (format stream "State for entry #~D~%" index))
                  collect
                    (list index
                          (accept 'boolean :default boolean
                                  :stream stream
                                  :prompt "  Enabled"
                                  :query-identifier
                                  `(,index boolean))
                          (accept 'keyword :default choice
                                  :stream stream
                                  :prompt "  What kind"
                                  :query-identifier
                                  `(,index choice))))))
```

**Window Layout**

### A Program Frame with More Than One Configuration

Frame-Up does not at this time support separating the design of a configuration from design of the whole program. You can, however, use it several times to design the configurations and then merge them yourself using the editor. Call Frame-Up from different editor buffers on dummy program names and then collect the forms together into one **dw:define-program-framework** form, like the one in the example below.

Use **dw:set-program-frame-configuration** to change configurations. If you need more than one command menu, give each command menu a separate **:menu-level** value.

```
(dw:define-program-framework my-program
  :command-definer define-my-command
  :panes ((main-command-menu
             :command-menu
             :menu-level :main)
           (secondary-command-menu
             :command-menu
             :menu-level :secondary)
           (listener :listener))
  :configurations
  '((main
      (:layout
        (main :column main-command-menu listener))
      (:sizes
        (main
          (main-command-menu
            :ask-window self
            :size-for-pane main-command-menu)
          :then
          (listener :even))))
    (secondary
      (:layout
        (secondary :column
                   main-command-menu
                   secondary-command-menu
                   listener))
      (:sizes
        (secondary
          (main-command-menu :ask-window
                             self
                             :size-for-pane main-command-menu)
          (secondary-command-menu :ask-window
                                  self
                                  :size-for-pane
                                  secondary-command-menu)
          :then
          (listener :even))))))
(define-my-command
  (com-command-1 :menu-accelerator "Something ordinary"
                 :menu-level :main)
  ()
  (format t "~&Here is a first level command.~%"))
```

```
(define-my-command (com-enable-secondary-commands
                     :menu-accelerator "More commands"
                     :menu-level :main)
                   ()
  (dw:set-program-frame-configuration 'secondary))

(define-my-command
  (com-command-2 :menu-accelerator "Something extraordinary"
                 :menu-level :secondary)
  ()
  (format t "~&Here is a second level command.~%"))

(define-my-command
  (com-disable-secondary-commands
    :menu-accelerator "Fewer commands"
    :menu-level :secondary)
  ()
  (dw:set-program-frame-configuration 'main))

(compile-flavor-methods my-program)
```

### Table of Advanced Facilities for Program Frames

**dw:default-command-top-level** *program* &rest *options* &key (*window-wakeup* **#'dw::default-window-wakeup-handler**) &allow-other-keys

The default command loop function for programs created with **dw:define-program-framework**.

**dw:read-program-command** *program* &rest *options* &key *(:stream* ***query-io****)* *:prompt (:dispatch-mode* **:command-only***)* *:keyboard-accelerators :environment :window-wakeup :input-wait-handler :intercept-function* &allow-other-keys

Default command reading function for programs created via **dw:define-program-framework**.

**dw:make-program** *name* &rest *options*

Makes an instance of the program flavor *name*.

**dw:set-program-frame-configuration** *configuration-name* &optional *(frame* **dw:*program-frame****)*

**dw:program-frame**

A resource of program frames (of the kind used by **dw:define-program-framework**). The resource is created via **tv:defwindow-resource** with the **:initial-copies** option set to **nil** and the **:reuseable-when** option set to **:deactivated**.

**dw:program-frame-top-level** *window* &rest *args*
Calls a program previously defined with **dw:define-program-framework**.

**cp:read-accelerated-command** &key (*command-table* **cp:*command-table***) (*stream* ***query-io***) (*help-stream* **stream**) (*echo-stream* **stream**) (*whostate* **nil**) (*prompt* **nil**) (*command-prompt* **cp::*full-command-prompt***) (*full-command-full-rubout* **nil**) (*special-blip-handler* **nil**) (*timeout* **nil**) (*input-wait* **nil**) (*input-wait-handler* **nil**) (*form-p* **nil**) (*handle-clear-input* **nil**) (*catch-accelerator-errors* **t**) (*unknown-accelerator-is-command* **nil**) (*unknown-accelerator-tester* **nil**) (*unknown-accelerator-reader* **nil**) (*unknown-accelerator-reader-prompt* **nil**) (*abort-chars* **nil**) (*suspend-chars* **nil**) (*status* **nil**) (*intercept-function* **nil**) (*window-wakeup* **nil**)
Reads a Command Processor command input as a single-key accelerator.

**dw:standard-command-menu-handler** *command-name* &rest *args*
Takes *command-name* and arguments *args* as passed to the command form of a **dw:define-command-menu-handler** form, and does the standard actions for two mouse gestures.

**dw:*command-menu-test-phase***
This variable is bound to **t** during the **:tester** phase when a command form body is running, and to **:documentation** during the documentation phase.

**dw:define-command-menu-handler** *(command-name command-table menu-levels* &key *(:gesture* **:left***) (:documentation* **t***) :tester) arglist* &body *command-form*
Defines a menu handler for the command named *command-name* in *command-table* for *menu-levels*.

**dw:after-program-frame-activation-handler** *program frame*
A generic function to do simple things after your program frame is activated.

**dw:after-program-frame-selection-handler** *program frame*
A generic function to do simple things after your program frame is selected.

**dw:before-program-frame-deactivation-handler** *program frame*
A generic function to do simple things before your program frame is deactivated.

**dw:before-program-frame-deexpose-handler** *program frame*
A generic function to do simple things before your program frame is deexposed.

**dw:before-program-frame-kill-handler** *program frame*
A generic function to do simple things before your program frame is killed.

# Substrate Facilities

## Using the Window System

### Introduction to Using the Window System

"Using the Window System" is intended to explain how you, as a programmer, can use the set of facilities in Genera known collectively as the window system. Specifically, this part explains how to create windows, and what operations can be performed on them. It also explains how you can customize the windows you produce, by mixing together existing flavors to produce a window with the combination of functionality that your program requires. This section does not explain how to extend the window system by defining your own flavors.

Most of the window system concepts and facilities covered in this part apply to Dynamic Windows as well as static windows. This is explicitly mentioned in a number of places. Where the two kinds of windows diverge, we also point that out. The reference documentation for **dw:dynamic-window** refers you to the particular sections in this part that describe facilities for use with Dynamic Windows. See the flavor **dw:dynamic-window**. For more general information on the relationship of static and Dynamic Windows, see the section "Window Substrate Facilities".

To get the most out of this material, you should have a working familiarity with Symbolics Common Lisp. You should also have some experience with the Genera user interface, including the ways of manipulating windows, such as the [Edit Screen], [Split Screen], and [Create] commands from the System menu. Furthermore, you must understand something about flavors. While you need not be familiar with how methods are defined and combined, you should understand what message passing is, how it is used in Genera, what a flavor is, what a "mixin" flavor is, and how to define a new flavor by mixing existing flavors.

See the section "Flavors".

## Concepts

### Purpose of the Window System

The term *window system* refers to a large body of software used to manage communications between Genera programs and the user, via the console. The console consists of a keyboard, a mouse, and one or more screens.

The window system controls the keyboard, encoding the shifting keys, interpreting special commands such as the FUNCTION and SELECT keys, and directing input to the right place. The window system also controls the mouse, tracking it on the screen, interpreting clicks on the buttons, and routing its effects to the right places. The most important part of the window system is its control of the screens, which it subdivides into windows so that many programs can co-exist, and even run simultaneously, without getting in each other's way, sharing the screens according to a set of established rules.

**Windows**

When you use Genera, you can run many programs at once. You can have a Lisp Listener, an editor, a mail reader, and a network connection program (you can even have many of each of these) all running at the same time, and you can switch from one to the other conveniently. Interactive programs get input from the keyboard and the mouse, and send output to a screen. Since there is only one keyboard, it can only talk to one program at a time. However, each screen can be divided into regions, and one program can use one region while another uses another region. Furthermore, this division into regions can control which program the mouse talks to; if the mouse blinker (the thing on the screen that tracks the mouse) is in a region associated with a certain program, this can be interpreted as meaning that the mouse is talking to that program. Allowing many programs to share the input and output devices is the most important function of the window system.

The regions into which the screen is divided are known as *windows*. In your use of Genera, you have encountered windows many times. Sometimes there is only one window visible on the screen; for example, when you cold boot a Symbolics machine, it initially has only one window showing, and it is the size of the entire screen. If you start using the System menu's [Create], [Edit Screen], or [Split Screen] commands, you can make windows in various places of various sizes and flavors. Usually windows have a border around them (a thin black rectangle around the edges of the window), and they also frequently have a label in the lower left-hand corner or on top. This is to help the user see where all the windows are, what parts of the screen they are taking up, and what kind of windows they are.

Sometimes windows overlap; two windows may occupy some of the same space. While the [Split Screen] command will never do this, you can make it happen by creating two windows and simply placing them so that they partially overlap, by using [Edit Screen]. If you have never done so, you should try it. The window system is forced to make a choice here: Only one of those two windows can be the rightful owner of that piece of the screen. If both of the windows were allowed to use it, then they would get in each other's way. Of these two windows, only one can be *visible* at a time; the other one has to be not fully visible, but either partially visible or not visible at all. Only the visible window has an area of the screen to use.

If you play around with this, you will see that it looks as if one window is on top of the other, as if they were two overlapping pieces of paper on a desk and one were on top. Create two Lisp Listeners using the [Create] command of the System menu or the [Edit Screen] menu, so that they partially overlap, and then click Left on the one that is on the bottom. It will come to the top. Now click Left on the other one; it will come back up to the top. The one on top is fully visible, and the other one is not. We will return to the concepts of visible and not-fully-visible windows later in more detail.

From the point of view of the Lisp world, each window is a Lisp object. A window is an instance of some flavor of window. There are many different window flavors available; some of them are described in this document.

Windows can function as streams by accepting all the messages that streams accept. If you do input operations on windows, they read from the keyboard; if you do output operations on windows, they type out characters on the screen. The value of ***terminal-io*** is normally a window, and so input/output functions in Genera do their I/O to windows by default.

Windows have internal state, contained in instance variables, that indicate which screen the window is on, where on the screen it is, where its cursor is, what blinkers it has, how it fits into the window hierarchy, and much more. You can get windows to do things by sending them messages; they accept a wide variety of messages, telling them to do such things as changing their position and size, writing characters and graphics, changing their labels and borders, changing status in various ways, redrawing themselves, and much more. The main business of this document is to explain the meaning of the internal state of windows, and to explain what messages you can send and what those messages do.

### Hierarchy of Windows

Several Genera system programs and application programs present the user with a window that is split up into several sections, which are usually called *window panes* or *panes*. For example, the Inspector has six panes in its default configuration: the one you type forms into at the top, the menu, the history list, and the three inspection panes below the first three. The Display Debugger and Zmail also use elaborate windows with panes. These panes are not exactly the same as the other windows we have discussed, because instead of serving to split up the screen, they serve to split up the program's window itself. Sometimes you don't see this, because often the program's window is taking up the whole screen itself. Try going into the [Edit Screen] system and reshaping a whole Inspector or Zmail window. You will see that the panes serve to divide this window up into smaller areas.

In fact, the same window system functionality is used to split up a paned window into panes as is used to split up a screen into windows. Each pane is, in fact, a window in its own right. Windows are arranged in a hierarchy, each window having a superior and a list of inferiors. Usually the top of the hierarchy is a screen. In the example above, the Inspector window is an inferior of the screen, and the panes of the window are inferiors of the Inspector window. The screen itself has no superior (if you were to ask for its superior, you would get **nil**).

The position of a window is remembered in terms of its relative position with respect to the its superior; that is, what we remember about each window is where it is within its superior. To figure out where a window is on the screen, we add this relative position to the absolute position of the superior (which is computed the same way, recursively; the recursion terminates when we finally get to a screen). The important thing about this is that when a superior window is moved, all its inferiors are moved the same amount; they keep their relative position within the superior the same. You can see this if you play with the [Move Window] command in [Edit Screen].

One effect of the hierarchical arrangement is that you can use [Edit Screen] to edit the configuration of panes in a frame as well as to edit the configuration of windows on the screen, by clicking right on [Edit Screen]. If you have ever clicked right on [Edit Screen] while the mouse was on top of a window with inferiors, such as an editor, you will have noticed that you get a menu asking which of these two things you want to do. In fact, that menu can have more than two items; the number of items grows as the height of the hierarchy.

So, what [Edit Screen] really does is to manipulate a set of inferiors of some specific superior window, which may or may not be a screen. The set of inferiors that you are manipulating is called the *active inferiors* set; each inferior in this set is said to be *active*. Windows can be activated and deactivated. The active inferiors are all fighting it out for a chance to be visible on their superior. If no two active inferiors overlap, there is no problem; they can all be uncovered. However, whenever two overlap, only one of them can be on top. [Edit Screen] lets you change which active inferiors get to be on top. There is also a part of the window system called the *screen manager* whose basic job is to keep this competition straight. For example, it notices that a window that used to be covering up part of a second window has been reshaped, and so the second window is no longer covered and can be brought to the top. Inactive windows are never visible until they become active; when a window is inactive, it is out of the picture altogether. For more on the screen manager, see the section "The Screen Manager".

Each superior window keeps track of all of its active inferiors, and each inferior window keeps track of its superior, in internal state variables. Superior windows do *not* keep track of their inactive inferiors; this is a purposeful design decision, in order to allow unused windows to be reclaimed by the garbage collector. So, when a window is deactivated, the window system doesn't touch it until it is activated again.

### Pixels and Bit-Save Arrays

A screen displays an array of *pixels*. Each pixel is a little dot of some brightness and color; a screen displays a big array of these dots to form a picture. On regular black-and-white screens, each pixel can have only two values: lit up, and not lit up. The way the display of pixels is produced is that inside the Lisp Machine, there is a special memory associated with each physical screen that has some number of bits assigned to each pixel of the screen; those bits say, for each pixel, what brightness and color it should display. For regular black-and-white screens, since a pixel can have only two values, only a single bit is stored for each pixel. If the bit is a one, the pixel is not lit up; if it is a zero, the pixel is lit up. (Actually, this sense can be inverted if you want.) Everything you see on the screen, including borders, graphics, characters, and blinkers, is made up out of pixels.

When a window is fully visible, its *contents* are displayed on a screen so that they can be seen. What happens to the contents when the window ceases to be fully visible? There are two possibilities. A window may have a *bit-save array*. A bit-save array is a Lisp array in which the contents of the window can be saved away when the window loses its visibility; if a window has a bit-save array, then the window

system will copy its contents out of the screen and into the bit-save array when the window ceases to be fully visible. If the window does not have a bit-save array, then there is no place to put the bits, and they are lost. When the window becomes visible again, if there is a bit-save array, the window system will copy the contents out of the bit-save array and back onto the screen. If there is no bit-save array, the window will try to redraw its contents; that is, to regenerate the contents from some state information in the window. Some windows can do this; for example, editor windows can regenerate their contents by looking at the editor buffer they are displaying. Genera windows cannot regenerate their contents, since they do not remember what has been typed on them. In lieu of regenerating their contents, such windows just leave their contents blank, except for the decorations in the margins of the window, which they are able to regenerate.

The advantage of having a bit-save array is that losing and regaining visibility does not require the contents to be regenerated; this is desirable since regeneration may be computationally expensive, or even impossible. The disadvantage is that the bit-save array uses up storage in the Lisp world, and since it can be pretty big, it may need to be paged in from the disk in order to be referenced (depending on how hard the virtual memory system is being strained). If the paging overhead for the bit-save array is very high, it might have been faster not to have one in the first place (although the system goes through some special trouble to try to keep the bit-array out of main memory when it is not being used).

The other important use of bit-save arrays is for windows that have inferiors. If the superior window is not visible, the inferiors can use the bit-save array of the superior as if it were a screen, and they can draw on it and become exposed on it. See the section "Screen Arrays and Exposure".

An additional benefit of having a bit-save array is that the screen manager can do useful things for partially visible windows when those windows have bit-save arrays; at certain times it can copy some of the pixels from the bit-save array onto the part of the screen in which the window is partially visible, so that when a window is only partially visible, you can see whatever part is visible. See the section "The Screen Manager".

### Screen Arrays and Exposure

This section discusses the concepts of screen arrays and of exposed windows. These have to do with how the system decides where to put a window's contents (its pixels), how the notion of visibility on the screen is extended into a hierarchy of windows, and how this interacts with the desire of a program or of the user to have some windows visible and other windows not visible at a particular time. These are complex concepts, which you don't have to understand completely to make use of the window system. You probably *do* need to understand these ideas thoroughly only if you plan to make advanced use of the window system, such as creating your own frame or customizing very basic aspects of the system's behavior.

The following discussion attempts to explain what it means for a window to be *exposed*. It will be necessary for us to refer to the concept of a window being ex-

posed before we explain exactly what that means. For the time being, the approximate meaning of "exposed" is that a window is exposed if it has somewhere for its typeout to go. A window that is fully visible on a screen is exposed, because its typeout can go on the screen. A window might be exposed even if it is not fully visible, because its typeout might be able to go to a bit-save array somewhere.

Each window has in it a set of all those inferiors that are "ready to be exposed". This set is a subset of the set of active inferiors, discussed above. When you send a window an **:expose** message, it becomes "ready to be exposed" and is added to the set; when you send a window a **:deexpose** message, it ceases being ready to be exposed and is removed from the set. These are the only ways anything ever gets into or out of the set. The meaning of "ready" to be exposed will be cleared up soon; for the time being, we will just say that either all the windows on that list are, in fact, exposed, or else none of them are exposed but they are all still "ready" to become exposed.

Each window has an internal state variable called its *screen-array*. The value of the screen-array variable is where output to the window should go; if a program draws a character "on a window" or draws a triangle "on a window", that means it is changing the values of pixels in the window's screen-array. The value of the screen-array variable is used in figuring out whether a window is exposed.

The screen-array of a screen (remember, a screen is a window itself) is the special memory that gets displayed on the physical screen. For any other window, if the window is exposed, then its screen-array is an indirect array that points into a section of the superior's screen-array; namely, it points into the area of the superior's screen-array where the inferior gets displayed on the superior. For example, consider a window whose superior is a screen, which is exposed, and whose upper-left-hand corner is at location (100,100) in the screen. Then the window's screen-array would be an indirect array whose (0,0) element is the same as the (100,100) element of the screen. If you were to set a pixel in the window's screen-array, the corresponding pixel in the screen (found by adding 100 to each coordinate) would be set to that value.

What happens to the screen-array variable if the window is not exposed? That depends on whether the window has a bit-save array or not. If there is a bit-save array, then the screen-array becomes the bit-save array. If there is no bit-save array, the screen-array becomes **nil**.

The most important thing to understand about the value of screen-array is that it is defined recursively, in terms of the superior's screen-array. Consider a window which is exposed, and all of whose ancestors are exposed: The superior is exposed, the superior's superior is exposed, and so on all the way back to the screen. Then each window has a screen-array that points into the middle of its superior's screen-array, all the way up the hierarchy, through the window whose screen-array points into the middle of the screen. When typeout is done on the window, it will appear on the screen, offset by the combined offsets of all the superiors, so that it will appear in the correct absolute position on the screen.

Now, suppose one of those ancestors becomes deexposed. There are two alternative things that might happen. First, consider the case in which that ancestor (the one

that got deexposed) has a bit-save array. That ancestor's screen-array will no longer point to its own superior; its screen-array will be its bit-save array. That means that our window's screen-array will be pointing, perhaps through several levels of indirection, into that ancestor's bit-save array. The ancestor window is not exposed, but our window *is* still exposed. If typeout is done on our window, it will appear on the bit-save array of the ancestor. This won't actually be visible to the user, since it is only a bit-save array and not an actual screen, but the typeout can proceed and the bits can be drawn into the bit-save array. Later, if and when the ancestor is exposed again, the window system will copy the bit-save array onto the screen, and the drawing that had been done will become visible.

There is another case: Suppose the ancestor is deexposed, and it does not have a bit-save array. Then the ancestor's screen-array becomes **nil**. Well, now we have a problem. The ancestor's inferior is exposed, and so its screen-array is supposed to point into the screen-array of its superior. However, there is no way to point into the middle of a **nil**. There just isn't anywhere for the screen-array to point to; the window doesn't have anywhere to type out. Since it has nowhere to type out, it gets deexposed too. In general: When a window is deexposed, and it has no bit-save array, all of its inferiors that are ready to be exposed (all of which are, in fact, exposed) become deexposed. They continue to be "ready to be exposed", though.

In fact, this is the distinction between "ready to be exposed" and actually being exposed. The rule is: A window is exposed when and only when it is "ready to be exposed" *and* its superior has a screen-array. That is what "exposed" means.

When a window is sent an **:expose** message, it always becomes "ready to be exposed". If the superior has a screen-array, then it immediately becomes exposed. If the superior does not have a screen array, then the window just stays "ready", and when the window's superior finally gets its screen array, the window itself is exposed. If a window is "ready to be exposed" but is not exposed yet, then it is waiting for its superior to acquire a screen-array; when the superior gets one, the window becomes exposed. The usual way that the superior gets a screen array is for it to get exposed itself; when this happens, the inferiors that are "ready to be exposed" will all get exposed.

Also, if the superior has no screen-array then obviously it has no bit-save array; it can be given one by the **:set-save-bits** message, which can change a window that doesn't have a bit-save array into a window that does have a bit-save array. You can dynamically change which windows have and don't have bit-save arrays, and windows that are affected will be exposed and deexposed accordingly. This is much less common, though; usually whether a window has a bit-save array or not is specified when the window is created, and it doesn't change.

So, the important point is that when a window is sent an **:expose** message, it may not become exposed then and there. If the superior has a screen-array, then the window will be exposed immediately. But if the superior does not have a screen array, then making the window exposed is delayed until the superior acquires a screen array. When the superior gets its screen array, then the window itself becomes exposed. So what the **:expose** always does is to add the window to the set of windows that are "ready to be exposed"; a window is exposed precisely when it is "ready to be exposed" and the window's superior has a screen-array. The **:deexpose**

message always removes a window from the set of windows "ready to be exposed", and therefore is always stops the window from being exposed.

Note well that "exposed" does not mean "visible". A window can be exposed by virtue of being able to type out on a bit-save array, and not be visible at all. A window is fully visible if and only if all its ancestors are exposed, and the top level ancestor is a screen.

(A detail: If a window is top-level (if it has no superior) then it is as if "its superior has a screen array"; sending a top-level window an **:expose** message always exposes it immediately. You usually don't deexpose top-level windows anyway.)

(Another detail: It is possible for a screen to be deexposed. In particular, if a Symbolics machine does not have a color display physically attached to it, there is still a "color screen" Lisp object in the Lisp world, but it is deexposed (and so are all its inferiors). This is so saved Lisp environments can be moved easily between machines with different hardware configurations. The screen object is left deexposed so that programs will not try to output to it.)

In order to maintain the model that windows are like pieces of paper on a desk, it is important that no two windows that both occupy some piece of screen space be exposed at the same time. To make sure that this is true, whenever a window becomes exposed, the system deexposes any of its exposed siblings that it overlaps. (Note: This is not true for temporary windows).

The window system uses conformal indirect arrays for its screen arrays. This means that the bit-array in which a window saves its bits when it is not visible does not have to be the full width of the screen; it is just the width of the window, rounded up to the next multiple of 32 bits. Screen arrays do not use multilevel indirection; the screen array of a nonscreen sheet always indirects either to a bit-save array or to the screen array of its screen. The screen array of a screen is always a displaced array to the hardware screen buffer.

## Window Exposure and Output

The main reason for worrying about whether a window is exposed or not is in order to figure out whether it should be allowed to type out. If a window is not exposed, either its superior has no screen-array (so there is no place for its output to go), or it is not ready to be exposed at all (so it is supposed to be hidden). Normally, when a process tries to do output to a window that is not exposed, by sending stream messages (such as **:tyo** and **:string-out**), the process waits in a state called **Output Hold**; the process continues to wait until the window becomes exposed again, at which time it proceeds with its typeout. The term "typeout" refers not only to character output, but to any form of modification of the window's contents, including drawing of graphics.

This is the normal case that you run into most of the time. However, there are some exceptions to this rule.

A process trying to output to a window does not actually decide to wait in the **Output Hold** state based on whether or not the window is exposed. There is actu-

ally a flag in each window, called the *output hold flag*, that is really being checked to see whether output can go ahead. The output hold flag is cleared when the window is exposed and set when the window is deexposed, and output is held when this flag is set. The complexity comes from other things besides exposing that clear this flag.

When a process attempts to type out on a window which is deexposed and has its output hold flag set, what happens depends on the window's *deexposed typeout action*. The deexposed typeout action can be any of certain keyword symbols, or it can be a list; it indicates an action that should be taken when there is an attempt to type out to a deexposed window. After the action is taken, if the output hold flag is still set, the process will wait for it to clear. The interesting thing is that the action may affect the value of the output hold flag.

By default, the deexposed typeout action is **:normal**, which means that no special action should be taken; hence the process will wait for the window to become exposed.

If the deexposed typeout action is **:expose**, however, then the action will be to send the window an **:expose** message. This may expose the window (if the superior has a screen-array), and if it does expose the window then the output hold flag will be cleared and typeout will be able to proceed immediately. If the superior is the screen, the **:expose** option provides a very different user interface from the **:normal** option.

If the deexposed typeout action is **:permit**, that means that typeout should be permitted even though the window is not exposed, as long as the window has a screen array; that is, it may type out on its own bit-save array even though it is not exposed. The next time the window is exposed the updated contents will be retrieved from the bit-save array. The action for **:permit** is to turn off the output hold flag if the window has a screen array. This mode has the disadvantage that output can appear on the window without anything being visible to the user, who might never see what is going on, and might miss something interesting.

The deexposed typeout action may also be **:notify**, which means that the user should be notified when there is an attempt to do output on the window. The action taken is to send the **:notice** message to the window with the argument **:output**. The default response to this is to notify the user that the window wants to type out and to make the window "interesting" so that `FUNCTION 0 S` can select it. Windows in the Terminal program have **:notify** deexposed typeout action by default.

Another permissible value is **:error**, which means that an error should be signalled.

If the deexposed typeout action is not any of these keywords, then it should be a list; the action will be to send the message specified by the first element of the list to the window, passing the rest of the elements of the list as arguments.

There is another exception to the rule that you can only type out on exposed windows: The special form **tv:sheet-force-access** allows you to do typeout on a window that has a screen array even if its output hold flag is set. Note that the screen ar-

ray must be this window's bit-save array (since the window is not exposed). What **tv:sheet-force-access** does is to temporarily turn off the output hold flag while executing its body. This is useful for drawing things on a window while the window is not visible on the screen. It is better to do it this way than to use a deexposed typeout action of **:permit**, in most cases, since the effect of **tv:sheet-force-access** is local to the program, while the deexposed typeout action affects anything that types out on the window. If the window does not have a screen-array, **tv:sheet-force-access** doesn't do anything at all; it just returns *without* evaluating its body.

Another way that typeout can be held up is if the window is *locked*. Locking is independent of the output hold flag and is not affected by the deexposed typeout action or by **tv:sheet-force-access**. There are two ways that a window can be locked. The normal form of locking is a mutual exclusion that guarantees that only one process at a time operates on the window's contents and attributes. If one process is working on the window and another tries to do so, the second process will wait until the first one is finished. In the absence of program bugs, this wait is for a very short time and should not be noticeable.

The other form of locking is called *temp-locking*. If a window is temp-locked, then any attempt to type out on it will wait, regardless of everything else. Temp-locking has to do with temporary windows: See the section "Temporary Windows". The functions used to control window exposure and output are:

**tv:sheet-force-access** (*sheet don't-prepare-sheet*) *body...*
Allows typeout on *sheet* if it has a screen array (that is, if it is exposed or has a bit-save array).

**tv:prepare-sheet** (*sheet*) *body...*
Prepares *sheet* for input or output.

### Temporary Windows

Normally, when a window is exposed in an area of the screen where there are already some other exposed windows, the windows that used to be there are deexposed automatically by the window system. This is because the window system normally doesn't leave two windows both exposed if they overlap. (In the absence of temporary windows, which we are about to introduce, the system never allows two overlapping windows to both be exposed.)

But sometimes there are windows that only get put up on the screen for a very short time. The most obvious examples of such windows are the momentary menus that only appear for long enough for you to select an item. It would be unfortunate if every time a momentary menu appeared, the windows under it had to be deexposed. The ones without bit-save arrays would have their screen image destroyed, forcing them to regenerate it or to reappear empty. The ones with bit-save arrays would not be damaged in this way, but they would have to be deexposed, and deexposure is a relatively expensive operation.

This problem is solved for momentary menus by making them out of *temporary windows*. In general, when you create a window, you can specify that you want it to be a temporary window. Temporary windows work differently from other win-

dows in the following way: When a temporary window is exposed, it saves away the pixels that it covers up. It restores these pixels when it is deexposed. These pixels may come from several different windows. This way it doesn't mess up the area of the screen that it uses, even if it covers up some windows that don't have bit-save arrays.

Also, a temporary window, unlike a normal window, does not deexpose the windows that it covers up. This way the covered windows need not try to save their bits away in their bit-save arrays (if they have them) or ever have to try to regenerate their contents (if they don't). They never notice that the temporary window was (temporarily) there.

There would be some problems if temporary windows were this simple. Suppose there is a normal window, and a temporary window has appeared over it; some of the contents of the normal window are being saved in an array inside the temporary window. Now, if the normal window is moved somewhere else, and possibly becomes deexposed or is overlapped by other windows, and then the temporary window is deexposed, the temporary window will dump back its saved bits where the normal window used to be, even though the normal window isn't there any more, and so some innocent bystander will be clobbered. Furthermore, suppose typeout were done on the normal window; we have not deexposed it, so nothing would prevent the typeout from overwriting the temporary window, nor prevent the typeout from being overwritten in return when the temporary window is deexposed. Because of problems like these, when a temporary window gets exposed on top of some other windows, all the windows that it covers up (fully or partially) are *temp-locked*. When a window is temp-locked, any attempt to type out on it will wait until it is no longer temp-locked. Furthermore, any attempt to deexpose, deactivate, move, or reposition a temp-locked window will wait until the window is no longer temp-locked.

Because of temp-locking, you should never write a program that will put a temporary window up on the screen for a "long" time. There should be some action by the user, such as moving the mouse, which will make the temporary window deexpose itself. It is best if any attempt by the user to get the system to do something makes the temporary window go away. While the temporary window is in place, it blocks many important window system operations over its area of the screen. The windows it covers cannot be manipulated, and programs that try to manipulate them will end up waiting until the temporary window goes away. Temporary windows should only be used when you want the user to see something for a little while and then have the window disappear. The temp-locking is undone when the temporary window is deexposed.

It works fine to have two or more temporary windows exposed at a time. If you expose temporary window **a** and then expose temporary window **b**, and they don't overlap each other, they can be deexposed in either order, and any windows that both of them cover up will be temp-locked until both of them are deexposed. If **b** covers up **a**, then **a** will be temp-locked just like any other window, and so it will not be possible to deexpose **a** until **b** has been deexposed.

### The Screen Manager

The *screen manager* is a subsystem of the window system that does various background jobs involved with keeping things straight in the window system. It has several responsibilities. One job of the screen manager is to find any window that is active and deexposed, but not covered up by any windows. There is no reason for such a window not to be exposed, so the screen manager exposes it. This is called *autoexposure*.

Another job of the screen manager is to manage those parts of the screen that are not currently part of any exposed window. When you first start using Genera, the entire screen is covered by a big Lisp Listener window, and the initially created windows for Zmacs, Zmail, and so on, are all as large as the entire screen, so this issue does not arise. Similarly, if you use [Split Screen] to divide the screen up into windows, the windows will use up all of the area of the screen. However, if you use the [Create] or [Edit Screen] commands, you can make windows of arbitrary shapes and sizes, and you can leave parts of the screen where there is no exposed window.

When the screen manager sees that there is such an area of the screen, it considers all of the active windows that aren't exposed. If it finds such a window, and that window has a bit-save array, then the screen manager displays the contents of the bit-save array for the corresponding portion of the screen. This gives the visual impression of overlapping pieces of paper on a desktop; the deexposed window is partially covered up by the exposed windows, but you can still see those parts that aren't covered.

If there is more than one active deexposed window that might be displayed in a given area of the screen, then the screen manager uses its priority ordering to decide which one to display.

Usually the screen manager only displays partially visible windows that have bit-save arrays. But if you want to make a window that doesn't have a bit-save array and you want the screen manager to try to display it when it is only partially exposed, use the mixin **tv:show-partially-visible-mixin**.

The screen manager not only manages screens; it can manage any window that has inferiors. Windows with panes are split up into windows just the same way screens are split up into windows, and so the screen manager can do the same thing to panes of paned windows that it does with windows directly on screens. The action of the screen manager on the inferiors of a window is controlled by that window's response to the **:screen-manage** message; the default is to do screen management in the same way as it is done on a screen. See the flavor **tv:no-screen-managing-mixin**.

Suppose there is a section of the screen in which there are no exposed windows, and more than one active, deexposed window could be exposed to fill this area, but the two could not both be exposed (because they overlap). Which one gets to be exposed? Here's another issue: When the screen manager wants to display pieces of partially visible windows, there might be more than one deexposed window that might be displayed in a given area of the screen. Somehow the screen manager must decide which window to display.

The way it decides is on the basis of a priority ordering. All of the active inferiors of a window are maintained in a specific order, from highest to lowest priority. When there is a section of the screen on which more than one active inferior might be displayed, the inferior that is earliest in the ordering, and so has the highest priority, is the one that gets displayed. This ordering is like the relative heights of pieces of paper on a desk; the highest piece of paper at any point on the desk is the one that you see, and all the rest are covered up.

The screen manager has a somewhat complicated algorithm for keeping track of this ordering. Part of the algorithm involves a value kept for each window called its *priority*, which may be a fixnum or **nil**. The general idea is that windows with higher numerical priority values have higher priority to appear on the screen. If a window has priority **nil**, then its priority is less than that of any window with numerical priority; that is, **nil** acts like the lowest possible number. The default value for priority is **nil**.

The ordering itself is not based on just the priorities. Instead, the way it works is that the ordering is remembered, and at various times, the windows are resorted according to the following set of rules:

1. Exposed windows go in front of nonexposed windows.

2. If two windows are both exposed or both have the same value of priority, their order is not changed by the sorting.

3. If two nonexposed windows have different values of priority, then the one with the higher value goes in front of the one with the lower value.

So not only the priority values make a difference; the relative positions of windows before the resorting matters too.

The resorting happens whenever some event occurs that might change the ordering. For example, when a window is exposed or deexposed, or when a window's priority changes, the ordering it is on must be resorted. Note that the sort is *stable*; that is, if we don't have any preference for one window over another then they keep their previous ordering. Since most of the time numerical priorities are not used anyway (the priorities of most windows are **nil**), this is generally what determines the ordering. When a window is exposed, it gets pulled up to the front of the ordering, and then as other windows later get exposed on top of it, it sinks back down. More recently exposed windows will be closer to the front.

There is also an operation called *burying* a window, which first deexposes the window, then moves it to the end of the ordering, and finally (since something interesting has happened) causes the ordering to be resorted. So burying a window essentially makes it be the farthest from the front of the ordering of all windows with the same priority as it. A program usually buries its window when it thinks that the user is not interested in that window and would prefer to see some other windows. The [Bury] command in [Edit Screen] is a way for the user to bury a window.

Negative priorities have a special meaning. If the value of a window's priority is **4294967295**, then the window will not ever be visible at all even if it is only partially covered; however, it will still get autoexposed. If the value of priority is **4294967294** or less, then the window will not even be autoexposed, and so it will simply not become exposed unless sent an explicit **:expose** message.

(Another minor point: Windows whose area of the screen does not lie within the boundaries of their superior cannot be exposed at all, and so the screen manager does not try to autoexpose such windows. However, they can be partially visible.)

You may have noticed a problem that screen management can cause. Suppose you send a **:deexpose** message to an exposed window. The window is no longer exposed, but since it is closer to the front of the ordering, and especially if numerical priorities are not being used much, then it may end up being the foremost window in the ordering that occupies its area of the superior, and so autoexposure is likely to expose it again immediately! If you want to do a series of deexposing and exposing operations, they can get messed up this way by the screen manager. In order to prevent this from happening, you can use the **tv:delaying-screen-management** special form to delay the actions of the screen manager until all of your operations have been done. In simple applications, you should not need to send your own **:deexpose** messages anyway (most deexposure is done automatically when new windows are exposed), and you should not need **tv:delaying-screen-management**; explicit deexposure and delaying of screen management is mostly used in advanced applications, and if you use these for something simple then you are probably doing something wrong.

While screen management is delayed, notes to the screen manager telling what areas of the screen have been played with are put on a queue. When the **tv:delaying-screen-management** form is returned from, all of the entries on the queue are examined, and the screen manager figures out all the things that need to be done and does them all at once. So, by delaying screen management, you prevent the screen manager from seeing various intermediate states and doing unnecessary work, which would consume computation time and make the windows on the screen visibly undergo unnecessary contortions.

When a **tv:delaying-screen-management** form is exited, normally or abnormally (that is, **throw**n through), the screen manager tries to run and empty the queue, using an **unwind-protect**. However, under some circumstances it cannot do screen management at this time. In these cases, it leaves the requests on the queue. There is a background process that runs all the time, called **Screen Manager Background**, that wakes up to do the screen management that these queue entries specify, when screen management stops being delayed. So the screen management does eventually happen, when the special form is exited and the background process wakes up. When **tv:delaying-screen-management** forms are nested, only the outermost one will do any screen management when it is exited.

This background process has another useful function, which is optional. Recall that if a window has its deexposed typeout action set to **:permit**, processes can type out on the window, but the typeout goes to the bit-save array rather than to the screen. The screen manager background process can be told to find any such windows on which some typeout has happened, and copy their partially visible parts to

the screen so that they can be seen. This way, you get to see the typeout that happens on the part of the window that isn't being covered by any other windows.

The screen manager also has another job. At the same time that it does autoexposing, it can also select a window if there isn't any selected window at the time.

The screen manager has a facility for *graying* areas of the screen that contain no windows or windows that are not fully exposed. See the section "Window Graying". The screen management facilities are:

**tv:delaying-screen-management**
**tv:screen-manage-update-permitted-windows**

### Window Graying

Screens and frames can *gray* areas that contain no windows or that contain windows that are not fully exposed. To gray an area of the screen is to cover it with a semitransparent texture pattern. There are two kinds of graying:

- *Background gray* is used to fill in areas of the screen that don't contain any windows. Normally this is just the borders around the screen, but if you reshape all the full-screen windows to be smaller, so that there is some area of the screen that doesn't have a window on it, the background gray appears there, also. The background gray in the two areas (the part of the screen where you can put windows and the part of the screen where you cannot put windows) joins smoothly.

- *Deexposed gray* is used to fill in the visible portion of a window that is not fully exposed. It tells you that you aren't seeing all of this window, because another window is covering part of it. Deexposed graying does not occur when a window is covered by a temporary window (like a momentary menu) because such a window isn't considered to be really deexposed and is often still a focus of the user's attention.

These concepts generalize to any window, dynamic or static, that has inferiors, not just the screen. You can make a flavor of frame that fills in any empty spots with gray or grays over any partially exposed panes.

Both kinds of graying are implemented by the screen manager, but are affected by messages to the screen and to the deexposed windows.

To disable both background and deexposed gray on the main screen:

```
(tv:set-screen-background-gray nil)
(tv:set-screen-deexposed-gray nil)
```

To get a light gray on both unused areas and deexposed windows:

```
(tv:set-screen-background-gray tv:6%-gray)
(tv:set-screen-deexposed-gray tv:6%-gray)
```

To get a light gray over deexposed windows and a darker gray in the background:

```
(tv:set-screen-background-gray tv:33%-gray)
(tv:set-screen-deexposed-gray tv:6%-gray)
```

### Window Graying Specifications

A *graying specification* determines what pattern to use in graying areas of the screen that contain no windows or that contain windows that are not fully exposed. These specifications are used as arguments to functions and messages that deal with graying. See the section "Functions, Flavors, and Messages for Window Graying".

Following are the possible values of a specification and their meanings:

**nil** Disable graying. Background gray is white (in black-on-white mode); deexposed gray is completely transparent.

Two-dimensional bit array
A stipple pattern to be replicated by **bitblt**.

**:white** Opaque white.

**:black** Opaque black.

Instance An object that must handle the **:draw-blank-rectangle** message to draw a gray rectangle.

Function A function to be called with standard arguments to draw a gray rectangle.

List The first element is a function to be called, and the remaining elements are arguments to the function to be supplied after the standard arguments.

Following are the arguments to the **:draw-blank-rectangle** message and to a function to be called:

*x-size* Horizontal size of the rectangle in pixels.

*y-size* Vertical size of the rectangle in pixels.

*x-pos* X-position of the top left corner of the rectangle on *sheet*.

*y-pos* Y-position of the top left corner of the rectangle on *sheet*.

*x-phase* Starting x-coordinate of the source array.

*y-phase* Starting y-coordinate of the source array.

*sheet* Sheet or array on which to draw the rectangle, or **nil**.

*raster* The raster to draw on, or **nil**. Both sheet and raster may be specified, in which case methods of the sheet should be used to draw on the raster.

*ones-alu* The alu to use to draw the "ones".

*zeros-alu* The alu to use to draw the "zeros"

The variable **tv:*gray-arrays*** contains a list of variables that are bound to available predefined graying specifications.

### Functions, Flavors, and Messages for Window Graying

The window graying facilities are:

**tv:set-screen-background-gray**
**tv:set-screen-deexposed-gray**
**:screen-manage-deexposed-gray-array**
**tv:gray-unused-areas-mixin**
**(flavor:method :gray-array-for-unused-areas tv:gray-unused-areas-mixin)**
**(flavor:method :gray-array-for-unused-areas tv:gray-unused-areas-mixin)**
**(flavor:method :set-gray-array-for-unused-areas tv:gray-unused-areas-mixin)**
**tv:gray-deexposed-inferiors-mixin**
**(flavor:method :gray-array-for-inferiors tv:gray-deexposed-inferiors-mixin)**
**(flavor:method :gray-array-for-inferiors tv:gray-deexposed-inferiors-mixin)**
**(flavor:method :set-gray-array-for-inferiors tv:gray-deexposed-inferiors-mixin)**

### Windows and Processes

The flavor **tv:process-mixin** creates a new process associated with each window of the dependent flavor, that is, the flavor with which this one is mixed with. The dynamic window flavor **dw:program-frame** is one such dependent flavor. The init option **:process** for this flavor allows you to specify options for the process. These are the same options that **make-process** has.

### Activities and Window Selection

The concepts and facilities discussed in this section apply to both Dynamic Windows and static windows.

### The Selected Window and the Selected Activity

When you type characters on the keyboard, they must be directed to some window. The window that receives keyboard input is the *selected window*. No more than one window can be selected at a time. Sometimes no window is selected, but usually this is a brief transitional state. **tv:selected-window** is a variable that is bound to the value of the currently selected window. Similarly, **tv:cold-load-stream-old-selected-window** is bound to the value of **tv:selected-window** at the time you entered the cold-load stream.

A window is selectable only if it has **tv:select-mixin** and **tv:stream-mixin** as components (**dw:dynamic-window** has both). **tv:select-mixin** allows the window to handle messages that select it. **tv:stream-mixin** provides the window an *I/O buffer*, which accumulates keyboard characters, and lets the window handle messages to

get input. **tv:stream-mixin** also provides the window with *input editing*. When input editing is enabled and a reading function tries to get input from the window, the user can edit typein before the reading function sees it. See the section "Input from Windows".

An *activity* is a group of windows that the user regards as a single unit. Typically an activity consists of a top-level window — one that is a direct inferior of a screen — and all its direct and indirect inferior windows. An example of an activity is a top-level Lisp Listener. Sometimes an activity consists of a non-top-level window and all its direct and indirect inferior windows. One example is a Lisp Listener inside a Split Screen frame.

The concept of activity is only partially implemented in the window system. No separate object represents an activity. Instead, an activity is designated by a representative window from that activity. In the usual case, where the windows in an activity form a tree, the root of the tree serves as the representative.

The system contains several generic tools for selecting among activities: These include the SELECT key, FUNCTION S, and the [Select] menu in the System Menu. The *selected activity* is the activity that contains the selected window. When you change the selected activity, you also change the selected window.

You usually select an activity by selecting the representative window of the activity. But this window might or might not be selectable itself; sometimes only its inferiors, or only some of its inferiors, can become the selected window. When you select an activity, the representative window of the activity usually decides which window within the activity should become the selected window.

We say that this window — the one that is to become the selected window when the activity is selected — is selected *relative* to its activity. When you select a window relative to its activity, you do not change the selected activity. If an activity happens to be the selected activity, then selecting a window relative to that activity also makes that window the new selected window. If an activity is not the selected activity, then selecting a window relative to that activity changes neither the selected activity nor the selected window.

Whenever you select a window that is part of an activity, that window is selected relative to its activity, and that activity becomes the selected activity.

## Frames and Panes

A *frame* is a window that is designed to contain other windows inside it. A direct inferior window of a frame is called a *pane*. Many activities consist of a frame and its direct and indirect inferior windows. The frame is the representative window of this kind of activity.

A window that is a direct or indirect inferior of a frame can be the *selected-pane* of the frame. The *selected-pane* is the window that is selected relative to the frame. A frame usually cannot become the selected window. Instead, when you select a frame, its selected-pane becomes the selected window, unless the selected-pane is itself a frame. In that case the selected-pane of the selected-pane becomes the selected window.

You can change the selected-pane of a frame without selecting the activity that the frame represents. The next time that activity is selected, the new selected-pane becomes the selected window. If that activity happens to be the selected activity, then changing the selected-pane of the frame causes the new selected-pane to become the selected window.

If you select a window that is a pane of a frame, that window becomes the selected-pane of the frame, and the activity that the frame represents becomes the selected activity.

For more about panes and frames, including constraint frames, see the section "Frames".

### Messages About Window Selection

These are all the messages having to do with window selection:

**:alias-for-selected-windows**
**:name-for-selection**
**:selectable-windows**
**:select-relative**
**:inferior-select**
**:select-pane**
**:selected-pane**
**(flavor:method :selected-pane tv:basic-constraint-frame)**
**:mouse-select**
**:select**
**:deselect**

### Flavors Related to Window Selection

The flavors related to window selection are:

**tv:select-mixin**
**tv:select-relative-mixin**
**tv:dont-select-with-mouse-mixin**
**tv:basic-frame**
**tv:pane-mixin**
**tv:pane-no-mouse-select-mixin**

### Selecting a Window Temporarily

These functions can be used to select a window temporarily:

**tv:window-call-relative**
**tv:window-call**
**tv:window-mouse-call**

### Window Status

The following methods respectively determine and set the status of a window. They may be used with static or Dynamic Windows.

**(flavor:method :status tv:essential-activate)**
**(flavor:method :set-status tv:essential-activate)**

## Window Flavors and Messages

### Overview of Window Flavors and Messages

In this section we present the actual messages that can be sent to windows to examine and alter their state and to get them to do things. Just how a window reacts to a message depends on what flavor it is an instance of, and so we will also explain the various flavors that exist. This section also explains how to create new windows, and how to compose new flavors of windows by mixing together existing flavors.

Windows have a wide variety of functions, and can respond to any of a large set of messages. To help you find your way around among all the messages, this chapter groups together messages that deal with the same facet of the functionality of windows. Here is a summary of the various groups of messages that are documented.

First of all, a window can be used as if it were the screen of a display computer terminal. You can output characters at a cursor position, move the cursor around, selectively clear parts of the window, insert and delete lines and characters, and so on, by sending stream messages to the window. This way, windows can act as output streams, and any function that takes a stream for its argument (such as **print** or **zl:format**) can be passed a window.

Characters can be drawn in any of a large set of *fonts* (typefaces). Prior to Genera 7.0, fonts for character output to a window were manipulated directly through various font messages. Currently, only a couple of these messages are supported. The preferred interface to character fonts is the *character style* system. Each window has a default character style, which you can specify as an init option. See the init option **(flavor:method :default-character-style tv:sheet)**. To override the default style, you can use one of several character style macros, see the section "Controlling Character Style". For more information on character styles generally, see the section "Character Styles".

Windows do useful things when you try to run the cursor off the right or bottom edges; they also have a facility called *more processing* to stop characters from coming out faster than you can read them.

In addition to characters, you can also display graphics (pictures) on windows. There are functions to draw lines, circles, triangles, rectangles, arbitrary polygons, circle sectors, and cubic splines.

A window can also be used for reading in characters from the keyboard; you do this by sending it stream input messages (such as **:tyi** and **:listen**). This way, windows can act as input streams, and any function that takes a stream for its argument (such as **zl:read** or **zl:readline**) can be passed a window. Each window has an *I/O buffer* holding characters that have been typed at the window but not read yet, and there are messages that deal with these buffered characters. You can *force keyboard input* into a window's I/O buffer; frequently two processes communicate by one process's forcing keyboard input into an I/O buffer which another process is reading characters from.

Each window can have any number of *blinkers*. The kind of blinker that you see most often is a blinking rectangle the same size as the characters you are typing; this blinker shows you the cursor position of the window. In fact, a window can have any number of blinkers; they need not follow the cursor (some do and some don't) and they need not actually blink (some do and some don't). For example, the editor shows you what character the mouse is pointing at; this blinker looks like a hollow rectangle. The arrow that follows the mouse is a blinker, too. Blinkers are used to add visible ornaments to a window; a blinker is visible to the user, but while programs are examining and altering the contents of a window the blinkers all go away. This means that blinkers do not affect the contents of the window as seen from programs; whenever a program looks at a window, the blinkers are all turned off. The reason for this is so that you can draw characters and graphics on the window without having to worry whether the flashing blinker will overwrite them. If you have anything that should appear to the user but not be visible to the program, then it should be a blinker. The window system provides a few kinds of blinkers, and you can define your own kinds. Blinkers are instances of flavors, too, and have their own set of messages that they understand.

Any program can use the mouse as an input device. The window system provides many ways for you to get at the mouse. Some of them are very easy to use, but don't have all the power you might want; others are somewhat more difficult to use but give you a great deal of control. The window system also takes responsibility for figuring out which of many programs have control over the mouse at any time.

There are a large number of messages for manipulating the size and position of a window. You can specify these numerically, ask for the user to tell you (using the mouse), ask for a window to be near some point or some other window, and so on.

A window's area of the screen is divided into two parts. Around the edges of the window are the four *margins*; while the margins can have zero size, usually there is a margin on each edge of the window, holding a border and sometimes other things, such as a label. The rest of the window is called the *inside*; regular character drawing and graphics drawing all occur on the inside part of the window. You have a great deal of control over what goes in the margins of a window. Control can be exercised either by mixing in different flavors that put different things in the margins or by specifying parameters such as the width of the borders or the text to appear in the label.

You can create windows with several panes (inferior windows). These are called *frames*, and there are messages that deal specifically with frames, their configuration, and their inferiors.

Sometimes a background process wants to tell the user something, but it does not have any window on which to display the information, and it does not want to pop one up just for one little message. A facility is provided wherein the process can send such *notification* messages to the selected window, and it will find some way to get the message to the user. Different windows do different things when someone tries to use them for notification.

Screens are windows themselves; they also have extra functions that windows don't have, since they do not have superiors and since they correspond to actual pieces of display hardware. Screens can be either black-and-white or color. Color screens have more than one bit for each pixel, and most operations on windows do something reasonable on color screens. But the extra bits give you extra flexibility, and so there are some more powerful things you can do to manipulate colors. Color screens also have a *color map*, that specifies which values of the pixels display which colors.

There are also messages for changing the status of windows: whether they are active, exposed, or selected. There are several options to exactly how exposure and deexposure should affect the screen. You can also ask windows to refresh their contents, kill them, and so on. There are also ways to deal with the screen manager, including messages to examine and alter priorities, and other functions and variables and flavors for affecting what the screen manager does.

You can define your own fonts, and/or convert fonts from other formats to the Lisp Machine's format. Font characters have various attributes such as their height, baseline, left kern, and so on.

The status line at the bottom of the screen shows the user something about the state of the Lisp Machine. There are several functions for controlling just what it does and for getting things to be displayed in it.

The window system provides a facility called *I/O buffers*. An I/O buffer is a general purpose first-in first-out ring buffer, with various useful features. Programs can use I/O buffers for anything else, too; it need not even have anything to do with the window system.

There are some interrelationships between windows and processes. Exactly how processes and windows relate depends on the flavor of the window, and, as usual, there are several messages to manipulate the connections.

## Getting a Window to Use

### Flavors of Basic Windows

Many programs never need to create any new windows. Often, all you are interested in doing is sending messages to ***standard-output*** and ***standard-input*** and performing the extended stream operations offered by windows to read and type characters, position the cursor (and other things that you do on display terminals), and draw graphics. Other programs want to create their own windows for various reasons; a common way to organize an interactive system in Genera is to create a

process that runs the command loop of the system, and have it use its own window or suite of windows to communicate with the user. This kind of system is what the editor and Zmail use, and it is very convenient to deal with.

Whichever of these you use, it is important for you to know what flavor of window you are getting. Some flavors accept certain messages that are not handled by others. The details of different flavors' responses to the same message may vary in accordance with what those flavors are supposed to be for. The following is a discussion of window flavors.

The most primitive flavor of window is called **tv:minimum-window**; it is the basic flavor on which all other window flavors are built, and it contains the absolute minimum amount of functionality that a window must have to work. **tv:minimum-window** itself is built on a number of other flavors that provide the "essential" attributes of windows. For reference, **tv:minimum-window** is defined as follows (ignoring **defflavor** options):

```
(defflavor tv:minimum-window ()
  (tv:essential-expose tv:essential-activate
   tv:essential-set-edges tv:essential-mouse
   tv:essential-window))
```

**tv:essential-window**, in turn, is built on the base flavor for all windows, **tv:sheet**.

There is another flavor called **tv:window**, which is built on **tv:minimum-window** and has about six mixins that do a variety of useful things. When you cold boot a Lisp Machine, the window you are talking to is of flavor **tv:lisp-listener**, which is built on **tv:window** and has three more mixins. **tv:window** has what you need to do the normal things that are done with windows; **tv:minimum-window** is missing messages for character output and input, selection, borders, labels, and graphics, and so there isn't much you can do with it. Anything built on **tv:window**, including Lisp Listeners, will be able to accept all the basic messages.

Some programs may benefit from more carefully tailored mixings of flavors. For the benefit of programmers who want to do this, we specify below, with each message and init option, which flavor actually handles it. If you are just using **tv:window** then you don't really care exactly what mixin specific features are in; you just need to know which ones are in **tv:window**. With the discussion of each flavor or group of messages, we will say which relevant flavors are in **tv:window** and which are not. For reference, **tv:window** is defined (ignoring **defflavor** options) as follows:

```
(defflavor tv:window ()
  (tv:stream-mixin tv:borders-mixin tv:label-mixin
   tv:select-mixin tv:graphics-mixin tv:minimum-window))
```

So, if you use **tv:window** then you have all the above mixins, and can take advantage of their features.

For information on Dynamic Window flavors, see the section "Window Substrate Facilities".

## Creating a Window

If you want to create your own window, static or dynamic, you use the **tv:make-window** function. Never try to instantiate a window flavor yourself with **make-instance**; always use **tv:make-window** which takes care of a number of internal system issues.

These are the facilities for creating windows:

**tv:make-window**
**(flavor:method :init tv:sheet)**
**(flavor:method :blinker-p tv:sheet)**
**(flavor:method :default-character-style tv:sheet)**
**(flavor:method :save-bits tv:sheet)**
**(flavor:method :superior tv:sheet)**
**(flavor:method :activate-p tv:essential-window)**
**(flavor:method :expose-p tv:essential-window)**
**tv:defwindow-resource**

## Character Output to Windows

The information included in this section applies to both Dynamic Windows and static windows.

## How Windows Display Characters

A window can be used as if it were the screen of a display computer terminal, and it can act as an output stream. The flavor **tv:sheet** implements the messages of the Genera output stream protocol. It implements a large number of optional messages of that protocol, such as **:insert-line**. The **tv:sheet** flavor is a component of all windows. Every window has a current *cursor position*; its main use is to say where to put characters that are drawn. The way a window handles the messages asking it to type out is by drawing that character at the cursor position, and moving the cursor position forward past the just-drawn character.

In the messages below, the cursor position is always expressed in "inside" coordinates; that is, its coordinates are always relative to the top-left corner of the inside part of the window, and so the margins don't count in cursor positioning. The cursor position always stays in the inside portion of the window--never in the margins. The point *(0,0)* is at the top-left corner of the window; increasing $x$ coordinates are further to the right and increasing $y$ coordinates are further towards the bottom. (Note that $y$ increases in the down direction, not the up direction!)

To draw a character "at" the cursor position basically means that the top-left corner of the character will appear at the cursor position; so if the cursor position is at position *(0,0)* and you draw a character, it will appear at the top-left corner of the window. (Things can actually get more complicated when fonts with left-kerns are used.)

When a character is drawn, it is combined with the existing contents of the pixels of the window according to an *alu function*. For a description of the different alu functions, see the section "Graphic Output to Windows". When characters are drawn, the value of the window's *char-aluf* is the alu function used. Normally, the *char-aluf* says that the bits of the character should be bit-wise logically *or*ed with the existing contents of the window. This means that if you type a character, then set the cursor position back to where it was and type out a second character, the two characters will both appear, *or*ed together one on top of the other. This is called overstriking.

The *character style* of characters output to the window is gotten by merging the character style specified for the output against the window's *default character style*. The resulting style maps to a particular font. (For more information on character styles, see the section "Character Styles". For more on specifying output character styles, see the section "Controlling Character Style". To specify a window's default character style, see the init option **(flavor:method :default-character-style tv:sheet)**. Details of fonts are gone into later. See the section "TV Fonts". For now, it is only important to understand what the *character-width* and *line-height* of the window are; these two units are used by many of the messages documented in this section.

Character-width is the *char-width* attribute — the width of a space character — of the font currently being used for character output, that is, the *current font*. The line-height is the sum of the *vsp* of the window and the *char-height*s of the current font. The *vsp* is an attribute of the window that controls how much vertical spacing there is between successive lines of text. That is, each line is as tall as the font is, plus vertical spacing added between lines by controlling the *vsp* of the window.

In some fonts, all characters have the same width; these are called *fixed-width fonts*. The default character style for the system, `(:fix :roman :normal)`, maps to a fixed-width font (**fonts:cptfont**) for character output to windows. In other fonts, each character has its own width; these are called *variable-width fonts*. In a variable-width font, expressing horizontal positions in numbers of characters is not meaningful, since different characters have different widths. Some of the functions below do use numbers of characters to designate widths; there are warnings along with each such use explaining that the results may not be meaningful if the current font has variable width.

Typing out a character does more than just drawing the character on the screen. The cursor position is moved to the right place; nonprinting characters are dealt with reasonably; if there is an attempt to move off the right or bottom edges of the screen, the typeout wraps around appropriately; *more* breaks are caused at the right time if *more processing* is enabled. Here is the complete explanation of what typing out a character does. You may want to remind yourself how the Symbolics character set works. See the section "The Character Set". You don't have to worry much about the details here, but in case you ever need to know, here they are. If you aren't interested, skip ahead to the definitions of the messages.

First of all, as was explained earlier, before doing any typeout the process must wait until it has the ability to output. See the section "Window Exposure and Output". The output hold flag must be off and the window must not be temp-locked.

Before actually typing anything, various exceptional conditions are checked for. If an exceptional condition is discovered, a message is sent to the window; the message keyword is the name of the condition. Different flavors handle the various exceptions different ways; you can control how exceptions are handled by what flavors your window is made of. First, if the *y*-position of the cursor is less than one line-height above the inside bottom edge of the window, an **:end-of-page-exception** happens. The handler for this exception in the **tv:sheet** flavor moves the cursor position to the upper-left-hand corner of the window and erases the first line, doing the equivalent of a **:clear-rest-of-line** operation.

Next, if the window's *more flag* is set, a **:more-exception** happens. The *more flag* gets set when the cursor is moved to a new line (for example, when a **#\return** is typed) and the cursor position is thus made to be below the *more vpos* of the window. (If **tv:more-processing-global-enable** is **nil**, this exception is suppressed and the *more flag* is turned off.) The **:more-exception** handler in the **tv:sheet** flavor does a **:clear-rest-of-line** operation, types out **MORE**, waits for any character to be typed, restores the cursor position to where it originally was when the **:more-exception** was detected, does another **:clear-rest-of-line** to wipe out the **MORE**, and resets the *more vpos*. The character read in is ignored.

Note that the *more flag* is only set when the cursor moves to the next line, because a **#\return** is typed, after a **:line-out**, or by the **:end-of-line-exception** handler described below. It is not set when the cursor position of the window is explicitly set (for example, with **:set-cursorpos**); in fact, explicitly setting the cursor position clears the *more flag*. The idea is that when typeout is being streamed out sequentially to the window, **:more-exception**s happen at the right times to give the user a pause in which to read the text that is being typed, but when cursor positioning is being used the system cannot guess what order the user is reading things in and when (if ever) is the right time to stop. In this case it is up to the application program to provide any necessary pauses.

The algorithm for setting the *more vpos* is too complicated to go into here in all its detail, and you don't need to know exactly how it works, anyway. It is careful never to overwrite something before you have had a chance to read it, and it tries to do a **MORE** only if a lot of output is happening. But if output starts happening near the bottom of the window, there is no way to tell whether it will just be a little output or a lot of output. If there's just a little, you would not want to be bothered by a **MORE**. So it doesn't do one immediately. This may make it necessary to cause a **MORE** break somewhere other than at the bottom of the window. But as more output happens, the position of successive **MORE**s is migrated and eventually it ends up at the bottom.

Finally, if there is not enough room left in the line for the character to be typed out, an **:end-of-line-exception** happens. The handler for this exception in the **tv:sheet** flavor advances the cursor to the next line just as typing a **#\return** character does normally. This may, in turn, cause an **:end-of-page-exception** or a **:more-exception** to happen. Furthermore, if the *right margin character flag* is on, then before going to the next line, an exclamation point in font zero is typed at the cursor position. When this flag is on, **:end-of-line-exception**s are caused a little bit earlier, to make room for the exclamation point.

The way the cursor position goes to the next line when it reaches the right edge of the window is called *horizontal wraparound*. You can make windows that truncate lines instead of wrapping them around by using **tv:truncating-lines-mixin**.

After checking for all these exceptions, the character finally gets typed out. If it is a printing character, it is typed in the current font at the cursor position, and the cursor position is moved to the right by the width of the character. If it is one of the format effectors **#\return**, **#\tab**, and **#\back-space**, it is handled in a special way to be described in a moment. All other special characters have their names typed out in tiny letters surrounded by a lozenge, and the cursor position is moved right by the width of the lozenge. If an undefined character code is typed out, it is treated like a special character; its code number is displayed in a lozenge.

**#\tab** moves the cursor position to the right to the next tab stop, moving at least one character-width. Tab stops are equally spaced across the window. The distance between tab stops is *tab-nchars* times the *character-width* of the window. *tab-nchars* defaults to **8** but can be changed.

Normally **#\return** moves the cursor position to the inside left edge of the window and down by one line-height, and clears the line. It also deals with more processing and the end-of-page condition as described above. However, if the window's *cr-not-newline-flag* is on, the **#\return** character is not regarded as a format effector and is displayed as "**return**" in a lozenge, like other special characters.

If the character being typed out is a **#\back-space**, the result depends on the value of the window's *backspace-not-overprinting-flag*. If the flag is **0**, as is the default, the cursor position is moved left by one character-width (or to the inside left edge, whichever is closer). If the flag is **1**, **#\back-space**s are treated like all other special characters.

### Messages to Display Characters on Windows

These are the messages used to display characteres on windows:

**(flavor:method :tyo tv:sheet)**
**(flavor:method :string-out tv:sheet)**
**(flavor:method :line-out tv:sheet)**
**(flavor:method :fresh-line tv:sheet)**
**(flavor:method :insert-char tv:sheet)**
**(flavor:method :insert-string tv:sheet)**
**(flavor:method :insert-line tv:sheet)**
**(flavor:method :set-default-character-style tv:sheet)**

### Messages to Read or Set Cursor Position

These are the messages used to read or set the cursor position:

**(flavor:method :read-cursorpos tv:sheet)**

**(flavor:method :set-cursorpos tv:sheet)**
**(flavor:method :home-cursor tv:sheet)**
**(flavor:method :home-down tv:sheet)**

### Messages to Remove Characters from Windows

These are the messages used to remove characters from windows:

**(flavor:method :refresh tv:sheet)**
**(flavor:method :clear-char tv:sheet)**
**(flavor:method :clear-rest-of-line tv:sheet)**
**(flavor:method :clear-rest-of-window tv:sheet)**
**(flavor:method :clear-window tv:sheet)**
**(flavor:method :delete-char tv:sheet)**
**(flavor:method :delete-string tv:sheet)**
**(flavor:method :delete-line tv:sheet)**

### Messages About Character Width and Cursor Motion

These are the messages that have to do with character width and cursor motion:

**(flavor:method :character-width tv:sheet)**
**(flavor:method :compute-motion tv:sheet)**
**(flavor:method :string-length tv:sheet)**

### Window Attributes for Character Output

The following messages and initialization options initialize, get, and set various window attributes which are relevant to the typing out of characters.

**(flavor:method :more-p tv:sheet)**
**(flavor:method :more-p tv:sheet)**
**(flavor:method :set-more-p tv:sheet)**
**tv:autoexposing-more-mixin**
**(flavor:method :vsp tv:sheet)**
**(flavor:method :vsp tv:sheet)**
**(flavor:method :set-vsp tv:sheet)**
**(flavor:method :reverse-video-p tv:sheet)**
**(flavor:method :set-reverse-video-p tv:sheet)**
**(flavor:method :deexposed-typeout-action tv:sheet)**
**(flavor:method :deexposed-typeout-action tv:sheet)**
**(flavor:method :set-deexposed-typeout-action tv:sheet)**
**(flavor:method :deexposed-typein-action tv:sheet)**
**(flavor:method :deexposed-typein-action tv:sheet)**

**(flavor:method :set-deexposed-typein-action tv:sheet)**
**(flavor:method :right-margin-character-flag tv:sheet)**
**(flavor:method :backspace-not-overprinting-flag tv:sheet)**
**(flavor:method :cr-not-newline-flag tv:sheet)**
**(flavor:method :tab-nchars tv:sheet)**

## Line-Truncating Windows

These facilities control how lines are truncated in windows:

**tv:truncatable-lines-mixin**
**tv:line-truncating-mixin**
**tv:truncating-lines-mixin**
**tv:truncating-window**
**(flavor:method :truncate-line-out tv:sheet)**
**(flavor:method :set-truncate-line-out tv:sheet)**

## Graphic Output to Windows

The facilities in this section can be used with both Dynamic Windows and static windows. For information on graphics functions introduced in Genera 7.0: See the section "Creating Graphic Output".

## How Windows Display Graphic Output

A window can be used to draw graphics (pictures). There is a set of messages for drawing lines, circles, sectors, polygons, cubic splines, and so on, implemented by the flavor **tv:graphics-mixin**. The **tv:graphics-mixin** flavor is a component of the **tv:window** and **dw:dynamic-window** flavors. Therefore, the messages documented below work on windows of these flavors or built on these flavors. (For information on a corresponding set of graphics functions: See the section "Creating Graphic Output".)

There are also some messages in this section that are in **tv:sheet** or **tv:stream-mixin** rather than **tv:graphics-mixin**, because they are likely to be useful to any window that can draw characters, but such windows might not want the full functionality of **tv:graphics-mixin**. These messages are **:draw-rectangle**, and the **:bitblt** message and its relatives. (If you are building on **tv:window** anyway, this doesn't affect you, since **tv:window** includes both of these flavors.)

The cursor position is not used by graphics messages; the messages explicitly specify all relevant coordinates. All coordinates are in terms of the inside size of the window, just like coordinates for typing characters; the margins don't count. Remember that the point *(0,0)* is in the upper left; increasing *y* coordinates are *lower* on the screen, not higher. Coordinates are always integers.

As with typing out text, before any graphics are typed the process must wait until it has the ability to output. The output hold flag must be off and the window must not be temp-locked. The other exception conditions of typing out are not relevant to graphics.

All graphics functions *clip* to the inside portion of the window. This means that when you specify positions for graphic items, they need not be inside the window; they can be anywhere. Only the portion of the graphic that is inside the inside part of the window will actually be drawn. Any attempt to write outside the inside part of the window simply won't happen.

There are a few simple microcoded primitives for drawing graphics. They can be used for drawing pictures into Lisp arrays. However, when drawing on windows you should send the documented messages rather than directly calling the microcode primitives because these messages provide several essential services which are too complex for the microcode, such as protecting blinkers from being affected from drawing, and locking out other processes.

### Alu Functions

Most of the messages that produce graphic output on windows take an *alu* argument, which controls how the bits of the graphic object being drawn are combined with the bits already present in the window. In most cases this argument is optional and defaults to the window's **char-aluf**, the same alu function as is used to draw characters, which is normally inclusive-or. The following variables have the most useful *alu* functions as their values:

**tv:alu-ior**
**tv:alu-andca**
**tv:alu-xor**
**tv:alu-seta**
**tv:alu-and**

### Drawing Points on Windows

These methods have to do with drawing points on windows.

**(flavor:method :point tv:graphics-mixin)**
**(flavor:method :draw-point tv:graphics-mixin)**

Also, see the function **graphics:draw-point**.

### Copying Bit Rectangles to and from Windows

These methods are for copying bit rectangles to and from windows:

**(flavor:method :bitblt tv:sheet)**
**:draw-1-bit-raster**

**(flavor:method :bitblt-from-sheet tv:sheet)**
**(flavor:method :bitblt-within-sheet tv:sheet)**

The function **tv:make-sheet-bit-array** is useful for creating arrays that are bit-blt'ed into and out of windows.

### Drawing Characters and Strings on Windows

These methods draw characters or strings on windows:

**(flavor:method :draw-char tv:sheet)**
**(flavor:method :draw-string tv:graphics-mixin)**

Also see the functions **graphics:draw-string**, **graphics:draw-image**, **graphics:draw-string-image** and **graphics:draw-glyph**.

### Drawing Lines on Windows

These methods draw lines on windows:

**(flavor:method :draw-line tv:graphics-mixin)**
**(flavor:method :draw-lines tv:graphics-mixin)**
**(flavor:method :draw-dashed-line tv:graphics-mixin)**
**(flavor:method :draw-curve tv:graphics-mixin)**
**(flavor:method :draw-closed-curve tv:graphics-mixin)**
**(flavor:method :draw-wide-curve tv:graphics-mixin)**

Also, refer to the graphics drawing functions described in "Drawing Functions".

### Drawing Polygons and Circles on Windows

These methods are used to draw polygons and circles on windows:

**(flavor:method :draw-rectangle tv:sheet)**
**(flavor:method :draw-triangle tv:graphics-mixin)**
**(flavor:method :draw-circle tv:graphics-mixin)**
**(flavor:method :draw-circular-arc tv:graphics-mixin)**
**(flavor:method :draw-filled-in-circle tv:graphics-mixin)**
**(flavor:method :draw-filled-in-sector tv:graphics-mixin)**
**(flavor:method :draw-regular-polygon tv:graphics-mixin)**

Also, refer to the graphics drawing functions described in "Drawing Functions".

### Drawing Splines on Windows

There are two ways to draw splines: **(flavor:method :draw-cubic-spline tv:graphics-mixin)** is the older way, and **graphics:draw-cubic-spline** the preferred way.

### Primitives for Drawing onto Arrays

The following functions are primitives for drawing pictures onto arrays. You should only use them on arrays and not directly on windows.

**sys:%draw-rectangle**
**sys:%draw-line**
**sys:%draw-triangle**

### Notifications and Progress Indicators

This section applies to both static and Dynamic Windows.

### Overview of Notifications

Notifications are messages that a process sends to the user asynchronously to inform the user of some change in the state of the process. Some examples:

- By default the garbage collector notifies the user as storage is used up and when the dynamic garbage collector flips and flushes oldspace.

- If a window's deexposed typeout action is **:notify**, the user is notified when an attempt is made to type out on that window.

- Converse messages can be received as notifications.

A process uses **tv:notify** to notify the user. This function constructs a notification and saves it on a queue. A central delivery process takes notifications from the queue and delivers them to the user. This process first gives the process associated with the selected window a chance to accept the notification itself. If the process associated with the selected window does not accept the notification within a short time, the delivery process usually tries to display the notification itself, in either the selected window or a pop-up window.

The notification delivery process tries to give the user process a chance to accept the notification by storing the notification in a locative obtained by sending the **:notification-cell** message to the selected window. If the user process wants to accept notifications, it usually checks the contents of this cell as part of the **:input-wait** wait function. The user process sends the **:receive-notification** message to the window to accept the notification. When the user process wants to display a notification it usually calls **sys:display-notification**. By default, if the user process does not accept a notification, the notification delivery process displays the notification in a pop-up window. The user process can use the **tv:with-notification-mode** special form to control what happens to notifications it does not accept.

All notifications received since cold booting are displayed in a scroll window obtained by pressing SELECT N or by calling **zl:display-notifications**. You can display some or all notifications by using the Show Notifications command.

### Notifying the User

The function **tv:notify** issues to the user an asynchronous notification, which is delivered by a central notification delivery process.

### Receiving and Displaying Notifications

When a process notifies the user, the central notification delivery process gives the process associated with the selected window a chance to accept the notification before the delivery process tries to display the notification itself. The notification delivery process stores the notification in a locative obtained by sending the **:notification-cell** message to the selected window, unless a notification is already there. In that case the notification delivery process usually tries to display the notification itself.

A user process that wants to accept notifications should send the selected window a **:notification-cell** message to find the locative that might contain a notification. The process should wait (usually in an **:input-wait** wait function) until the locative contains something other than **nil**. A user process that does not want to accept notifications and does not want pop-up notification windows to occur can set the variable **tv:*allow-pop-up-notifications*** to **nil**. See the section "Pop-up notifications". The **:notification-cell** message to an interactive stream returns the locative in which the notification delivery process stores notifications. When a notification cell contains a notification, a process can accept the notification by sending the selected window a **:receive-notification** message. If the process wants to display the notification, it usually passes it on to the function **zl:display-notifications**.

Following is a simple example of a command loop that waits for input, a notification, or a new selected-pane. When a notification arrives, it displays it in a pane reserved for notifications. When input arrives, it just displays a representation of the input in the selected pane.

```
(defun my-top-level (frame)
  (let ((notification-pane (send frame :get-pane 'notification-pane)))
    (error-restart-loop ((error sys:abort) "My top level")
      (let ((selected-pane (send frame :selected-pane))
            (note))
        (when selected-pane
          (send selected-pane :input-wait nil
              #'(lambda (note-cell)
                  (declare (sys:downward-function))
                  (or (neq selected-pane (send frame :selected-pane))
                      (not (null (location-contents note-cell)))))
              (send selected-pane :notification-cell))
          (cond
              ((neq selected-pane (send frame :selected-pane)))
              ((setq note (send selected-pane :receive-notification))
               (sys:display-notification notification-pane note :stream))
               (t
```

```
(let ((char (send selected-pane :any-tyi-no-hang)))
  (cond
     ((null char))
     ((fixp char)
      (format selected-pane "~&Character: ~C" char))
     ((listp char)
      (format selected-pane "~&Blip: ~S" char))
     (t (format selected-pane
                "~&Unknown object: ~S" char))))))))))
```

After storing a notification in the selected window's notification cell, the notification delivery process gives the process associated with the selected window some time to accept the notification. The amount of time is determined by the variable **tv:*notification-deliver-timeout***.

If the process associated with the selected window does not accept a notification within the specified time, or if the window's notification cell already contains a notification, the window's *notification mode* determines what the delivery process does with the notification. You can use the **:notification-mode** message to get the notification mode and the **:set-notification-mode** message to set it.

If you want to execute some code with a stream's notification mode bound to some value, use the special form **tv:with-notification-mode**.

### Pop-up notifications

When a notification is displayed in a pop-up window, the user is alerted with a beep and given some time to notice the beep and stop typing. Until that time elapses, all typein is directed to the previously selected window, except that the user can press ABORT to deexpose the pop-up window immediately. The amount of time is determined by the variable **tv:unexpected-select-delay**.

After the select delay, typing any character or selecting another window deexposes the pop-up window. If a "window of interest" was supplied as the first argument to **tv:notify**, a message is displayed that informs the user that FUNCTION 0 S or a mouse click on the pop-up window selects the window of interest. If another notification arrives while the pop-up window is exposed, the notification is displayed on the window. If after a time the user has typed nothing, the pop-up window is deexposed automatically. The amount of time the pop-up window remains exposed is determined by the variable **tv:*notification-pop-down-delay***.

If pop-up notifications are not desired, they can be suppressed by setting the value of **tv:*allow-pop-up-notifications*** to **nil**, with, for example,

```
(setq tv:*allow-pop-up-notifications* nil)
```

## Progress Indicator Facilities

Facilities in this category of basic output facilities provide a way of communicating the progress of some operation to your users:

**tv:noting-progress**
**tv:note-progress**
**tv:dolist-noting-progress**
**tv:dotimes-noting-progress**

Progress is indicated by the advance of a progress bar in the lower, right corner of the screen or, alternatively, by a wide bar across the entire width of the screen. (Which is determined by the setting of the "Progress area" option in the Set Screen Options command.) Also displayed is a string naming the operation being noted.

The general-purpose facility is **tv:noting-progress**, within which the **tv:note-progress** function is used. **tv:note-progress** is the one that decides when and how much progress has occurred. This is shown in the following example:

```
(tv:noting-progress ("Working Away By Fifths")
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 1 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 2 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 3 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 4 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 5 5)
  (sleep 1))
```

**tv:dolist-noting-progress** and **tv:dotimes-noting-progress** implement the Common Lisp special forms **zl:dolist** and **zl:dotimes** in a noting-progress environment. They take care of most simple cases.

## Input from Windows

The material presented in this section applies to both static and Dynamic Windows.

## Windows as Input Streams

A window can be used as if it were the keyboard of a computer terminal, and it can act as an input stream. The flavor **tv:stream-mixin** implements the messages of the Genera input stream protocol. The **tv:stream-mixin** flavor is a component of the **tv:window** and **dw:dynamic-window** flavors.

**tv:stream-mixin** includes **si:interactive-stream**, and windows support all the operations that interactive streams in general do. See the section "Interactive Streams". Windows have specialized versions of some input operations (see the section "Messages for Input from Windows").

You do input from windows rather than only from the keyboard so that many programs can share the keyboard without getting in each other's way. If two processes try to read from the keyboard at the same time, they can do it by going through windows. Characters from the keyboard go only to the selected window, and not to any of the others; this way, you can control which process you are typing at, by selecting the window you are interested in.

If a process tries to do input from a window that does not have any characters in its input buffer, what happens depends on the window's *deexposed typein action*. It may be either **:normal** or **:notify**. If the deexposed typein action is **:normal**, and/or the window is exposed, then the process waits until something appears in the input buffer. If it is **:notify** and the window is not exposed, the user is notified with a message such as "Process X wants typein", and the window is "made interesting" so that FUNCTION 0 S can select it.

Reading characters from a window normally returns an integer that represents a character in the Symbolics character set, possibly with extra bits that correspond to the CONTROL, META, SUPER, and HYPER keys. For information on the format of such integers and the symbolic names of the bit fields, see the section "The Character Set".

Note that reading characters from a window does not echo the characters; it does not type them out. If you want echoing, you can echo the characters yourself, or call the higher-level functions such as **zl:tyi**, **zl:read**, and **zl:readline**; these functions accept a window as their stream argument and will echo the characters they read.

Every window (that has **tv:stream-mixin** as a component) has an *I/O buffer* that holds characters that are typed by the user before any program reads the characters. When you type a character, it enters this buffer, and stays there until a program tries to read characters from this window. There are some messages that deal with the I/O buffer, letting you clear it and ask whether there is anything in it; see the section "Messages for Input from Windows".

Normally, integers get into the I/O buffer because characters were typed on the keyboard. However, you can also get any Lisp object into a window's I/O buffer under program control by sending a **:force-kbd-input** message to the window. One common use of this feature is for the mouse process to tell a user process about activity on the mouse buttons. That is how characters with the **%%kbd-mouse** bit can get read from the window. It is possible to put Lisp objects other than integers into an I/O buffer; by convention, such objects are usually lists whose first element is a symbol saying what kind of a "message" this object is. (Such lists are

sometimes called *blips*.) You can also get the mouse to send blips instead of integers, in order to find out the mouse position at the time of the click. Using the mouse is explained in the section "Mouse Input" .

You can explicitly manipulate I/O buffers in order to get certain advanced functionality by using the **:io-buffer** init option and the **:io-buffer** and **:set-io-buffer** messages. For example, you can make several windows use the same I/O buffer; this is often used to make panes of a paned window all share the same I/O buffer. You can also put properties on the I/O buffer's property list; this lets you request various special features.

The console hardware actually sends codes to Genera whenever a key is pressed or lifted; thus, Genera knows at all times which keys are pressed and which are not. You can use the **tv:key-state** function to ask whether a key is down or up. Also, you can arrange for reading from a window to read the raw hardware codes exactly as they are sent by putting a non-**nil** value of the **:raw** property on the property list of the I/O buffer; however, the format of the raw codes is complicated and dependent on the hardware implementation. It is not documented here.

The window system intercepts some characters specially. Some are intercepted when the user process is about to read the character from a window; others are intercepted as soon as they are typed. In the first category, the **io-buffer-output-function** of the I/O buffer defaults to **tv:kbd-default-output-function**, which intercepts certain characters when they are read. The value of the variable **sys:kbd-intercepted-characters** is a list of characters that are intercepted and not returned as input from the window. These characters default to **#\abort**, **#\m-abort**, **#\suspend**, and **#\m-suspend**. For more information, see the section "Intercepted Characters".

The second category of specially handled characters is those handled *asynchronously*. See the section "Asynchronous Characters".

### Messages for Input from Windows

Windows support all the input operations that interactive streams in general do (see the section "Messages for Input from Interactive Streams").

Windows have specialized versions of some of these operations, mainly involved in reading characters from I/O buffers. These are:

**(flavor:method :any-tyi tv:stream-mixin)**
**(flavor:method :any-tyi-no-hang tv:stream-mixin)**
**(flavor:method :untyi tv:stream-mixin)**
**(flavor:method :listen tv:stream-mixin)**
**(flavor:method :clear-input tv:stream-mixin)**

### SELECT and FUNCTION Keys

These facilities can be used to control the SELECT and FUNCTION keys:

**tv:add-function-key** *char function documentation* &rest *options*
Adds *char* to the list of keys that can follow the FUNCTION key.

**tv:*function-keys***
An alist, each entry of which describes a subcommand of the FUNCTION key.

**tv:add-select-key** *char flavor name* &optional (*create-p* **t**) *clobber-p*
Adds *char* to the list of keys that can follow the SELECT key.

**tv:*select-keys***
An alist, each entry of which describes a subcommand of the SELECT key. Obsolete as of Genera 7.3 Ivory.

**sys:set-select-key-activity** *char activity-name* &key *:clobber-p*

### Asynchronous Characters

The FUNCTION and SELECT keys are always intercepted as soon as they are typed; they cause the **Keyboard** process to take special action to handle the command that the user is giving. You can add your own FUNCTION and SELECT commands, using the functions **tv:add-function-key** and **tv:add-select-key**. See the section "SELECT and FUNCTION Keys".

Other characters can also be intercepted as soon as they are typed. A special system process called the keyboard process calls a user-defined function as soon as the key is pressed. The main process of the program is left undisturbed. This function runs in parallel with the main program and could communicate with it.

Asynchronous character handling is available to any window that includes **tv:stream-mixin**. The window has a list that associates keyboard characters with functions. The default list contains c-ABORT, c-SUSPEND, c-m-ABORT, and c-m-SUSPEND. The default actions are the same as those of the corresponding keys without c- modifiers, except that the window's process is sent an **:interrupt** message so that the actions take place immediately.

The keyboard process checks each character coming in to see if it is defined as an asynchronous character for the selected window. When it is, the keyboard process calls the associated function in the context of the keyboard process.

The function that runs as a result of an asynchronous character is running in the keyboard process. It is called with two arguments, the character and **self**. It should be very short and must not do any I/O. An error in one of these functions would break the keyboard process and the keyboard along with it and you would have to warm boot. To avoid any possibility of errors, you can have the function create a new process with **process-run-function** and make the new process handle the real work.

You can set up your own handling of asynchronous characters by using the **:asynchronous-character-p**, **:handle-asynchronous-character**, **:add-asynchronous-character**, and **:remove-asynchronous-character** messages and the **:asynchronous-characters** init option for **si:interactive-stream**. See the section "Interactive-Stream Operations for Asynchronous Characters".

# TV Fonts

## Using TV Fonts

In Genera, characters can be typed out in any of a number of different typefaces. Some text is printed in characters that are small or large, boldface or italic, or in different styles altogether. Each such typeface is called a *font*. A font is conceptually an array, indexed by character code, of pictures showing how each character should be drawn on the screen. The Font Editor (FED) is a program that allows you to create, modify, and extend fonts: See the section "Font Editor".

A font is represented internally as a Lisp object. Each font has a name. The name of a font is a symbol, usually in the **fonts** package, and the symbol is bound to the font. A typical font name is **tr8**. In the initial Lisp environment, the symbol **fonts:tr8** is bound to a font object whose printed representation is something like:

```
#<FONT TR8 234712342>
```

The interface to fonts is provided by *character styles* (for more information: See the section "Character Styles".) You can (indirectly) control which font is used when output is done to a window by specifying the default character style for that window: See the init option **(flavor:method :default-character-style tv:sheet)**. Additional control over character styles is provided by several output macros: See the section "Controlling Character Style".

The character style resulting from merging the output character style against a window's default character style maps to a particular font. This is true of both static and Dynamic Windows. This font is the *current font* for the window; to access it you can use the **:current-font** message. To discover what font corresponds to a particular character style, use the function **si:backtranslate-font**.

When you create a font of your own, there are basically two ways you can make use of it: 1) for defining a new character style; and 2) as a collection of glyphs for graphics output. To define a new character style and associate your font with it, use the function **si:define-character-style-families**: See the section "Mapping a Character Style to a Font". To draw a glyph included in a font array, use **graphics:draw-glyph**: See the function **graphics:draw-glyph**. One additional facility provided for interfacing with TV fonts is the **:baseline** method of **tv:sheet**.

## Standard TV Fonts

You can use `Show Font HELP` in the Lisp Listener or the List Fonts (`m-X`) command in Zmacs to get a list of all the fonts that are currently loaded into the Lisp environment. The **fonts** package contains the names of all fonts. Here is a list of some of the useful fonts:

**fonts:cptfont** — This is the default font, used for almost everything.

**fonts:jess14** — This is the default font in menus. It is a variable-width rounded font, slightly larger and more attractive than medfnt.

| | |
|---|---|
| **fonts:cptfonti** | This is a fixed-width italic font of the same width and shape as **fonts:cptfont**, the default screen font. It is most useful for italicizing running text along with **fonts:cptfont**. |
| **fonts:cptfontcb** | This is a fixed-width bold font of the same width and shape as **fonts:cptfont**, the default screen font. |
| **fonts:medfnt** | This is a fixed-width font with characters somewhat larger than those of **cptfont**. |
| **fonts:medfnb** | This is a bold version of **medfnt**. When you use Split Screen, for example, the [Do It] and [Abort] items are in this font. |
| **fonts:hl12i** | This is a variable-width italic font. It is useful for italic items in menus; Zmail uses it for this in several menus. |
| **fonts:tr10i** | This is a very small italic font. It is the one used by the Inspector to say *"More above"* and *"More below"*. |
| **fonts:hl10** | This is a very small font used for nonselected items in Choose Variable Values windows. |
| **fonts:hl10b** | This is a bold version of **hl10**, used for selected items in Choose Variable Values windows. |

### Attributes of TV Fonts

Fonts, and characters in fonts, have several interesting attributes.

#### *Character Height* Font Attribute

One attribute of each font is its *character height*. This is a nonnegative integer used to figure out how tall to make the lines in a window. Each window has a certain *line height*. The line height is computed by examining each font in the font map, and finding the one with the largest character height. This largest character height is added to the vertical spacing (in pixels) between the text lines (*vsp*) specified for the window, and the sum is the line height of the window. The line height, therefore, is recomputed every time the font map is changed or the *vsp* is set. This ensures that any line has enough room to display the largest character of the largest font and still leave the specified vertical spacing between lines. One effect of this is that if you have a window that has two fonts, one large and one small, and you do output in only the small font, the lines are still spaced far enough apart to accommodate characters from the large font. This is because the window system cannot predict when you might, in the middle of a line, suddenly switch to the large font.

#### *Baseline* Font Attribute

Another attribute of a font is its *baseline*. The baseline is a nonnegative integer that is the number of raster lines between the top of each character and the base of the character. (The base is usually the lowest point in the character, except for letters that descend below the baseline, such as lowercase p and g.) This number is stored so that when you are using several different fonts side-by-side, they are aligned at their bases rather than at their tops or bottoms. So when you output a character at a certain cursor position, the window system first examines the baseline of the current font, then draws the character in a position adjusted vertically to make the bases of the characters all line up.

### *Character Width* Font Attribute

The *character width* can be an attribute either of the font as a whole, or of each character separately. If there is a character width for the whole font, it is as if each character had that character width separately. The character width is the amount by which the cursor position should be moved to the right when a character is output on the window. This can be different for different characters if the font is a variable-width font, in which a W might be much wider than an i. Note that the character width does not necessarily have anything to do with the actual width of the bits of the character (although it usually does); it is merely defined to be the amount by which the cursor should be moved.

### *Left Kern* Font Attribute

The *left kern* is an attribute of each character separately . Usually it is zero, but it can also be a positive or negative integer. When the window system draws a character at a given cursor position, and the left kern is nonzero, the character is drawn to the left of the cursor position by the amount of the left kern, instead of being drawn exactly at the cursor position. In other words, the cursor position is adjusted to the left by the amount of the left kern of a character when that character is drawn, but only temporarily; the left kern only affects where the single character is drawn and does not have any cumulative effect on the cursor position.

### *Fixed-width* Font Attribute

A font that does not have separate character widths for each character and does not have any nonzero left kerns is called a *fixed-width* font. The characters are all the same width and so they line up in columns, as in typewritten text. Other fonts are called *variable-width* because different characters have different widths and things do not line up in columns. Fixed-width fonts are typically used for programs, where columnar indentation is used, while variable-width fonts are typically used for English text, because they tend to be easier to read and to take less space on the screen.

### *Blinker Width* and *Blinker Height* Font Attributes

The *blinker width* and *blinker height* are two nonnegative integers that tell the window system an attractive width and height to make a rectangular blinker for characters in this font. These attributes are completely independent of all other attributes and are only used for making blinkers. Using a fixed width blinker for a variable-width font causes problems; the editor actually readjusts its blinker width as a function of what character it is on top of, making a wide blinker for wide characters and a narrow blinker for narrow characters. The easiest thing to do is to use the blinker width as the width of the blinker. This works well with a fixed-width font.

### *Chars-exist-table* Font Attribute

The *chars-exist-table* is **nil** if all characters exist in a font, or an **sys:art-boolean** array. This table is not used by the character-drawing software; it is for informational purposes. Characters that do not exist have pictures with no bits "on" in them, just like the Space character. Most fonts implement most of the printing characters in the character set, but some are missing some characters.

## Format of TV Fonts

The array leader of a font is a structure defined by **zl:defstruct**. Here are the names of the accessors for the elements of the array leader of a font:

**zl:font-name**
**zl:font-char-height**
**zl:font-char-width**
**zl:font-baseline**
**zl:font-char-width-table**
**zl:font-left-kern-table**
**zl:font-blinker-width**
**zl:font-blinker-height**
**zl:font-chars-exist-table**
**zl:font-raster-height**
**zl:font-raster-width**
**zl:font-indexing-table**

## Blinkers

Each static or Dynamic Window can have any number of *blinkers*. The kind of blinker that you see most often is a blinking rectangle the same size as the characters you are typing; this blinker shows you the cursor position of the window. In fact, a window can have any number of blinkers. They need not follow the cursor (some do and some don't); the ones that do are called *following* blinkers; the others have their position set by explicit messages.

Also, blinkers need not actually blink; for example, the mouse arrow does not blink. A blinker's *visibility* may be any of the following:

**:blink** The blinker should blink on and off periodically. The rate at which it blinks is called the *half-period*, and is an integer giving the number of 60ths of a second between when the blinker turns on and when it turns off.

**:on** *or* **t** The blinker should be visible but not blink; it should just stay on.

**:off** *or* **nil** The blinker should be invisible.

Usually only the blinkers of the selected window actually blink; this is to show you where your typein will go if you type on the keyboard. The way this behavior is obtained is that selection and deselection of a window have an effect on the visibility of the window's blinkers.

When the window is selected, any of its blinkers whose visibility is **:on** or **:off** has its visibility set to **:blink**. Blinkers whose visibility is **t** or **nil** are unaffected (that is the difference between **t** and **:on**, and between **nil** and **:off**); blinkers whose visibility is **:blink** continue to blink.

Each blinker has a *deselected visibility*, which should be one of the symbols above; when a window is deselected, the visibilities of all blinkers that are blinking (whose visibility is currently **:blink**) are set to the deselected visibility.

Most often, blinkers have visibility **:on** when their window is not selected, and visibility **:blink** when their window is selected. In this case, the deselected visibility is **:on**.

Blinkers are used to add visible ornaments to a window; a blinker is visible to the user, but while programs are examining and altering the contents of a window the blinkers all go away. The way this works is that before characters are output or graphics are drawn, the blinker gets turned off; it comes back later. This is called *opening* the blinker. You can see this happening with the mouse blinker when you type at Genera. To make this work, blinkers are always drawn using exclusive ORing. See the variable **tv:alu-xor**.

Every blinker is associated with a particular window. A blinker cannot leave the area described by its window; its position is expressed relative to the window. When characters are output or graphics are drawn on a window, only the blinkers of that window and its ancestors are opened (since blinkers of other windows cannot possibly be occupying screen space that might overlap this output or graphics). The mouse blinker is free to move all over whatever screen it is on; it is therefore associated with the screen itself, and so must be opened whenever anything is drawn on any window of the screen.

The window system provides a few kinds of blinkers. Blinkers are implemented as instances of flavors, too, and have their own set of messages that they understand, which is distinct from the set that windows understand.

Positions of blinkers are always expressed in pixels, relative to the inside of the window (that is, the part of the window that doesn't include the margins).

### General Blinker Operations

These are the general blinker operations:

**tv:make-blinker**
**(flavor:method :x-pos tv:blinker)**
**(flavor:method :y-pos tv:blinker)**
**(flavor:method :read-cursorpos tv:blinker)**
**(flavor:method :set-cursorpos tv:blinker)**
**(flavor:method :follow-p tv:blinker)**
**(flavor:method :set-follow-p tv:blinker)**
**(flavor:method :visibility tv:blinker)**
**(flavor:method :set-visibility tv:blinker)**
**(flavor:method :deselected-visibility tv:blinker)**
**(flavor:method :deselected-visibility tv:blinker)**
**(flavor:method :set-deselected-visibility tv:blinker)**
**(flavor:method :half-period tv:blinker)**
**(flavor:method :half-period tv:blinker)**
**(flavor:method :set-half-period tv:blinker)**
**(flavor:method :set-sheet tv:blinker)**
**tv:sheet-following-blinker**
**tv:turn-off-sheet-blinkers**

### Specialized Blinkers

These are the specialized blinkers:

**tv:rectangular-blinker**
**(flavor:method :width tv:rectangular-blinker)**
**(flavor:method :height tv:rectangular-blinker)**
**(flavor:method :set-size tv:rectangular-blinker)**
**tv:hollow-rectangular-blinker**
**tv:box-blinker**
**tv:ibeam-blinker**
**(flavor:method :height tv:ibeam-blinker)**
**tv:character-blinker**
**(flavor:method :font tv:character-blinker)**
**(flavor:method :char tv:character-blinker)**
**(flavor:method :set-character tv:character-blinker)**

### Mouse Input

### Introduction

The "Mouse Input" section describes the mouse process and mouse facilities in the pre-Genera 7.0 context of static windows. In this context, the mouse process has broader responsibilities than it does in Dynamic Windows, and many applications have included considerable amounts of code running in the mouse process in addition to that running in the user process. Coordinating the two processes is sometimes tricky, and the facilities described for "grabbing the mouse", "usurping the mouse", and so on are helpful in providing more control in the user process. (See the sections "Grabbing the Mouse" and "Usurping the Mouse".)

In Dynamic Windows, the mouse process has fewer duties, being responsible primarily for communicating to the user process where the mouse cursor is and whether any actions involving the mouse have occurred. With Dynamic Windows and the presentation-type system, mouse sensitivity of displayed items is a built-in feature. Facilities in Dynamic Windows forming the interface to the mouse process are **dw:tracking-mouse** and the mouse handler facilities ( see the section "Programming the Mouse: Writing Mouse Handlers").

### Handling the Mouse

Along with the keyboard, the mouse can be used by any program as an input device. The functions, variables, and flavors described in the sections under "Mouse Input" allow you to use the mouse to do some simple things. To get advanced mouse behavior in your own programs, such as the way the editor gets the mouse to put a box around the character being pointed at, you must extend the window system by writing your own methods, which is beyond the scope of this document. Of course, you can invoke the built-in choice facilities, such as menus and multiple-choice windows. These high-level facilities are described elsewhere. See the section "Window System Choice Facilities".

The window system includes a process called `Mouse` that normally *tracks* the mouse. To track the mouse means to examine the hardware mouse interface, noting how the mouse is moving, and adjust Lisp variables and the mouse blinker to follow the position being indicated by the user. The mouse process also keeps track of which window *owns* the mouse at any time. For example, when the mouse enters an editor window, the editor window becomes the owner. To indicate this, the mouse process changes the blinker from a northwest arrow to a northeast arrow.

In general, the window that owns the mouse is the window that is under the mouse; but since the windows are arranged in a hierarchy, generally a window, its superior, its superior's superior, and so on, are all under the mouse at the same time. So the window that owns the mouse is really the lowest window in the hierarchy (farthest in the hierarchy from the screen) that is visible (it and all its ancestors are exposed). If you move the window to part of the screen occupied by a partially visible window, one of its ancestors (often the screen itself) becomes the owner. The screen handles single-clicking on the Left button by selecting the window under it, allowing you to select partially visible windows with the mouse.

In general, the mouse process decides how to handle the mouse based on the flavor of the window that owns the mouse. Some flavors handle the mouse themselves, running in the mouse process, in order to be able to put boxes around

things, usually to indicate what would happen if you were to click a button. (This has changed in Dynamic Windows; see the section "Introduction to Mouse Input".)

As noted, to do this you must extend the window system, creating your own methods to be run in the mouse process; that is beyond the scope of this document. The flavor of the window owning the mouse is also what usually controls the effect of clicking the mouse buttons. There are three ways for you to use the mouse without writing your own methods.

- You can mix in flavors to your window to tell the mouse process to let you know when the mouse is clicked.

- You can watch the mouse moving and watch the buttons, letting the mouse process do the tracking.

- You can turn off the mouse process and do your own tracking.

You must choose one of these three ways to use the mouse; you cannot mix them. Note that you can also use various high-level facilities to get certain specific mouse behavior: for example, you can create windows with mouse-sensitive items (like the List Buffers (m-X) command in the Editor), menus, and multiple-choice windows.

Several of the following facilities are methods for **tv:essential-mouse**. This is a component flavor of both **tv:window** and **dw:dynamic-window**. These are the substrate facilities for handling the mouse:

**tv:mouse-sheet**
**(flavor:method :handle-mouse tv:essential-mouse)**
**(flavor:method :mouse-moves tv:essential-mouse)**
**(flavor:method :set-mouse-position tv:essential-mouse)**
**(flavor:method :who-line-documentation-string tv:sheet)**
**tv:mouse-warp**
**tv:mouse-set-blinker-cursorpos**
**sys:mouse-wakeup**
**tv:window-under-mouse**

### Mouse Blips

Mouse blips are lists inserted into the input buffer of a window when the mouse is clicked within that window. (Do not confuse these blips with presentation blips generated by translating mouse handlers when the mouse is clicked on a presentation in a Dynamic Window: see the section "Presentation Input Blip Facilities".) The list contains five elements:

1. The keyword **:mouse-button**.

2. A mouse character corresponding to which button (Left, Middle, Right) was clicked.

3. The window that received the blip.

4. The x-coordinate of the mouse cursor when the mouse was clicked.

5. The y-coordinate of the mouse cursor when the mouse was clicked.

Blips representing mouse clicks are sent by the **:mouse-click** method of **tv:essential-mouse**, a component of **tv:minimum-window**. You can receive mouse blips by sending the window a **:list-tyi** or **:any-tyi** message. (For an example, see the section "Mouse Characters".)

**(flavor:method :mouse-click tv:essential-mouse)** *buttons x y* *Method*

Called by the **:mouse-buttons** method of **tv:essential-mouse**, which is called by the default mouse handler when mouse buttons are pushed. *buttons* is a structure representing the buttons pushed; use reader macros like **#\Mouse-R** to handle these structures in your program. (See the section "Mouse Characters".) *x* and *y* represent the position of the mouse at the time of the click, in the window's outside coordinates.

If the click is **#\sh-Mouse-R**, the **:mouse-buttons** method pops up a system menu. Otherwise, if the window has an I/O buffer, **:mouse-click** sends it a blip of the form (**:mouse-button** *buttons window x y*). In addition, if the click is **#\Mouse-L**, the window is selected.

**:mouse-click** methods are combined using **:or** combination, so the **:mouse-click** method of **tv:essential-mouse** runs only if no earlier method handles the message (and all earlier methods return **nil**). This allows a method to intercept only certain clicks and return non-**nil**, and to pass on other clicks and return **nil**.

### Mouse Characters

Mouse characters are implemented as structures, not character objects, but the printed representation is similar. **#\Mouse-L**, **#\Mouse-M**, and **#\Mouse-R** correspond to Left, Middle and Right clicks. Mouse characters can be qualified by shift keys. For example, **#\c-Mouse-M** indicates a Middle click with the CONTROL key pressed.

Mouse characters prefixed with **sh-**, such as **#\sh-Mouse-R**, can be generated by the user in two ways.

- Pressing the SHIFT key while clicking the Right mouse button.

- Clicking the Right mouse button twice in rapid succession.

(The latter interpretation is possible only if the variable **tv:mouse-double-click-time** has not been set to **nil**.)

Because mouse characters are not implemented as other characters, they require their own set of manipulation functions. For example, the function **char-mouse-equal** compares mouse characters and the predicate **mouse-char-p** determines whether an object is a mouse character. **char-mouse-equal** checks that its arguments are really mouse characters and signals an error otherwise. You can also use **eql**, which is slightly faster, to compare mouse characters, when you do not require the argument checking. **char-mouse-equal** and **mouse-char-p** are commonly used when handling mouse blips, as shown in the following example:

```
(defun get-mouse-char ()
  (let (blip mouse-char)
    (setq blip (send *graphics-window* :list-tyi))
    (setq mouse-char (second blip))
    (if (mouse-char-p mouse-char)
        (cond ((char-mouse-equal mouse-char #\mouse-l)
               (left-click-function))
              ((char-mouse-equal mouse-char #\mouse-r)
               (right-click-function))
              (t (send *graphics-window* :beep)))
        (send *graphics-window* :beep))))
```

These are the mouse character functions:

**char-mouse-equal**
**mouse-char-p**
**char-mouse-button**
**char-mouse-bits**
**make-mouse-char**

### Grabbing the Mouse

When the mouse is grabbed, the mouse process is told that no window owns the mouse, and it changes the mouse blinker back to the default (a northeast arrow). The mouse process continues to track the mouse, and your process can now watch the position and the buttons by using the variables and functions described below. (The corresponding facility for Dynamic Windows is **dw:tracking-mouse**: see the function **dw:tracking-mouse**.)

These are the facilities to use with the mouse grabbed:

**tv:with-mouse-grabbed**
**tv:with-mouse-grabbed-on-sheet**
**tv:with-mouse-and-buttons-grabbed**
**tv:with-mouse-and-buttons-grabbed-on-sheet**

**sys:mouse-x**
**sys:mouse-y**
**tv:mouse-last-buttons**
**tv:mouse-wait**
**tv:wait-for-mouse-button-down**
**tv:wait-for-mouse-button-up**
**tv:mouse-button-encode**
**tv:who-line-mouse-grabbed-documentation**

### Usurping the Mouse

You can tell the mouse process not to do anything, and track the mouse in your own process. This is called *usurping* the mouse. The mouse blinker disappears, so if you want any visual indication of the mouse to appear, you must do it yourself. Here are the facilities to use:

**tv:with-mouse-usurped**
**tv:mouse-input**
**sys:mouse-buttons**

### Controlling the Mouse Outside a Window

These are the facilities for controlling the mouse outside a window:

**tv:hysteretic-window-mixin**

**(flavor:method :hysteresis tv:hysteretic-window-mixin)**

**(flavor:method :hysteresis tv:hysteretic-window-mixin)**

**(flavor:method :set-hysteresis tv:hysteretic-window-mixin)**

### Scaling Mouse Motion

The following two variables apply to Dynamic Windows as well as static windows.

**tv:mouse-x-scale-array**
**tv:mouse-y-scale-array**

### The Keyboard

Another way of using the keyboard, different from reading a stream of input characters from a window, is to treat it as a "random access" device and look at the instantaneous state of particular keys.

One application for checking the state of keys is in user interfaces where the action of mouse clicks is modified by the shift keys on the keyboard; you can have one hand on the mouse and the other on the keyboard. You can use the variables **tv:mouse-double-click-time** and **tv:*mouse-incrementing-keystates*** to augment or replace double clicks with shifted clicks.

Mouse characters can be modified with the modifier keys CONTROL, META, SUPER, and HYPER, just as keyboard characters can. Which of these keys modify mouse characters depends on the value of the variable **tv:*mouse-modifying-keystates***.

The editor considers each modified mouse click to be a separate command. You can bind commands to particular modified mouse clicks. You can also use Install Mouse Macro (m-X) with modified mouse clicks to increase the number of mouse macros available.

You can use **login-forms** in an init file to set the variables **tv:mouse-double-click-time**, **tv:*mouse-incrementing-keystates***, and **tv:*mouse-modifying-keystates*** and customize the behavior of the mouse.

**tv:key-state** *key-name*

*Function*

Returns **t** if the keyboard key named *key-name* is currently pressed, **nil** if it is not.

*key-name* may be the symbolic name of a modifier key, from the table below, or a character object. Modifier keys that come in pairs have three symbolic names; one for the left-hand key, one for the right-hand key, and one for both, which is considered to be pressed if either member of the pair is.

The modifier key names are:

| | | |
|---|---|---|
| :shift | :left-shift | :right-shift |
| :symbol | :left-symbol | :right-symbol |
| :control | :left-control | :right-control |
| :meta | :left-meta | :right-meta |
| :super | :left-super | :right-super |
| :hyper | :left-hyper | :right-hyper |
| :caps-lock | :repeat | :mode-lock |

**tv:mouse-double-click-time**

*Variable*

The maximum period of time (in microseconds) between mouse clicks for

which the clicks are interpreted as a double click instead of two single clicks. Default: **200000** (decimal). If you set this to **nil**, disabling double clicking entirely, mouse response time improves slightly in static windows and appreciably in Dynamic Windows. This is the recommended setting.

**tv:*mouse-incrementing-keystates***

*Variable*

A list of names of keys, acceptable to **tv:key-state**. If one or more of these keys are pressed, single mouse clicks are interpreted as double clicks. Default: **(:shift)**.

**tv:*mouse-modifying-keystates***

*Variable*

A list of names of keys acceptable to **tv:key-state**. If one or more of these keys are pressed, sets the corresponding modifier bits in the mouse character. Default: **(:control :meta :super :hyper)**. If a key appears as an element of both this list and the list that is the value of **tv:*mouse-incrementing-keystates***, the modifier bit is set and the click is interpreted as a double click.

**Window Sizes and Positions**

The messages and init options in this section are used to examine and set the sizes and positions of windows, both static and dynamic. There are many different messages, which lets you express things in different forms that are convenient in varying applications. Usually, sizes are in units of pixels. However, sometimes we refer to widths in units of characters and heights in units of lines. The number of horizontal pixels in one character is called the character-width, and the number of vertical pixels in one line is called the line-height.

See the section "Character Output to Windows".

A window has two parts: the inside and the margins. The margins include borders, labels, and other things; the inside is used for drawing characters and graphics. Some of the messages below deal with the outside size (including the margins) and some deal with the inside size.

Since a window's size and position are usually established when the window is created, we will begin by discussing the init options that let you specify the size and position of a new window. To make things as convenient as possible, there are many ways to express what you want. The idea is that you specify various things, and the window figures out whatever you leave unspecified. For example, you can specify the right-hand edge and the width, and the position of the left-hand edge

will automatically be figured out. If you underspecify some parameters, defaults are used. Each edge defaults to being the same as the corresponding inside edge of the superior window; so, for example, if you specify the position of the left edge, but don't specify the width or the position of the right edge, then the right edge will line up with the inside right edge of the superior. If you specify the width but neither edge position, the left edge will line up with the inside left edge of the superior; the same goes for the height and the top edge.

In order for a window to be exposed, its position and size must be such that it fits within the *inside* of the superior window. If a window is not exposed, then there are no constraints on its position and size; it may overlap its superior's margins, or even be outside the superior window altogether.

All positions are specified in pixels and are relative to the *outside* of the superior window.

### Initializing Window Size and Position

The following options set various position and size parameters. The size and position of the window are computed from the parameters provided by these and other options, and the set of defaults described above. Note that all edge parameters are relative to the *outside* of the superior window.

**(flavor:method :left tv:sheet)**
**(flavor:method :x tv:sheet)**
**(flavor:method :top tv:sheet)**
**(flavor:method :y tv:sheet)**
**(flavor:method :position tv:sheet)**
**(flavor:method :right tv:sheet)**
**(flavor:method :bottom tv:sheet)**
**(flavor:method :width tv:sheet)**
**(flavor:method :height tv:sheet)**
**(flavor:method :size tv:sheet)**
**(flavor:method :inside-width tv:sheet)**
**(flavor:method :inside-height tv:sheet)**
**(flavor:method :inside-size tv:sheet)**
**(flavor:method :edges tv:sheet)**
**(flavor:method :character-width tv:sheet)**
**(flavor:method :character-height tv:sheet)**
**(flavor:method :integral-p tv:sheet)**
**(flavor:method :edges-from tv:essential-window)**
**(flavor:method :minimum-width tv:essential-window)**
**(flavor:method :minimum-height tv:essential-window)**
**tv:set-default-window-size**

### Messages for Window Size and Position

Many messages that change the window's size or position take an argument called *option*. The reason that this argument exists is that certain new sizes or positions are not valid. One reason that a size may not be valid is that it may be so small that there is no room for the margins; for example, if the new width is smaller than the sum of the sizes of the left and right margins, then the new width is not valid. Another reason a new setting of the edges may be invalid is that if the window is exposed, it is not valid to change its edges in such a way that it is not enclosed inside its superior. In all of the messages that take the *option* argument, *option* may be either **nil** or **:verify**. If it is **nil**, that means that you really want to set the edges, and if the new edges are not valid, an error should be signalled. If it is **:verify**, that means that you only want to check whether the new edges are valid or not, and you don't really want to change the edges. If the edges are valid, the message will return **t**; otherwise it will return two values: **nil** and a string explaining what is wrong with the edges. (Note that it is valid to set the edges of a deexposed inferior window in such a way that the inferior is not enclosed inside the superior; you just can't expose it until the situation is remedied. This makes it more convenient to change the edges of a window and all of its inferiors sequentially; you don't have to be careful about what order you do it in.)

These messages are used to examine or change the size or position of a window:

**(flavor:method :change-of-size-or-margins tv:sheet)**
**(flavor:method :size tv:sheet)**
**(flavor:method :set-size tv:essential-set-edges)**
**(flavor:method :inside-size tv:sheet)**
**(flavor:method :set-inside-size tv:essential-set-edges)**
**(flavor:method :size-in-characters tv:sheet)**
**(flavor:method :set-size-in-characters tv:sheet)**
**(flavor:method :position tv:sheet)**
**(flavor:method :set-position tv:essential-set-edges)**
**(flavor:method :edges tv:sheet)**
**(flavor:method :set-edges tv:essential-set-edges)**
**(flavor:method :margins tv:sheet)**
**(flavor:method :left-margin-size tv:sheet)**
**(flavor:method :top-margin-size tv:sheet)**
**(flavor:method :right-margin-size tv:sheet)**
**(flavor:method :bottom-margin-size tv:sheet)**
**(flavor:method :inside-edges tv:sheet)**
**(flavor:method :center-around tv:essential-set-edges)**
**(flavor:method :expose-near tv:essential-set-edges)**

### Window Margins, Borders, and Labels

There is a distinction between the inside and outside parts of the window. The part of the window that is not the inside part is called the *margins*. There are four margins, one for each edge. The margins sometimes contain a *border*, which is a rectangular box drawn around the outside of the window. Borders help the

user see what part of the screen is occupied by which window. The margins also sometimes contain a *label*, which is a text string. Labels help the user see what a window is for.

A label can be inside the borders or outside the borders (usually it is inside). In general, there can be lots of things in the margins; each one is called a *margin item*. Borders and labels are two kinds of margin items. In any flavor of window, one of the margin items is the innermost; it is right next to the inside part of the window. Each successive margin item is outside the previous one; the last one is just inside the edges of the window. Each margin item is created by a flavor's being mixed in. You can control which margin items your window has by which flavors you mix in, and you can control their order by the order in which you mix in the flavors. Margin item flavors closer to the front of the component flavor list are further outside in the margins. The **tv:window** flavor has as components **tv:borders-mixin** and **tv:label-mixin**, in that order, and so the label is inside the border. You can ask for the size of the margins with the **:margins** message.

Here is a list of the margin facilities that you can mix in. With few exceptions, all of these facilities are intended for static windows, not dynamic ones. For information on equivalent facilities for use with Dynamic Windows: See the section "Window Substrate Facilities". More detailed information is available in the reference documentation for **dw:dynamic-window**: See the flavor **dw:dynamic-window**.

**tv:margin-space-mixin**
**(flavor:method :margin-space tv:margin-space-mixin)**
**(flavor:method :margin-space tv:margin-space-mixin)**
**(flavor:method :set-margin-space tv:margin-space-mixin)**

### Window Borders

Here is a list of facilities for creating window borders.

**tv:borders-mixin**
**(flavor:method :borders tv:borders-mixin)**
**(flavor:method :set-borders tv:borders-mixin)**
**(flavor:method :border-margin-width tv:borders-mixin)**
**(flavor:method :border-margin-width tv:borders-mixin)**
**(flavor:method :set-border-margin-width tv:borders-mixin)**

### Window Labels

Of the following facilities, only the **:name** and **:label** init options and the **:name** method apply to Dynamic Windows; the rest are intended for static windows. For information on equivalent facilities intended for use with Dynamic Windows: See the section "Window Substrate Facilities".

**tv:label-mixin**
**(flavor:method :name tv:sheet)**

**(flavor:method :name tv:sheet)**
**(flavor:method :label-size tv:label-mixin)**
**(flavor:method :set-label tv:label-mixin)**
**tv:top-box-label-mixin**
**tv:changeable-name-mixin**
**(flavor:method :set-name tv:changeable-name-mixin)**
**tv:delayed-redisplay-label-mixin**
**(flavor:method :delayed-set-label tv:delayed-redisplay-label-mixin)**
**(flavor:method :update-label tv:delayed-redisplay-label-mixin)**

## Text Scroll Windows

### Concepts

A *text scroll window* maintains and displays an ordered list of Lisp objects, one on each line. The caller inserts objects into or deletes objects from the list by sending messages, and the window dynamically redisplays to show the changes. If there are more items in the list than lines in the window, the text scroll window displays some portion of the items. The portion that is shown is controlled by *scrolling* the window. The caller scrolls the window by sending messages, and the user scrolls it by using the mouse scroll bar.

### Text Scroll Window Flavors

**tv:text-scroll-window** is the most basic text scroll window mixin. It simply displays the items and allows you to scroll the window using the mouse against the left edge.

**tv:function-text-scroll-window** lets you provide a function to print an item, replacing **prin1**, to give you finer control over how each item is displayed.

**tv:mouse-sensitive-text-scroll-window** makes the items displayed on the window sensitive to mouse clicks.

**tv:margin-scrolling-with-flashy-scrolling-mixin** provides the *More above/More below* facility.

### Basic Use of Text Scroll Windows

You can use any of the usual options to **tv:make-window** to control such parameters as the size and shape of the window. When the window is first created, its item list is empty and it displays as an empty window. Here is a list of other facilities that you can use for scrolling windows.

**tv:text-scroll-window**
**(flavor:method :insert-item tv:text-scroll-window)**
**(flavor:method :append-item tv:text-scroll-window)**

**(flavor:method :delete-item tv:text-scroll-window)**
**(flavor:method :replace-item tv:text-scroll-window)**
**(flavor:method :set-items tv:text-scroll-window)**
**(flavor:method :items tv:text-scroll-window)**
**(flavor:method :number-of-items tv:text-scroll-window)**
**(flavor:method :top-item tv:text-scroll-window)**
**(flavor:method :last-item tv:text-scroll-window)**
**(flavor:method :put-item-in-window tv:text-scroll-window)**
**(flavor:method :put-last-item-in-window tv:text-scroll-window)**
**(flavor:method :item-value tv:text-scroll-window)**
**(flavor:method :scroll-to tv:basic-scroll-bar)**

### Example of a Text Scroll Window

This example creates a small text scroll window in the upper left corner of the screen and uses most of the text scroll window methods. It then leaves the window on the screen so that you can also scroll the window using the mouse. Reselect the original window to deexpose it.

```
(defflavor test-window ()
           (tv:text-scroll-window tv:window)
  )

(defvar *test-window*
        (tv:make-window 'test-window
                        :edges '(0 0 400 100)
```

```
                        :expose-p nil))
(defun test-basic-scroll-window ()
  ;; Initialize window
  (send *test-window* :set-items 0)             ; Clear the items
  (send *test-window* :expose)
  (send *test-window* :scroll-to 0 :absolute)   ; Scroll to the top
  ;; Demonstrate appending of items to the end of the list
  (loop for i from 0 to 10
        do
    (send *test-window* :append-item (list 'appended i))
    (process-sleep 60 (format nil "appending ~d" i)))
  ;; Demonstrate absolute scrolling
  (loop for i from 1 to 10 by 2
        do
    (send *test-window* :scroll-to i :absolute)
    (process-sleep 60 (format nil "scrolled to item ~d" i)))
  ;; Scroll to a arbitrary point in the middle of the item list
  (send *test-window* :scroll-to 3 :absolute)
  ;; Demonstrate insertion of items
  (loop for i from 1 to 10
        for j from 10 by -1
        do
    (send *test-window* :insert-item j (list 'inserted i))
    (process-sleep 60 (format nil "inserting ~d at ~d" i j)))
```

```
;; Demonstrate replacement of items
  (loop for i from 1
        for j from 1 by 3
        until (> j (send *test-window* :number-of-items))
        do
    (send *test-window* :replace-item j (list 'replaced i))
    (process-sleep 60 (format nil "replacing ~d at ~d" i j)))
  ;; Scroll to bottom of item list
  (send *test-window* :put-last-item-in-window)
  (process-sleep 60 "put last item in window")
  ;; Demonstrate relative scrolling
  (loop until (zerop (send *test-window* :top-item))
        do
    ;; Scroll back two items
    (send *test-window* :scroll-to -2 :relative)
    (process-sleep 60 "scrolled back 2"))
  ;; Demonstrate deletion of items
  (loop until (< (send *test-window* :number-of-items) 10)
          do
    (send *test-window* :delete-item 0)
    (process-sleep 60 "deleting the first item")))
```

**Formatting Text Scroll Window Items**

The simple **tv:text-scroll-window** calls **prin1** on each item to display it on a line of the screen. **tv:function-text-scroll-window** lets you provide a function of your own to replace **prin1**.

When the window displays a line, the function is called with four arguments:

- The item to be printed.
- An object associated with the window. See the method **(flavor:method :set-print-function-arg tv:function-text-scroll-window)**.
- The window itself.
- The number of the item in the window's item list.

When the function is called, the window's cursor is positioned to the beginning of the appropriate line on the window, so you can just send stream output messages to the window (the third argument). Do *not* output the new-line character to the window.

Here is a list of facilities that you can use to format text scroll window items.

**tv:function-text-scroll-window**
**(flavor:method :set-print-function tv:function-text-scroll-window)**
**(flavor:method :print-function tv:function-text-scroll-window)**
**(flavor:method :set-print-function-arg tv:function-text-scroll-window)**
**(flavor:method :print-function-arg tv:function-text-scroll-window)**

### Example of Formatting Text Scroll Window Items

Change the previous example (see the section "Example of a Text Scroll Window".) as follows:

```
(defflavor test-window ()
          (tv:function-text-scroll-window tv:window)
  )

        .
        .
        .

(defun test-basic-scroll-window ()
  (send *test-window* :set-print-function
        #'(lambda (item object window number)
           (format window "~:r item which was ~a."
                   (second item)
                   (first item))))
  (send *test-window* :set-items 0)

        .
        .
        .
```

### Mouse-Sensitive Items in Text Scroll Windows

The flavors **tv:mouse-sensitive-text-scroll-window** and **tv:mouse-sensitive-text-scroll-window-without-click** allow you to create *mouse-sensitive items*; that is, regions of each line can be made sensitive to mouse clicks.

Note that the word "item" is being used in two ways. One "item" of the item list is displayed on every line, but each line might have many "mouse-sensitive items" on it.

When the mouse is clicked, a *blip* is forced into the window's input buffer. The elements of the blip are:

- The type of the mouse-sensitive-item.

- The "item" which the "mouse-sensitive item" was in.
- The window itself.
- The mouse click character. See the section "The Character Set".

Here is a list of facilities to use for creating mouse-sensitive items in text scroll windows.

**tv:mouse-sensitive-text-scroll-window**
**tv:mouse-sensitive-text-scroll-window-without-click**
**:print-item**
**(flavor:method :item tv:mouse-sensitive-text-scroll-window-without-click)**
**(flavor:method :mouse-sensitive-item tv:mouse-sensitive-text-scroll-window-without-click)**
**tv:sensitive-item-types**
**tv:displayed-item-item**
**tv:displayed-item-type**

### Example of Mouse-Sensitive Items in Text Scroll Windows

This example creates a frame with a text scroll window and a plain window as panes. Clicks on the text scroll window display the blips on the plain window and toggle the mouse-sensitivity of the items.

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(defflavor test-pane ()
           (tv:mouse-sensitive-text-scroll-window
            tv:pane-no-mouse-select-mixin
            tv:window)
  (:default-init-plist
   :sensitive-item-types :sensitive-type-p))

(defmethod (:print-item test-pane) (item ignore ignore)
  (send self :item item :whole-item
        #'(lambda (item window)
            (send window :item item :first-part
                  #'(lambda (item window)
                      (format window "~r" (car item))))
            (format window " and ")
            (send window :item item :second-part
                  #'(lambda (item window)
                      (format window "~r" (cdr item)))))))
```

```
(defmethod (:who-line-documentation-string test-pane) ()
  (let ((superior (send self :superior)))
    (format nil "L: Turn left ~:[on~;off~]; ~
            M: Turn whole item ~:[on~;off~]; R: Turn right ~:[on~;off~]"
            (send superior :left)
            (send superior :both)
            (send superior :right))))

(defmethod (:sensitive-type-p test-pane) (mouse-sensitive-item)
  (let ((superior (send self :superior)))
    (case (tv:displayed-item-type mouse-sensitive-item)
      (:first-part (send superior :left))
      (:whole-item (send superior :both))
      (:second-part (send superior :right)))))

(defflavor test-frame ((left t) (both t) (right t))
           (tv:select-mixin
            tv:process-mixin
            tv:bordered-constraint-frame-with-shared-io-buffer)
  :settable-instance-variables
  (:default-init-plist
   :panes
   '((display-pane tv:window-pane)
     (scroll-pane test-pane))
   :constraints
   '((only . ((scroll-pane display-pane)
              ((scroll-pane .4))
              ((display-pane :even)))))
   :selected-pane 'display-pane
   :configuration 'only
   :process '(main-loop)))

(defun main-loop (frame)
  (send frame :main-loop))
```

```
(defmethod (:main-loop test-frame) ()
  (let* ((scroll-pane (send self :get-pane 'scroll-pane))
         (display-pane (send self :get-pane 'display-pane))
         (*terminal-io* display-pane))
    (loop for i from 1 to 5
          do
      (loop for j from 10 to 50 by 10
            do
        (send scroll-pane :append-item (cons i j))))
    (error-restart-loop ((sys:abort error)
                         "Silly program top level")
      (let ((blip (send display-pane :list-tyi)))
        (format t "~&Blip received was: ~%  ~s" blip)
        (case (if (eq (first blip) :mouse-button)
                  (second blip)
                  (fourth blip))
          (#\mouse-l (setq left (not left)))
          (#\mouse-m (setq both (not both)))
          (#\mouse-r (setq right (not right))))))))

(defvar *test-frame*
        (tv:make-window 'test-frame
                        :expose-p t))
```

### Flashy Scrolling in Text Scroll Windows

To scroll a display with the familiar *More above* and *More below* style scrolling, use **tv:margin-scrolling-with-flashy-scrolling-mixin**.

When this flavor is used, **tv:borders-mixin** should be included in the flavor definition before **tv:margin-scrolling-with-flashy-scrolling-mixin**. If it isn't, the *More above* and *More below* messages appear outside the borders.

If a label is required, **tv:top-box-label-mixin** should be placed after **tv:borders-mixin** and before **tv:margin-scrolling-with-flashy-scrolling-mixin** to put the label in the right place.

Here is a list of facilities for flashy scrolling within text scroll windows.

**tv:margin-scrolling-with-flashy-scrolling-mixin**
**(flavor:method :margin-scroll-regions tv:margin-scroll-mixin)**
**(flavor:method :flashy-scrolling-region tv:flashy-scrolling-mixin)**

### Example of Flashy Scrolling in Text Scroll Windows

Alter the previous example (See the section "Example of Mouse-Sensitive Items in Text Scroll Windows". ) as follows:

```
(defflavor test-pane ()
           (tv:borders-mixin
            tv:top-box-label-mixin
            tv:margin-scrolling-with-flashy-scrolling-mixin)
            tv:mouse-sensitive-text-scroll-window
            tv:pane-no-mouse-select-mixin
            tv:window)
  (:default-init-plist
   :sensitive-item-types :sensitive-type-p
   :margin-scroll-regions '(:top :bottom)))
```

### Typeout Windows

Here is a list of facilities that you can use for controlling typeout windows.

**tv:window-with-typeout-mixin**
**(flavor:method :typeout-window tv:essential-window-with-typeout-mixin)**
**tv:typeout-window**
**tv:typeout-window-with-mouse-sensitive-items**
**tv:temporary-typeout-window**
**tv:with-terminal-io-on-typeout-window**

### Scrolling Windows

Use these flavors to control how scrolling occurs in windows.

**tv:basic-scroll-bar**
**tv:margin-scroll-mixin**
**tv:flashy-scrolling-mixin**

### Frames

The concepts and facilities discussed in this section apply generally to Dynamic Window-based frames created with the Frame-Up Layout Designer and **dw:define-program-framework**. (For an overview of these facilities and references to additional documentation, see the section "Defining Your Own Program Framework".) In particular, the subsections on specifying panes and constraints, specification examples, and frame messages are relevant:

See the section "Specifying Panes and Constraints".

See the section "Examples of Specifications of Panes and Constraints".

See the section "Messages to Frames".

A *frame* is a window that is divided into subwindows, using the hierarchical structure of the window system. The subwindows are called *panes*. The panes are the inferiors of the frame, and the frame is the superior of each pane. Several heavily used systems programs use frames. For example, Inspector windows are frames. The default Inspector window has six panes: the interaction pane on top, the history pane and command menu pane below it, and three Inspect panes below that. The Window Debugger and Zmacs also use frames. In Zmacs, each new editor window is a pane of the Zmacs Frame. Zmail uses frames heavily.

From these examples, you can see some of the things that frames are good for. In general, by using a frame as a user interface to an interactive subsystem, you get a convenient way to put many different things on the screen, each in its own place. Generally you can split up the frame into areas in which you can display text or graphics, areas where you can put menus or other mouse-sensitive input areas, and areas to interact with, in which keyboard input is echoed or otherwise acknowledged.

If you use [Edit Screen] to change the shape of an Inspector or Window Debugger frame, the shapes of the panes are all changed so that the proportions come out looking as they are supposed to. If you play around with [Edit Screen] enough, you can even see the menus reformat themselves (changing their numbers of rows and columns) in order to keep all of their items visible. The way all this works is that the positions and shapes of the panes, instead of being explicitly specified in units of pixels, are specified symbolically. When the window changes shape, the symbolic description is elaborated again in light of the new shape, and the panes are reshaped appropriately.

This set of symbolic descriptions is called a set of constraints, and the kind of frame that implements the constraint mechanism is a flavor called **tv:basic-constraint-frame**. While there are other, more basic frame flavors, you cannot use them alone; you must write a new flavor that includes the more basic frame flavors in its components, and has new methods. Since writing new methods is beyond the scope of this document, we will simply explain how to use constraint frames.

When you make a constraint frame, you specify the configuration of panes within the frame by creating list structure to represent the layout. The format of this list structure is called the constraint language. It lets you say things like "give this pane one third of the remaining room, then give that pane 17 pixels, and then divide what remains between these two panes, evenly." The constraint language is fairly complex. For full details, see the section "Specifying Panes and Constraints". In general, a frame can have many different *configurations*. Each configuration is described in the constraint language, and each specifies one way of splitting up the frame. While the program is running, it can switch a frame from one configuration to another. Some panes may appear in more than one configuration, but other panes may be left out of one configuration, and may only be visible when the frame is switched to another configuration. For example, in Zmail, when you click

on [Mail], the frame changes to a new configuration showing the Headers and Mail panes.

### Flavors for Panes and Frames

To have a frame with panes, you must have a frame, which is a window, and you must have panes, each of which is a window. The flavor of each pane of a frame must have, as one of its components, the flavor **tv:pane-mixin**. Some system facilities provide flavors for you that already have this flavor mixed in. For example, the flavor **tv:command-menu-pane** is a flavor that consists of **tv:command-menu** and **tv:pane-mixin**. (This is the kind of menu most often used in frames; menus are a higher-level facility.) In general, you can take any flavor of window that you might want to use in a pane, and make a new flavor suitable to actually be a pane simply by mixing in **tv:pane-mixin**.

(For information on Dynamic Window-based frames and related facilities, see the section "Defining Your Own Program Framework".)

The flavor of the frame itself might be any of several flavors. The simplest flavor of constraint frame is **tv:constraint-frame**.

Here is a list of flavors for use with panes and frames.

**tv:pane-mixin**
**tv:pane-no-mouse-select-mixin**
**tv:window-pane**
**tv:basic-frame**
**tv:constraint-frame**
**tv:bordered-constraint-frame**

Bordered constraint frames are used most often. Usually, each of the panes has borders, and the frame does too. A reason for this is that when two of the panes are right next to each other, as they usually are, their borders are side by side, and so look like a double-thick line. In order to make the edges of the panes that are at the edge of the frame (rather than up against another pane) look as if they are the same thickness, the frame has a border itself.

It is common in frame-oriented interactive subsystems for all of the panes to use the same I/O buffer. The reason for this is that such subsystems are usually organized as a single process that reads commands and executes them. But with a many-paned frame, there may be many windows (each pane is a window) at which characters might be typed or mouse-clicks might be clicked. When the process is waiting for its next command, it would be inconvenient for it to have to wait for the complex condition that any of these windows has input available in its I/O buffer. Instead, since the command stream is only one serial stream of commands anyway, it is common to have all the panes of a frame share the same I/O buffer.

What happens when many windows share an I/O buffer is that any characters typed at any of them, or any mouse-clicks that generate forced keyboard input, are all put into the same I/O buffer, in the chronological order in which they are generated. The process then does successive **:tyi** stream operations from any pane of

the frame, and it receives anything that has been typed at any pane. When the I/O buffer is shared like this, it doesn't matter which pane is selected: All the characters go to the same place anyway, and the information as to which pane was typed at is lost. However, the forced keyboard input generated by mouse clicks at a facility that is designed to be used as a pane of a frame (**tv:command-menu-pane** for instance) will return all useful and relevant information to the sender of the **:tyi** message, including which pane the mouse was pointing at when it was clicked.

To have all of the panes share the same I/O buffer, use one of the following flavors:

**tv:constraint-frame-with-shared-io-buffer**
**tv:bordered-constraint-frame-with-shared-io-buffer**
**(flavor:method :io-buffer tv:constraint-frame-with-shared-io-buffer)**

### Specifying Panes and Constraints

When you create a constraint frame, you must supply two initialization options. The **:panes** option specifies what panes you want the frame to have, and the **:configurations** option specifies the set of constraints for each of the configurations that the window may assume. For the purposes of these two options, windows are given internal names, which are Lisp symbols, used only by the flavors and methods that deal with constraint frames. These names are not used as the actual names of the windows (as in the **:name** message).

These facilities specify the panes of a constraint frame, and the constraint language.

**(flavor:method :panes tv:basic-constraint-frame)**
**(flavor:method :configurations tv:basic-constraint-frame)**

### :layout Constraint Frame Specification

A configuration is itself a stack. Thus, the symbol that names a configuration must appear in that configuration's **:layout** list as the name of either a row or a column.

A configuration specification includes a list of layout specifications, introduced by the keyword **:layout**. Each layout specification defines one row, column, or fill. (The panes are defined by the **:panes** init option to the frame. See the init option **(flavor:method :panes tv:basic-constraint-frame)**.)

A layout specification for a *row* takes the following form:

(*name* :row *name1 name2*...)

*name* is a symbol, the name of the row. *name1*, *name2*, and so on are symbols, the names of the members of the row. The members are listed in left-to-right order.

A layout specification for a *column* takes the following form:

(*name* `:column` *name1 name2*...)

*name* is a symbol, the name of the column. *name1*, *name2*, and so on are symbols, the names of the members of the column. The members are listed in top-to-bottom order.

A layout specification for a *fill* takes one of the following forms. In each of these *name* is a symbol, the name of the fill.

**(***name* **:fill :white)** The area is filled with zero pixels (normally displayed as white).

**(***name* **:fill :black)** The area is filled with one pixels (normally displayed as black).

**(***name* **:fill** *array***)** The area is filled with the contents of the array, using **bitblt**. You probably want to use backquote (‘) to create the configuration description and insert the array at the appropriate point.

**(***name* **:fill** *symbol***)** The symbol should be the name of a function of six arguments. The function is expected to fill the rectangle that has been allocated to this part of the section with some pattern. The following values are passed to the function:

*constraint-node*
: This is an internal data structure. You should not need to do anything with this argument.

*x-position*
: X-coordinate of the top left corner of the rectangle to be filled.

*y-position*
: Y-coordinate of the top left corner of the rectangle to be filled.

*width* Width in pixels of the rectangle to be filled.

*height* Height in pixels of the rectangle to be filled.

*screen-array*
: This is a two-dimensional array into which the function should write the pattern it wants to put into the window.

**(***name* **:fill** *list***)** This is similar to the case in which *pattern* is a symbol, but it lets you pass extra arguments. The first element of the list is the function to be called, and that function is passed all of the objects in the rest of the list, after the six arguments enumerated above.

### :sizes Constraint Frame Specification

A configuration specification includes a list of size specifications, introduced by the keyword **:sizes**. Each size specification defines how a stack is divided up among its members; it controls the width of each member of a row, or the height of each member of a column. No size specification exists for fills and panes.

A size specification is a list whose first element is the name of the relevant stack. The remaining elements consist of groups of *constraints* separated by the keyword **:then**. The groups are processed sequentially; all the constraints in a group are processed in parallel. Each constraint allocates some of the space available in a stack to a single member of that stack. (This space is width if the stack is a row, height if the stack is a column). After one group has been processed, the amount of space available is decreased by the sum of the space that was allocated, and then the next group is processed. This is the meaning of the parallel versus sequential distinction.

The division of constraints into groups matters when a constraint specifies the size of a member as some fraction of the space available. For example, suppose two constraints each specify that a member is to receive 50% of the available space. If these two constraints are in the same group (processed in parallel) they will allocate 100% of the space. If they are in separate groups (processed sequentially) they will allocate 75% of the space, and the first member will be twice as large as the second member. The first member gets 50% of the total space, then the second member gets 50% of what remains, which is 25% of the total space.

Note that the order of the constraints in a size specification is unrelated to the actual order of the members on the screen, which is controlled solely by the layout specification.

A constraint can take any of several forms. In each case the constraint is a list whose first element is the name of the member (a symbol). Here is a list of constraints.

"Integer Constraint Size Specification"
"Fraction Constraint Size Specification"
"**:even** Constraint Size Specification"
"**:ask** Constraint Size Specification"
"**:ask-window** Constraint Size Specification"
"**:funcall** Constraint Size Specification"
"**:eval** Constraint Size Specification"
"**:limit** Constraint Size Specification"

**Examples of Specifications of Panes and Constraints**

The first set of examples below, 1-6, is meant to give you a feel for the basics of constraint-frame specification. These are followed by two, more complex examples. Additional examples can be found in the files `sys:examples;constraint-frame-language-1.lisp`, `-2.lisp`, and `-3.lisp`. Also, note that for Dynamic Window-based frames you can use the Frame-Up Layout Designer to help with the initial specification of a variety of layouts. (See the section "Defining Your Own Program Framework".)

**Example 1**

```
;;; Two windows of equal size, one on top of the other.

(defflavor cframe1 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes `((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))

:configurations
          `((config1 (:layout (config1 :column pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))
```

**Example 2**

```
;;; Two windows of equal size, side by side

(defflavor cframe2 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes `((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))

:configurations
          `((config1 (:layout (config1 :row pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))
```

**Example 3**

```
;;; Here we have created a constraint frame with two
;;; possible configurations.  You can switch between these
;;; configurations at run time.
```

```
(defflavor cframe3 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label"))

:configurations
          '((config1 (:layout (config1 :column pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :even) (pane-2 :even))))
            (config2 (:layout (config2 :row pane-1 pane-2))
                     (:sizes  (config2 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))

;;; Before going on with more complex constraint frames,
;;; you have to know how to access the various panes of a
;;; constraint frame and how to tell the window to change
;;; to a different configuration.  Notice what happens to
;;; the circle drawn in Pane-2.
(defvar *win* (tv:make-window 'cframe3))

(defun one ()
  (let ((first-pane (send *win* :get-pane 'pane-1))
        (second-pane (send *win* :get-pane 'pane-2)))
    (send *win* :set-configuration 'config1)
    (send *win* :expose)
    (send *win* :send-all-panes :clear-window)
    (send first-pane :draw-circle 500 100 20)
    (send second-pane :draw-circle 600 100 20)
    (sleep 2)
    (send *win* :set-configuration 'config2)))
```

**Example 4**

```
;;; Now lets try organizing the panes in interesting patterns.
;;; For this we will need additional panes.  Also notice in
;;; config3 that you don't always have to use all the panes
;;; defined.
```

```
(defflavor cframe4 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes `((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label")
            (pane-3 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-3 label")
            (pane-4 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-4 label"))

:configurations
          `((config1 (:layout (config1 :column pane-1 row-panes)
                              (row-panes :row pane-2 pane-3 pane-4))
                     (:sizes  (row-panes (pane-2 :even)
                                (pane-3 :even) (pane-4 :even))
                              (config1 (pane-1 :even)
                                       (row-panes :even))))
            (config2 (:layout (config2 :row pane-1 column-panes)
                              (column-panes :column pane-2 pane-3 pane-4))
                     (:sizes  (column-panes (pane-2 :even)
                                (pane-3 :even) (pane-4 :even))
                              (config2 (pane-1 :even)
                                       (column-panes :even))))
            (config3 (:layout (config3 :row pane-1 pane-2 pane-4))
                     (:sizes  (config3 (pane-1 :even)
                                (pane-2 :even) (pane-4 :even)))))
   :configuration 'config1))
;
;;; Display all configurations of any constraint frame that is
;;; the value of *WIN*.  This function is useful for Examples 4,
;;; 5, and 6.
;
(defvar *win* (tv:make-window 'cframe4))
```

```
(defun display-all-configs ()
  (let ((configurations (loop for thing in (send *win* :constraints)
                              collect (car thing))))
    (loop for config in configurations
          do
      (print config)
      (sleep 2)
      (send *win* :set-configuration config)
      (send *win* :expose)
      (sleep 3)
      (send *win* :bury))))
```

**Example 5**

```
;;; Now we turn our attention to controlling the sizes of
;;; the panes.  In place of :EVEN, if you put a integer it
;;; will allocate that many pixels.  If you put a fraction
;;; 0.2 the pane will be allocated that percent of the room
;;; remaining to be allocated.  A size value of 10 :lines
;;; or 20 :characters will create a pane large enough to hold
;;; them.
(defflavor cframe5 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes `((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label")
            (pane-3 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-3 label")
            (pane-4 tv:window-pane
                    :blinker-p nil
```

```
              :save-bits t:configurations
          `((config1 (:layout (config1 :row pane-1 pane-2 pane-3))
                     (:sizes  (config1 (pane-1 200)
                                       :then (pane-2 200)
                                       :then (pane-3 :even))))

            (config2 (:layout (config2 :row pane-1 pane-2 pane-3))
                     (:sizes  (config2 (pane-1 0.5)
                                       :then (pane-2 0.5)
                                       :then (pane-3 :even))))

            (config3 (:layout (config3 :row pane-1 pane-2 pane-3))
                     (:sizes  (config3 (pane-1 50 :characters)
                                       :then (pane-2 50 :characters)
                                       :then (pane-3 :even))))
            (config4 (:layout (config4 :column pane-1 pane-2 pane-3 pane-4))
                     (:sizes  (config4 (pane-1 200)
                                       :then (pane-2 0.5)
                                       :then (pane-3 5 :lines)
                                       :then (pane-4 :even)))))
   :configuration 'config1))
```

**Example 6**

```
;;; You might want to try reshaping the window by clicking on
;;; the system menu choice "Edit Screen" and then on "Reshape".
;;; You should notice two things:
;;;   1) The panes adjust themselves to fit into the space given.
;;;   2) If you make the window small enough the panes will
;;;      become uselessly small.
;;;
;;; To give your constraint frame size limits try the following:
(defflavor cframe6 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes `((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))

:configurations
          `((config1 (:layout (config1 :column pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :limit (5 10 :lines) :even)
                                       (pane-2 :even)))))
   :configuration 'config1))
```

Following are two examples of configuration definitions, slightly edited from the system source code.

**Example 7**

```
;;;Here is how the Font Editor (FED) used to specify its
;;;standard configuration.  This code is extracted from a
;;;source file with package zl:fed and base 8.(defmethod (fed :before :init) (init-plist)
  ...
  (setf (get init-plist :configurations)
        '((:standard
           (:layout
            (:standard :column character-pane prompt-pane top-section)
            (top-section :row fed-pane other-slab)
            (other-slab :column
                        draw-mode-menu
                        command-menu-1
                        command-menu-2
                        command-menu-3
                        status-pane
                        alphabet-menu
                        param-chvv
                        register-pane))
           (:sizes
            (other-slab (draw-mode-menu :ask :pane-size)
                        :then (command-menu-1 :ask :pane-size)
                        :then (command-menu-2 :ask :pane-size)
                        :then (command-menu-3 :ask :pane-size)
                        :then (status-pane 3 :lines)
                        :then (alphabet-menu :ask :pane-size)
                        :then (param-chvv 5 :lines)
                        :then (register-pane :even))
            (top-section (other-slab :limit (24 144 :characters prompt-pane)
                                     0.3)
                         :then (fed-pane :even))
            (:standard
             (character-pane :ask :wanted-size)
             :then (prompt-pane 4 :lines)
             :then (top-section :even))))
          (:wide ...))))
```

**Example 8**

```
;;;Here is how an early implementation of the
;;;Document Examiner specified its frame configuration.
;;;This code is extracted from a source file with package
```

```
;;;sage and base 10.(defconst *dex-frame-constraints*
          '((main
             (:layout
              (main :column top-part bottom-part)
              (top-part :row title&viewer-pane candidates-and-bookmarks)
              (bottom-part :row command-pane menu-pane)
              (title&viewer-pane :column title-pane viewer-pane)
              (candidates-and-bookmarks :column candidate-pane bookmark-pane))
             (:sizes
              (main (bottom-part 4 :lines command-pane)
                    :then (top-part :even))
              (bottom-part (command-pane 660)
                           :then (menu-pane :even))
              (top-part (title&viewer-pane 660)
                        :then (candidates-and-bookmarks :even))
              (title&viewer-pane (title-pane 0 :lines)  ;label only
                                 :then (viewer-pane :even))
              (candidates-and-bookmarks (candidate-pane 0.5)
                                        :then (bookmark-pane :even))))))(defmethod (:init
x-frame :before) (plist)
  (unless (variable-boundp tv:panes)
    (setq tv:panes *dex-frame-panes*))
  (unless (get plist :configurations)
    (setf (get plist :configurations) *dex-frame-constraints*))
  ...)
```

### Messages to Frames

Here is a list of messages to frames.

**:select-pane**
**:selected-pane**
**(flavor:method :selected-pane tv:basic-constraint-frame)**
**(flavor:method :get-pane tv:basic-constraint-frame)**
**(flavor:method :pane-name tv:basic-constraint-frame)**
**(flavor:method :send-pane tv:basic-constraint-frame)**
**(flavor:method :send-all-panes tv:basic-constraint-frame)**
**(flavor:method :configuration tv:basic-constraint-frame)**
**(flavor:method :configuration tv:basic-constraint-frame)**
**(flavor:method :set-configuration tv:basic-constraint-frame)**
**(flavor:method :constraints tv:basic-constraint-frame)**

For information on select menus and frames, see the message **:name-for-selection**.

### Specifying Panes and Constraints in Non-Dynamic Windows

This section gives the complete rules for specifying the panes of a constraint frame, and for the constraint language, in releases before the introduction of Dynamic Windows. The specification method described in this section is obsolete but supported for compatibility. (The function,**tv:back-convert-constraints** can be used to convert from the earlier specification form to the later.)

When you create a constraint frame, you must supply two initialization options. The **:panes** option specifies what panes you want the frame to have, and the **:constraints** option specifies the set of constraints for each of the configurations that the window may assume.

When you use the following two options, windows are given internal names, which are Lisp symbols, used only by the flavors and methods that deal with constraint frames. These names are not used as the actual names of the windows (as in the **:name** message).

**(flavor:method :panes tv:basic-constraint-frame)**
**(flavor:method :constraints tv:basic-constraint-frame)**

A configuration-description-list is a list of configuration-descriptions. There is one configuration-description in the list for each of the possible configurations that the frame can assume. Each configuration is named by a symbol, called the configuration-name. A configuration-description-list is an alist that associates the configuration-descriptions with the names. It looks like this:

((*configuration-name-1* **.** *configuration-description-1*)
 (*configuration-name-2* **.** *configuration-description-2*)
 ...)

Each configuration-description describes the layout of the panes in a single configuration. The description has two parts. The first part specifies the order in which the windows appear, and the second part specifies how the sizes are computed. Actually, in addition to windows, there can also be *dummies* in the configuration-descriptor. A dummy is used either to hold empty space that is not used by any window, or it can reserve a region of space to be divided up by another configuration-description.

A configuration-description splits up one of the dimensions of a rectangular area into many parts. Such an area is called a *section*. Which of the two dimensions is being split up is determined by the *stacking*. If the stacking is **:vertical** then the section is being split up vertically; that is, the parts are stacked on top of each other. If the stacking is **:horizontal** then the section is being split up horizontally; that is, the parts are side-by-side. The stacking of the top-level configuration-descriptions in the **:constraints** option is always **:vertical**, but there can be more configuration-descriptions nested inside of them, and these can have either stacking.

Each part has a name, represented as a symbol. A part may either hold an actual pane, or it may hold something else; if it holds something else, it is called a *dummy* part. Dummy parts can be further subdivided into more panes and dummies using another constraint-description, or their pixels can be blank or filled with some pattern.

A configuration-description looks like this:

(*ordering* . *description-groups*)

*ordering* is a list of names of panes and of dummies, each represented by a symbol; the order of this list is the order that the panes and dummies appear in the space being split up by the configuration-description. For vertical stacking the list goes top to bottom. For horizontal stacking the list goes left to right. A *description-group* is a list of *descriptions*. Each description describes either exactly one pane or one dummy. A configuration-description must have one description for each element of the *ordering* list.

All of the descriptions in a description-group are processed together ("in parallel"); each of the description-groups is processed in turn, starting with the first one. By grouping the descriptions this way, you can control which constraints are elaborated together and which are elaborated at different times; when two constraints are elaborated at different times you can control which one is elaborated first. The reason that the ordering-list in the configuration-description is separate from the description-groups is so that the order in which the panes and dummies appear in the frame can be independent of the order in which their constraints are elaborated.

Each description describes one pane or one dummy. We'll get back to dummies later. A description that describes a pane looks like this:

(*pane-name* . *constraint*)

*pane-name* is the name of the pane being described; *constraint* is the constraint that describes the pane. We will return later to what descriptions of dummies look like. The constraint will be elaborated, and will yield a size in pixels; this size will be used for the width or height being computed.

Finally we get to constraints themselves. The basic form of a constraint is as follows:

(*key arg-1 arg-2* ...)

*key* may be an integer, a flonum, or one of various keyword symbols. Each type of constraint may take arguments, whose meaning depends on which kind of constraint this argument is passed to.

While descriptions of panes do not have the same format as descriptions of dummies, the same kind of constraints are used in both of them. So all the formats given below may be used inside the descriptions of either panes or dummies.

Any constraint may, optionally, be preceded by a **:limit** clause. If a constraint has a **:limit** clause, the constraint looks like:

(:limit *limit-specification key arg-1 arg-2* ...)

The **:limit** clause lets you set a minimum and a maximum value that will be applied to the size computed by the constraint. If the constraint returns a value smaller than the minimum, then the minimum value will be used; if it returns a value larger than the maximum, then the maximum value will be used. The *limit-specification* is normally a two-element list, whose elements are integers giving the minimum and maximum values in pixels. If the list has a third element, it should

be one of the symbols **:lines** or **:characters**, and it means that the integers are in units of lines or characters, computed by multiplying by the line-height or char-width of the pane. If there is a fourth element, it should be the name of a pane, and that pane's line-height or char-width is used instead of that of the pane being constrained. (If this constraint applies to a dummy instead of a pane, and the third element of the list is present, then the fourth must be present as well, since dummies do not have their own line-height or char-width.)

The following Lisp objects may be used as values of *key* in a constraint. Note: The **:funcall** and **:eval** constraints are rarely used and you probably don't need to worry about them. The other kinds are used frequently.

*integer* This lets you specify the absolute size. The value computed by the constraint is simply this integer. Optionally, an argument may be given: it may be the symbol **:lines** or the symbol **:characters**, meaning that the integer is in units of lines or characters, and should be computed by multiplying by the line-height or char-width of the window. If a second argument is also present, it should be the name of a pane, and that pane's line-height or char-width is used instead of that of the pane being constrained. (If this constraint applies to a dummy instead of a pane, and the first argument is given, then the second must be present as well, since dummies do not have their own line-height or char-width.)

*flonum* This lets you specify that a certain fraction of the remaining space should be taken up by this window. Optionally, an argument may be given: It may be **:lines** or **:characters**, and it means to round down the size of the pane to the nearest multiple of the pane's line-height or char-width. A second argument may be given; it is just like the second argument when *key* is an integer.

The distinction between descriptors in the same group and descriptors in different groups is important when you use this kind of constraint. If you have one descriptor group with two descriptors, both of which requests **0.2** of the remaining space, then both panes will get the same amount of space. However, if you have the same two descriptors but put them in successive descriptor groups, then the first one will get **0.2** of the remaining space, and then the second one will get **0.2** of what remains after the first one was allocated; thus, the second pane will be smaller than the first pane. In other words, the amount of space remaining is recomputed at the end of each descriptor group, but not at the end of each descriptor.

**:even** This constraint has a special restriction: You can only use it for descriptors in the last descriptor group of a configuration. Furthermore, if any of the descriptors in that group use **:even**, then *all* of the descriptors in the group *must* use **:even**. The meaning is that all of the panes in the last descriptor group evenly divide all of the remaining space.

It is usually a good idea to use **:even** for at least one pane in every configuration, so that the entire frame will be taken up by panes that all fit together and extend to the borders of the frame. **:even** is careful to choose exactly the right number of pixels to fill the frame completely, avoiding

roundoff errors that might cause an unsightly line of one or a few extra pixels somewhere.

Remember that just because the **:even**s must be in the last descriptor group does not mean that the panes that they apply to must be at the bottom or right-hand end of the frame! The ordering of the panes in the frame is controlled by the ordering list, not by the order in which the descriptors appear.

**:ask** This constraint lets you ask the window how much space it would like to take up. The format of a constraint using **:ask** is as follows:

(:ask *message-name arg-1 arg-2 ...*)

A message whose name is *message-name* and whose arguments are some extra arguments passed by the constraint mechanism followed by *arg-1*, *arg-2*, and so on, is sent to the pane; its answer says how much space the pane should take up. Note that *arg-1*, and so on, are not forms: They are the values of the arguments themselves (that is, they are not evaluated; if you want to compute them, you must build the constraint language description at run-time, which is usually written using a backquoted list).

The arguments that are actually sent along with the message are the same as the arguments passed when you use the **:funcall** option except that the *constraint-node* is not passed; see below. You don't have to worry about these unless you want to define your own methods to be used by **:ask** constraints, and definition of new methods is generally beyond the scope of this document anyway.

Various different flavors of windows accept some messages suitable for use with **:ask**. By convention, several kinds of windows, such as menus, accept a message called **:pane-size**. For example, the **:pane-size** method for menus figures out how much space in the dimension controlled by the **:ask** constraint is needed to display all the items of the menu, given the amount of space available in the other dimension. No arguments are specified in the constraint. Another useful message, handled by **tv:pane-mixin** (and therefore by *all* panes) is **:square-pane-size** (also with no arguments), which makes the window take up enough room to be square.

**:ask-window**

This constraint is a variation on **:ask**. Its format is:

(:ask *pane-name message-name arg-1 arg-2 ...*)

It works like **:ask** except that the message is sent to the pane named *pane-name* instead of the pane being described. This is primarily used for dummies, when the size of a dummy should be controlled by the needs of a pane inside it.

**:funcall**

This constraint lets you supply a function to be called, which should compute the amount of space to use. The format is:

(:funcall *function arg-1 arg-2 ...*)

The specified *function* is called. It is first passed six arguments from inside the workings of constraint frames, and then the *arg-1*, *arg-2*, and so on, values. The six arguments are:

*constraint-node*
: This is an internal data structure. [Not yet documented; you should not need to look at this anyway.]

*remaining-width*
: The amount of width remaining to be used up at the time this description is elaborated, after all of the panes in previous description groups and all of the earlier panes in this description group are allocated.

*remaining-height*
: Like *remaining-width*, but in the height direction.

*total-width*
: The amount of width remaining to be used up by all of the parts of this description group. This is the amount of room left after all of the panes in previous description groups have been allocated but none of the panes in this description group have been allocated.

*total-height*
: Like *total-width*, but in the height direction.

*stacking*
: Either **:vertical** or **:horizontal**, depending on the current stacking.

**:eval** This is like **:funcall**, but instead of providing a function and arguments, you provide a form. The format is:

(:eval *form*)

The six special values that are passed as arguments when the **:funcall** constraint-type is used can be accessed by *form* as the values of the following special variables:

**tv:**constraint-node****
**tv:**constraint-remaining-width****
**tv:**constraint-remaining-height****
**tv:**constraint-total-width****
**tv:**constraint-total-height****
**tv:**constraint-stacking****

This finishes the discussion of descriptions of panes. Descriptions of dummies are different; they may be in any of several formats, identified by the following keywords:

**:blank** This description is used if you want this part of the section to be filled up with some constant pattern. The format of the description is:

(*dummy-name* :blank *pattern* **.** *constraint*)

The *constraint* is used to figure out the size of the part of the section, in the usual way. *pattern* may be any of the following:

**:white** The part is filled with zeroes.

**:black** The part is filled with the maximum value that the pixels can hold (if the pixels are one bit wide, as on a black-and-white TV, this value is 1).

*an array*
The part is filled with the contents of the array, using the **bitblt** function.

*a symbol*
The symbol should be the name of a function of six arguments. The function is expected to fill up the rectangle that has been allocated to this part of the section with some pattern. The following values are passed to the function:

*constraint-node*
This is an internal data structure. [Not yet documented; you should not need to look at this anyway.]

*x-position*
*y-position*
*width*
*height*
These four arguments tell the function the position and size of the rectangle that it should fill.

*screen-array*
This is a two-dimensional array into which the function should write the pattern it wants to put into the window.

*a list* This is similar to the case in which *pattern* is a symbol, but it lets you pass extra arguments. The first element of the list is the function to be called, and that function is passed all of the objects in the rest of the list, after the six arguments enumerated above.

**:horizontal** or **:vertical**
This description is used if you want to subdivide the part into more panes and dummies, using a configuration-description. If you use **:vertical**, it will be split up with vertical stacking, and if you use **:horizontal**, it will be split up with horizontal stacking. You must use only the opposite kind of stacking from the kind currently happening; that is, successive levels of configuration-description must use alternating kinds of stacking. The format is as follows:

(*dummy-name* `:horizontal` *constraint* **.** *configuration-description*)
or
(*dummy-name* `:vertical` *constraint* **.** *configuration-description*)

*constraint*, as usual, specifies the size of this part; it can be in any of the formats given above. Note that in this format, *constraint* appears as an element of a list rather than as the tail of a list, and so the printed representation of the list will include a pair of parentheses around the constraint. *configuration-description* tells how this part is subdivided into parts of its own.

## Examples of Specifications of Panes and Constraints for Non-Dynamic Windows

This section gives some examples of specifications of panes and constraints in the constraint language used before Dynamic windows. The full description of how to use constraint frames, including the full constraint language, is rather complicated. For complete specifications of the pre-Dynamic windows language, see the section "Specifying Panes and Constraints in Non-Dynamic Windows".

The following form creates a constraint frame with two panes, one on top of the other, each of which takes up half of the frame.

```
(tv:make-window 'tv:constraint-frame
                ':panes
                  '((top-pane tv:window-pane)
                    (bottom-pane tv:window-pane))
                ':constraints
                  '((main . ((top-pane bottom-pane)
                             ((top-pane 0.5))
                             ((bottom-pane :even))))))
```

Two initialization options were given to the **tv:constraint-frame** flavor: the **:panes** option and the **:constraints** option. The meaning of the **:panes** specification is: "This frame is made of the following panes. Call the first one **top-pane**; its flavor is **tv:window**. Call the second one **bottom-pane**; its flavor is **tv:window**". The meaning of the **:constraints** specification is: "There is just one configuration defined for this pane; call it **main**. In this configuration, the panes that appear are, in order from top to bottom, **top-pane** and **bottom-pane**. **top-pane** should use up **0.5** of the room. **bottom-pane** should use up all the rest of the room."

This example demonstrates some more features:

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':panes
    '((graphics-pane tv:window-pane
         :label nil :blinker-p nil)
      (message-pane tv:window-pane
         :label "Message Pane" :blinker-p nil)
      (interaction-pane tv:window-pane))
  ':constraints
    '((main . ((interaction-pane graphics-pane message-pane)
               ((message-pane 4 :lines))
               ((graphics-pane 400))
               ((interaction-pane :even))))))
```

This frame has a border around the edges (because of the flavor of the frame itself), and it has three panes. The panes are given some initialization options themselves. The topmost pane is **interaction-pane**, **graphics-pane** is in the middle, and **message-pane** is on the bottom. **message-pane** is four lines high, **graphics-pane** is **400** pixels high, and **interaction-pane** uses up all remaining space.

Here is a window that has two possible configurations. In the first one, there are three little windows across the top of the frame and a big window beneath them; in the second one, the same big window is at the top of the frame, and underneath it is a strip split between a menu and another window.

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':panes
    '((huey tv:window-pane)
      (dewey tv:window-pane)
      (louie tv:window-pane)
      (main-pane tv:window-pane)
      (random-pane tv:window-pane)
      (menu tv:command-menu-pane
            :item-list ("Foo" "Bar" "Baz")))
  ':constraints
    '((first-config . ((top-strip main-pane)
                       ((top-strip :horizontal (.3)
                                   (huey dewey louie)
                                   ((huey :even)
                                    (dewey :even)
                                    (louie :even))))
                       ((main-pane :even))))
      (second-config . ((main-pane bottom-strip)
                        ((bottom-strip :horizontal (.2)
                                       (random-pane menu)
                                       ((menu :ask :pane-size))
                                       ((random-pane :even))))
                        ((main-pane :even))))))
```

In this example, the frame has two different configurations. When the frame is first created, it will be in the first of the configurations listed, namely **first-config**. In this configuration, the top three-tenths of the frame are split equally, horizontally, between three windows, and the rest of the frame is occupied by **main-pane**. The frame can be switched to a new configuration using the **:set-configuration** message. If we switch it to **second-config**, then **main-frame** will appear on top of a strip one-fifth of the height of the window. This strip will contain a menu on the right that is just wide enough to display the strings in the menu's item list, and another pane using up the rest of the strip. When the configuration of the window is switched, **main-pane** must be reshaped.

Another thing to notice is that the list of items in the menu was present in the **:panes** option, rather than a form to be evaluated. If the list had been in a variable, it would have been necessary to write the **:panes** option using backquote, like this:

```
':panes
   `((huey tv:window-pane)
     (dewey tv:window-pane)
     (louie tv:window-pane)
     (main-pane tv:window-pane)
     (random-pane tv:window-pane)
     (menu tv:command-menu-pane
           :item-list ,the-list-of-items))
```

For an explanation of how to use menus, see the section "Window System Choice Facilities".

Following is the last example, using the **:configurations** init option instead of the **:constraints** option used before Dynamic windows:

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':panes
    '((huey tv:window-pane)
      (dewey tv:window-pane)
      (louie tv:window-pane)
      (main-pane tv:window-pane)
      (random-pane tv:window-pane)
      (menu tv:command-menu-pane
            :item-list ("Foo" "Bar" "Baz")))
  ':configurations
    '((first-config (:layout
                     (first-config :column top-strip main-pane)
                     (top-strip :row huey dewey louie))
                    (:sizes
                     (top-strip (huey :even) (dewey :even) (louie :even))
                     (first-config (top-strip 0.3)
                                   :then (main-pane :even))))
      (second-config (:layout
                      (second-config :column main-pane bottom-strip)
                      (bottom-strip :row random-pane menu))
                     (:sizes
                      (bottom-strip (menu :ask :pane-size)
                                    :then (random-pane :even))
                      (second-config (bottom-strip 0.2)
                                     :then (main-pane :even))))))
```

For a description of the constraint language used in Release 6.0, see the section "Specifying Panes and Constraints".

In this example, the window is divided into two windows, side by side.

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':edges '(100 100 600 600)
  ':panes
    '((left tv:window-pane)
      (right tv:window-pane))
  ':constraints
    '((main . ((whole-thing)
           ((whole-thing :horizontal (:even)
                         (left right)
                         ((left :even)
                          (right :even))))))))
```

This example also points out that constraint frames are windows too, and you can use init options acceptable to **tv:minimum-window** with them. In this case, we give the edges of the frame as a whole, in absolute numbers. Remember that frames are *not* built out of **tv:window**.

**Window Substrate Facilities**

The facilities described in this section represent only that fraction of the window substrate that are specific to dynamic windows and not to static windows. The dictionary entry for **dw:dynamic-window** provides references to other relevant sections. See the flavor **dw:dynamic-window**.

The following table lists the Dynamic Window substrate facilities:

Dynamic Window Facilities
- **dw:dynamic-window**
- **dw:margin-borders**
- **dw:margin-white-borders**
- **dw:margin-whitespace**
- **dw:margin-drop-shadow-borders**
- **dw:margin-ragged-borders**
- **dw:margin-label**
- **dw:margin-scroll-bar**
- **(flavor:method :set-margin-components dw:margin-mixin)**
- **(flavor:method :set-borders dw:margin-mixin)**
- **(flavor:method :set-label dw:margin-mixin)**
- **(flavor:method :delayed-set-label dw:margin-mixin)**
- **(flavor:method :update-label dw:margin-mixin)**
- **dw:set-default-end-of-page-mode**

Dynamic Frame Facilities
- **dw:program-frame**

**dw:dynamic-window** is the basic window flavor in the Dynamic Window substrate. It is the dynamic equivalent of **tv:window**, the basic static window flavor. Unlike **tv:window**, however, **dw:dynamic-window** has built into it a variety of desirable window features. **dw:dynamic-window** also refers to a resource of Dynamic Windows.

The basic Dynamic Window flavor supports an output-history, that is, presentation recording, is scrollable, includes a visible scroll bar, has a label, and is surrounded by a simple, one-pixel-wide border. The last three attributes — the scroll bar, label, and border — are margin components made available via a single mixin flavor, **dw:margin-mixin**.

Most of the remaining Dynamic Window facilities listed in the above table relate to margin components. They provide a set of flavors and methods allowing you to customize the appearance of your program's windows, from a variety of border designs to labels and scroll bars. The following example shows how to make a Dynamic Window with a customized set of margin components:

```
(defun dynamic-window-margin-example ()
  (let ((test (tv:make-window 'dw:dynamic-window
                 :edges-from :mouse
                 :margin-components
                 '((dw:margin-borders :thickness 1)
                   (dw:margin-white-borders :thickness 3)
                   (dw:margin-borders :thickness 10)
                   (dw:margin-white-borders :thickness 8)
                   (dw:margin-borders :thickness 3)
                   (dw:margin-whitespace :margin
                                         :left :thickness 10)
                   (dw:margin-scroll-bar)
                   (dw:margin-whitespace :margin
                                         :bottom :thickness 7)
                   (dw:margin-scroll-bar :margin :bottom)
                   (dw:margin-whitespace :margin :left
                                         :thickness 10)
                   (dw:margin-label :margin :bottom
                                    :style (:sans-serif
                                            :italic :normal))
                   (dw:margin-whitespace :margin :top
                                         :thickness 10)
                   (dw:margin-whitespace :margin :right
                                         :thickness 13))
                 :expose-p t
                 :mouse-blinker-character #\mouse:fat-circle)))
  (send test :set-label "Margin Test Window")))
```

When you create this window and run the mouse cursor over it, you will notice the cursor changing shape. The shape, in this case a "fat circle", is specified via the **:mouse-blinker-character** init option. Other available mouse blinker characters are listed in the section that follows.

Additional Dynamic Window methods are included in the program output category, because of their usefulness in that context. See the section "Controlling Location and Other Aspects of Output".

Dynamic frame facilities considered to be substrate-level are limited to **dw:program-frame**. This is the building-block flavor used by the Frame-Up Layout Designer and **dw:define-program-framework** to create program frames. For an overview of these facilities and some frame functions, see the section "Defining Your Own Program Framework".) Also, as is the case with Dynamic Windows generally, static window system facilities for programming with frames are applicable to dynamic frames as well. See the section "Frames". **dw:program-frame** is also a window resource.

### Mouse-Blinker Characters

Through the **:mouse-blinker-character** init option to **dw:dynamic-window**, the mouse blinker, when moved over a Dynamic Window, can assume any of the shapes available in the mouse font (**fonts:mouse**). To see the glyphs included in this font, use the Show Font Command Processor command on "mouse". Each glyph in the font maps to a unique mouse-blinker character. The following lists these in the order in which they appear in the font:

**#\mouse:Up-Arrow**
**#\mouse:Right-Arrow**
**#\mouse:Down-Arrow**
**#\mouse:Left-Arrow**
**#\mouse:Vertical-Double-Arrow**
**#\mouse:Horizontal-Double-Arrow**
**#\mouse:NW-Arrow**
**#\mouse:Times**
**#\mouse:Fat-Up-Arrow**
**#\mouse:Fat-Right-Arrow**
**#\mouse:Fat-Down-Arrow**
**#\mouse:Fat-Left-Arrow**
**#\mouse:Fat-Double-Vertical-Arrow**
**#\mouse:Fat-Double-Horizontal-Arrow**
**#\mouse:Paragraph**
**#\mouse:NW-Corner**
**#\mouse:SE-Corner**
**#\mouse:Hourglass**
**#\mouse:Circle-Plus**
**#\mouse:Paintbrush**
**#\mouse:Scissors**
**#\mouse:Trident**
**#\mouse:NE-Arrow**
**#\mouse:Circle-Times**
**#\mouse:Big-Triangle**
**#\mouse:Medium-Triangle**
**#\mouse:Small-Triangle**
**#\mouse:Inverse-Up-Arrow**
**#\mouse:Inverse-Down-Arrow**
**#\mouse:Filled-Lozenge**
**#\mouse:Dot**
**#\mouse:Fat-Times**
**#\mouse:Small-Filled-Circle**
**#\mouse:Filled-Circle**
**#\mouse:Fat-Circle**
**#\mouse:Fat-Circle-Minus**
**#\mouse:Fat-Circle-Plus**
**#\mouse:Down-Arrow-To-Bar**
**#\mouse:Short-Down-Arrow**
**#\mouse:Up-Arrow-To-Bar**

**#\mouse:Short-Up-Arrow**
**#\mouse:Boxed-Up-Triangle**
**#\mouse:Boxed-Down-Triangle**

Note that mouse-blinker characters are non-printing; that is, they are intended for on-line use only.

## Window System Choice Facilities

### The Choice Facilities

The Genera window system contains a variety of facilities to allow the user to make choices interactively. These all work by displaying some arrangement of items in a window. By pointing to an item with the mouse and pressing a mouse button, the user selects the item. The choice facilities are implemented in and accessed with the Flavors feature of Lisp.

### Overview of the Choice Facilities

This section is a capsule description of the choice facilities. This should familiarize you with the possibilities, thereby helping you to decide which facility is appropriate to your application, without reading through each detailed description. (For an overview of choice facilities intended for use with Dynamic Windows, see the section "Using Presentation Types for Input".)

### List of Choice Facilities

Here is a brief explanation of each of the choice facilities.

*Pop-up Menus*
: This facility puts a menu with items on the screen. The user is forced to make a choice among the items. (The menu does not disappear until a choice has been made.) See the section "Instantiable Pop-up and Momentary Menus".

*Momentary Menus*
: Momentary menus appear on the screen with a list of choices. The user does not have to make a choice, however. By moving the mouse outside of the menu, the user can make the menu disappear. See the section "Basic and Mixin Pop-up and Momentary Menus".

*Command Menus*
: Command menus are used when you want to pass a command to your own controlling process from a menu. The command is sent to the process via an input buffer that can be shared with other windows or processes. This way, the controlling process can be looking in the buffer for commands from several windows as well as for keyboard input. See the section "Command Menus".

*Dynamic Item List Menus*

A dynamic item list menu is provided for menus whose items change over time. The item list is updated whenever the menu is displayed. Both momentary and pop-up dynamic item list menus are available. See the section "Dynamic Item List Menus".

*Multiple Menus*

Multiple menus are provided for situations in which the user can select *several* items at a time. The selected items are displayed in inverse video. *Special choices* allow the user to specify operations on all the items selected. Both momentary and pop-up multiple menus are available. see the section "Multiple Menus".

*Multiple Menu Choose Menus*

This facility provides for menus with several columns. The user picks one item from each column. Special choices [Do It] and [Abort] are used to execute the choices and and deactivate the menu, respectively. See the section "The Multiple Menu Choose Facility".

*Multiple Choice Menus*

This facility displays a menu in which each item is displayed on a separate line. Each item is associated with several yes/no choices, in *choice boxes*. By pointing to a box and pressing the left mouse button, the user complements the yes/no state of the choice box for that item. Constraints can be imposed among the choices for an item, ensuring, for example, that if one box is selected, the others are automatically deselected. See the section "The Multiple Choice Facility".

*Choose Variable Values*

Each item is associated with a value printed next to it. Many different types of values can be specified, or the programmer can create new types. In operation, users select items and then alter the values associated with the item. See the section "The Choose Variable Values Facility".

*User Options*

The user option facility is based on the choose variable values facility. It is used to keep track of options to a program of the sort that users would want to specify once and then save. The option list can be associated with particular programs. See the section "The User Option Facility".

*Mouse-sensitive Items and Areas*

Mouse-sensitive behavior underlies all of the choice facilities. This mixin facility lets areas of the screen be sensitive to the mouse. Moving the mouse into such an area causes a box to be drawn around it. At this point, clicking the mouse invokes a user-defined operation. See the section "The Mouse-Sensitive Items Facility".

*Margin Choices*

Windows can be augmented with choice boxes in their margins. Choice boxes give the user a few mouse-sensitive points that are independent of anything else in the window. Margin choices can be added to any flavor of window in a modular fashion. See the section "The Margin Choice Facility".

### Standard and Customizable Facilities

From the programmer's viewpoint, there are two ways of invoking the choice facilities.

- *Standard* facilities are provided with a reasonable set of defaults predefined in the system code. They are invoked with a simple function call.

- *Customizable* facilities require you to provide more specifications, but they allow more flexibility in the layout and behavior of the facilities. Customizable facilities are manipulated by the Flavor system, and include instantiable, basic, and mixin flavors.

Many of the documented choice facilities are provided in both standard and customizable forms.

### Choice Facilities Use the Flavor System

The window system and the choice facilities are implemented using the Flavor system in Lisp. When a menu is instantiated, users communicate with it by pressing mouse buttons (sometimes called "mouse-clicking"), or by typing in values. Internally, programs communicate with a menu by sending it a message using the **send** function of Lisp.

Useful *initialization property-list options* (hereafter called *init-plist options*) and *messages* associated with each flavor are specified in this document.

### Combining Choice Facilities

Since the choice facilities are implemented with the Flavor system, many of the behaviors listed previously can be integrated into one menu by means of flavor combination.

For example, one menu might include both of these features:

- Pop-up behavior, meaning that the window does not disappear until a choice has been made.

- Multiple menu behavior, allowing several menu items to be selected.

### Instantiable, Basic, and Mixin Flavors

Each choice facility is based on either an *instantiable,* a *basic,* or a *mixin* flavor. Even the standard choice facilities (invoked by simple Lisp function calls) are based on these flavors.

Instantiable flavors are self-contained objects that are ready to be invoked. Instantiable facilities are built out of the basic and mixin facilities. An example of an instantiable facility is the **tv:momentary-menu** flavor.

Basic flavors (often denoted by the prefix **basic-** in the code) define a whole family of related flavors. Most of the basic flavors are noninstantiable and merely serve as a base on which to build other flavors. An example of a noninstantiable basic facility is the **tv:basic-mouse-sensitive-items** flavor.

Mixin flavors (often denoted by the suffix **-mixin** in the code) define a particular feature of an object. A mixin flavor cannot be instantiated, because it is not a complete object. An example of a mixin flavor is **tv:dynamic-multicolumn-mixin**.

In the descriptions of the different choice facilities that follow, the instantiable flavors will be discussed first, followed by the basic and mixin flavors.

### Modifying the Choice Facilities

Although this document explains how to combine the features of the different choice facilities to suit different applications, it does not tell you how to modify the facilities provided with the system, except in the simplest of ways. In order to change the basic behavior of the choice facilities you will need to read some of the code that implements the facility in question. (For example, you should study window instance variables and internal messages that you might want to put daemons on or redefine.)

### The User's Process and the Mouse Process

An asynchronous process called the *mouse process* handles interaction with the mouse. Some portions of these choice facilities execute in the process that calls them, while other portions execute in the mouse process. For example, when menu items are displayed on the screen and the mouse points to them, a box is drawn around the items. This drawing is performed by the mouse process.

This document does not attempt to explain the details of how the mouse and the window system interact. Indeed, the choice facilities are supposed to shield the user from such details, and they can be used effectively with no knowledge of how they are implemented internally.

However, the cases in which a portion of a facility runs in the mouse process are noted where they occur in this text. Excepting these cases, you can freely use side-effects (both special variables and **throw**), and not worry about errors in your program corrupting the system.

The choice facilities described in this document respond to messages sent by the mouse process. For example, **:mouse-buttons**, **:mouse-click**, and **:mouse-select** are all handled by any flavor built on **tv:menu**.

### Introduction to the Menu Facilities

From the user's point of view, a *menu* is a group of choices, each identified by a word or short phrase. To see an example of a menu, click the right mouse button while pressing the `SHIFT` key in a Lisp Listener; this should cause the System

menu to appear (Figure !).

**The System Menu**

| *Windows* | *This window* | *Programs* |
|---|---|---|
| Create | Move | Lisp |
| Select | Shape | Edit |
| Split Screen | Expand | Inspect |
| Layouts | Hardcopy | Mail |
| Edit Screen | Refresh | Trace |
| Set Mouse Screen | Bury | Emergency Break |
| | Kill | Frame-Up |
| | Reset | Namespace Editor |
| | Arrest | Hardcopy |
| | Un-Arrest | File System |
| | Attributes | Document Examiner |

Figure 57. System menu.

You can select one of the choices by moving the mouse near it, which causes it to be highlighted (a box appears around it), and then clicking any mouse button. What happens when you select one of the choices depends on the particular type of menu. Typically the choices in a menu might be commands to some program or choices on which a command should operate.

The window system software automatically chooses the arrangement of the choices and the size and shape of the window. Naturally, there are ways for programmers to control these parameters if they desire. See the section "Init-plist Options for **tv:menu**".

The inverse-video *mouse documentation line* is provided near the bottom of the screen in order to convey the meaning of the mouse buttons at a given time. For example, in the System menu, with the mouse positioned over the "Create" item, the mouse documentation line normally displays the following text:

```
Create a new window.  Flavor of window selected from a menu.
```

The abbreviations L, M, and R stand for the left, middle, and right mouse buttons, respectively. The numeral 2 indicates a quick double click of the mouse button. (Note that the "double-click" effect can also be obtained by clicking once on the mouse while holding down the `SHIFT` key. )

## Components of a Menu

It is important to understand the terminology for describing a menu. The components of a menu are shown in Figure !.

## Menu Items

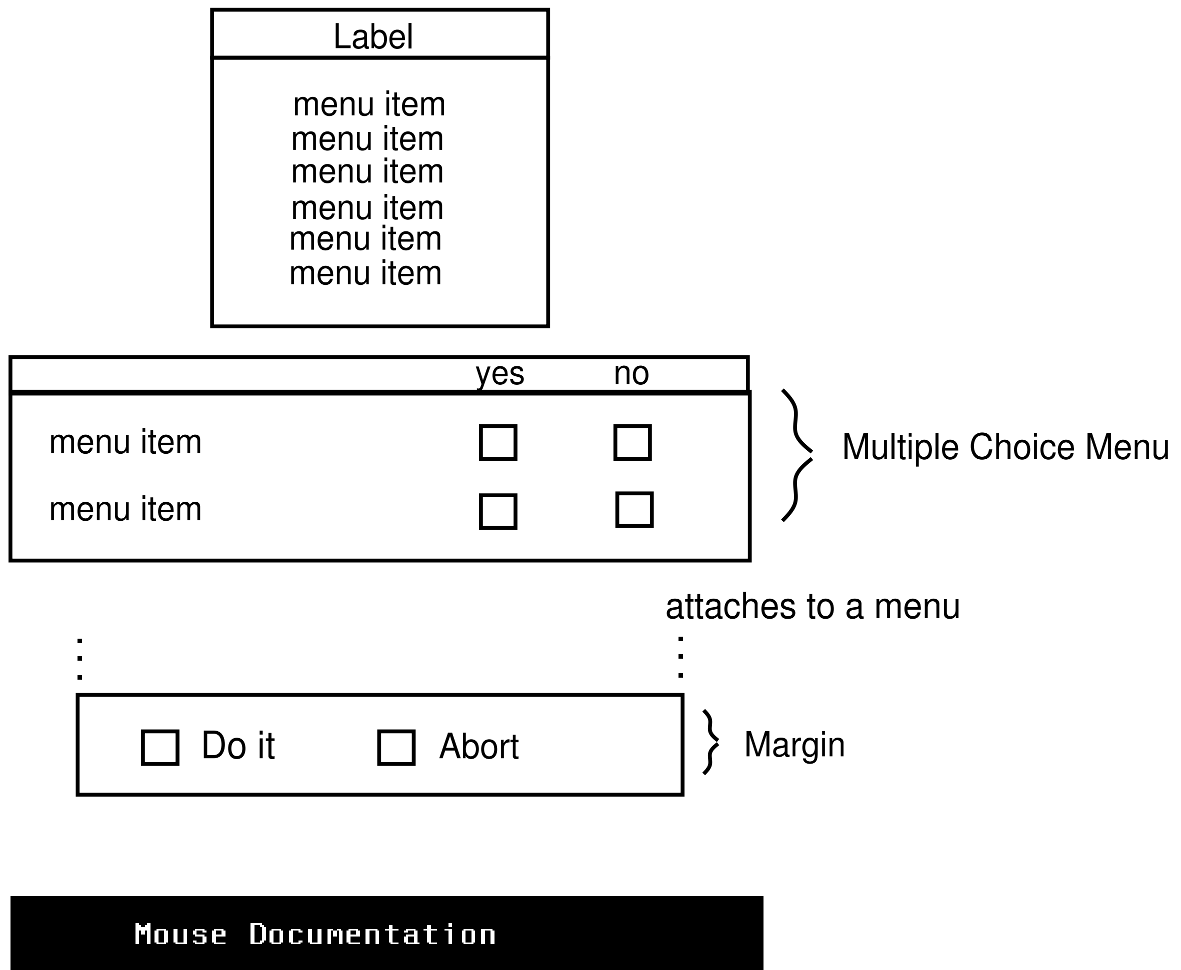



Figure 58. Components of a menu.

From the viewpoint of the programmer, a menu has a list of *items;* each item represents one of the displayed choices. The user chooses an item, and then the program executes it.

An item, then, has three parts:

- A representation in the item list
- A displayed representation
- A specified action when it is executed; this can include a value (or values) to return as well as side-effects

### The Form of a Menu Item

Generally speaking, a menu item can take any of several forms, noted in the list below. In practice, you find these forms in the specification of particular item types, described in the next section.

a string or a symbol
: The string or symbol is both what is displayed and what is returned. There are no side-effects when the item is executed. (Note: **nil** is not a valid menu item.)

a cons
: This is like an alist entry. The **car** is a string or symbol to display and the **cdr** is what to return. The **cdr** must be atomic to distinguish this case from the remaining ones. There are no side-effects.

a list **(***name value***)**
: Another form of alist entry. *name* is a string or a symbol to display, and *value* is any arbitrary object to return. There are no side-effects when the item is executed.

a list **(***name type arg option1 arg1 option2 arg2...***)**
: This is the "general list" form, described in more detail below. *name* is a string or a symbol to display. *type* is a keyword symbol specifying what to do when the item is executed, and *arg* is an argument to it. The *options* are keyword symbols specifying additional features desired, and the *args* following them are arguments to those options.

### Types of Menu Items

Each menu item is an instance of a particular *type*. In most menus, you may not want to explicitly specify the type of the menu item. This is because in simple menus all the menu items are of the same type. Your code (which processes the selected items) presumably knows this type.

It is possible to specify the type of the menu items, however. This provides another dimension of flexibility in menu design. Since items of different types can be intermingled in a single menu, selecting different items can generate a variety of interesting effects. For example, some items can return a value, while others can generate new menus or perform other computations.

### The "General List" Form of Item

To specify the type of an item, use the "general list" form of item.

*(name type arg option1 arg1 option2 arg2 ...)*

As described, an *arg* (argument) field follows each type specification. The predefined types of menu items and the meaning of their arguments are listed here.

**:value** *arg* is what to return when the item is executed. There are no side-effects.

**:eval** *arg* is a form to be evaluated in null environment. Its value is returned.

**:funcall**
*arg* is a function of no arguments to be called. Its value is returned.

**:funcall-with-self**
Like the **:funcall** item type, **:funcall-with-self** calls a function. However, the specified function is called with one argument: **self**, that is, the menu itself.

An example demonstrates its use:

```
;;; Specify the item list
...
;;; Specify the :funcall-with-self item
("Option 1" :funcall-with-self do-option-1)
...
(defun do-option-1 (menu)
  ;; send the :option-1 message
  (send menu ':option-1))
```

**:no-select**
This item cannot be selected. Moving the mouse near it does *not* cause it to be highlighted. This is useful for putting comments, headings, and blank spaces into menus. *arg* is ignored, but it must be present for syntactic consistency.

**:kbd** *arg* is sent to the selected window via the **:force-kbd-input** message. Typically it is either a character code that is to be treated as if it were typed in from the keyboard, or a list that is a command to the program. It is almost always preferable to use a command menu rather than **:kbd** menu items. See the section "Command Menus".

**:menu** *arg* is a new menu to choose from; it is sent a **:choose** message and the result is returned. Normally *arg* would be a momentary menu. If *arg* is a symbol it gets evaluated.

**:buttons**
*arg* is a list of three menu items. The item actually chosen (that is, the item to be executed) is one of these three, depending on which mouse button was clicked. The order in the list is **(***left middle right***)**.

**:window-op**

*arg* is a function of one argument. The argument is a list of three elements: the window the mouse was in before this menu was exposed and the X and Y coordinates of the mouse at that time. For a description of the **tv:window-hacking-menu-mixin**: See the section "Basic and Mixin Pop-up and Momentary Menus". This type is not useful unless the **tv:window-hacking-menu-mixin** is present in the window flavor.

### Menu Item Options

Menu item options follow the arguments in the "general list" form of item. They have two purposes:

- Specifying the character style of a menu item
- Specifying the *mouse line documentation string* associated with an item

The menu item option keywords are as follows:

**:character-style**

This keyword is followed by a character style specification. The item is displayed in that character style, merged against the menu's default character style.

The **:character-style** option is for use with static-window-based menu facilities. For **dw:menu-choose** menu items and the **alist-member** presentation type, use the **:style** option instead.

**:style**

This keyword is followed by a character style specification. The item is displayed in that character style, merged against the menu's default character style.

The **:style** option is for use with Dynamic Window-based menu facilities, **dw:menu-choose** and **alist-member** in particular. For static-window-based facilities, use the **:character-style** option instead.

**:documentation**

This keyword is followed by a string that briefly describes this menu item. When the mouse is pointing at this item, so that it is highlighted, the documentation string is displayed in the documentation line at the bottom of the screen. It is considered good practice to include documentation for all menu items.

An example of the use of menu item options is shown here:

```
("Item 2" :value 1.5 :documentation "Costs $1.50"
          :character-style (nil :bold nil))
```

The character style of the displayed item will be of the same family and size as the default character style for the menu, but its face will be bold.

### Choosing and Executing

After an item has been chosen, it is *executed.* Executing a menu item does what the item type tells it to do. Depending on the type of item being executed, executing produces a value, performs a side-effect, or both.

Execution always takes place in the user process (rather than the mouse process). Thus, execution can depend on the special-variable environment and can perform actions that take a long time, interact with the user, or depend on being able to use the mouse.

The responsibility for executing the chosen menu item rests with either the system or the programmer, depending on how the menu is used. The **tv:menu-choose** function and the **:choose** message execute the chosen item and return its *value*, or they return **nil** if no item was chosen. When using command menus the chosen *item* is returned to the user program. See the section "Command Menus". The user program can execute it by sending the **:execute** message. See the section "Useful **tv:menu** Messages".

The importance of executing menu items depends on the function of the menu. Some menus contain items that act as "nouns". The user simply chooses one out of a group of similar items. In this case, executing the item serves only to translate from the item list. The item list contains the printed representation displayed in the menu and the documentation displayed in the mouse documentation line. For this kind of item, the **:value** item type is often used.

Other menus contain items that act more like "verbs". The program operating the menu might not be aware of the details of each item; it simply allows the user to choose one and then executes it. In this case, most of the complicated behavior is within the menu item. Typically, the **:eval** or **:funcall** item type is used.

### The Geometry of a Menu

A menu has a *geometry* that controls its size, its shape, and the arrangement of displayed choices. The creator of a menu may specify some aspects of the geometry explicitly, leaving other aspects to be given by the system according to default specifications.

There are two ways the choices can be displayed. They can be shown in an array of rows and columns, or they can be in a "filled" format with as many to a line as will reasonably fit. Filled format is specified by giving zero as the number of columns.

The geometry of a menu is represented by a list of six elements:

*columns*
: The number of columns (0 for filled format).

*rows*
: The number of rows.

*inside width*
: The *inside width* of the window, in units of the screen (pixels). If you set the size or edges of the window the inside width is remembered here and acts as a constraint on the menu afterwards.

*inside height*
: The *inside height* of the window, in pixels. If you set the size or edges of the window the inside height is remembered here and acts as a constraint on the menu afterwards.

*maximum width*
: The maximum (inside) width of a window, in pixels. The window system prefers to choose a tall skinny shape rather than exceed this limit.

*maximum height*

The maximum (inside) height of a window, in pixels. The system prefers to choose a short fat shape rather than exceed this limit. If both the maximum width and the maximum height are reached, the system displays only some of the menu items and enables scrolling to make the rest accessible.

Values of **nil** for parts of the geometry can be specified to leave that part unconstrained.

### Geometry Init-plist Options

The init-plist options listed here initialize the geometry of any menu built on the **tv:menu** flavor.

**(flavor:method :geometry tv:menu)**
**(flavor:method :rows tv:menu)**
**(flavor:method :columns tv:menu)**
**(flavor:method :fill-p tv:menu)**

### Geometry Messages

These messages may be sent to any flavor of menu to access and manipulate its geometry:

**(flavor:method :geometry tv:menu)**
**(flavor:method :current-geometry tv:menu)**
**(flavor:method :set-geometry tv:menu)**
**(flavor:method :fill-p tv:menu)**
**(flavor:method :set-fill-p tv:menu)**

Note that the messages **:set-default-character-style** and **:set-item-list** (which do what they say) also cause the geometry of a menu to be recomputed.

### Geometry Example 1: A Multicolumned menu

It is not necessary to explicitly specify all six values for the geometry list. In the following example, only the *columns* value is supplied, and a one-column menu is specified. The rest of the geometry values are computed by using the column value to constrain the system-default settings.

```
(setq geometry-list (list 1))
```

Figures !a and !b show the result of setting the geometry of a menu first to a one-column form (a), then a multicolumn format (b, using the three-column code example below). In the example, the variable **result** holds the value of the item selected by the mouse.
The code used to generate Figure 59b is next.

Figure 59. Adjusting a menu's column geometry. (a) One column (b) Three columns

```
;;; Geometry Example 1

;;; First element in the geometry list specifies three columns
(setq geometry-list (list 3))

;;; Make the menu
(setq my-menu (tv:make-window 'tv:momentary-menu
        ':label '(:string " Selection"
                  :character-style (:swiss :bold :normal))
        ':geometry geometry-list
        ':borders 3
        ':item-list '(("First"  :value 100)
                      ("Second" :value 200)
                      ("Third"  :value 300)
                      ("Fourth" :value 400)
                      ("Fifth"  :value 500)
                      ("Sixth"  :value 600)
                      ("None"   :value   0))))
;;; Expose the window, make a choice,
;;;   and leave the value in the variable "result"
(setq result (send my-menu ':choose))
```

### Geometry Example 2: Retrieving Geometry Information

Figure ! is an example of a simple menu from which we would like to retrieve geometry information.

The code that produced Figure 60 uses the **:current-geometry** message, which retrieves a description of a menu's geometry. Border and character-style specifications are used to customize the menu. (A list of the loaded screen fonts is accessible by using List Fonts (`m-X`) in the Zmacs editor.)

Figure 60. Simple menu from which geometry information is obtained.

```
;;; Menu Geometry Example 2

;;; z is an instance of a momentary window created
;;;  by the make-window function
(setq z  (tv:make-window 'tv:momentary-menu
           ':borders 6
           ':label '(:string " Select Color of Issue "
                     :character-style (:swiss :italic :normal))))

;;; item-list is a list of menu items
(setq item-list '("Blue" "Red" "Yellow" "Green" "Orange"))

;;; This sends a message to set up an item list
(send z ':set-item-list item-list)
```

The next expression interrogates the menu and returns a list that describes its geometry. The list is returned in **geometry-facts.** (Nothing in particular is done with **geometry-facts** in this example).

```
(setq geometry-facts (send z ':current-geometry))
```

The final expression exposes the menu, allows a choice to be made, and returns the selected string in the variable **result**.

```
(setq result (send z ':choose))
```

### Momentary and Pop-up Menus

A momentary menu appears on the screen with a list of items. The user does not have to make a choice, however. By moving the mouse outside the menu, the menu is made to disappear.

By contrast, a pop-up menu forces the user to make a choice. The menu does not disappear until an item has been selected.

### The Standard Momentary Menu Interface

The standard form of a choice facility provides a simple function-call mechanism for invoking it without specifying its details. The standard momentary menu inter-

face is based on this function:

**tv:menu-choose**

### Standard Momentary Menu Example

The following code is an example of how to instantiate a simple momentary menu. Once the menu pops up, the user can make a choice with the mouse. (Any mouse button selects the chosen item.) The *item-list* is a list of menu items in the "general list" form. The **price** variable is set to the value of the selected item, specified in the item list.

```
(setq item-list
  '(("Meat and potatoes" :value 3.49 :documentation "Costs $3.49")
    ("Fish and chips" :value 3.79 :documentation "Costs $3.79")
    ("Hash" :value 1.49 :documentation "Costs $1.49")
    ("Chicken stew" :value 2.99 :documentation "Costs $2.99")))
(setq price (tv:menu-choose item-list "F & T Diner "))
```

### The tv:mouse-y-or-n-p Facility

One of the simplest choice facilities in the system is based on the **tv:menu-choose** function. This function is useful for quick yes-or-no queries in a user interface:

**tv:mouse-y-or-n-p**

### Basic and Mixin Pop-up and Momentary Menus

The *basic* and *mixin* flavors for ordinary kinds of menus are listed in this section. They cannot be instantiated themselves but they are the building blocks of the instantiable menus.

**tv:basic-menu**
**tv:basic-momentary-menu**
**tv:window-hacking-menu-mixin**

### Instantiable Pop-up and Momentary Menus

The instantiable menu flavors are listed below. Two of the most important menu flavors are **tv:menu** and **tv:momentary-menu** , since many other menu flavors are built on them. For a diagram of the flavor network on which **tv:menu** and **tv:momentary-menu** are built, see the section "The Flavor Network of **tv:menu**". For an enumeration of many of **tv:menu**'s init-plist options and messages, see the section "Init-plist Options for **tv:menu**" , and see the section "Messages Accepted

by **tv:menu**".

**tv:menu**
**tv:momentary-menu**
**tv:pop-up-menu**
**tv:momentary-window-hacking-menu**
**tv:momentary-menu**

### Useful tv:menu Init-plist Options

This is a list of some of the most frequently used init-plist options for the **tv:menu** flavor and menu flavors built on it, such as **tv:momentary-menu** and **tv:pop-up-menu**. For a list of more window-related init-plist options associated with any flavor built on **tv:menu**, see the section "Init-plist Options for **tv:menu**".

**(flavor:method :borders tv:menu)**
**(flavor:method :default-character-style tv:menu)**
**(flavor:method :item-list tv:menu)**
**(flavor:method :label tv:menu)**
**(flavor:method :vsp tv:menu)**

See the section "Geometry Init-plist Options".

### Useful tv:menu Messages

This is a list of some useful window and menu-related messages associated with the **tv:menu** flavor and any flavor built on it. For a list of more window-related messages to **tv:menu**, see the section "Messages Accepted by **tv:menu**".

**(flavor:method :choose tv:menu)**
**(flavor:method :execute tv:menu)**
**(flavor:method :deactivate tv:menu)**

### tv:momentary-menu Example 1: Simple Momentary Menu

An example of a simple momentary window with three items in it from which to select is shown in Figure !.

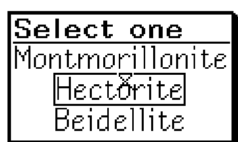



Figure 61. Momentary menu example.

The code to produce such a menu is given next. (In the example, there are no actions specified when an item is selected.)

```
;;; Momentary Menu Example 1

;;; z is an instance of a momentary menu created by the
;;;  make-window function
(setq z (tv:make-window 'tv:momentary-menu
            ':label '(:string "Select one"
                      :style (:swiss :bold :normal))))

;;; item-list is a list of menu items
(setq item-list '("Montmorillonite" "Hectorite" "Beidellite"))

;;; This passes a message to set up an item list
(send z  ':set-item-list item-list)

;;; The :choose message exposes the window and allows the mouse
;;;  to select an item.  choice holds the result.
(setq choice (send z ':choose))
```

### tv:momentary-menu Example 2: Item List as Init-plist Option

Another way to set up the item list is to specify it as an init-plist option.

```
;;; Example 2
;;; Shows use of the init-plist to specify items

(setq z (tv:make-window 'tv:momentary-menu
                ':label " New Selection "
                ':item-list '("First" "Second" "Third")))

(setq choice (send z ':choose))
```

### tv:momentary-menu Example 3: Centered Label and Use of General List Items

In Example 3, two new principles are shown. First, in order to have a centered label for the menu, the new flavor **momentary-menu-with-centered-label** is created.

Second, the "general list" form of item list is used. See the section "The "General List" Form of Item". This allows your program to invoke an operation or return a value when an item is selected. In the example, the variable *choice* is set to **nil** or one of the numbers 1.0, 2.0, or 3.0, depending upon the action taken by the user.

The **:documentation** option keyword has the following effect. When an item with the **:documentation** keyword is pointed at by the mouse, the specified documentation string appears in the inverse-video mouse documentation line at the bottom of the screen.

```
;;; Example 3
;;;  Shows use of flavor mixing and "general list" menu items

;;; Define a flavor with the centered-label-mixin
(defflavor momentary-menu-with-centered-label ()
           (tv:centered-label-mixin tv:momentary-menu))

;;; Create an instance of the window
(setq z (tv:make-window
           'momentary-menu-with-centered-label
           ':label "Selection"
           ':item-list '(("Orange" :value 1.0
                            :documentation "Select orange.")
                          ("Red" :value 2.0
                            :documentation "Select red.")
                          ("Yellow" :value 3.0
                            :documentation "Select yellow."))))

(setq choice (send z ':choose))
```

**tv:momentary-menu Example 4: Using the Mouse Buttons**

The general list form can include choices that depend on the three mouse buttons. **:buttons** is a menu item type that takes three arguments (*left middle right*), each of which specifies what to do if a particular button is pressed. If any argument to **:buttons** is **nil ,** a click on that button is ignored. See the section "Types of Menu Items". An example demonstrates its use.

```
;;; Example 4, shows the use of different mouse buttons

;;; Specify the item list
(setq button-list
      '(("One Item, Three Ways"
         :buttons ((l :eval (print "left"))
                   (m :eval (print "middle"))
                   (r :eval (print "right")))
         :documentation
         "L: Print left, M: Print middle, R: Print right")))

;;; Make the menu
(setq menu-1 (tv:make-window 'tv:momentary-menu
                             ':label "Test Buttons"
                             ':item-list button-list))

;;; Expose the window and choose
(setq choice (send menu-1 ':choose))
```

**tv:pop-up-menu Example**

Since a pop-up menu does not operate as automatically as a momentary menu, it requires a slightly different treatment. The normal mode of operation is to allow **:choose** to activate and expose it, and then send it a **:deactivate** message when done. This does not "destroy" the menu, it just makes sure it does not appear on the screen.

Figure ! shows a simple example of a pop-up menu. We use the "general list" form of item to invoke a function that exposes a second menu and stores the results of the two selections in the variables **drink** and **price.**

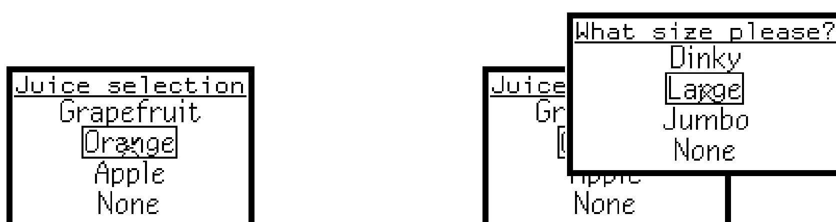



Figure 62. Pop-up menu example.

The code that generated Figure 62 follows.

```
;;; Pop-up menu example

(defvar drink nil)
(defvar grapefruit "Grapefruit Juice")
(defvar orange "Orange Juice")
(defvar apple "Apple Juice")

;;; This function dispatches according to the kind of
;;;  juice selected, and calls the second menu
(defun juice (fruit)
  (selectq fruit
    (gr (setq drink grapefruit))
    (oj (setq drink orange))
    (ap (setq drink apple)))
  (setq price (send two ':choose)))

;;; This function handles the no-juice item
(defun no-juice ()
  (setq drink nil))
```

```
;;; This the first menu, a pop-up menu that allows the user
;;;  to select a juice
(setq one (tv:make-window
             'tv:pop-up-menu
             ':label "Juice selection"
             ':borders 3
             ':item-list '(("Grapefruit" :eval (juice 'gr))
                           ("Orange" :eval (juice 'oj))
                           ("Apple" :eval (juice 'ap))
                           ("None" :funcall no-juice))))

;;; This is the second menu, a momentary menu that allows the user
;;;  to select a size of drink
(setq two (tv:make-window
             'tv:momentary-menu
             ':label "What size please?"
             ':borders 3
             ':item-list
                '(("Dinky" :value  .5
                  :documentation "Smallest size costs 50 cents.")
                  ("Large" :value 1.0
                  :documentation "Actually medium size, costs $1.")
                  ("Jumbo" :value 1.5
                  :documentation "Big, costs $1.50.")
                  ("None"  :value 0
                  :documentation "Cheapest selection by far."))))

;;; Operate the menu; explicit exposing and
;;;  deactivating are necessary for pop-up menus
(defun operate ()
  (send one ':expose-near '(:mouse))
  (send one ':choose)
  (send one ':deactivate))

;;; Invoke the juice selection menu
(operate)
```

Another way to implement this example would have been to use the **:menu** item type to invoke the second menu. See the section "Types of Menu Items".

## Command Menus

Command menus are used when a menu does not stand alone but is part of a frame of several window panes, which can include other menus. The entire frame is controlled by a single process; each frame sends *commands* (or *blips*) to the con-

trolling process from a menu. (For Dynamic Window-based frames, various high-level facilities are available for creating command menus: See the section "Managing the Command Processor".)

In order to understand the operation of a command menu, it is necessary to understand the difference between a menu item and a menu item's value.

### Menu Items and Menu Values

A menu item consists of a list supplied by the programmer in the item list of a menu specification. In most menus, your program rarely receives menu items back from the window system; usually the *values* of the items are returned. There are two exceptions to this situation:

- Certain messages deal explicitly with items, such as the **:item-list** message, which returns the list of items associated with a menu.

- In command menus, your program receives a command (or blip) back from the window system. The blip contains an entire item as well as other information (explained in the next section). You send the **:execute** message to the menu to extract the item's value and perform side-effects.

### Command Blips

Since the **:choose** message (which gets a value and not an item) does not operate on a command menu, the command is sent to the user process through an *I/O buffer* associated with the menu. (Many windows have an I/O buffer associated with them. See the section "Overview of Window Flavors and Messages".) Your controlling process can be looking in its I/O buffer for commands from several windows as well as for keyboard input.

The command chosen by the user is sent to the I/O buffer as a list in the following form:

(**:menu** *chosen-item button-mask window*)

Note: The button-mask is a bit mask with a bit for each button on the mouse. This provides the option of taking different actions depending on which mouse button was pressed. The bit assignments are as follows:

1 Left button
2 Middle button
4 Right button

### Responsibilities of Your Program

Your program is responsible for performing each of the actions that the **:choose** message would normally do, including the following:

- Deciding where to put the menu. Usually this is specified in the definition of the frame, via **:panes** and **:constraints** specifications in a **tv:bordered-constraint-frame-with-shared-io-buffer** flavor.
- Exposing the menu. Usually the command menu is part of a frame and the entire frame is exposed.
- Receiving a choice from the mouse. This is received via an I/O operation like the **:any-tyi** message.
- Executing the choice. Example: (**send** *window* **':execute** *chosen-item*)
- Deciding whether to deactivate the frame. This is not normally performed on an individual command menu pane.

### Command Menu Mixins

Here is a list of command menu mixin flavors.

**tv:command-menu-mixin**
**tv:command-menu-abort-on-deexpose-mixin**

### Instantiable Command Menus

Here is a list of instantiable command menu flavors.

**tv:command-menu**
**tv:command-menu-pane**

### tv:command-menu Init-plist Options

This is an init option to the tv:command-menu

**(flavor:method :io-buffer tv:command-menu)**

Note: By making a command-menu to be a pane in a a **tv:bordered-constraint-frame-with-shared-io-buffer**, you are supplied with an I/O buffer automatically. The frame puts an **:io-buffer** option into the init-plist of each pane. See the section "Frames".

### tv:command-menu Messages

Here is a list of tv:command-menu messages.

**(flavor:method :io-buffer tv:command-menu)**

**(flavor:method :set-io-buffer tv:command-menu)**

### tv:command-menu Example

Figure ! shows a simple command menu. The top pane contains a command menu that allows the user to draw an object on the screen. The middle pane is the drawing surface. The bottom pane is another command menu that allows the user to refresh the drawing surface or exit.

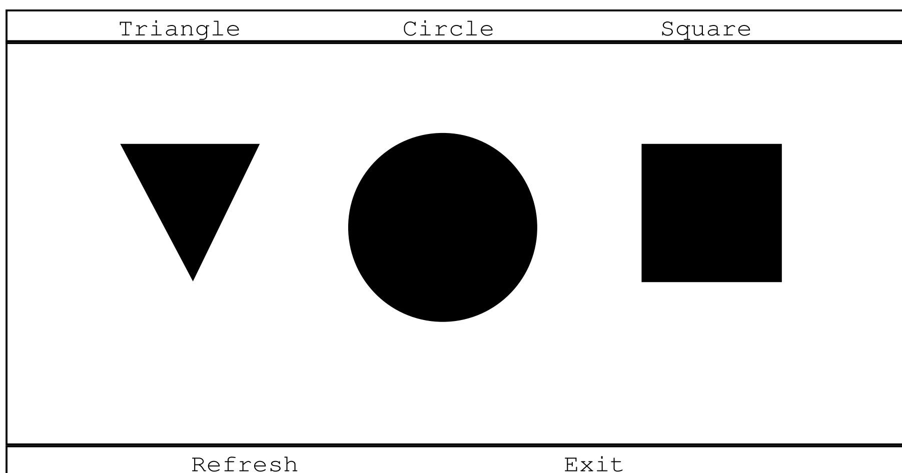



Figure 63. Command menu example.

The Lisp code to produce the window in Figure 63 is shown next.

```
;;; Define the frame and its panes
(setq *test-frame*
      (tv:make-window
        'tv:bordered-constraint-frame-with-shared-io-buffer
        ;; Select the graphics pane when it is exposed
        ':selected-pane 'graphics-pane
        ;; Specify the panes
```

```
        ':panes
        '((lower-menu-pane
            tv:command-menu-pane
            :item-list
            (("Refresh" :value :refresh
              :documentation "Refresh graphics pane")
             ("Exit" :value :exit
              :documentation "Exit this frame.")))
          (graphics-pane tv:window :label nil :blinker-p nil)
          (upper-menu-pane
            tv:command-menu-pane
            :item-list
            (("Triangle" :value :triangle
              :documentation "Draw a triangle.")
             ("Circle" :value :circle
              :documentation "Draw circle.")
             ("Square" :value :square
              :documentation "Draw square."))))

        ;; Specify the size constraints and ordering
        ':constraints
        '((main . ((upper-menu-pane graphics-pane lower-menu-pane)
                   ;; Big enough for the menu
                   ((upper-menu-pane :ask :pane-size))
                   ;; Big enough for graphics pane
                   ((graphics-pane :400.))
                   ;; Big enough for the menu
                   ((lower-menu-pane :ask :pane-size)))))))
;;; This function accesses the panes and looks for a blip
;;;  in the I/O buffer. It then draws, refreshes the
;;;  graphics pane, or exits
(defun work ()
  ;; Get access to the panes
  (let ((graphics-pane
          (send *test-frame* ':get-pane 'graphics-pane))
        (upper-menu-pane
          (send *test-frame* ':get-pane 'upper-menu-pane))
        (lower-menu-pane
          (send *test-frame* ':get-pane 'lower-menu-pane)))
    (send *test-frame* ':expose)
    ;; blip holds the list returned by :any-tyi
    (loop as blip = (send graphics-pane ':any-tyi)
          as result-value =
                 (cond ((and (listp blip) (eq (car blip) ':menu))
                        (send (fourth blip) ':execute (second blip)))
```

```
                (t nil))  ;just ignore keyboard input
        do
        ;; Check the value and draw the appropriate object
        (selectq result-value
          (:square
           (send graphics-pane ':draw-rectangle 180. 180. 800. 110.))
          (:circle
           (send graphics-pane ':draw-filled-in-circle 530. 200. 94.))
          (:triangle
           (send graphics-pane ':draw-regular-polygon
                 82. 120. 282. 120. 3))
          (:refresh
           (send graphics-pane ':refresh))
          (:exit
           (send *test-frame* ':deactivate)
           (return))))))

(work)
```

**Dynamic Item List Menus**

A dynamic item list menu is a menu in which the items change in between exposures. You see an example of a dynamic item list menu when you click on the [Select] item on the System menu (Figure !). At different times, a different item list appears, depending upon how many different processes were activated by the user.

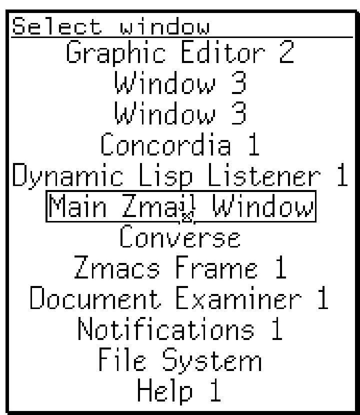



Figure 64. Select menu, an example of a dynamic item list menu.

You can add an item to the menu by changing the value of the variable supplied as the **:item-list-pointer** init-plist option. At appropriate times the menu checks to see if this variable has been changed. If it has, the menu automatically updates the item list. (Do not directly modify the item list yourself, as it is part of the

menu.) For a description of the times when the menu checks the state of **:item-list-pointer** option, See the section "Messages to Dynamic Menus".

The dynamic item list feature is provided only for momentary and pop-up menus; it is not available for use in menus within fixed frames.

### Dynamic Item List Mixins

Here is a list of noninstantiable mixins for dynamic item lists.

**tv:abstract-dynamic-item-list-mixin**
**tv:dynamic-item-list-mixin**
**tv:dynamic-multicolumn-mixin**

### Instantiable Dynamic Item List Menus

Here is a list of instantiable dynamic item list menus.

**tv:dynamic-momentary-menu**
**tv:dynamic-momentary-window-hacking-menu**
**tv:dynamic-pop-up-menu**
**tv:dynamic-pop-up-command-menu**
**tv:dynamic-pop-up-abort-on-deexpose-command-menu**

### Init-plist Option for Dynamic Menus

The init options on this list are for use with dynamic menus.

**(flavor:method :column-spec-list tv:dynamic-multicolumn-mixin)**
**(flavor:method :item-list-pointer tv:dynamic-...-menu)**

### Messages to Dynamic Menus

This method is a message accepted by menus with the dynamic item-list mixin.

**(flavor:method :update-item-list tv:dynamic-...-menu)**

### Dynamic Menu Example

A graphic example of a dynamic-momentary-menu is given in Figure !. The menu is shown in its state before updating (a) and after updating (b). This is followed by a listing of the code that produces it.

```
;;; Dynamic Menu Example
```

Figure 65. Dynamic menu example.

```
;;; Set up the initial item list and define the
;;;  dynamic-item-list pointer.
(defvar pointer
        '("Door Number 1"
          "Door Number 2"
          "Door Number 3"))

;;; Make the dynamic menu
(defvar doors (tv:make-window 'tv:dynamic-momentary-menu
                              ':borders 4
                              ':default-character-style
                                (:dutch :bold :normal)
                              ':label "CHOICES"
                              ':item-list-pointer 'pointer))

;;; Expose the menu, allowing a choice to be made
(send doors ':choose)
```

(In the example, nothing is being done with the result.)

Here is an example of dynamically updating the item list. The **:update-item-list** message is sent automatically and transparently by the menu to itself. The user does not have to explicitly send it.

```
;;; Add entries to the item list
(setq pointer
  (append pointer (list "Door Number 4" "Door Number 5")))

;;; Expose the menu with the new choices added
(send doors ':choose)
```

### Adding an Item to the System Menu

Although they are not specifically a part of the dynamic item list facility, two functions exist for adding an item (such as the name of a program) to the System menu.

### Adding an Item to the Programs Column

To add an item to the *Programs* column of the System menu, use the following function:

**tv:add-to-system-menu-programs-column**

### Adding an Item to the Create Column

To add an item to the *Create* menu used in the System Menu and the Screen Editor, use the following function:

**tv:add-to-system-menu-create-menu**

### tv:select-or-create-window-of-flavor Function

This function selects or creates a window of a specified flavor.

**tv:select-or-create-window-of-flavor**

### Multiple Menus

Multiple menus allow several items to be selected at a time. The selected items are highlighted in inverse video. Clicking the mouse on an item complements its selected state. Clicking the default special choice [Do It] associated with a multiple menu completes the selection, and returns the result of executing all the highlighted choices. The lower portion of Figure 23 is an example of a hardcopy multiple menu with several items selected.

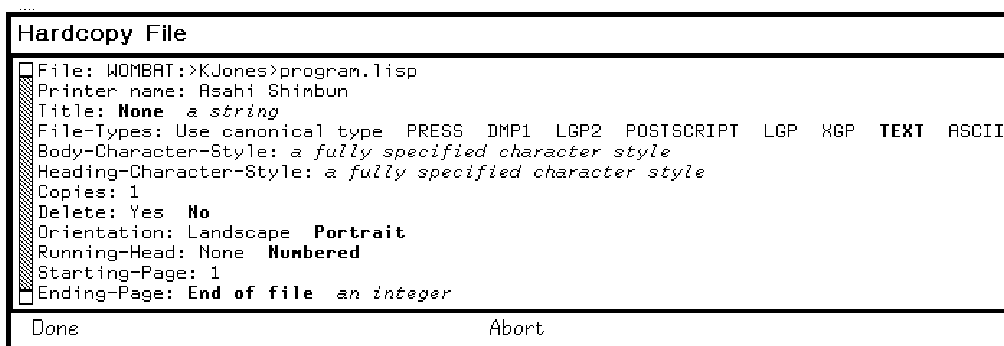



Figure 66. Hardcopy multiple menu.

### Multiple Menu Mixins

These are the noninstantiable flavors that add multiple menu behavior to a window.

**tv:menu-highlighting-mixin**

**tv:multiple-menu-mixin**

### Instantiable Multiple Menus

These are the instantiable flavors that add multiple menu behavior to a window.

**tv:multiple-menu**
**tv:momentary-multiple-menu**

### tv:multiple-menu-mixin Init-plist Options

Use these init options with multiple menu mixins.

**(flavor:method :highlighted-items tv:menu-highlighting-mixin)**
**(flavor:method :special-choices tv:multiple-menu-mixin)**

### tv:multiple-menu-mixin Messages

When using the following methods, note that for those requiring an item from the menu's item list, the item must be **eq** to the **:item-list** item, that is, the item itself.

**(flavor:method :set-highlighted-items tv:menu-highlighting-mixin)**
**(flavor:method :add-highlighted-item tv:menu-highlighting-mixin)**
**(flavor:method :remove-highlighted-item tv:menu-highlighting-mixin)**
**(flavor:method :highlighted-values tv:menu-highlighting-mixin)**
**(flavor:method :set-highlighted-values tv:menu-highlighting-mixin)**
**(flavor:method :add-highlighted-value tv:menu-highlighting-mixin)**
**(flavor:method :remove-highlighted-value tv:menu-highlighting-mixin)**

Consider the following example:

You make a menu (probably in a constraint frame description):

```
    ...
 :panes '(...
          (tv:command-menu-pane :item-list
                ((“This” :funcall this
         (“That” :funcall that)
                 (“The other” :funcall the-other)))
          ...)
```

Later, in some function, you want to highlight the "This" menu item. So you use the **:set-highlighted-items** message:

```
          ...
        (send menu :set-highlighted-items
            '(("This" :funcall this)))
          ...
```

Doing this does not highlight anything. What you need to do instead is:

```
(defvar *item-list* '(("This" :funcall this)
                      ("That" :funcall that)
                      ("The other" :funcall the-other)))

;;; make the constraint frame, but use backquote:

   ...
:panes `(...
          (tv:command-menu-pane :item-list ,*item-list*)
          ...)
   ...

;;; And in the function, do this:

   ...
(send menu :set-highlighted-items (list (first *item-list*)))
   ...
```

### tv:momentary-multiple-menu Example

A simple example of defining a momentary multiple menu is given in Figure !. The example of a Thai restaurant is used to illustrate the situation where more than one choice is appropriate.

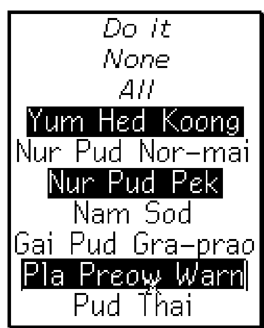



Figure 67. Momentary multiple menu.

The Lisp code used to generate Figure 67 is given in this example of setting up

and using a multiple menu. The variable **selections** is used to contain the selected items.

```
;;; Multiple Menu Example

;;; Set up the item list. Each of the dishes has a name and
;;;  a number. When selected, the names are highlighted.
(setq items '(("Yum Hed Koong" 1)
              ("Nur Pud Nor-mai" 2)
              ("Nur Pud Pek" 3)
              ("Nam Sod" 4)
              ("Gai Pud Gra-prao" 4)
              ("Pla Preow Warn" 5)
              ("Pud Thai" 6)))

;;; This handles the "Do It" special item
(defun do-it ()
  ;; Get the names of the selected dishes
  (setq names
        (mapcar 'car (tv:menu-highlighted-items Thai-menu)))
  ;; Get the numbers of the selected dishes
  (setq selections
        (send Thai-menu ':highlighted-values)))
;;; This handles the "None" special item
(defun none ()
  (send Thai-menu ':set-highlighted-items nil)
  (setq selections nil)
  (setq names nil))
;;; This handles the "All" special item
(defun all ()
  ;; Make all the items selected
  (send Thai-menu':set-highlighted-items items)
  ;; Get the names of the selected dishes
  (setq names (mapcar 'car (send Thai-menu ':highlighted-items)))
  ;; Get the numbers of the selected dishes
  (setq selections (send Thai-menu ':highlighted-values)))

;;; This sets up the special choice list.
;;; When one of these is selected, the menu exits.
(setq choices '(("Do it" :eval (do-it))
                ("None" :eval (none))
                ("All" :eval (all))))

;;; This instantiates the menu
(setq Thai-menu (tv:make-window
        'tv:momentary-multiple-menu
        ':item-list items
        ':special-choices choices))
```

```
;;; This exposes the menu, allowing choices to be made.
(send Thai-menu ':choose)
```

### The Multiple Menu Choose Facility

The multiple menu choose facility provides menus with several columns. The user may choose one item from each column. The selected choice in each column is highlighted with inverse video. At the bottom of the leftmost two columns are two special choices, in italics. The [Do It] choice selects all the highlighted choices. [Abort] deactivates the menu with no further action.

An example of the multiple menu choose facility can be displayed by clicking right on the [Reply] item in the main Zmail window, as in Figure ! below.

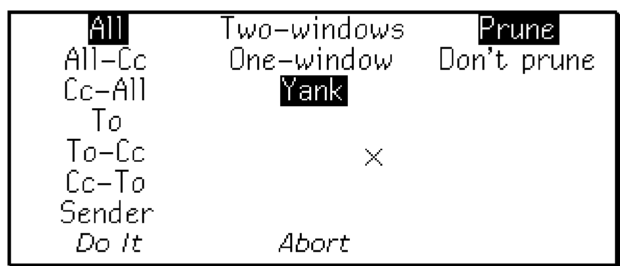



Figure 68. Multiple menu choose facility in Zmail.

Menus of this type are operated by the **:multiple-choose** message rather than the **:choose** message.

### The Standard Multiple Menu Choose Function

Use these functions for a simple multiple-menu-choose menu.

**tv:multiple-menu-choose**
**tv:defaulted-multiple-menu-choose**

### tv:multiple-menu-choose Example

An example of a simple multiple-menu-choose menu is shown in Figure !.
The code to produce the menu in Figure 69 follows.

Figure 69. A standard multiple-menu-choose menu.

```
;;;This sets up a three-row item list.
(setq possibilities
        '((Item-AA Item-AB Item-AC)
          (Item-BA Item-BB Item-BC)
          (Item-CA Item-CB Item-CC)))

;;;Evaluate this to instantiate the menu.
(setq new-menu (tv:multiple-menu-choose
                 possibilities '(Item-AA Item-BA Item-CA)))

;;;This also sets up a three-row item list. Evaluate it then
;;;re-evaluate the form above and note the difference.
(setq possibilities
 '(((Item-AA :value 1)(Item-AB :value 2)(Item-AC :value 3))
          (Item-BA Item-BB Item-BC)
          (Item-CA Item-CB Item-CC)))

;;;This also instantiates the menu illustrated.
;;;Notice especially the value returned for a 1st-column choice.
(setq new-menu (tv:defaulted-multiple-menu-choose
                 possibilities '(1 Item-BA Item-CA)))
```

### Multiple Menu Choose Mixin and Resource

Use these facilities for a multiple menu-choose menu.

**tv:multiple-menu-choose-menu-mixin**

**tv:pop-up-multiple-menu-choose-resource**

### Instantiable Multiple Menu Choose Flavors

Here are two instantiable multiple menu choose flavors.

**tv:multiple-menu-choose-menu**
**tv:pop-up-multiple-menu-choose-menu**

### tv:multiple-menu-choose-menu Example

Figure ! shows an example of a momentary-multiple-item-list menu generated using the flavor **tv:multiple-menu-choose-menu.** The figure is followed by the code that generated the menu.

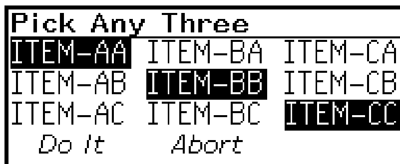



Figure 70. Momentary multiple-menu-choose menu.

```
;;; -*- Mode: LISP; Package: USER; Base: 10; Syntax: Zetalisp -*-
;;; Multiple-menu-choose-menu Example

;;; Define the item list of lists
(setq items-3x3
        '((Item-AA Item-AB Item-AC)
          (Item-BA Item-BB Item-BC)
          (Item-CA Item-CB Item-CC)))
;;; Specify the default, highlighted items
(setq default-items '(Item-AA Item-BB Item-CC))

;;; Make the menu
(setq newer-menu
   (tv:make-window
            'tv:multiple-menu-choose-menu
            ':label
            '(:string "Pick Any Three"
              :character-style (:swiss :bold :normal))
            ':borders 2))
;;; Choose an item from each column; resultat holds result
(setq resultat
        (send newer-menu
                ':multiple-choose items-3x3 default-items))
```

### The Multiple Choice Facility

The *Multiple Choice* facility produces a window containing several items, one per text line. For each item, there can be several yes/no choices for the user to make. For an example of a multiple-choice window, try selecting the [Kill or Save

Buffers] operation in the Zmacs editor menu (see Figure !).

```
  Buffer                                      Save Kill UnMod Hardcopy
* menus7.sab >sys>doc>menus H:                [X]  [ ]  [ ]   [ ]
* menus8.sab >sys>doc>menus H:                [X]  [ ]  [ ]   [ ]
+ doc-59-104.lisp >sys>doc>patch>doc-59 H:    [X]  [ ]  [ ]   [ ]
  *Buffer-1*                                  [ ]  [ ]  [ ]   [ ]
  *Buffer-2*                                  [ ]  [ ]  [ ]   [ ]
  *Buffer-3*..................................[ ]  [ ]  [ ]   [ ]
  *Bug-Mail-Frame-Mail-1*                     [ ]  [ ]  [ ]   [ ]
  menus10.sab >sys>doc>menus H:               [ ]  [ ]  [ ]   [ ]
  menus11.sab >sys>doc>menus H:               [ ]  [ ]  [ ]   [ ]
  menus12.sab >sys>doc>menus H:               [ ]  [ ]  [ ]   [ ]
  menus9.sab >sys>doc>menus H:                [ ]  [ ]  [ ]   [ ]
              Do It [ ]                          Abort [ ]
```

Figure 71. Multiple choice facility in the Zmacs menu.

Note that the window is arranged in columns, with headings at the top. The left-most column contains the text naming each item. The remaining columns contain small boxes (called *choice boxes*). A "no" box has a blank center, while a "yes" box contains an "X".

Pointing the mouse at a choice box and clicking the left button complements its yes/no state. Each choice can be initialized by the program to yes or no as appropriate for a default set-up. Note that some items cannot allow some choices, so there can be blank places in the array of choice boxes.

There can be constraints among the choices for an item. For example, if they are mutually exclusive then clicking one choice box to "yes" automatically sets the other choice boxes on the same line to "no".

Several parameters are associated with a multiple-choice window:

- *Item-name* -- a string which is the column heading for the leftmost column.

- *Item-list* -- a list of representations of items. Each element is a list, **(***item name choices***)**. *item* is any arbitrary object. *name* is a string which names that object; it is displayed on the left on the line of the display devoted to this item. *choices* is a list of keywords representing the choices the user can make for this item. Each element of *choices* is either a symbol, *keyword*, or a list, *(keyword default)*. If *default* is present and non-**nil**, the choice is initially "yes"; otherwise it is initially "no".

- *Keyword-alist* is a list defining all the choice keywords allowed. Each element takes the form **(***keyword name***)**. *keyword* is a symbol, the same as in the *choices* field of an *item-list* element. *name* is a string used to name that keyword. It is used as the column heading for the associated column of choice boxes.

- An element of *keyword-alist* can have up to four additional list elements, called *implications*. These control what happens to other choices for the same item when this choice is selected by the user. Each implication can be **nil**, meaning no implication, a list of choice keywords, or **t** meaning all other choices.

- The first implication is *on-positive*; it specifies what other choices are also set to "yes" when the user sets this one to "yes."

  The second implication is *on-negative*; it specifies what other choices are set to "no" when the user sets this one to "yes."

  The third and fourth implications are *off-positive* and *off-negative*; they take effect when the user sets this choice to "no."

  The default implications are **nil t nil nil**, respectively. In other words the default is for the choices to be mutually exclusive. (If the implications are not specified, the defaults are **rplacd**'ed into the *keyword-alist* element by the system.)

- *Finishing-choices* -- the choices displayed in the bottom margin. When users click on one of these they are done. The variable **tv:default-finishing-choices** contains a reasonable pair of default finishing choices: [Do It] and [Abort].

### The Standard Multiple Choice Function

This function interface to the multiple choice facility provides all the default values needed for a simple multiple choice menu.

**tv:multiple-choose**

### tv:multiple-choose Menu Example

An example of a multiple-choice menu is shown in Figure !.

| Today's selections | Yes, please. | No, thanks. | What is it? |
|---|---|---|---|
| Selection 1 | ☒ | ☐ | ☐ |
| Selection 2 | ☐ | ☒ | ☐ |
| Selection 3 | ☐ | ☐ | ☒ |
| **Selection 4..** | ☒ | ☐ | ☐ |
| Selection 5 | ☐ | ☐ | ☐ |

Do It ☐ Abort ☐

Figure 72. Multiple choice menu example.

The code to produce the multiple-choice menu in Figure 72 follows.

```
;;; Multiple Choice Example

;;; These are the possible choices the user can make
(setq choices '(Yes No Explain))

(setq selection-item-list
  (list (list 1 " Selection 1" choices)
        (list 2 " Selection 2" choices)
        (list 3 " Selection 3" choices)
        (list 4 " Selection 4" choices)
        (list 5 " Selection 5" choices)))
;;; Set the choice boxes
(setq selection-keyword-alist
        (list '(Yes "Yes, please. ")
              '(No "No, thanks. ")
              '(Explain "What is it? ")))

;;;  Expose the menu,
(setq appetizer-order-list
        (tv:multiple-choose
             " Today's selections" selection-item-list
             selection-keyword-alist))
```

If a selection is made for each item, an example of the values assigned to the variable **appetizer-order-list** is the following:

```
((1 YES) (2 NO) (3 EXPLAIN) (4 NO) (5 NO))
```

If only one selection is made, the values assigned to the **appetizer-order-list** might look like this:

```
((1 YES) (2) (3) (4) (5))
```

### The Basic Multiple Choice Flavor

The default multiple-choice facility shown previously is useful for many applications, but sometimes more customization is desirable. This basic facility provides options that allow you to tailor a multiple-choice menu to specific needs.

**tv:basic-multiple-choice**

### Instantiable Multiple Choice Menu Flavors

These instantiable facilities provide options that allow you to tailor a multiple-choice menu to specific needs.

**tv:multiple-choice**

**tv:temporary-multiple-choice-window**
**tv:temporary-multiple-choice-window**

### tv:multiple-choice Menu Messages

The following messages are useful to send to a multiple-choice window.

**(flavor:method :setup tv:multiple-choice)**
**(flavor:method :choose tv:multiple-choice)**

### tv:multiple-choice Example

This example shows how the **tv:multiple-choice** flavor can be used to define a multiple-choice menu.

```
;;; Specify the choice keywords
(setq choices '(Yes No))

;;; Set the choice boxes
(setq x-keyword-alist
        (list '(Yes "Yes")
              '(No "No")))

;;; Specify the item list
(setq x-item-list
  (list (list "Blue" "Blue" choices)
        (list "Red" "Red" choices)
        (list "Yellow" "Yellow" choices)
        (list "Green" "Green" choices)))

;;; Make the window
(setq x (tv:make-window 'tv:multiple-choice))

;;; Setup the window
(send p ':setup "Select Mode " x-keyword-alist
      tv:default-finishing-choices x-item-list)

;;; Expose the window and make a choice
(setq result (send p ':choose))
```

### The Choose Variable Values Facility

The choose-variable-values facility is used throughout the Lisp Machine system software. The basic idea of choose-variable-values is to allow the user to interac-

tively adjust the *value* of variables used in a program. (For an overview of related facilities intended for use with Dynamic Windows, see the section "Using Presentation Types for Input".)

More specifically, this facility displays a menu of names (standing for Lisp variables), followed by colons, and their values. After selecting a value with the left mouse button, users can interactively modify the value of the variable. Pressing the middle button preloads the input editor with the value of the variable, allowing the user to edit it. After the values are modified, the user can exit the menu.

For an example of a choose-variable-values window, try the [Attributes] option of the System menu (see Figure !).

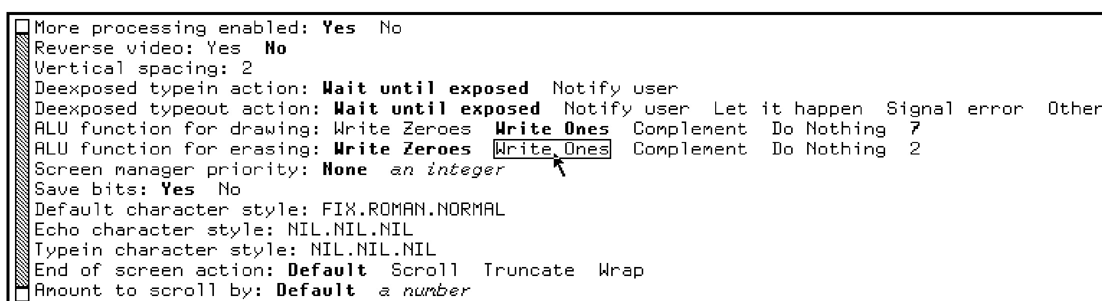



Figure 73. Choose-variable-values window accessed via the System menu.

## Variables and Types

Each variable has a *type* that limits the values it can assume. The way the value is displayed and the way the user enters a new value depend on the type. The types fall into two categories:

Those with a small number of valid values.

Those with a large or infinite number of valid values.

The first category displays all the choices, with the current value of the variable in boldface. The second category displays the current value until it is selected, at which point the value disappears until the user types in a new value. If the user rubs out more characters than were typed in, the original value is restored.

Note that the type definition mechanism is extensible. You can define new types at any time. See the section "Defining Choose Variable Values Types".

All variables whose values are to be chosen must be declared **special**, so that they are represented by Lisp symbols and can be accessed non-locally to your program. (Note that the compiler automatically declares certain variables to be special. Good programming practice mandates that this should be done explicitly by the programmer.)

In most cases, the syntax for input and output is controlled by the binding of the Lisp system variables **zl:base**, **zl:ibase**, **zl:*nopoint**, **zl:prinlevel**, **zl:prinlength**, **zl:package**, and **zl:readtable** , as usual. However, the **:number**, **:number-or-nil**, **:integer**, and **:integer-or-nil** types take a **:base** parameter to specify the base for input and output. The default base is decimal.

Each line of the display is represented by an *item*, which can be one of the following:

*String* The string is displayed; strings are useful for putting headings and blank separating lines into the display.

*Symbol* The symbol is a variable whose type is **:sexp**; that is, its value can be any Lisp object. The name of the variable on the display is simply its print-name.

*List* in the form: (*variable name type args...*)

- *variable* is the object whose value is being chosen.
- *name* is optional; if it is omitted it defaults to the print-name of *variable*. If *name* is supplied it can be a string, which is displayed as the name of the variable, or it can be **nil**, meaning that this line should have no variable name, but only a value.
- *type* is an optional keyword giving the type of variable; if omitted it defaults to **:expression**.
- *args* are possible additional specifications dependent on *type*.

A list is the most general form of item. It is possible to omit *name* and supply *type* since *name* is always a string and *type* is always a symbol. For example, both of the following forms are valid item lists:

```
(base "Output Base" :integer)
```

and

```
(base :integer)
```

It is also possible to specify a locative in place of a variable. The value displayed and modified is the contents of the cell designated by the locative.

### Predefined tv:choose-variable-values Variable Types

The following are the types of variables supported by default, along with any *args* that can be put in the item after the *type* keyword:

**:boolean**

The value of the variable is either **t** or **nil**. The choices are displayed as "Yes" for **t** and "No" for **nil**.

**:inverted-boolean**
The value of the variable is either **t** or **nil**. The choices are displayed as "Yes" for **nil** and "No" for **t**.

**:expression**
The value is any Lisp expression, read with **zl:read** and printed with **prin1**.

**:sexp** The same as **:expression**. This type is obsolete.

**:princ** The value is any Lisp expression, read with **zl:read** and printed with **princ**.

**:eval-form**
The value is the result of evaluating a Lisp form, read and evaluated with **zl:read-and-eval** and printed with **prin1**.

**:choose** *values-list print-function*
The value of the variable must be one of the elements of the list *values-list*. Comparison is by **zl:equal** rather than **eq**. All the choices are displayed, with the current value in boldface. A new value is entered by pointing to it with the mouse and clicking. *print-function* is the function to print a value; it is optional and defaults to **princ**.

**:assoc** *values-list print-function*
The displayed object is the **car** of one of the elements of *values-list*, while the **cdr** of the element is the value that goes in the variable. *print-function* is the function to print a value; it is optional and defaults to **princ**.

**:choose-multiple** *values-list print-function*
This type takes arguments like the **:assoc** type, but permits the user to choose more than one element in the values list. The variable is set to a list of all the values chosen.

**:menu-alist** *item-list*
The items are specified in an *item-list*. See the section "Types of Menu Items". The usual menu mechanisms for specifying the string to display, the value to return, the function to call, and the mouse documentation work with this. **:menu-alist** is often used for its mouse documentation feature.

**:character**
The value is an integer that is a character code. It is printed as the character name (using the **~:@c zl:format** operator), and it is read as a single keystroke.

**:character-or-nil**
This is an integer like **:character**, but **nil** is also allowed as the value. **nil** displays as "none" and can be entered by pressing CLEAR-INPUT.

**:string** This value is a string, printed with **princ** and read with **zl:readline**.

**:string-list**
This value is a list of strings, whose printed representation for input and output consists of the strings separated by commas and optional spaces.

**:string-or-nil**
This value is a string or **nil** if the user just presses RETURN, LINE, or END.

**:number :base** *base* **:or-nil** *or-nil*
This value is a number. It is printed with **prin1** and read with **sys:read-number**. If **:base** is specified, the number is read and printed in base *base*. By default, the number is read and printed in decimal. If **:or-nil** is specified with a value other than **nil**, a value of **nil** is accepted when the user just presses RETURN, LINE, or END. **nil** displays as "none". The default for *or-nil* is **nil**.

**:number-or-nil :base** *base*
The same as **:number :base** *base* **:or-nil t**. This type is obsolete.

**:decimal-number**
The same as **:number :base 10** This type is obsolete.

**:decimal-number-or-nil**
The same as **:number :base 10**. **:or-nil t**. This type is obsolete.

**:integer :base** *base* **:or-nil** *or-nil*
This value is an integer. It is printed with **prin1** and read with **sys:read-integer**. If **:base** is specified, the integer is read and printed in base *base*. By default, the integer is read and printed in decimal. If **:or-nil** is specified with a value other than **nil**, a value of **nil** is accepted when the user just presses RETURN, LINE, or END. **nil** displays as "none". The default for *or-nil* is **nil**.

**:date** This value is a universal date-time. An ambiguous date is interpreted as being in the future. (Compare this with **:past-date**.)

**:date-or-never**
This value is a universal date-time or **nil** if the user types "never". An ambiguous date is interpreted as being in the future.

**:past-date**
The value is a universal date-time. An ambiguous date is interpreted as being in the past.

**:past-date-or-never**
This value is a universal date-time or **nil** if the user types "never". An ambiguous date is interpreted as being in the past.

**:time-interval-or-never**
The value is an integer representing the number of seconds in a time interval, or **nil** if the user types "never". The interval is read and printed as either "never" or alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years.

**:time-interval-60ths**
The value is an integer representing the number of sixtieths of a second in a time interval. The interval is read and printed as alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years. The smallest unit read or displayed is second.

**:pathname**
The value is a pathname, represented as a string. The pathname read is merged with the result of **(fs:default-pathname)** and has a default version of **:newest**.

**:pathname-or-nil**
The value is a pathname, represented as a string, or **nil** if the user just presses RETURN, LINE, or END. The pathname read is merged with the result of **(fs:default-pathname)** and has a default version of **:newest**.

**:pathname-list**
The value is a list of pathnames, read as a series of pathnames separated by commas and optional spaces, and merged with the result of **(fs:default-pathname)**. The default version is **:newest**. The list is printed as a series of pathnames separated by commas and spaces.

**:host** The value is a network host, read and printed as the name of the host.

**:host-or-local**
The value is a network host. It is read as the name of a host or the string "local" to represent the local host. If the host is the local host, it is printed as "Local"; otherwise, it is printed as the name of the host.

**:host-list**
The value is a list of network hosts, read as a series of host names separated by commas or spaces, and printed as a series of host names separated by commas and spaces.

**:pathname-host**
The value is a pathname host, read and printed as the name of the host. The name can be "local", "sys", or the name of another logical host as well as the name of a physical host.

**:keyword-list**
The value is a list of symbols in the **keyword** package, read as a series of symbol names separated by commas or spaces, and printed as a series of symbol names separated by spaces. Symbol names are read and printed without package prefixes (that is, not preceded by colons).

**:font-list**
The value is a list of fonts, read as a series of font names separated by commas or spaces, and printed as a series of font names separated by commas and spaces. Font names are read and printed without package prefixes (that is, not preceded by **fonts:**).

A **:documentation** specification can be inserted where a variable type would normally be expected.

**:documentation** *doc type args...*
The actual type of the variable is *type*. *doc* is a string that is displayed in the mouse documentation line when the mouse is pointing at this item. The default, if no documentation is sup-

plied using the **:documentation** specification, depends on the variable type. It is generally something like "Click Left to input a new value from the keyboard".

### The Optional Constraint Function

It sometimes is necessary to ensure that when one variable's value is changed, one or more of the others is changed as well. As an init-plist option, a choose-variable-values window can have an associated function, which is called whenever a variable's value is changed. This function can implement constraints among the variables.

The constraint function is specified by the **:function** init-plist option. See the section "**tv:choose-variable-values** Options". It is called with arguments *window*, *variable*, *old-value*, and *new-value*. The function should return **nil** if just the original variable needs to be redisplayed, or **t** if no redisplay is required; in this case it would usually **setq** several of the variables then send a **:refresh** message to the window to redisplay everything.

### The Standard Choose Variable Values Function

**tv:chose-variable-values**, the standard function interface to the choose-variable-values feature chooses the dimensions of the window and enables scrolling if there are too many variables to fit in the chosen height. It exposes a window and displays the values of the specified variables, permitting the user to alter them.

### tv:choose-variable-values Examples

Here are some examples of how to call **tv:choose-variable-values**. The simplest kind of example is to display some variable names and values and let the user change them, as in Figure !. To see how it works, point at one of the variables, press the left mouse button, and then type in a new value and press RETURN. Recall that **zl:*nopoint** is a Lisp variable.

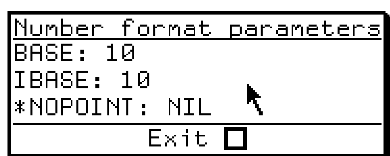



Figure 74. Choose-variable-values example 1.

The Lisp code used to produce Figure 74 is shown here.

```
;;; Choose Variable Values Example 1
```

```
; Invoke the window
(tv:choose-variable-values '(zl:base zl:ibase zl:*nopoint)
                           ':label "Number format parameters")
```

The same example can be done with better menu formatting in the next example (shown in Figure !).

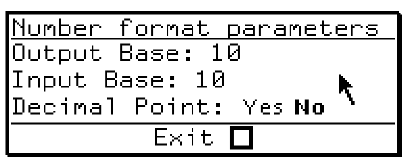



Figure 75. Choose-variable-values example 2: better formatting.

The Lisp code used to produce Figure 75 is given here.

```
;;; Choose Variable Values Example 2

(tv:choose-variable-values
        '((zl:base "Output Base" :number)
          (zl:ibase "Input Base" :number)
          (zl:*nopoint "Decimal Point"
                  :assoc (("Yes" . nil)
                          ("No" . t))))
        ':label "Number format parameters")
```

If we had not wanted to reverse the sense of **t** and **nil** the entry for **zl:*nopoint** would have been the following:

```
(*nopoint "No Decimal Point" :boolean)
```

If we wanted to use the name of the variable as the menu item, rather than spelling it out, we could have used the following expression:

```
(*nopoint :boolean)
```

As another example, we consider shopping for groceries via Lisp Machine. We have variables **fish**, **crustaceans**, **seafood-specialties**, **lettuce**, and **apples.** Many stores accept coupons for discounts on purchases, so the **Coupon-value** variable (a floating-point number) allows users to enter a dollar value representing the value of the coupons they are redeeming.

As mentioned, clicking [Middle] on the mouse puts the variable in the input editor, allowing you to make changes in it. In Figure ! we display this situation and allow it to be modified, using several different kinds of items:
The Lisp code used to produce Figure 76 is provided next. Each "STORE" in the example is implemented with a different variation of the choose variable value facility. Note the use of strings to provide labels for the sections, and null strings to separate the sections with blank lines.

Figure 76. Choose-variable-values window: grocery store example.

```
;;; Choose Variable Values Example 3

;;; Set up the variables
(setq fish '("Salmon"))
(setq crustaceans '("Clams"))
(setq seafood-specialties '("Flying-fish roe"))
(setq lettuce "Boston")
(setq apples "Pippin")
(setq Coupon-value 0)
(setq result (tv:choose-variable-values
   '("FISH STORE"
     (fish "Fish" :string-list)
     (crustaceans "Shellfish" :string-list)
     (seafood-specialties "Other Seafood" :string-list)
     ""
     "PRODUCE STORE"
     (lettuce "Lettuce" :choose ("Boston" "Red" "Iceberg"))
     (apples "Apples" :choose ("Macintosh" "Jonathan" "Pippin"))
     ""
     "VALUE OF YOUR COUPONS"
     (Coupon-value "Coupons"
        :documentation
          "Click left to enter the value of your coupons."
          :number))
     ':label "Today's Food Selections"))
```

## The User Option Facility

The user option facility provides a simple window interface that allows you to set parameter options to your programs. The user option facility is based on the choose-variable-values facility.

A typical use would be in a program that requires several variables to be set before it is run. In a conventional system, a standard way to alter these values would be to alter the code, recompile the program, and then run it. By contrast, the user option facility generates a window with the names and default values of the variables. This gives you the option of resetting these variables before execution of the program. When the window is exited, the rest of the program runs.

For an example of a user option window, type the following function at a Lisp Listener window:

```
(choose-user-options zwei:*zmail-user-option-alist*)
```

The **choose-user-options** function is also used by the Zmail Profile mode, and elsewhere throughout the system.

Special forms are provided for defining options, and the **choose-user-options** function exists for putting all the options into a choose-variable-values window so that the user can alter them. In addition, the current state of the options can be written into an initialization file, or all the options can be set to their default initial values.

### Functions for Defining User Option Variables

Use these special forms to define user option variables.

**define-user-option-alist**
**define-user-option**

### Functions for Altering User Option Variables

Use these functions to alter user option variables.

**choose-user-options**
**reset-user-options**
**write-user-options**

### User Options Example

Figure ! is an example of a user option window that sets three variables of a simple graphics program.
The Lisp code used to produce Figure 77 is shown between the asterisk-marked (****) lines. The rest of the code generates the graphics.

```
;;; User Option Example
```

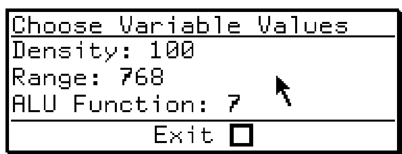



Figure 77. User options window example.

```
;;;****
;;; This names the user option alist
(define-user-option-alist options)

;;; These expressions set up the options
(define-user-option (alu-function options)
        tv:alu-ior :decimal-number "ALU Function")
(define-user-option (range options) 768. :decimal-number "Range")
(define-user-option (density options) 100. :decimal-number "Density")

;;; Expose the choose-option window
(choose-user-options options)
;;;****
;;; This is a random line-drawing function
(defun image (alu-function range density)
 (setq x (tv:make-window 'tv:window))
 ;; Temporarily select a window; the arguments
 ;;  are the window x and the final action on it
 (tv:window-call (x :deactivate)
  (setq n range)
  (loop for i below density do
        (send x ':draw-lines alu-function
              (random n) (random n) (random n) (random n)
              (random n) (random n) (random n) (random n))
        (send x ':draw-circle
              (random n) (random n) (random n)))
  (send x ':tyi)))

;;; Draw the image
(image alu-function range density)
```

## Defining Choose Variable Values Types

The standard choose-variable-values facility supplies programmers with a range of predefined types. See the section "Predefined **tv:choose-variable-values** Variable Types". However, this list is extensible through two mechanisms:

1. Adding a type keyword property to a new type name

2. Adding a type decoding method

### Adding a Type Keyword Property

The basic type definition mechanism is simple: put a **tv:choose-variable-values-keyword** property on the type name. In the following example, the new type is called **new-type**, the property value is *type-list*, and the property name is **tv:choose-variable-values-keyword**.

```
(defprop new-type type-list tv:choose-variable-values-keyword)
```

For a discussion of the contents of *type-list*: See the section "Elements of the **tv:choose-variable-values-keyword** Property". See the section "Type Decoding Message".

### Adding a Type Decoding Method

The second way to extend the range of standard types is to define a new flavor of choose-variable-values window and give it a **:decode-variable-type** method — circumventing the use of the standard variable types.

### Type Decoding Message

The method **(flavor:method :decode-variable-type tv:basic-choose-variable-values)** must implement the **:documentation** keyword, which can appear in an item where a variable type would normally appear. The system sends the **:decode-variable-type** message to a choose-variable-values window when it needs to understand an item. The argument of this message is a list whose **car** is the keyword for the item whose **cdr** is a list of the arguments of the keyword. The default method for **:decode-variable-type** looks for two properties on the keyword's property list: **tv:choose-variable-values-keyword**, which is a list of six values; and **tv:choose-variable-values-keyword-function**.See the section "**tv:choose-variable-values** Type Definition Example".

### tv:choose-variable-values Type Definition Example

```
;;; Defining a Choose Variable Values Type Example
;;;  Adding the type keyword property

(defvar candidate-1 nil)
(defvar candidate-2 nil)
(defvar candidate-3 nil)
```

```
;;; Set up the type list
(setq type-list '(princ nil ("Yes" "No" "Abstain") nil nil nil))

;;; Put the type-list value on the
;;;  tv:choose-variable-values-keyword property
(putprop 'mytype type-list
   'tv:choose-variable-values-keyword)

;;; Use the newly created type
(tv:choose-variable-values
  '((candidate-1 " John Q. Public " mytype)
    (candidate-2 " Jane Doe " mytype)
    (candidate-3 " John Blevins " mytype))
  ':label "*** Select  One  Candidate ***")
```

### Defining a Choose Variable Values Window

Up to this point, an easy-to-use but limited form of the choose-variable-values facility has been discussed, namely, the standard **tv:choose-variable-values** function.

In order to create a new flavor of window with choose-variable-values behavior, the *basic* and *instantiable* choose-variable-values window flavors are needed. These are described in this section.

### The Basic Choose Variable Values Flavor

**tv:basic-choose-variable-values** is the basic flavor that makes a window implement the choose-variable-values facility. It requires more parameter specifications from the programmer, but it is also the most flexible.

### Instantiable Choose Variable Values Flavors

Use windows with these instantiable choose-variable-values flavors as panes in a frame and as pop-up windows.

**tv:choose-variable-values-window**
**tv:choose-variable-values-pane**
**tv:temporary-choose-variable-values-window**

A resource of this type of window is:

**tv:temporary-choose-variable-values-window**

### I/O Buffers for Choose Variable Values Windows

I/O buffers can be associated with choose-variable-values windows. See the section "Menu Items and Menu Values". A choose-variable-values window has an I/O buffer, which the window uses to send commands (also known as *blips*) back to its controlling process. As usual these commands are lists, to distinguish them from keyboard characters that are numbers. If all panes send commands to the same I/O buffer, then when one of these commands arrives it can be processed in the appropriate pane. At the same time, the controlling process can be looking in the I/O buffer for other commands from other panes and for input from the keyboard. A choose-variable-values window uses the same I/O buffer to read a new value from the keyboard as it uses to send blips to the controlling process.

The following I/O buffer commands (blips) are sent by the choose-variable-values window to the user process.

(**:variable-choice** *window item value line-number mouse-gesture*)

This indicates that the user clicked on the value of a variable, expressing a desire to change it. *window* is the choose-variable-values window instance, *item* is the complete item specification, *value* is the value that was clicked on, and *line-number* is the line on which the item appears in the menu; the lines are numbered starting at 0. *mouse-gesture* is the mouse character (for example, **#\mouse-m**) corresponding to the gesture used for clicking.

(**:choice-box** *window box*)

This indicates that the user clicked on one of the choice boxes in the bottom margin. *window* is the window instance, and *box* is the choice box specification.

The following sequence of events is a typical model for implementing a choose-variable-values window.

1. Set up and expose the window. The window is gotten from the window resource, **tv:choose-variable-values-window**.

2. Loop within an **:any-tyi**, or **tv:io-buffer-get** loop, checking to see if a variable-choice or a choice-box selection has been made.

3. If a choice-box selection has been made, your "choice-box handler" routine is called. This routine returns the choice-box descriptor. If the choice-box was an [Abort] item, your process typically sends the window the **:deactivate** message.

A function that implements the response to these commands is:

**tv:choose-variable-values-process-message**

**tv:basic-choose-variable-values Init-plist Options**

The following init-plist options are relevant to choose-variable-values windows. Note that if no dimensions are specified in the init-plist, the width and height are automatically chosen according to the other init-plist parameters. The height is dictated by the number of elements in the *item-list*. Specifying a height in the init-plist, using any of the standard dimension-specifying init-plist options, overrides the automatic choice of height.

*Note:* the **:stack-group** option is required, unless the **:setup** message is used to initialize the window. See the section "**tv:choose-variable-values-window** Messages".

**(flavor:method :function tv:basic-choose-variable-values)**
**(flavor:method :variables tv:basic-choose-variable-values)**
**(flavor:method :stack-group tv:basic-choose-variable-values)**
**(flavor:method :name-style tv:basic-choose-variable-values)**
**(flavor:method :value-style tv:basic-choose-variable-values)**
**(flavor:method :string-style tv:basic-choose-variable-values)**
**(flavor:method :unselected-choice-style tv:basic-choose-variable-values)**
**(flavor:method :selected-choice-style tv:basic-choose-variable-values)**
**(flavor:method :margin-choices tv:choose-variable-values-window)**
**(flavor:method :io-buffer tv:choose-variable-values-window)**

### tv:choose-variable-values-window Messages

The following messages are useful to send to a choose-variable-values window.

**(flavor:method :setup tv:choose-variable-values-window)**
**(flavor:method :set-variables tv:choose-variable-values-window)**
**(flavor:method :appropriate-width tv:choose-variable-values-window)**
**(flavor:method :adjust-geometry-for-new-variables tv:choose-variable-values-window)**
**(flavor:method :redisplay-variable tv:choose-variable-values-window)**

### tv:choose-variable-values-window Example

As we have discussed, in the simplest mode of operation, the **tv:choose-variable-values** function takes care of creating the window and establishes all necessary communication with it. When you make a choose-variable-values window (as in the example below), you need to handle the communication yourself, using the information given below. An example of a situation in which this is necessary is when you have a frame, some panes of which are choose-variable-values windows.
The Lisp code used to generate Figure 78 is given next.

```
;;; Choose Variable Values Example 4

;;; In this example, the user specifies the number of
;;;  instrumentalists of each kind needed to define an orchestra.
```

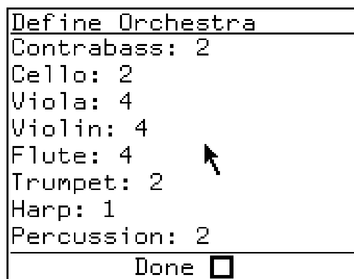



Figure 78. Example of making a choose-variable-values menu.

```
(defvar contrabass 2)
(defvar cello 2)
(defvar viola 4)
(defvar violin 4)
(defvar flute 4)
(defvar trumpet 2)
(defvar harp 1)
(defvar percussion 2)

;;;; Define the variable list
(defvar instrument-list
          '((contrabass "Contrabass" :number)
            (cello "Cello" :number)
            (viola "Viola" :number)
            (violin "Violin" :number)
            (flute "Flute" :number)
            (trumpet "Trumpet" :number)
            (harp "Harp" :number)
            (percussion "Percussion" :number)))
;;; Define the margin choice list
(defvar margin-list '(("Done" nil
       tv:choose-variable-values-choice-box-handler nil nil)))
```

```
;;; Make the window
(defvar choix
        (tv:make-window 'tv:choose-variable-values-window))

;;; This function sets up the window, exposes it,
;;;  and calls appropriate routines
(defun display ()
        (let ((base 10.) (ibase 10.)) ; Set the base to 10
          (send choix ':setup
                       instrument-list
                       "Define Orchestra"
                       nil
                       margin-list)
          ;; The :setup message is normally followed by the
          ;;  :adjust-geometry-for-new-variables message in order
          ;;  to coordinate the size of the window with the number
          ;;  of variables. The numerical argument (180.) tells
          ;;  it to adjust the width of the window to the precise
          ;;  size I want it to be. I could also have sent
          ;;  the :appropriate-width message.
          (send choix ':adjust-geometry-for-new-variables 180.)
          (send choix ':set-position 200. 200.)
          (tv:window-call (choix :deactivate)
            ;; blip holds the list returned by :any-tyi
            ;;  Look for a :choice-box blip
            (loop as blip = (send choix ':any-tyi)
                  until (eq (car blip) ':choice-box)
                  do (tv:choose-variable-values-process-message
                             choix blip)))))
```

In order to invoke this menu, type the following form at the Lisp input editor:

```
(display)
```

The results are stored in **contrabass, cello, viola,** and the other instrument variables.

### The Mouse-Sensitive Items Facility

The mouse-sensitive items facility is related to certain choice facilities such as the pop-up menus described previously. Like these facilities, the mouse is used to point at an object on the screen, and a box is drawn around an object when the mouse is over it. (Mouse sensitivity is a basic feature of Dynamic Windows and the presentation-type system. For an introduction to these facilities, see the section "Overview of User Interface Programming Facilities".)

In contrast to a menu, in which mouse-sensitive behavior is limited to a relatively permanent item list, mouse-sensitive items are not a permanent part of a window. They disappear if the screen is cleared, for example. A main feature of a mouse-

sensitive window is that graphical objects and text can be intermingled. The graphical objects themselves can be made mouse-sensitive. See the section "Mouse-Sensitive Areas Example".

For an example of mouse-sensitive items, try the [List Buffers] command in the Zmacs editor command menu (Figure !). Move the mouse over the list of buffers and click Right. Another menu, keyed from a mouse-sensitive-item, is exposed.

```
Buffers in Zmacs:
  Buffer name:                              File Version:                       Major mode:

* menus12.sab >sys>doc>menus H:             (19)                                (WriterTools)
* menus11.sab >sys>doc>menus H:             (18)                                (WriterTools)
* menus10.sab >sys>doc>menus H:             (18)                                (WriterTools)
+ *Buffer-1*                                [67 lines]                          (Fundamental)
* menus9.sab >sys>doc>menus H:              (20)                                (WriterTools)
* menus7.sab >sys>doc>menus H:              (27)                                (WriterTools)
* menus8.sab >sys>doc>menus H:              (18)                                (WriterTools)
  *Buffer-2*                                [1 line]                            (WriterTools)
+ doc-59-104.lisp >sys>doc>patch>doc-59 H:                                      (LISP)
  *Buffer-3*                                [1 line]                            (WriterTools)
  *Bug-Mail-Frame-Mail-1*                   (To: Bug-Concordia Re: Trying t•    (Text)

+ means new file or non-empty non-file buffer.  * means modified file.
• means field truncated.
```

Figure 79. Mouse-sensitive items.

Mixing **tv:basic-mouse-sensitive-items** into a window flavor equips the window with mouse-handling according to the paradigm described in this section. Mouse-sensitive items are something you add in when defining your own window, rather than a complete facility. Consequently, there is no instantiable version.

*Note:* The word "typeout" appears here and there in the mouse-sensitive items facility for historical reasons. Often mouse-sensitive items are typed out on top of some other display, such as an editor buffer. However, the mouse-sensitive-item facility has nothing to do with the *typeout-window* facility. See the section "Typeout Windows".

Use the following mixin flavor to create mouse-sensitive areas on the screen:

**tv:basic-mouse-sensitive-items**

### Attributes of a Mouse-sensitive Item

A mouse-sensitive item has three main attributes:

- A *type* -- a keyword that controls what you can do to it
- An *item* -- an arbitrary Lisp object associated with it
- A *rectangular area* of the window -- typically something is displayed in that area at the same time as a mouse-sensitive item is created, using normal stream output to the window.

Unlike things such as menu items, mouse-sensitive items are not a permanent property of the window. They are just as ephemeral as the displayed text. This means they go away if you clear the window or if typeout wraps around and types over them.

### Associating Actions with Mouse-sensitive Items

The **:item-type-alist** init-plist option specifies an alist that associates actions with types of items. Each element of the list contains the following elements:

- A *type keyword* -- for example, **:value**
- A *default operation* -- for example, a function name
- A *documentation string* -- displayed in the mouse documentation line when the mouse is pointing at an object of this type
- A *list of all the operations* -- (the default doesn't necessarily have to be a member of this list) This list is in the form of menu items, so typically each element is **(name . operation)** where the user sees the string *name* but the program identifies the operation by the symbol *operation*. In most cases *operation* is a function to be called, but it can be any atom.

Here is an example of an item-type-alist:

```
((zwei:file
   zwei:find-defaulted-file
   "Left: Find file this file.  Right: menu of Load, Find, Compare."
   ("Load" :value zwei:load-defaulted-file
    :documentation "Load this file.")
   ("Find" :value zwei:find-defaulted-file
    :documentation "Find file this file.")
   ("Compare" :value zwei:srccom-file
    :documentation "Compare file with newest version (srccom)."))
 (zwei:function-name
   zwei:edit-definition
   "Left: Edit function. Right: menu (Arglist, Edit, Disassemble, Document.)."
   ("Arglist" :value zwei:typeout-menu-arglist
    :documentation "Print arglist for this function.")
   ("Edit" :value zwei:edit-definition
    :documentation "Edit this function.")
   ("Disassemble" :value zwei:do-disassemble
    :documentation "Disassemble this function.")
   ("Documentation" :value zwei:typeout-long-documentation
    :documentation "Print long documentation for this function.")))
```

The **tv:item-type-alist** instance-variable can be initialized via the init-plist when the window is created. Normally, you do not create this alist directly. Instead, you use **tv:add-typeout-item-type** to build it up incrementally. See the section "**tv:basic-mouse-sensitive-items** Messages and Functions".

### Mouse Behavior

The mouse works with a mouse-sensitive item in the following manner:

- Mouse-left -- Perform the default operation
- Mouse-right -- Pop up a menu of all the operations. Selecting one of these items performs it.
- Mouse-right-twice -- Call the System menu.
- Other mouse clicks and clicking on an item whose type is not in the type alist -- Cause a beep (the screen flashes) and generate an error.

Performing an operation means that a command (also known as a *blip*) is sent to the controlling process through the **:force-kbd-input** message to the window. This command is a list **(:typeout-execute** *operation item***)**, where *operation* is the operation and *item* is the arbitrary object remembered by the mouse-sensitive item. The ramifications of this, and how the *operation* is performed, are up to the application program.

You can use the **tv:add-typeout-item-type** special form to declare information about a mouse-sensitive type by adding an entry to an alist kept in a special variable. This alist can be put into the item-type alist of a mouse-sensitive window.

### tv:basic-mouse-sensitive-items Init-plist Options

Use this init option with **tv:add-typeout-item-type**.

**(flavor:method :item-type-alist tv:basic-mouse-sensitive-items)**

### tv:basic-mouse-sensitive-items Messages and Functions

The following messages are useful to send to a window with mouse-sensitive items. To create and display a list of mouse-sensitive items, use the function **si:display-item-list**.

**(flavor:method :item tv:basic-mouse-sensitive-items)**
**(flavor:method :primitive-item tv:basic-mouse-sensitive-items)**
**si:display-item-list**

### tv:basic-mouse-sensitive-items Example

An example of a mouse-sensitive items window is shown in Figure !. It shows four mouse-sensitive items in a window. One of the items has been selected. Some graphic figures (not mouse-sensitive) have also been drawn in the window. For a

description of the graphics operations, see the section "Graphic Output to Windows".

The point of this figure is to show how in mouse-sensitive windows (unlike in regular menus) graphics and text can be intermingled. Notice the technique of combining the mixin flavors **tv:borders-mixin** and **tv:top-box-label-mixin** before **tv:window** to generate the boxed-in label at the top of the window.

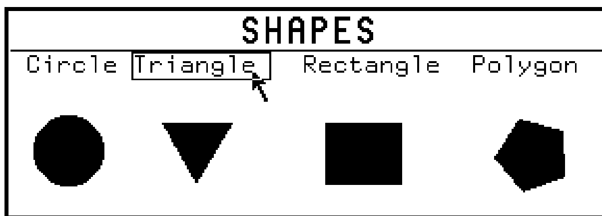



Figure 80. Mouse-sensitive items example.

In Figure ! one of the items [Triangle] has been selected, causing a menu of alternative actions to the the default action (default function) to appear next to it.

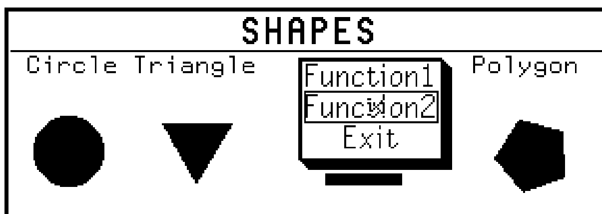



Figure 81. Result of selecting a mouse-sensitive item.

The Lisp code used to produce Figure 81 is listed next.

```
;;; Mouse-sensitive Example

;;; The functions called by the menus do nothing except increment
;;;  some values. Check their values after instantiating the
;;;  window to verify that the values were incremented. Also
;;;  look at the value of the variable "blip".

;;; Initialize variables
(zl:defconst c1 0)
(zl:defconst c2 0)
(zl:defconst default 0)
(defvar alist-alpha nil)
```

```
;;; Define a new flavor of window, with a
;;;  centered top-label and a mouse-sensitive-item mixin
(defflavor new ()
           (tv:centered-label-mixin
            tv:borders-mixin tv:top-box-label-mixin
            tv:basic-mouse-sensitive-items
            tv:window))
;;; These define mouse-sensitive items
(tv:add-typeout-item-type alist-alpha
                          :new-type "Exit" (exit)
                          nil "Exit and kill window")

(tv:add-typeout-item-type alist-alpha
                          :new-type "Function2" (function2)
                          t "Add one to c2")
(defun function2 ()
  (setq c2 (+ 1 c2)))

(tv:add-typeout-item-type alist-alpha
                          :new-type "Function1" (function1)
                          nil "Add one to c1")
(defun function1 ()
  (setq c1 (+ 1 c1)))

;;; Make the mouse-sensitive window
(defvar sensitive-window
        (tv:make-window
          'new ; This is the flavor specification
          ':borders 2
          ':top 200.
          ':bottom 310.
          ':right 488.
          ':width 316.
          ':blinker-p nil
          ':label '(:string "SHAPES"
                    :character-style (:fix :roman :very-large))
          ':item-type-alist alist-alpha))
```

```
;;; Expose the window and draw the objects
(defun set-up ()
  (tv:window-call (sensitive-window :deactivate)
    (send sensitive-window ':item ':new-type " Circle ")
    (send sensitive-window ':item ':new-type "Triangle ")
    (send sensitive-window ':item ':new-type "  Rectangle ")
    (send sensitive-window ':item ':new-type " Polygon ")
    (send sensitive-window ':draw-filled-in-circle 30. 50. 18.)
    (send sensitive-window ':draw-triangle 79. 36. 116. 36. 97. 68.)
    (send sensitive-window ':draw-rectangle 40. 32. 164. 36.)
    (send sensitive-window
          ':draw-regular-polygon 265. 34. 288. 40. 5.)
    ;; blip holds the list returned by :any-tyi
    (loop as blip = (send sensitive-window ':any-tyi)
          ;; Invoke the operation returned by the blip
          ;;  unless the operation is (exit)
          until (equal (cadr blip) '(exit))
          do (eval (cadr blip)))))

; Do it
(set-up)
```

### Mouse-Sensitive Areas Example

In Figure !, we show how *areas* of the screen can be made mouse-sensitive, allowing the mouse to be used to select graphical entities, as well as text items.

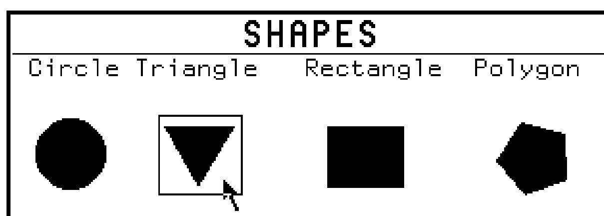



Figure 82. Mouse-sensitive areas example.

To make the shapes mouse-sensitive, within the function **set-up**, add several lines of Lisp code after the following line:

```
(send sensitive-window ':draw-regular-polygon 250. 34. 272. 40. 5.)
```

Next is the code to add to **set-up.**

```
(defun set-up ()
  .
  .
  .

;; The boxes are associated with the graphic area
(send sensitive-window
     ':primitive-item ':new-type 'box-1 10. 30. 52. 74.)
(send sensitive-window
     ':primitive-item ':new-type 'box-2 77. 31. 120. 72.)
(send sensitive-window
     ':primitive-item ':new-type 'box-3 160. 31. 201. 72.)
(send sensitive-window
     ':primitive-item ':new-type 'box-4 250. 31. 295. 75.)
  .
  .
  .
)
```

### The Margin Choice Facility

A window can be augmented with choice boxes in its bottom margin using the flavor **tv:margin-choice-mixin**. See the section "The Multiple Choice Facility". Margin choice boxes give the user a few labelled mouse-sensitive points that are independent of anything else in the window. Thus margin-choices can be added to any flavor of window in a modular fashion. They are commonly used to implement "confirmation" choices (for example, [Do It] and [Abort]) following another selection.

Margin choices are not a complete choice facility and consequently do not come supplied in an instantiable version. The margin choice facility must be combined with another window flavor. For an example of a window with margin choices (as well as choice boxes in its interior), try the [Kill or Save Buffers] operation in the Zmacs editor menu (refer to Figure 68 shown previously, page 2033.)

### The tv:margin-choice-mixin Flavor

Use this mixin flavor to provide choice boxes in a window's margin.

**tv:margin-choice-mixin**

**tv:margin-choice-mixin Init-plist Option**

Use this init option to create a line of choice boxes in a window's margin.

**(flavor:method :margin-choices tv:margin-choice-mixin)**

### tv:margin-choice-mixin Messages

Use this method to change the set of margin choices.

**(flavor:method :set-margin-choices tv:margin-choice-mixin)**

### tv:margin-choice-mixin Example

A simple example of the margin choice facility is shown in Figure !. In the example, the user can select one of three actions to be taken within a graphics window.

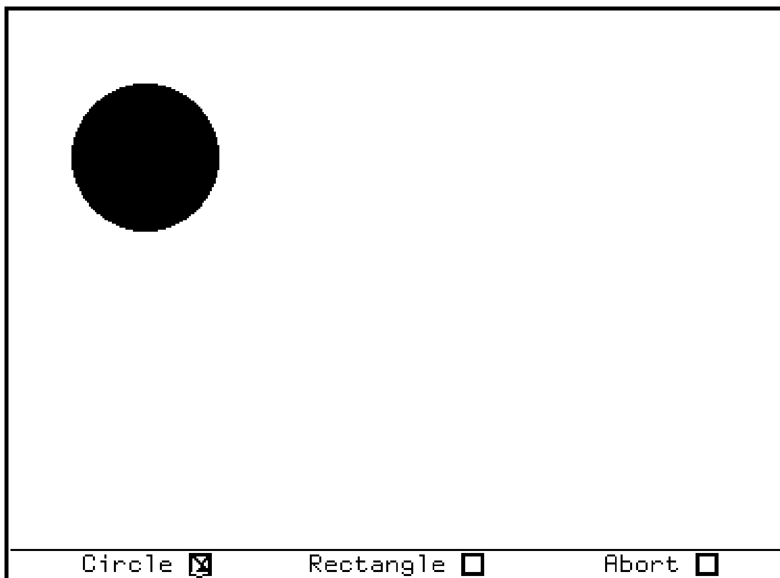



Figure 83. Example of a margin choice facility added to a window.

The Lisp code used to produce Figure 83 is listed below.

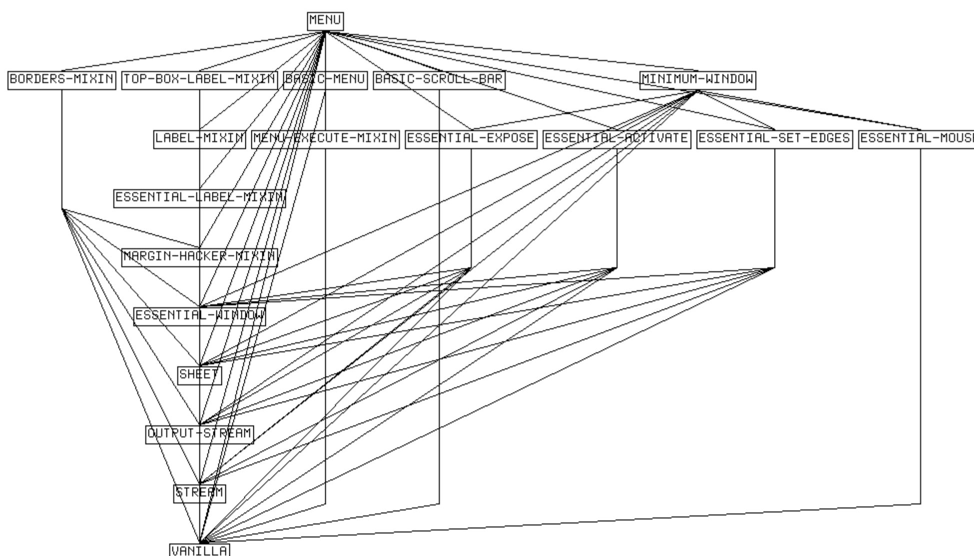
```
;;; Margin Choice Facility Example
;;;  Draws shapes or aborts based on the margin-choice selection.

;;; Specify the margin choice-box descriptors
```

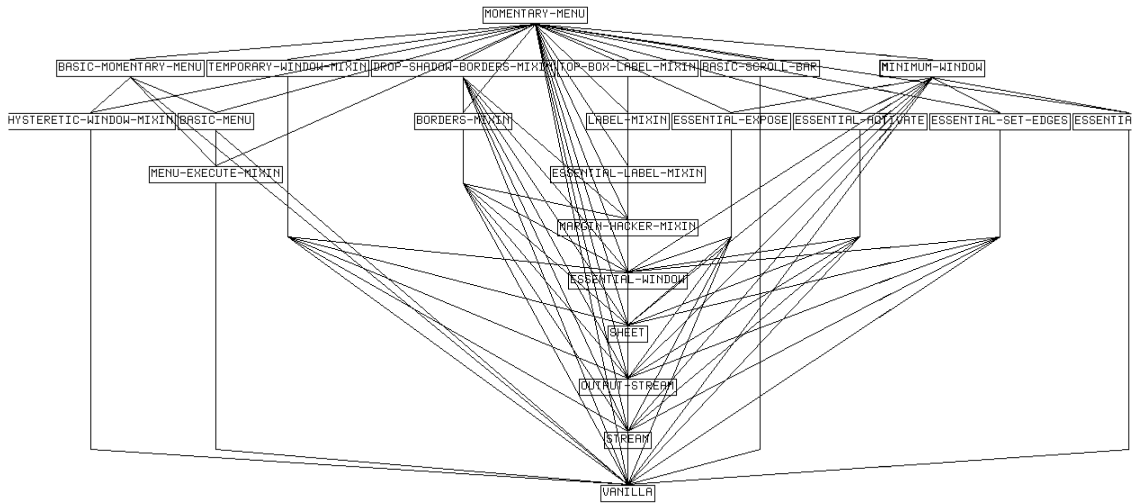
```
(defvar choice-box-1 '(" Circle" nil shape-handler x y
                       (:draw-filled-in-circle 70. 75. 38.)))
(defvar choice-box-2 '("Rectangle" nil shape-handler x y
                       (:draw-rectangle 70. 70. 170. 50.)))
(defvar choice-box-3 '(" Abort" nil Abort-handler x y))
(defvar margin-list (list choice-box-1 choice-box-2 choice-box-3))

;;; Name of the window we create

(defvar test-window)

;;; Mixin the margin-choice facility with a window
(defflavor window-with-margin-choices ()
        (tv:borders-mixin tv:margin-choice-mixin tv:window))

;;; Define a handler for the choice boxes that draw shapes
(defun shape-handler (window choice-box region y-pos)
  y-pos ;not used, suppress compiler warning
  ;; Make just this box be lit
  (clear-other-choice-boxes choice-box)
  ;; Erase the window
  (send window :clear-window)
  ;; Refresh the margin so new choice box X's are displayed
  (send (tv:margin-region-function region) :refresh window region)
  ;; Draw the shape the user requested
  (apply window (nth 5 choice-box)))

;;; Define a handler for the "Abort" box
(defun Abort-handler (window choice-box region y-pos)
  y-pos ;not used, suppress compiler warning
  ;; Make just this box be lit
  (clear-other-choice-boxes choice-box)
  ;; Refresh the margin so new choice box X's are displayed
  (send (tv:margin-region-function region) :refresh window region)
  ;; Remove the window from the screen
  (send window :deactivate))

;;; This function clears the non-selected choice boxes
;;; and sets the selected one
(defun clear-other-choice-boxes (selected-box)
  (dolist (box margin-list)
    (setf (tv:choice-box-state box) (eq box selected-box))))
```

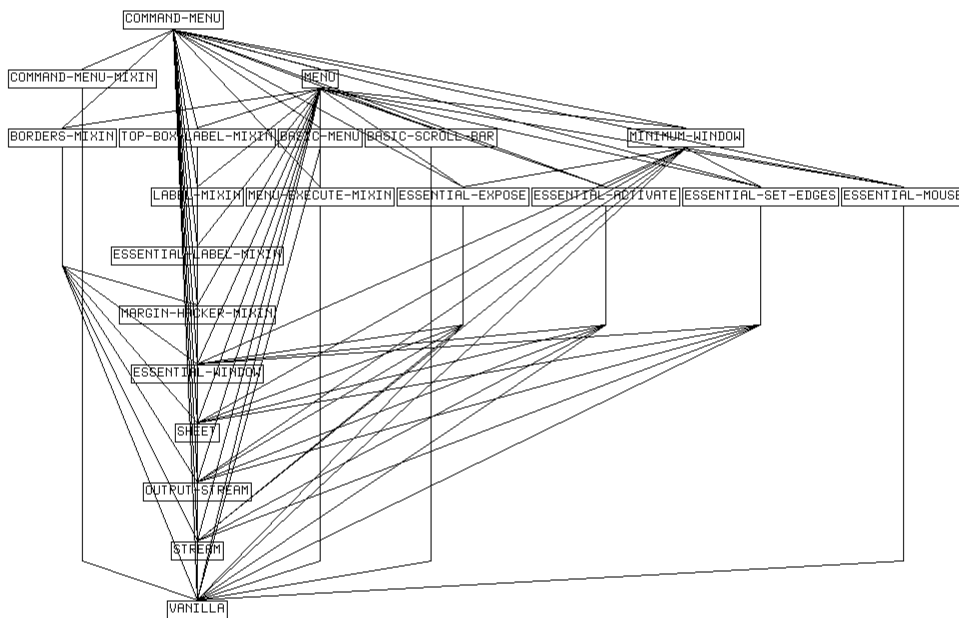
```
;;; Test the window.
(defun Shapes (&optional (test-window (tv:make-window
        'window-with-margin-choices
        :borders 2
        :label nil
        :vsp 2  ; vertical spacing
        :top 200.
        :bottom 500.
        :right 650.
        :width 410.
        :margin-choices margin-list
        :blinker-p nil)))
  (send Test-Window :Expose))

;;; Type (SHAPES) to try this out.
```

### The Flavor Network of tv:menu

**tv:menu** is the basis of many of the choice facilities described in this text. **tv:menu** is itself built on a network of flavors, shown in this diagram. **tv:momentary-menu** has a different network, which gives the flavor its own behavior. **tv:command-menu** is based on both **tv:menu** and the **tv:command-menu-mixin**. Knowing the derivation of these flavors can be useful in investigating all the available options and in modifying them for special applications.






### Init-plist Options for tv:menu

This is a list of some useful window-oriented init-plist options accepted by the **tv:menu** flavor and flavors built on it. It is not meant to be a comprehensive list. Use the Flavor Examiner to find out all the init-plist options of a particular flavor. Most of these options are also documented elsewhere: See the section "Using the

Window System".

**(flavor:method :activate-p tv:menu)**
**(flavor:method :borders tv:menu)**
**(flavor:method :bottom tv:menu)**
**(flavor:method :character-height tv:menu)**
**(flavor:method :character-width tv:menu)**
**(flavor:method :columns tv:menu)**
**(flavor:method :default-character-style tv:menu)**
**(flavor:method :edges tv:menu)**
**(flavor:method :edges-from tv:menu)**
**(flavor:method :expose-p tv:menu)**
**(flavor:method :fill-p tv:menu)**
**(flavor:method :geometry tv:menu)**
**(flavor:method :height tv:menu)**
**(flavor:method :inside-height tv:menu)**
**(flavor:method :inside-size tv:menu)**
**(flavor:method :inside-width tv:menu)**
**(flavor:method :item-list tv:menu)**
**(flavor:method :label tv:menu)**
**(flavor:method :left tv:menu)**
**(flavor:method :minimum-height tv:menu)**
**(flavor:method :minimum-width tv:menu)**
**(flavor:method :name tv:menu)**
**(flavor:method :position tv:menu)**
**(flavor:method :reverse-video-p tv:menu)**
**(flavor:method :right tv:menu)**
**(flavor:method :rows tv:menu)**
**(flavor:method :screen tv:menu)**
**(flavor:method :top tv:menu)**
**(flavor:method :vsp tv:menu)**
**(flavor:method :width tv:menu)**
**(flavor:method :x tv:menu)**
**(flavor:method :y tv:menu)**

### Messages Accepted by tv:menu

These are some of the messages (arranged in alphabetical order) accepted by menu flavors built on **tv:menu.** The list is not meant to be comprehensive. Use the Flavor Examiner to find out all the messages accepted by a particular flavor. Most of these messages are also documented elsewhere: See the section "Using the Window System".

**(flavor:method :deactivate tv:menu)**
**(flavor:method :deexpose tv:menu)**
**(flavor:method :expose tv:menu)**
**(flavor:method :refresh tv:menu)**

**(flavor:method :set-default-character-style tv:menu)**
**(flavor:method :set-edges tv:menu)**
**(flavor:method :set-item-list tv:menu)**
**(flavor:method :set-label tv:menu)**

## Scroll Windows

### Introduction to Scroll Windows

Scroll windows are a flavor of window provided by the Genera window system to facilitate building programs that display information that updates itself, changes its format, responds to the mouse, and shows other evidences of "live" behavior. To see many examples of this type of window, press SELECT P to invoke the **Peek** subsystem, and observe the behavior of its various displays as the objects they represent change state.

The basic service performed by scroll windows is that of *redisplay*. You provide a scroll window with a data structure defining what is to be displayed and how to display it. This is very different from other windows that you simply *instruct* to display text (and sometimes graphics) by telling them what to display. While a normal window simply draws what it has been asked to display, a scroll window remembers *how to display again* what it is now displaying, when instructed to do so. Also, a scroll window knows how to *update* its display, changing only those portions of the display that need changing. This is very much like what a real-time editor does when you change text. (Redisplay facilities for Dynamic Windows are introduced in another section; see the section "Displaying Output: Replay, Redisplay, and Formatting".)

A typical use of scroll windows is to display a structured representation of some data structure in your program. By clicking on mouse-sensitive items, you can ask to "display more detail" about some item on display. Your program and the scroll window would negotiate to display the more detailed items under the selected item, and move other items around. The file system editor and the **Window** hierarchy display in **Peek** do this. Another typical use is to display data about activity in the Lisp Machine going on simultaneously in other processes, while you watch the display. Such a display might have lines consisting of fixed text followed by numbers or strings that are the "values" of the quantities being "watched". For instance, some lines of such a display might read as follows:

```
Total polyhedra measured    603
Global eccentricity (av.)   .82%
```

while you watched; the numbers change (*update*) as the program measures new polyhedra.

Note that "scroll windows" have nothing, in particular, to do with the concepts of scrolling of windows in general and of mouse scrolling commands in particular. The name "scroll window" is something of a misnomer and a historical accident. Scrolling is not really what is important about scroll windows: the important thing that they provide is a convenient mechanism for getting information to redisplay.

Scroll window displays are exciting and enjoyable to watch and use, and add a touch of class to any program that uses them.

## Basics of Scroll Windows

The flavor of scroll window most often used is **tv:scroll-window**. You can call **tv:make-window** to make a scroll window. There is also **tv:basic-scroll-window** that contains nothing more than the feature of being a scroll window, and can be used to build more highly specialized flavors. You might also be interested in **tv:scroll-window-with-typeout**. It provides an inferior typeout window should random program output occur directed at it.

The various fields to be displayed are described by *items*. Each item corresponds to some logical portion of the display, always an integral number of lines. Items often contain other items (in a hierarchical fashion), and items can be added and removed from items dynamically (which, as is the whole point of scroll windows, causes the objects on display to appear and reappear when the scroll window's display is *redisplayed*).

A scroll window displays exactly one *top-level item*. The top-level item is simply an item corresponding to *all* the data to be displayed in in the scroll window. You normally create and set the top-level item just once, when you create and initialize the scroll window. When you have constructed the top-level item, you hand it to the scroll window using this message:

**(flavor:method :set-display-item tv:basic-scroll-window)**

The display created by the items given to a scroll window may well be larger than the physical dimensions of the window. Scroll windows handle this elegantly by showing only a portion of the total display, and allowing the user to scroll the data of the display in the window by using the mouse scrolling commands.

You cause a redisplay by sending the window this message:

**(flavor:method :redisplay tv:basic-scroll-window)**

There are two types of items: *line items* and *list items*. A line item describes information to be displayed on exactly one line of the display; that is, if the portion of the display controlled by a certain line item is visible in the window, then it uses up exactly one line of the window, and all of the information of the line item must fit in that line. Drawing a line item must not ever try to move to the next line (you shouldn't use RETURN characters).

A line item is built up of a sequence of *entries*. Each entry is responsible for controlling how one field of the line is drawn. The entries in a line item can be any mixture of constant strings or dynamically updated quantities. The descriptions of the dynamic quantities provide instructions for obtaining and displaying their values. The formats of these descriptions are given below. When the window is asked to redisplay, all of the dynamic entries of the line items on display are computed according to these instructions, and the fields of the line to which they correspond are dynamically and incrementally updated if they need to be.

List items describe multiple-line objects to be displayed. A list item is little more than a list of other items, themselves line items or list items. A list item is displayed by displaying all of the elements in it, in the order in which they appear in the list. The way you insert and remove lines of the display is by adding elements to and deleting elements from list items.

A list item is simply a Lisp list. Its first element is a *list item plist*, specifying some advanced options to be discussed below, and its remaining elements are the items logically comprising the list item. In most cases, the list item plist may be left empty (that is, **nil**).

### Constructing Items

Line items are constructed by a specialized function, described below. List items are constructed by the standard Lisp list-building functions.

### Constructing Line Items

Line items are constructed with the following function:

**tv:scroll-parse-item**

The line item spec consists of two portions: *global line attributes* that are optional, and *entries*, specifying the fields to be displayed, in the order they are to be displayed on the line. The global line attributes are keyword/value pairs of elements. The first even-numbered element of the line item spec that is not a symbol is the first entry (all keywords are symbols). **nil**s are ignored in any position of the line item spec; this sometimes makes the specs easier to construct. Every occurrence of **nil** is deleted from the spec before further processing.

Here is a simple call to **tv:scroll-parse-item**.

```
(tv:scroll-parse-item
        ':mouse '(DOUGHNUTS)
        "Number of doughnuts:  "
        '(:symeval food:doughnut-holes nil ("~D")))
```

Here the global line attributes are present, and consist of the following:

```
':mouse '(DOUGHNUTS)
```

There are two entries:

```
"Number of doughnuts:  "

(:symeval food:doughnut-holes nil ("~D"))
```

In the above example, the **:mouse** global line attribute makes the line displayed by this line item be mouse-sensitive, and the data item (**DOUGHNUTS**) will be encoded in the blip fed to the window's input buffer when this line is clicked upon. The meanings of the various global line attributes will be discussed in detail later.

There will be two fields displayed on this line: the fixed string **"Number of Doughnuts: "**, and the value of the global variable **food:doughnut-holes**. The latter value will be displayed as a decimal number (the **"~d"** is a **zl:format** control string), immediately after the **"Number of doughnuts: "** string, on the same line.

Whenever the window displaying this item is asked to redisplay, the displayed value of **food:doughnut-holes** will be updated if the value of that variable has changed.

### Line Item Entries

An *entry* in a line item spec can either specify a constant string to be displayed, or it can specify how to find a value to be displayed. There are four types of entries: *string, symeval, function,* and *value*. An entry is ordinarily represented as a list, whose first element is one of the keywords **:string , :symeval, :function,** or **:value**.

There are two exceptions. First, when an entry is to be made mouse-sensitive, two extra elements are included at the front of the list. See the section "Mouse Sensitivity". Secondly, there are shorthand forms for some of the formats; they are listed in the table below.

Here are the four types of entries, and their respective formats:

**:string**

Format: **(:string** *string***)**
Shorthand format: *string*

where *string* is a string. This entry will display as the string, occupying as much of the line as it takes up.

**:symeval**

Format: **(:symeval** *symbol width* **(***format-ctl base *nopoint***))**
Shorthand format: *symbol*

where *symbol* is a symbol to be evaluated to produce the value to be displayed. The syntax *symbol* is equivalent to

```
(:symeval symbol nil ("~A" base *nopoint))
```

The third and fourth elements of the entry are optional. *width* specifies the field width in characters, on the line, to be allocated to the displayed data. If omitted, or given as **nil**, as much space as needed will be allocated. If a value is given, it must be a positive number that must fit in the window's line length. The printed representation of the value should not use more than this many characters.

The value is printed using the **format** function. The fourth element of the entry is a list, whose first element specifies the **format** control string to be used. If there is no fourth element, **"~a"** is used. The second and third elements of this last element of the entry (which are also optional) give the values of the global variables **zl:base** and **zl:*nopoint** to be set up when **format** is called. If not given, the current values of these variables at redisplay time will be used.

Note that if you use the shorthand form of the **:symeval** entry type as the first entry in the line item spec, it will be mistaken for a keyword in the global line attributes. If you want the first entry to be a **:symeval** entry, you must use the longer syntax.

Here are some examples of **:symeval** entries:

```
(:symeval number-of-dogs)              ; Just display the value.
number-of-dogs                         ; (The same.)
(:symeval number-of-dogs 6 ("~S"))     ; Use six columns and
                                       ;   use slashification.
```

**:function**

Format: **(:function** *function arglist width* **(***format-ctl base *nopoint***))**
Shorthand format: **(lambda .....)**
Shorthand format: **(named-lambda ....)**
Shorthand format: *<an actual compiled code object>*

This is the most general type of entry. It specifies a function to be called at redisplay time, and the actual arguments to which it is to be applied. If obtaining the data to be displayed for an entry involves any action more complicated than the evaluation of a variable, you will need a **:function** entry. *function* specifies the function to be called. It may be a symbol, lambda expression, or named-lambda expression, or compiled code object. It will be applied to *arglist* at redisplay time to produce the value to be displayed. Keep in mind that *arglist* is a list of actual values, *not* a list of forms to be evaluated. If *arglist* is not given, it is assumed to be **nil**. It is often useful to use the backquote list-templating facility to create **:function** entries whose argument lists contain actual data objects obtained at the time **tv:scroll-parse-item** is called. See the section "Backquote-Comma Syntax".

*width*, *format-ctl*, *base*, and **nopoint* are optional, and have the same meaning as they do with **:symeval** entries.

In the shorthand forms, in which only a function is supplied, *arglist* is assumed **nil** and default assumptions about the printing format are made as for **:symeval** entries.

Here are some examples of **:function** entries:

```
(:function #'compute-number-of-items '(dogs))
(:function #'compute-number-of-items '(dogs) 6 ("~S"))
(lambda () (compute-number-of-cats))
```

**:value**

Format: **(:value** *index width* **(***format-ctl base *nopoint***))**

**:value** entries are a trick to obtain multiple results or decompose structured results from functions. Since **:function** entries can return only one value to be displayed, it is more difficult to display a complicated result, or multiple values returned by a function, than to display a single result. Scroll windows provide a one-hundred element array in which functions called by **:function** entries may store extra results. **:value** accesses elements of this array for display: *index* is a number that specifies what element of the array to access. By using this array as a temporary holding place, values computed by a **:function** entry early in the line item can be accessed by **:value** and **:function** entries later in the line item.

The array can also be accessed via the accessor **tv:value** from functions in **:function** entries. This accessor is applied to the array element index into the array **tv:value** in question. **zl:setf** may be used to store values into this array.

*width*, *format-ctl*, *base*, and **nopoint* are optional, and have the same meaning as they do with **:symeval** entries.

Here is an example of the use of a **:value** entry. We wish to display a line item that contains two constant strings and two variable fields. The line will represent the result of calling a function, **current-horse-lister**, that returns lists such as:

```
(Seabiscuit Silver Horace)
```

This function interrogates the state of some horse-processing system that is assumed to be running and continually processing horses. We wish to display on one line the number of horses currently being processed, and the actual list of their names.

A first attempt might look like

```
(tv:scroll-parse-item
        "Number of horses  :"
        '(:function (lambda ()
                      (length (current-horse-lister)))
                    5
                    ("~5D"))
        "Their names:   "
        '(:function #'current-horse-lister))
```

Although this will produce a display of the right format, it is inadequate because it calls **current-horse-lister** twice. It is possible that between the two calls to **current-horse-lister** the set of horses may have changed. Or we could be dealing with a function that has side effects, and must not be called twice if we really only want one answer. **:value** solves this problem. Here is the correct code.

```
(tv:scroll-parse-item
        "Number of horses  :"
        '(:function
             (lambda ()
               (setf (tv:value 0)
                     (current-horse-lister))
               (length (tv:value 0)))
             5 ("~5D"))
        "Their names:   "
        '(:value 0))
```

In this example, element **0** of the array is used to save the horse list between the display of the second and fourth entries in this item.

You should not use **tv:value** except for this purpose, and you should only expect its values to be saved during the display of one line item. It cannot be counted on to retain values between displays of different items, or repetitive displays of one item.

### Mouse Sensitivity

Entire line items or individual entries in a line item may be made mouse-sensitive. This means that the display corresponding to the item or entry will be highlighted as the user moves the mouse over it, and if the user clicks on it, the program controlling the scroll window will be notified.

If you want to use any of the mouse sensitivity features, you must include the flavor **tv:scroll-mouse-mixin** in the flavor of window to be used. This mixin is not included in **tv:scroll-window**. (Note: this has nothing to do with mouse scrolling; the name means that it is the flavor of the scroll facility that deals with the mouse.)

To make a line item mouse-sensitive, put a specification of the form

`:mouse` *action*

or

`:mouse-self` *action*

in the global line attributes of the line item spec when constructing the line item. *action* must be a list (actually, a cons). When a mouse-sensitive item is clicked on, the scroll window's handler, running in the mouse process, does one of the things described below, depending on the car of *action*.

If the car of *action* is **nil**, *action* is interpreted as a menu item. Clicking causes an **:execute** message is sent to the window, with *action* as its argument. Only those menu item types that produce side effects are meaningful here (that is, **:funcall**, **:eval**, **:kbd**, **:menu**, and **:buttons**). You can also use **:documentation** to provide a string to be displayed in the mouse documentation window in the who-line. Note that the car of *action* is not significant to **:execute**. For example:

```
(tv:scroll-parse-item
 ':mouse '(nil :eval (set-balance 0)
               :documentation "Set the balance to zero.")
 "Current balance:  " balance)
```

When you move the mouse over this line of the display, the entire line is highlighted, and the documentation string appears in the who line. If you click on the line, the function **set-balance** is applied to **0**.

If the car of *action* is a symbol other than **nil**, that symbol is looked up in the *type alist*, which is an association list. If the car is found, an **:execute** message is sent to the window. The argument to the message is the list

(nil *op* . *action*)

where *op* is the cadr of the entry found in the type alist for the car of *action*. The type alist can be set with the **:set-type-alist** message, or initialized with the **:type-alist** init option.

If the car of *action* is not found in the type alist (which will happen if you aren't using the alist feature) and is not **nil**, a blip of the form

(*type action window button*)

is forced into the window's input buffer. Here, *type* is the car of *action*, *window* is the window itself, and *button* is a mouse button encoding. See the section "The Character Set". This is the standard way to "read" the event of clicking on a sensitive item. The doughnut example above used this technique, putting blips of type **DOUGHNUT** in its input stream.

**:mouse-self** is just like **:mouse**, except that before returning the line item, **tv:scroll-parse-item** walks over *action*, and substitutes the actual line item that it constructed for all occurrences of the symbol **self** in *action*, so you can access its array leader. See the section "Line Item Array Leaders".

Individual entries in a line item can be made mouse-sensitive, as well. To make an entry mouse-sensitive, express it in the standard form, that is, (as opposed to the shorthand form), as follows:

```
(:string "Differential Amplifiers")
```

Then place either of the following at the head of the list:

**:mouse** *action*

or

**:mouse-item** *action*

The new entry will precede what was there before. For example:

```
(:mouse (nil :menu parts-menu
             :documentation "Pop up a menu of parts.")
   :string "Differential Amplifiers")
```

**:mouse** acts just like it does for entire line items, and *action* has precisely the same interpretation. Instead of **:mouse-self**, use **:mouse-item** to get the substitution feature: for mouse-sensitive entries, the *item* (that is, the item for the whole line) is substituted for all occurrences of the symbol **item** in *action* if **:mouse-item** is employed.

### Line Item Array Leaders

You can use the array leader of a line item for arbitrary data storage. You can use **:mouse-self** or **:mouse-item** to get the items back at mousing time. Scroll windows use the first few entries in the array leader of a line item for its own purposes. The index of the first item available for your use is stored in the variable **tv:scroll-item-leader-offset**.

To specify that you want array leader space to be reserved at line item creation time, you must use the **:leader** global line attribute. Its formats are

`:leader` *size*

`:leader` *init-list*

*size* is the amount of array leader to be reserved for your purposes, and *init-list* is a list of elements to be placed at line item creation time in as many array leader elements as they require.

### Constructing List Items

List items are normally constructed with the function **list**. The first element of a list item is the list item plist, and the rest of the elements are items that make up the list item.

Here is an example of constructing a list item for a three-line display:

```
(list ()                                ;list item plist
   (tv:scroll-parse-item ...)
   (tv:scroll-parse-item ...)
   (tv:scroll-parse-item ...))
```

The list item plist is a list of alternating keyword symbols and values. There are two defined keywords, as follows:

**:pre-process-function**

The **:pre-process-function** keyword takes any function object as an argument. This function is called at redisplay time, with the entire list item as as its one argument. Its returned value is ignored. The idea of this is to allow you to compute, at redisplay time, whether or not you still want all the items currently in the list item to remain in it, or want to add new ones and so on. Your "pre-process function" will have to walk over the cdr of the

list item, and be aware that lists therein are list items and arrays are line items in whose array leader you may have stored identifying information meaningful to you.

**:function**

(Not to be confused with the **:function** entry type in line items.) The **:function** keyword takes any function object as an argument. When it is time to redisplay this list item, the function is called to process every item of this list item, and the returned value of the function is **rplaca**'ed back into the list item before the redisplay is done. This processing occurs *after* the pre-process function, if any, has been called.

The idea of the **:function** list item property is to allow scroll window redisplay to actually cause your subsystem to update its own data. Some subsystems might want or require this, although it is very uncommon.

The function is called on three arguments: *inferior-item*, *position*, and *plist*. *inferior-item* is the particular constituent item of the list item, *position* is an internal item index, and *plist* is a locative to the list item plist of the current list item. The result of *function* is **rplaca**'ed back into the list item when *function* returns.

**Virtual List Maintenance**

An elegant facility to construct and maintain list items is provided by **tv:scroll-maintain-list**. If you intend to construct displays in which lines and subdisplays dynamically appear and disappear, you probably want to use this facility to construct and update list items. It uses the list item plist facilities described above for its implementation.

The function **tv:scroll-maintain-list** constructs (and returns to you) a list item that updates itself to represent some object of yours and its inferior objects every time the scroll window is asked to redisplay. You provide **tv:scroll-maintain-list** with two functions, one (the *init function*) that will be called at redisplay time to produce some object of yours corresponding to a set of your objects that require associated displays, and a second (the *item function*) that, given an object of yours, produces the display item (line or list) representing it.

As just described, the set of objects is expected to be a list of your objects. **tv:scroll-maintain-list** will ask for it at each redisplay, and cdr down it, applying your item function to get display items. It is also possible to return a set of your objects in some other form than a list; in this case, you must provide a *stepper function* that knows how to extract the next object, the "rest" of the set, and tell whether the end has been reached.

Here is an example of code to construct a list item that displays the contents of a Lisp list on separate lines. The variable ***important-data*** contains the list.

```
(tv:scroll-maintain-list
   #'(lambda () *important-data*)       ; The init-fun.
   #'(lambda (list-element)             ; The item-fun.
       ;; Create an item from the list element.
       (tv:scroll-parse-item
          `(:string ,(format nil "~S" list-element)))))
```

## Interactive Streams

### Introduction to Interactive Streams

An interactive stream is a bidirectional stream designed for interaction with human users. It supports input editing, which lets the user edit input before a function that reads from the stream sees it. Interactive streams are built on the flavor **si:interactive-stream**, which in turn includes one of the following mixins: **si:display-input-editor**, **si:printing-input-editor**, or **si:half-duplex-input-editor**.

To find out whether or not a stream is interactive, send the stream an **:interactive** message.

Interactive streams are generally connected to a terminal of some kind. Windows built on **tv:stream-mixin** are one kind of interactive stream (see the section "Input from Windows"). Remote terminals are another (see the section "Remote Login").

Some reading functions can be used to get input from both interactive and noninteractive streams; others are designed to read only from interactive streams. See the section "Input Functions for Interactive Streams".

Interactive streams support general operations on input and output streams. For more information on these operations, see the section "Stream Operations". Interactive streams also have specialized input operations, mainly to handle interactions with the input editor. See the section "Messages for Input from Interactive Streams". They also intercept some characters when read and maintain a list of characters to be handled asynchronously. See the section "Intercepted Characters". See the section "Interactive-Stream Operations for Asynchronous Characters". (Remote terminals do not handle asynchronous characters.)

Some interactive streams can display mouse-sensitive items. See the section "Interactive Streams and Mouse-Sensitive Items".

For information on the program interface to the input editor, see the section "The Input Editor Program Interface".

The command processor is a utility that reads commands from an interactive stream. For more information, see the section "Communicating with Genera" and see the section "Managing the Command Processor".

One common use for interactive streams is to ask a question of the user. See the section "Querying the User".

## Input Functions for Interactive Streams

The general reading functions like **zl:read**, **zl:readline**, and **zl:read-delimited-string** can be used to read from either interactive or noninteractive streams. See the section "Input Functions". The functions described here are designed to read only from interactive streams. The functions that read Command Processor commands, **cp:read-command** and **cp:read-command-or-form**, are described elsewhere:

See the section "Managing the Command Processor". See the section "Table of Basic Command Facilities". Other input functions and facilitites for interactive streams are:

**sys:read-character**
**zl:read-expression**
**zl:read-form**
**zl:*read-form-edit-trivial-errors-p***
**zl:*read-form-completion-alist***
**zl:*read-form-completion-delimiters***
**read-or-end**
**zl:read-or-character**
**zl:read-and-eval**
**zl:readline-no-echo**

## Messages for Input from Interactive Streams

All interactive streams — that is, streams that have the flavor component **si:interactive-stream** — support these input operations. Some streams have specialized versions of some operations, partly because different kinds of streams have different sources of input when input is to come from the stream instead of the input buffer. Windows, for example, take input from an I/O buffer. See the section "Messages for Input from Windows". The messages for input from interactive streams are:

**(flavor:method :any-tyi si:interactive-stream)**
**(flavor:method :any-tyi-no-hang si:interactive-stream)**
**(flavor:method :tyi si:interactive-stream)**
**(flavor:method :tyi-no-hang si:interactive-stream)**
**(flavor:method :list-tyi si:interactive-stream)**
**(flavor:method :untyi si:interactive-stream)**
**(flavor:method :listen si:interactive-stream)**
**(flavor:method :clear-input si:interactive-stream)**
**(flavor:method :line-in si:interactive-stream)**
**(flavor:method :string-in si:interactive-stream)**
**(flavor:method :string-line-in si:interactive-stream)**

### Intercepted Characters

Interactive streams specially intercept some characters. Some are intercepted when some user process is about to read the character from a stream; others are intercepted as soon as they are typed. This section describes the first kind of interception. For information on asynchronously intercepted characters, see the section "Asynchronous Characters" and see the section "Interactive-Stream Operations for Asynchronous Characters".

The value of the variable **sys:kbd-intercepted-characters** is a list of characters that are intercepted and not returned as input from the stream. These characters default to **#\ABORT**, **#\M-ABORT**, **#\SUSPEND**, and **#\M-SUSPEND**. Following are the standard actions to be taken when these characters are intercepted:

| | |
|---|---|
| **#\ABORT** | Signal **sys:abort** |
| **#\M-ABORT** | Reset the current process |
| **#\SUSPEND** | Call the **break** function |
| **#\M-SUSPEND** | Break to the Debugger |

By convention, programs are all expected to use the ABORT key as a command to abort things in some appropriate sense for that program. If you do not do anything special, ABORT is intercepted automatically. Most interactive programs just set up restart handlers for **sys:abort**. But some programs may want to do something specific when the user presses ABORT (or SUSPEND).

You can replace the system default action by binding the variable **sys:kbd-intercepted-characters**. By default, this variable is bound to the value of **sys:kbd-standard-intercepted-characters**. If you want the system to intercept only the standard abort characters, you can bind this variable to the value of **sys:kbd-standard-abort-characters**. If you want the system to intercept only the standard break characters, you can bind this variable to the value of **sys:kbd-standard-suspend-characters**.

### Interactive-Stream Operations for Asynchronous Characters

The keyboard process intercepts some characters as soon as they are typed. See the section "Asynchronous Characters". All interactive streams maintain a list of characters to be handled asynchronously. Remote terminals, however, do not handle asynchronous characters.

You can set up your own handling of asynchronous characters by using the **:asynchronous-character-p**, **:handle-asynchronous-character**, **:add-asynchronous-character**, and **:remove-asynchronous-character** messages and the **:asynchronous-characters** init option for **si:interactive-stream**.

### Interactive Streams and Mouse-Sensitive Items

Some windows support mouse sensitivity. They can display representations of items in such a way that moving the mouse onto the item causes it to be highlighted, and clicking the mouse on the item does something with the item. One example is the basic mouse-sensitive items facility. See the section "The Mouse-Sensitive Items Facility". (Note that the referenced section and the facilities discussed here are based on static windows. Mouse sensitivity is a built-in feature of Dynamic Windows and the SemantiCue input system. See the section "Using Presentation Types for Input".)

The fundamental message that creates and displays a mouse-sensitive item is **:item**. All interactive streams support this message, whether or not they support mouse sensitivity. If they do not support mouse sensitivity, they just display a printed representation of the item.

Any interactive stream can also display an ordered list of items, using the function **si:display-item-list**. This displays each item by sending an **:item** message to the stream.

## The Input Editor Program Interface

### How the Input Editor Works

The input editor is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to let you edit minor mistakes in typein. At the same time, it is not supposed to get in the way; Lisp is to see the input as soon as you have typed a syntactically complete form. The definition of "syntactically complete form" depends on the function that is reading from the stream; for **zl:read**, it is a Lisp expression. This section describes the general protocol used for communication between the input editor and reading functions such as **zl:read** and **zl:readline**.

By *reading function* we mean a function that reads a number of characters from a stream and translates them into an object. For example, **zl:read** reads a Lisp expression and returns an object. **zl:readline** reads a line of characters and returns a string as its first value. Reading functions do not include the more primitive **:tyi** and **:any-tyi** stream operations, which take and return one character or blip from the stream.

The tricky thing about the input editor is the need for it to figure out when you are all done. The idea of an input editor is that as you type in characters, the input editor saves them up in an *input buffer* so that if you change your mind, you can edit them and replace them with different characters. However, at some point the input editor has to decide that the time has come to stop putting characters into the input buffer and let the reading function start processing the characters. This is called "activating".

The right time to activate depends on the function calling the input editor, and determining it may be very complicated. If the function is **zl:read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed

representations, what all currently defined reader macros do, and so on. The input editor should not have to know how to parse the characters in the input buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The input editor interface is organized so that the calling function can do all the parsing, while the input editor does all the handling of editing commands, and the two are kept completely separate.

Following is a summary of how the input editor works. The input editor used to be called the rubout handler, and some operations and variables still have "rubout-handler" in their names.

When a reading function is called to read from a stream that supports the **:input-editor** operation, that function "enters" the input editor. It then goes ahead **:tyi**'ing characters from the stream. Because control is inside the input editor, the stream echoes these characters so the user can see the input. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the input editor it is also handing it the responsibility for echoing). The input editor is also saving all these characters in the input buffer, for reasons disclosed in the following paragraph. When the reading function decides it has enough input, it returns and control "leaves" the input editor. That was the easy case.

If you press RUBOUT or a keystroke that represents another editing command, the input editor processes the command and lets you insert characters before the last one in the line. The input editor modifies the input buffer and the screen accordingly. Then, when you type the next nonediting character at the end of the line, a **throw** is done, out of all recursive levels of **zl:read**, reader macros, and so forth, back to the point where the input editor was entered. Now the **zl:read** is tried over again, rereading all the characters you had typed and not rubbed out, but not echoing them this time. When the saved characters have been exhausted, additional input is read from you in the usual fashion.

The input editor has options that can cause the **throw** to occur at other times as well. With the **:activation** option, when you type an activation character a **throw** occurs, a rescan is done if necessary, and a final blip is returned to the reading function. With the **:preemptable** and **:command** options, a blip or special character in the input stream causes control to be returned from the input editor immediately, without a rescan. These options let you process mouse clicks or special keystroke commands as soon as they are read.

The effect of all this is a complete separation of the functions of input editing and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, **zl:read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have nontrivial side-effects, since it may be called multiple times.

If an error occurs while inside the input editor, the error message is printed and then additional characters are read. When you press RUBOUT, it rubs out the error message as well as the last character. You can then proceed to type the corrected expression; the input is reparsed from the beginning in the usual fashion.

### Invoking the Input Editor

The variable **sys:rubout-handler** indicates the current state of input editing. This variable is not **nil** if the current process is already inside the input editor.

The input editor is invoked on a stream when the stream receives an **:input-editor** message. The **:input-editor** and **:tyi** methods of **si:interactive-stream** contain the code of the input editor. The **:input-editor** method initializes the input editor, establishes its **catch**, and then calls back to the reading function with **sys:rubout-handler** bound to **:read**. When the reading function sends the **:tyi** or **:any-tyi** message, input is taken from the input buffer. If no input is available, the editing or **:tyi** portion of the input editor is invoked, and **sys:rubout-handler** is bound to **:tyi**.

The first argument to the **:input-editor** message is the function that the input editor should call to do the reading, and the rest of the arguments are passed to that function. If the reading function returns normally, the values returned by the **:input-editor** message are just those returned by the reading function. If the input editor returns by throwing out of the reading function, the return values depend on which option caused the input editor to throw: See the option **:full-rubout**. See the option **:preemptable**. See the option **:command**.

The input editor can take a series of options. These are specified dynamically by the special forms **with-input-editing-options** and **with-input-editing-options-if**. For a description of the options: See the section "Input Editor Options".

This example illustrates the use of the **:command**, **:preemptable**, and **:prompt** input editor options. It is a simple command loop that reads different kinds of commands — typed Lisp expressions, single-keystroke commands, and mouse clicks. The Lisp expressions are read using the **read-or-end** function. You can provide four kinds of input:

| *Input* | *Action* |
|---|---|
| END | Exit the command loop |
| Lisp form | Print form on next line |
| Mouse click | Display type of click and mouse coordinates |
| Single-key command | Display keystroke |

The predicate for detecting a single-keystroke command simply checks for the Super bit. In a more complex program, it might look up the character in a command table.

```
(defun command-char-p (c) (char-bit c :super))
```

```
(defun command-loop ()
  (loop
    do (multiple-value-bind (value flag)
           (with-input-editing-options
               ((:command 'command-char-p)
                (:preemptable :blip)
                (:prompt "Command loop input: "))
             (read-or-end))
         (selectq flag
           (:end
            (format t "Done")
            (return t))
           (:blip
            (selectq (car value)
              (:mouse-button
               (destructuring-bind (click nil x y) (cdr value)
                 (format t "~C click at ~D, ~D" click x y)))
              (otherwise (format t "Random blip -- ~S" value))))
           (:command
            (format t "Execute ~:C command" (second value)))
           (otherwise
            (format t "~&Value is ~S" value))))))
```

To write a reading function that invokes the input editor, you should use the **with-input-editing** special form instead of sending the **:input-editor** message directly. Such functions as **zl:read** and **zl:readline** use this special form to provide input editing.

## Input Editor Options

The input editor can take a series of options, specified by the special forms **with-input-editing-options** and **with-input-editing-options-if**. These options are listed next. Their descriptions are in *User Interface Dictionary*.

**:full-rubout** *token*
If the user rubs out all the characters that were typed, control is returned from the input editor immediately.

**:pass-through** &rest *characters*
The characters in *characters* are not to be treated as special by the input editor.

**:prompt** &rest *prompt-option*
When it is time for the user to be prompted, the input editor displays *prompt-option*.

**:reprompt** &rest *prompt-option*
When it is time for the user to be reprompted, the input editor displays *prompt-option*.

**:complete-help** &rest *help-option*
When the user presses HELP, the input editor types out a message determined by *help-option*.

**:partial-help** &rest *help-option*
When the user presses HELP, the input editor first types out a message determined by *help-option*.

**:merged-help** *function* &rest *arguments*
When the user presses HELP, the input editor types out a message determined by the arguments.

**:brief-help** &rest *help-option*
When the user presses HELP, the input editor displays a message determined by *help-option* on the same line as the typein.

**:initial-input** *string* &optional *begin end cursor-position*
When the input editor is entered, *string* is inserted into the input buffer as if the user had typed it.

**:input-history-default** *string*
Specifies *string* as the default to be yanked by c-m-Y. *string* is temporarily placed at the head of the input history.

**:blip-handler** *function*
Specifies a function to handle blips received while inside the input editor.

**:do-not-echo** &rest *characters*
The characters in *characters* are interpreted as activation characters and are not echoed.

**:activation** *function* &rest *arguments*
For each character typed, the input editor invokes *function* with the character as the first argument and *arguments* as the remaining arguments.

**:preemptable** *token*
A blip in the input stream causes control to be returned from the input editor immediately.

**:no-input-save**
The input editor does not save the scanned contents of the input buffer on the input history when returning from the reading function.

**:command** *function* &rest *arguments*
This option is used to implement nonediting single-keystroke commands.

**:editor-command** &rest *command-alist*
This option lets you specify your own input editor editing commands.

**:input-wait** &optional *whostate function* &rest *arguments*
When the input editor waits for input, it sends the stream an **:input-wait** message with the arguments to the **:input-wait** option as arguments.

**:input-wait-handler** *function* &rest *arguments*
When the input editor is waiting for input it sends the stream an **:input-wait** message.

**:suppress-notifications** *flag*
If a notification is received while in the input editor, and *flag* is supplied as **nil**, the input editor itself handles the notification, regardless of any other way you have specified that notifications should be handled.

**:notification-handler** *function* &rest *arguments*
If a notification is received while in the input editor, *function* is called to handle it.

### Displaying Prompts in the Input Editor

The input editor options **:prompt** and **:reprompt** and the functions **zl:readline-no-echo** and **sys:read-character** take *prompt* arguments that let you specify an input editor prompt. *prompt* can be **nil**, a string, a function, a symbol other than **nil**, or a list (for the input editor options, the list is an **&rest** argument):

**nil** — No prompt is displayed.

string — A **format** control string to be passed to **format** with one argument, the stream on which the prompt is displayed.

function or symbol other than **nil** — A function to display the prompt. The function should take two arguments: the first is the stream on which the prompt is displayed, and the second is a keyword that indicates the origin of the function call.

list — If the first element is **nil**, no prompt is displayed. If the first element is a string, it is a **format** control string to be passed to **format** with the remaining elements of the list as arguments. If the first element is a function or a symbol other than **nil**, it is a function to display the prompt. The first argument to the function is the stream on which the prompt is displayed. The second argument is a keyword that indicates the origin of the function call. The remaining arguments are the remaining elements of the list.

When a function is called to display the prompt, the second argument to the function is a keyword that indicates the origin of the function call:

| *Keyword* | *Function called from* |
|---|---|
| **:prompt** | **:input-editor** method of **si:interactive-stream**, when the input editor is entered |
| **:restore** | **:restore-input-buffer** method of **si:interactive-stream** |
| **:finish-typeout** | **:finish-typeout** method of **si:interactive-stream** |
| **:refresh** | Body of the input editor, when the user presses REFRESH |
| **:erase-typeout** | Body of the input editor, when the user presses PAGE |

### Displaying Help Messages in the Input Editor

The input editor options **:brief-help**, **:partial-help**, and **:complete-help** and the functions **zl:readline-no-echo** and **sys:read-character** take *help* arguments that let you specify input editor help messages. *help* can be a string, a function, a symbol, or a list (for the input editor options, the list is an **&rest** argument):

string — A **format** control string to be passed to **format** with one argument, the stream on which the help message is displayed.

function or symbol — A function to display the help message. The function should take one argument, the stream on which the help message is displayed.

list — If the first element is a string, it is a **format** control string to be passed to **format** with the remaining elements of the list as arguments. If the first element is a function or a symbol, it is a function to display the help message. The first argument to the function is the stream on which the help message is displayed, and the remaining arguments are the remaining elements of the list.

### Examples of Use of the Input Editor

This series of examples shows several different ways of using the input editor, gradually increasing in complexity. The examples are also available in the file sys: examples; interaction.lisp.

We refer to functions whose names begin with "read-" as "reading functions" or "readers", since they read individual characters and construct a Lisp object as a returned value. Examples of readers the Lisp system provides are **read**, **readline**, and **read-delimited-string**. **read** returns Lisp objects of many types. **readline** and **read-delimited-string** return strings.

**read-two-lines-1** reads two lines of input from the console. You type each line in its own editing context. After you enter the first line by pressing RETURN, LINE, or END, you can no longer rub out or otherwise edit any of the characters in the first line. You can type and edit only the second line at that point.

```
(defun read-two-lines-1 () (list (readline) (readline)))
```

**read-two-lines-2** lets you edit both lines in a single context by using the **with-input-editing** special form. Even after entering the first line you can edit it. For example, the m-< input editor command moves the cursor to the first character of the first line. **read-two-lines-2** also adds a stream parameter so that you can read from different streams without having to bind ***standard-input***. You can also use this function for reading from noninteractive streams, such as file streams.

```
(defun read-two-lines-2 (&optional (stream *standard-input*))
  (with-input-editing (stream) (list (readline stream) (readline stream))))
```

**read-two-lines-3** demonstrates the use of the **:prompt** input editor option and the **:end-activation** option for **with-input-editing**. When you invoke this function on an interactive stream you receive a prompt. This prompt is redisplayed if typeout to the stream occurs. This might happen if you press HELP or the window receives a notification.

The **:end-activation** option defines **#\end** as an activation character. This lets you activate previous input to **read-two-lines-3**, after yanking and editing it, by pressing END. The **:prompt** and **:end-activation** options have no effect on the behavior of the function for noninteractive streams.

```
(defun read-two-lines-3 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type two lines: "))
    (with-input-editing (stream :end-activation)
      (list (readline stream) (readline stream)))))
```

**read-n-lines** is like **read-two-lines** except that you specify the number of lines to be read using the **n-lines** argument. It also uses a prompt function instead of a string to generate the prompt.

```
(defun read-n-lines-prompt (stream ignore n-lines)
  (format stream "Type ~R line~:P:~%" n-lines))

(defun read-n-lines (n-lines &optional (stream *standard-input*))
  (with-input-editing-options ((:prompt 'read-n-lines-prompt n-lines))
    (with-input-editing (stream :end-activation)
      (loop repeat n-lines collect (readline stream)))))
```

Next is an example of a simple sentence parser. It builds a list of strings and symbols that represent the words and punctuation marks of the sentence. A sentence may be any number of lines long. It is delimited by a period or a question mark. Words are delimited by a space, newline, or punctuation mark. This is also an example of a reading function written entirely in terms of **:tyi** as the primitive input operation.

```
(defun read-sentence-1 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream)
      (loop named sentence
            with sentence = nil
            for word = (make-array 20. :type art-string :fill-pointer 0)
            do (loop for char = (send stream :tyi)
                     do
                 (cond ((memq char '(#\space #\return #/. #/? #/,))
                        (if (not (equal word ""))
                            (push word sentence))
                        (selectq char
                          ((#\space #\return #/,)
                           (return))
                          (#\.
                           (push :period sentence)
                           (return-from sentence (nreverse sentence)))
                          (#\?
                           (push :question-mark sentence)
                           (return-from sentence (nreverse sentence)))))
                       (t (array-push-extend word char))))))))
```

Following is a different sentence parser that calls **read-delimited-string** to accumulate characters into a string. It uses the **:end-activation** option for **with-input-editing** so that previous input to **read-sentence-2** can be yanked, edited, and activated using the END key. When it detects incorrect uses of punctuation, it calls **user::parse-ferror** to signal an error caught by the input editor.

```
(defun read-sentence-2 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream :end-activation)
      (loop with sentence = nil
            do (multiple-value-bind (word nil delimiter)
                   (read-delimited-string
                     '(#\space #\return #/. #/? #/, #/: #/;) stream)
                 (if (not (equal word ""))
                     (push word sentence))
                 (cond ((memq delimiter '(#\space #\return)))
                       ((null sentence)
                        (if (eq delimiter #\end)
                            (return nil)
                            (sys:parse-ferror
                              "The punctuation mark /"~C/" occurred at the ~
                             beginning of the sentence."
                              delimiter)))
                       ((symbolp (car sentence))
                        (sys:parse-ferror
                          "The punctuation mark /"~C/" was typed after a ~@∧."
                          delimiter (car sentence)))
                       (t (selectq delimiter
                            (#/,
                             (push ':comma sentence))
                            (#/:
                             (push ':colon sentence))
                            (#/;
                             (push ':semicolon sentence))
                            (#/.
                             (push ':period sentence)
                             (return (nreverse sentence)))
                            (#/?
                             (push ':question-mark sentence)
                             (return (nreverse sentence))))))))))
```

Sometimes an error in parsing is detected not by the function that invokes the input editor, but by some function that it calls. In the next example, **read-time** invokes **time:parse-universal-time** to do its parsing. If we did not use the **condition-case** form in **read-time**, we would enter the Debugger when **time:parse-universal-time** encountered incorrect input. The **condition-case** form encapsulates the original error in one of flavor **zl:parse-ferror** so that the input editor catches it. Alternately, we could define **time:parse-error** to be a subflavor of **sys:parse-error**.

```
(defun read-time (&optional (stream *standard-input*))
  (with-input-editing (stream :line)
    (let ((string (readline-or-nil stream)))
      (when string
        (condition-case (error)
            (time:parse-universal-time string)
          (time:parse-error
            (sys:parse-ferror "~A" error)))))))
```

### Input Editor Messages to Interactive Streams

These are the input editor messages that can be sent to interactive streams:

**(flavor:method :input-editor si:interactive-stream)**
**(flavor:method :start-typeout si:interactive-stream)**
**si:*typeout-default***
**(flavor:method :finish-typeout si:interactive-stream)**
**(flavor:method :rescanning-p si:interactive-stream)**
**(flavor:method :force-rescan si:interactive-stream)**
**(flavor:method :replace-input si:interactive-stream)**
**(flavor:method :read-bp si:interactive-stream)**
**(flavor:method :noise-string-out si:interactive-stream)**

### Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read on the stream ***query-io***, which normally is synonymous with ***terminal-io*** but can be rebound to another stream for special applications.

**y-or-n-p**
**zl:y-or-n-p**
**yes-or-no-p**
**zl:yes-or-no-p**
**fquery**
**prompt-and-read**
**define-prompt-and-read-type**

## Digital Audio Facilities

### Introduction to the Digital Audio Facilities

The 3600-family audio facilities consist of two 16-bit digital audio channels and supporting microcode. The facilities read arrays of samples from memory and feed

them to the console at a rate of 50,000 pairs of samples per second. This rate is controlled in hardware by a crystal. When active, the audio microcode reads a pair of samples from main memory every 20 microseconds, supplying one 16-bit value to each channel.

In the standard console, the samples are sent to a 12-bit digital-to-analog converter (DAC). The signal emanating from the DAC is routed to a small speaker and an 8-ohm headphone jack, as well as a low-level analog output compatible with standard "auxiliary" inputs to consumer audio equipment. In the standard console, the monaural output sound is produced by combining the two DAC channels and routing the signal through a simple two-pole low-pass filter at 8 KHz.

The audio microcode also supports a *polyphony feature*. The polyphony feature allows the use of the audio facility for the performance of music, obviating the need to generate samples for an entire performance.

The digital audio facilities are demonstrated through several code examples. See the section "Examples of Using the Audio Facilities".

The code examples are distributed in the following file:

```
SYS:EXAMPLES;AUDIO-EXAMPLES.LISP.
```

Note: the digital audio facility works only on 3600-family computers.

### Setting the Console Volume

Use this function to check and set the volume (loudness) of the console audio.

**sys:console-volume** &optional *(console* **sys:*console***)*
Returns the current volume setting for the console.

### Microcode Support for the Digital Audio Facilities

### The Audio Microtask

This section discusses the microcode interface, that is, the formats of commands and samples interpreted by the audio microcode. This is the lowest-level interface to this facility, and only the barest primitives are described here.

The audio microcode runs in its own *microtask* and thus operates parallel with the execution of Lisp. The audio microtask is either *active* or *stopped* at any time. Since the microtask scheduler works according to a priority queue, when the audio task is active, it "wakes up" every 20 microseconds, and executes, preempting Lisp, until it either outputs an audio sample pair or stops. The generation of audio samples is not affected by the behavior of Lisp programs, including the masking of process preemption, and so forth.

When active, the audio microtask follows a *command list*, or program of its own, consisting of *audio commands*, stored by the programmer in main memory before the audio microcode is started. The command list is stored in sequential *physical*

memory locations (although it can contain "jumps"). Each command occupies one or more 3600 words. The words are expected to be fixnums. The 32 data bits of each fixnum contain the data interpreted by the audio microtask. The commands include directives to control the flow of the command list as well as directives to output data to the console DAC. The audio microcode also maintains a *repeat counter* to facilitate generation of repetitive or continuous waveforms. See the section "Looping Through Audio Command Lists".

The audio microtask is started by the execution of the **%audio-start** instruction by Lisp; the evaluation of the form **(sys:%audio-start)** effects this. When this instruction is executed, the audio microtask fetches the physical address of the beginning of the command list from the variable **sys:%audio-command-pointer**. Therefore, this variable must be set to the physical address of the beginning of the command list *prior* to the execution of the form **(sys:%audio-start)**. The audio microcode stops when it encounters an explicit command to this effect in its command list.

The audio microtask is coded for real-time performance; it does no validity checking, and issues no diagnostics. If you program the audio microtask via the techniques described in this document, it is your responsibility, as always, to create valid programs. In the case of the digital audio facilities, however, the result of an invalid program could be a machine halt or destruction of the integrity of virtual memory, or both. If certain bit patterns are interpreted as audio commands, they can modify storage locations. Save your editor buffers often when debugging code for the audio microcode.

### Sample Format

Each sample pair is expected to be a fixnum. The 32 data bits of each fixnum include two samples, one for each channel. The sample pair is read by the audio microtask in one operation, and the samples are sent to each channel in parallel. Each sample is a 16-bit unsigned integer, one in the lower (bits 0-15) half word (channel 0), and one in the upper (bits 16-31) half word (channel 1).

A sample value of 0 produces the lowest analog output voltage, and a sample value of all 1s (65535, octal 177777) produces the highest. A voltage of zero is represented by the midpoint value, 32768 (octal 100000).

Channel 0 is currently supplied with analog output hardware in the console; Channel 1 is not. The digital-to-analog converter in the console is only of 12-bit precision, and thus, it ignores the low 4 bits of Channel 0 samples.

### Audio Command Format

Audio commands occupy one or more words of sequential physical memory. The command words are expected to be fixnums. The fixnum data (32 bits) for each command is described in this section.

The format of the first word of each command is as follows, described by byte specifiers in the **sys** package:

**%%audio-command-op**

A 4-bit *opcode* selecting the action to be performed by the audio microcode. Each of the currently assigned opcodes is described elsewhere. See the section "Audio Command Opcodes". See the section "Polyphony Command Opcodes".

**%%audio-command-arg**

A 28-bit quantity, whose meaning differs for each opcode. When the contents of this field, known as the *operand*, is described as an *address*, it must be a physical address. The usual way to obtain such a physical address is via the function **si:%vma-to-pma** (which does a virtual-to-physical translation). This function is given a fixnum virtual memory address. The usual way to derive such addresses, which are usually references to array element cells, is via the **%pointer** and **aloc** functions. A physical address computed from a virtual address in this way cannot be validly used unless the relevant virtual address has been wired in advance. See the section "Notes on Wired Structures".

### Audio Command Opcodes

These are the valid opcodes of audio commands, with the exception of those commands associated with the polyphony feature. See the section "The Polyphony Feature". The descriptions tell what action is performed by the audio microtask when a command having this opcode is encountered by the microtask. The opcodes are listed under the the name of the system constant (also in the **sys** package) that gives the opcode value.

**%audio-command-stop**

Causes the audio microtask to halt execution. No more commands are fetched, or samples sent to the console, until the next execution of the **sys:%audio-start** instruction. The operand is ignored.

**%audio-command-jump**

Causes the audio microtask to fetch its next instruction not from the next sequential location, but from the physical address that is the value of the operand. Sequential execution of commands continues at that physical address.

**%audio-command-load-repeat**

Loads the repeat register with the value of the operand. The operand is an unsigned 28-bit number to be loaded into the repeat register, not an address. See the description of the **%audio-command-loop** opcode for the use of this register.

**%audio-command-loop**

Decrements the repeat register by 1. If the result is greater than zero, the operand is interpreted as a jump address, and

execution of commands continues at that address, as with **%audio-command-jump**. Otherwise, if the result is less than or equal to zero, command execution continues with the next sequential command.

**%audio-command-samples**

Designates a vector of sample pairs to be sent to the console. The operand is the physical address of the first sample pair; the remaining samples are fetched from successive words of physical memory. The word in the command stream after the **%audio-command-samples** command contains a fixnum that is the count of the number of sample pairs to be fetched and sent to the console before the execution of **%audio-command-samples** terminates, and the microtask proceeds to the next sequential command. The **%audio-command-samples** command is thus a two-word command.

**%audio-command-zero**

A synchronization primitive. The operand is the *physical* address of a cell, usually an array element. The audio microcode stores a fixnum zero in that cell as the result of executing the command having the opcode **%audio-command-zero**. The software can use this facility to test if the audio microtask has passed a given point in its command list. This enables the software to ascertain when it is safe to unwire or reuse data structures containing audio commands and/or samples. It is important to remember that the audio task, when active, locks out Lisp execution until it either sends a sample or goes idle. For example, if **%audio-command-zero** is immediately followed by **%audio-command-stop**, the observation of the zeroed cell by Lisp software implies that the microtask has already read, interpreted, and executed the **%audio-command-stop**.

**%audio-command-immediate**

Designates a vector of sample pairs to be sent to the console. Unlike **%audio-command-samples**, the sample pairs appear in the command list, in consecutive physical memory locations immediately following the the **%audio-command-immediate** command word. The operand of **%audio-command-immediate** is a number, which is the count of sample pairs. That number of sample pairs is fetched from the command list and sent to the console, one every 20 microseconds (at a 50 KHz sampling rate). Execution of the command list proceeds with the next command after the vector of sample pairs, after all samples have been sent to the console.

It is critically important that the operand is equal to the number of samples provided, lest commands be interpreted as samples or vice versa.

## The Polyphony Feature

The polyphony feature of the Symbolics audio microcode provides a way to generate polyphonic music in real time. There is no need to precompute the samples and store them before playback from disk. The polyphony feature can produce six *voices*, where a voice is a rhythmically independent sequence of musical notes. Each voice can be assigned a predefined, programmer-specified waveform, which determines the spectrum and the amplitude of the notes that appear in that voice, regardless of their pitch (frequency). The waveform specification determines the shape and amplitude of *one cycle* only of the waveform. This waveform is repeated at different frequencies to produce musical tones.

The polyphony feature is not intended as a general-purpose music synthesis facility. For example, no control over the amplitude envelopes (attack, decay, and so forth) of the sounds produced is provided. The polyphony feature is intended for use in music system prototyping, that is, composition research, music editing programs, and so forth. Nevertheless, the square-envelope notes it produces are not very different from those produced by some electronic organs. When properly programmed and amplified, the digital audio facility is capable of reasonably authentic performance of much of the organ literature.

## Operation of Polyphony

The basic function of the polyphony feature is to generate, in parallel, six separate wave signals, usually of different frequencies, and sum them, at the sampling times of the audio facility. The audio microcode accomplishes this by maintaining, for each voice, a *wavetable*, a *wavetable cursor*, and an *increment*.

The wavetable for each voice consists of 1024 fixnums stored in consecutive locations in physical memory, defining the *waveform* for notes in that voice. The fixnums constitute *wave values*, which digitally describe the waveform of the voice.

The detailed interpretation of the wave values is as follows: Each fixnum wavetable element is interpreted as the algebraic sum of the wave values for the channels 0 and 1, channel 1 having been shifted 16 bits left. In detail, the value for channel 0 is a 32-bit signed (31 bits and sign, 2's complement) value between -2**15 and 2**15-1, inclusive. The value for channel 1, also in the range -2**15 to 2**15-1, is shifted left 16 bits and added algebraically to the value for channel 0. The resulting number (which is always a fixnum) is the value of the wavetable entry. Note that this is not the same format as that of audio samples used by other parts of the audio facility.

When polyphony is running (that is, when the audio microtask is interpreting the command **%audio-command-polyphony**), one value from each of the six tables is extracted, and these values are added algebraically. The resulting value is then offset by 2**15 *in each halfword*, and the resulting two halfwords are sent as audio samples to the two audio channels.

You must ensure that the sum of the values from each table never exceeds the range -2**15 to 2**15-1 for either channel. The audio microcode clips or overflows into the other channel if this range is exceeded.

Associated with each voice is also a counter/pointer called the *wavetable cursor*. This quantity is a 32-bit unsigned number. The high-order ten bits of the wavetable cursor for each voice constitute an index, which selects the entry of its wavetable to be summed into the audio sample to be produced. The low bits are used to measure the passage of time, overflowing into the high bits 1024 times per cycle of that voice.

Also associated with each voice is a quantity called an *increment*. The increment is a 32-bit fixnum. It controls the frequency, or pitch, of the note in each voice, by controlling the rate of incrementing of the wavetable cursor for that voice. When the command **%audio-command-polyphony** is being interpreted by the audio microtask, the increment for each voice is added to the wavetable cursor for that voice, and the resulting quantity is made the new wavetable cursor. (This addition is performed *after* the wavetable sample is extracted). Thus, when this repeated addition produces enough change in the value of the wavetable cursor such that the top ten bits are affected, a different wavetable entry for that voice is fetched at the next sampling time. Note that continued incrementing in this manner "wraps around". In this way, the wavetable cursor is way reset to the beginning of the wavetable, after the last entry in the wavetable has been used.

The following function (available in the **audio** package) computes the increment for a voice from the frequency:

```
(defun frequency-polyphonic-increment (frequency)
  (round (* frequency (float 1_32.)) audio:*sample-rate*))
```

You simultaneously establish the increment and wavetable location for a voice by the audio command **%audio-command-load-voice**. You instruct the polyphony facility to output samples by the audio command **%audio-command-polyphony**. This command uses all of the wavetables and increments previously established by **%audio-command-load-voice**, and outputs as many samples as requested, one every 20 microseconds, generated by summing entries from the six wavetables, incrementing the six wavetable cursors by the six associated increments as each sample is generated.

Note: changing the wavetable and/or increment for a voice does not affect any other voice in any way. Since the audio microtask is awakened by an external timer, and runs until it either outputs a sample pair or stops, no discontinuity in notes played by other voices is observed when **%audio-command-load-voice** is interpreted to change the note in one voice.

### Polyphony Command Opcodes

**%audio-command-load-voice**

Establishes a wavetable and increment for one voice of the polyphony feature The operand is the physical address of the base of the wavetable for the voice. The word in the command stream after **%audio-command-load-voice** is, in its 32 data bits, the increment for the voice. The low three (that is, the least significant) bits of this increment are the binary number

of the voice whose wavetable and increment are to be established. **%audio-command-load-voice** is effectively a two-word command.

When polyphony is being performed, the audio microcode uses, for each voice, the wavetable and increment established for that voice. There is no way to assert that a voice does not exist, or has no wavetable, or no increment. A valid wavetable and increment must be established for each of the polyphonic voices before **%audio-command-polyphony** is executed by the audio microcode, regardless of whether that voice is needed for the performance of the particular composition.

**%audio-command-load-voice** does not affect the value of the wavetable cursor for the voice involved.

**%audio-command-polyphony**

The operand is an unsigned 28-bit number. The audio microcode sends out that many samples, one each 20 microseconds, generated from the currently established wavetables of the polyphony feature. The wavetable cursors of each voice used by the polyphony feature are incremented by the increment established for that voice as each sample is sent out. The values of the increments and the wavetable cursors are not reset in any way by either the start of **%audio-command-polyphony**, or its completion.

### Simple Tone Generation with sys:%beep and sys:%slide

Use these functions to generate tones on 3600-family consoles.

**sys:%beep**
**sys:%slide**

### Notes on Wired Structures

The audio microtask fetches commands from sequential locations of physical memory. Branch addresses in the command list are physical addresses. Audio sample data pointed to by the command list are also described by physical address. Wavetables used by the polyphony feature are also described and accessed by physical address.

The audio microtask does not perform virtual address translation. Thus, the command list and sample data must be stored in data structures *wired*, or locked, in main memory. That is, they must be prevented from being paged out or moved by the Genera system. As a digital audio programmer, you must therefore be aware of page boundaries.

Audio command lists and sample vectors must be stored in wired pages consecutive in main memory, or scattered throughout main memory. If commands are stored in pages scattered throughout main memory, jumps must be programmed at the end of each page, to send the audio microcode on to the next page. If sample vectors are stored in pages scattered throughout main memory, you must use a separate **%audio-command-samples** command to describe the samples on each page. Wavetables for the polyphony feature must be in consecutive locations in main memory.

It is conventional to use Lisp arrays as the data structure containing audio commands, samples, and wavetables. Any type of array is usable for this purpose. **art-q** arrays allow one audio command or sample pair per element, and are also the only type of array whose elements can validly be addressed by the **zl:aloc** function.

### Lisp Primitives for Wiring Memory

The relevant Lisp primitives to wire data structures for the digital audio facility are **storage:wire-structure**, **storage:wire-words**, and **storage:wire-consecutive-words**. **storage:wire-words** wires any extent of virtual memory into physical memory, although the page frames into which successive pages are wired cannot be contiguous. **storage:wire-consecutive-words** also wires any extent of virtual memory into physical memory, but successive pages are guaranteed to be stored in successive page frames in physical memory. **storage:wire-structure** wires an entire structure (a convenience device to avoid having to calculate the location and extent of the virtual memory occupied by a structure) in the manner of **storage:wire-words**.

Since commands must be stored in consecutive locations in physical memory, **storage:wire-consecutive-words** suggests itself as the natural primitive for this application. However, success of this primitive depends on the availability of consecutive page frames of main memory not already containing wired pages, and it is thus less likely to succeed as more pages are wired. Use of **storage:wire-structure** and **storage:wire-words** for audio data does not encounter this problem, but requires explicit programmer handling of page boundaries, as outlined previously.

**sys:%find-structure-header** and **sys:%structure-total-size** are used to find the virtual memory location and extent of whole arrays or other structures to be wired. **storage:page-array-calculate-bounds** can be used to calculate the virtual memory location and extent of portions of array that are to be be wired, when **storage:wire-words** or **storage:wire-consecutive-words** is used. **sys:%pointer-difference** can also be used to determine the length of the extent, in words, between two addresses obtained via these primitives or the **zl:aloc** function.

Structures, or portions thereof, wired by any of these primitives, should be unwired by **storage:unwire-structure** or **storage:unwire-words** (as appropriate) only after it has been ensured (via the techniques described) that the audio microtask is not fetching commands or samples from these structures.

### Lisp Primitives for the Digital Audio Facilities

## Functions, Variables, and Macros for Digital Audio

This section describes the functions, variables, and macros available to the Lisp programmer to aid in programming the 3600-family Digital Audio Facilities. All of these objects are tools for programming the audio microtask. Therefore, this section assumes that you already understand the microcode capabilities. See the section "Microcode Support for the Digital Audio Facilities".

All of the digital audio functions, variables, and macros appear in the **audio** package. Several comprehensive examples of their use are provided in the file sys:examples;audio-examples.lisp. See the section "Examples of Using the Audio Facilities".

These Lisp tools assume the existence of an audio *command array*, in which audio microtask commands are placed, and out of which they are executed by the audio microtask. A macro (**audio:with-audio**) manages the wiring and unwiring of command arrays within the scope of a program.

A default audio command array is provided as part of these audio support primitives. All of these primitives, however, allow the specification of any suitable user-provided array as a command array. Such an array must be a nonindirect, single-dimensional **sys:art-q** array, with a fill pointer, allocated in a static area (such as **audio:audio-area**).

Command arrays, as all arrays, are finite in extent. Carefully planned synchronization techniques must be utilized to allow uninterrupted sound to be produced from a single command array that is being serially reused for sequences of audio commands. See the section "Examples of Using the Audio Facilities".

### Digital Audio Parameters

These are the critical constants of the audio facility.

**audio:*sample-rate***
**audio:*number-of-polyphonic-voices***
**sys:%%audio-increment-integer**

### Testing for the Existence of Audio

Use this variable to test for the existence of audio on a 3600-family machine.

**audio:audio-exists**

### The Audio Wrapping Form

Use this macro to prepare an audio command array. The macro globally binds scheduler parameters to allow process generating audio commands to gain control when necessary.

**audio:with-audio**

### Building Audio Command Lists

The functions listed in this section prepare arguments for, build, and store audio commands in a command array. They assume that the fill pointer of the array describes the next available location in the array, and they update the fill pointer as needed. The array must be wired, as some of these functions compute and store physical addresses of locations in the command array. Calling these functions does not produce sound. Sound is produced when the audio facility is directed (via **audio:audio-start**) to a command list produced by calling these functions.

The fill pointer of the array defines a logical pointer called the *audio index*. The function **audio:audio-index** (which defines a location accessible with **zl:setf**) is used to access this index (for example, for use as an argument to a later function call).

The current implementation uses command arrays that are wired into successive, contiguous page frames of physical memory. The exclusive use of these primitives hides this implementation detail. Do not perform calculations on audio indices. Instead, request them whenever needed via **audio:audio-index**, and use them only as arguments to the primitives provided.

Use of the macro **audio:with-audio** is the recommended way to establish the proper context in which these functions can be validly used. Each of them takes an optional argument, which specifies the command array in question. This argument always defaults to the facility's default command array.

**audio:audio-index**
**audio:audio-room**
**audio:audio-limit**
**audio:push-audio-jump**
**audio:push-audio-zero-flag**
**audio:push-audio-load-voice**
**audio:push-audio-polyphony**
**audio:modify-audio-command-arg**

### Storing Samples

The functions and macros described in this section place audio sample pairs into the command program. These commands can be either immediate (**%audio-command-immediate**) or stored elsewhere (**%audio-command-samples**).

**audio:push-array-of-audio-samples**
**audio:computing-immediate-audio-samples**
**audio:push-immediate-audio-sample**

### Looping Through Audio Command Lists

These two macros facilitate the use of **%audio-command-loop** to create loops in audio command lists. Keep in mind that the audio microcode does not support nested loops.

**audio:audio-loop**
**audio:set-audio-repeat-count**

### Synchronization Flags

These functions allocate, in the command array specified, locations to be used as synchronization flags (for **%audio-command-zero**), and allow the flags to be waited for and reset. The "reset", or "normal", state of these flags, is non-zero. The audio microcode "sets" them, by setting them to zero, when a **%audio-command-zero** is executed. By means of these flags, the real-time progress of the audio microtask can be monitored.

**audio:reserve-audio-flags**
**audio:wait-for-audio-flag**

### Starting and Stopping the Audio Microtask

These functions are used to start and stop the audio microtask.

**audio:audio-start**
**audio:audio-stop**

### Conversions Between Sample Formats

The following functions encode and decode sample pairs. They are provided to hide the internal representation of sample pairs. Some of these "functions" are actually implemented as macros to help make code that prepares audio samples as fast as possible.

These functions convert between three formats of samples, *float*, *fixnum*, and *sample*. Float and fixnum formats describe channel values. Sample format is the actual format of sample pairs stored in command arrays and sample arrays.

Fixnum format consists of integers in the range -1**15 $\le x <$ 1**15. Float format consists of floating numbers and float channels are in the range -1.0 $\le x <$ 1.0. You must ensure that a float format value is never +1.0.

**audio:float-channel-fix**
**audio:fix-channel-float**
**audio:fix-sample**

**audio:float-sample**
**audio:sample-channels**
**audio:sample-add-fix**
**audio:sample-add-float**
**audio:sample-add-sample**

### Conversions for the Polyphony Feature

These functions convert between fixnum and float format channel values and the values stored in wavetables used by the polyphony feature. See the section "The Polyphony Feature".

**audio:fix-polyphonic-wave-table-entry**
**audio:float-polyphonic-wave-table-entry**
**audio:polyphonic-wave-table-entry-channels**

### Computing Polyphonic Increments

This function computes the appropriate wavetable increment to specify the frequencies in polyphonic textures.

**audio:frequency-polyphonic-increment**

### Examples of Using the Audio Facilities

This chapter presents seven program examples that use the digital audio facilities, in both real-time and non-real-time synthesis applications.

### Sine Wave Example

This example generates a sine wave at a specified frequency.

```
(defun sine-wave (frequency)
  (audio:with-audio ()                  ;Set up the audio environment
    (let* ((start (audio:audio-index))  ;Get the current (starting) index
           (samples-per-cycle (round audio:*sample-rate* frequency))
           ;; Spread out several cycles to get a more accurate
           ;; frequency.  Extra factor of 2 makes sure there is room.
           (number-of-cycles (max 1 (floor (audio:audio-limit) (* samples-per-cycle 2))))
           ;; Actual number of samples we are going to produce
           (number-of-samples (* samples-per-cycle number-of-cycles)))
      ;; Make sure we have room to play this frequency
      (when (> (+ number-of-samples 2) (audio:audio-limit))
        (ferror "Frequency too low"))
      ;; This form allows us to compute number-of-samples inline
      ;; (as opposed to computing them in a separate array). If we
      ;; didn't know how many samples we were going to produce we could
      ;; supply NIL for number-of-samples and the form will keep track
      ;; and adjust the command array when the form is exited.  Since we
      ;; do supply the number of samples, the form will check to make
      ;; sure we supply exactly that many.  This helps us to avoid writing
      ;; incorrect audio programs.
      (audio:computing-immediate-audio-samples (number-of-samples)
        (loop for sample-number below number-of-samples
              as phase =
                 ;; This is the phase (angle) that is passed to sin
                 ;; to get the sine wave.  (This will cons double-floats in
                 ;; systems where cl:pi is a double-float.)
                 (// (* 2 cl:pi sample-number number-of-cycles)
                     number-of-samples)
              as sample =
                 ;; Take the sin of the phase.  Also multiply it
                 ;; by something less than 1 so we never get a
                 ;; value of 1.0 (a restriction, see
                 ;; documentation).  Take the resulting floating
                 ;; point number in the range [-1.0, +1.0) and
                 ;; create a 'sample.'
                 (audio:float-sample (* (sin phase) 0.9))
              do ;; Now actually push the sample into the command array.
          (audio:push-immediate-audio-sample sample)))
      ;; All of the samples are computed and an appropriate command has
      ;; been generated to output them.  Now we cause a jump back to the
      ;; beginning to keep the sound going.
      (audio:push-audio-jump start)
      ;; The program is complete, we can now start the audio facility.
      (audio:audio-start start)
      ;; When you've heard enough, just type anything.  with-audio
      ;; supplies code to turn off the audio facility when exited and do
      ;; other bookkeeping.
      (tyi))))
```

### Sawtooth Wave Example

This is roughly the same as sine wave, but instead produces a sawtooth and only generates one cycle for it.

```
(defun saw-wave (frequency)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (round audio:*sample-rate* frequency)))
      (audio:computing-immediate-audio-samples (samples-per-cycle)
        (loop for sample-number below samples-per-cycle
              as value =
                 ;; create a sawtooth value in the range [-1.0,1.0).
                 ;; Note this can never be exactly 1.0 since
                 ;; sample-number never quite gets as large as
                 ;; samples-per-cycle.
                 (- (// (* 2.0 sample-number) samples-per-cycle) 1.0)
              do (audio:push-immediate-audio-sample (audio:float-sample value)))
        (audio:push-audio-jump start)
        (audio:audio-start start)
        (tyi)))))
```

### Square Wave Example

This example demonstrates yet another type of waveform: a square wave. The **audio:audio-loop** form is also exemplified.

```
(defun square-wave (frequency)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (round audio:*sample-rate* frequency))
           ;; Compute the number of samples for the high value and
           ;; low value.  Divide them as evenly as possible.
           (samples-first-half (// samples-per-cycle 2))
           (samples-second-half (- samples-per-cycle samples-first-half)))
      ;; Create a loop that will repeat samples-first-half times.  If we
      ;; weren't sure how many times we want to repeat, we could specify
      ;; NIL and then use set-audio-repeat-count to set the count.
      (audio:audio-loop  (samples-first-half)
        ;; Compute 1 value (the high value) for output.
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample 0.9))))
      ;; Do the same for the second half.
      (audio:audio-loop (samples-second-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample -0.9))))
      ;; Jump back to the beginning so we get more than one cycle.
      (audio:push-audio-jump start)
      (audio:audio-start start)
      (tyi))))
```

### Beep Example

This is basically a modified square-wave.

```
(defun %beep-ignoring-most-issues (frequency duration)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (sys:round audio:*sample-rate* frequency))
           (samples-first-half (// samples-per-cycle 2))
           (samples-second-half (- samples-per-cycle samples-first-half)))
      ;; Can't nest loops, so we have to do the outer loop with a jump
      ;; and bash the location when time has elapsed.
      (audio:audio-loop (samples-first-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample 0.9))))
      (audio:audio-loop (samples-second-half)
        (audio:computing-immediate-audio-samples (1)
```

```
    (audio:push-immediate-audio-sample (audio:float-sample -0.9))))
;; This is the tricky part.  We need to put a jump to the
;; beginning, but we need to know where it is so we can cause it
;; to fall through.  We also need a flag so we know when the audio
;; has stopped so we can exit.  If we simply exited without
;; waiting, the with-audio form could turn off the sound prematurely.
(let* (;; get the index that we will eventually bash and put in a
       ;; jump back to the start.
       (jump-index (prog1 (audio:audio-index) (audio:push-audio-jump start)))
       ;; reserve (and reset) an audio flag.
       (flag-index (audio:reserve-audio-flags 1))
       ;; reserve-audio-flags puts in a jump command around the
       ;; flags it reserves, so we could have gotten the
       ;; fall-through index after pushing the jump command.
       ;; Anyway, get the index of the fall-through location.
       (fall-through-index (audio:audio-index)))
  ;; When we bash the jump command the microcode will jump to here
  ;; instead, which will cause the flag to get zeroed and the
  ;; audio facility to stop.  Both events happen atomically as far
  ;; as Lisp can tell because no samples are output in the
  ;; intervening time.
  (audio:push-audio-zero-flag flag-index)
  (audio:push-audio-stop)
  ;; Start the audio
  (audio:audio-start start)
  ;; Wait the appropriate number of microseconds.
  (loop with start-time = (sys:%microsecond-clock)
        until
        (≥ (%32-bit-difference (sys:%microsecond-clock) start-time) duration))
  ;; Here is where we bash the argument of the jump command to
  ;; instead jump to the fall-through code.
  (audio:modify-audio-command-arg fall-through-index :index jump-index)
  ;; Wait for the microcode to get to the flag and stop before we exit.
  (audio:wait-for-audio-flag flag-index "%BEEP")))))
```

### Non-real-time Synthesis Example

Certain kinds of very high quality sound cannot be generated in real time (one sample computed every 20 microseconds). Small pieces (pieces that can fit in physical memory) can be computed and then played later.

```
(defun play-audio-sample-array
  (array &optional (from 0) (to (array-active-length array)))
  (audio:with-audio ()
    ;; with-wired-structure wires the structure on entry
    ;;  and unwires on exit. External sample arrays must be wired.
    (si:with-wired-structure array
      (let* ((flag-index (audio:reserve-audio-flags 1))
             (start (audio:audio-index)))
        ;; Cause the samples to be played.  If we supplied a non-NIL
        ;; immediate-p argument, we wouldn't have to wire the
        ;; structure, since the samples would be put in the command
        ;; array which is already wired.  However, most command arrays
        ;; are not very large and probably couldn't hold all the
        ;; samples.  It's a tradeoff.
        (audio:push-array-of-audio-samples array from to)
        ;; When the microcode finishes the samples, cause it to clear
        ;; the flag and stop.
        (audio:push-audio-zero-flag flag-index)
        (audio:push-audio-stop)
        ;; Start it up and wait for it to finish.
        (audio:audio-start start)
        (audio:wait-for-audio-flag flag-index "Play samples")))))
```

## Playing Large Pieces Example

Larger pieces (those that are too big to fit in physical memory) can still be played. This program plays data that is stored on the FEP filesystem. Storage must be on the FEP filesystem for several reasons. The digital audio system must produce data at the rate of one sample every 20 microseconds (including all overhead). This is 1.6 megabits per second, which is a small factor away from raw disk speed. After overhead, this is getting close to the limits of the system. The LMFS file system incurs too much overhead. Also, we cannot copy (as LMFS would try to do if we used **:string-in** into an array) and we cannot spend time wiring buffers (as we would need to do with LMFS if we used **:read-input-buffer**).

The FEP filesystem allows us to do disk direct memory access (DMA) directly into a buffer that we can keep wired. We can also setup the audio facility to point to these buffers (using **push-array-of-audio-samples**) once so we do not have to do it often.

The macro **with-multi-disk-buffering** takes care of multibuffering bookkeeping. The user decides how many pages to devote to each buffer and the number of buffers. Disk arrays (the buffers) are allocated and wired on entry and unwired on exit.

```
(defmacro with-multi-disk-buffering
    ((npages nbuffers) (array-of-buffers size-of-each-buffer) &body body)
  "npages and nbuffers are inputs, array-of-buffers and size-of-each-buffer are outputs"
  `(let ((,array-of-buffers (make-array ,nbuffers))
         (,size-of-each-buffer (* ,npages 288.)))
     (unwind-protect
       (progn (loop for .idx. below ,nbuffers
                    as .buffer. = (allocate-resource 'si:disk-array
                                                     (+ ,size-of-each-buffer 288.))
                    do (setf (aref ,array-of-buffers .idx.) .buffer.)
                    (si:wire-structure .buffer.))
              ,@body)
       (loop for .idx. below ,nbuffers
             as .buffer. = (aref ,array-of-buffers .idx.)
             do (when (si:structure-wired-p .buffer.)
                  (si:unwire-structure .buffer.))
             (deallocate-resource 'si:disk-array .buffer.)))))
```

The function **play-disk-file** is the workhorse. There are many "if we are fast enough" clauses in this example. As long as there is not much other activity (especially paging activity) we usually are fast enough.

```
(defun play-disk-file (pathname)
  (setq pathname (fs:merge-pathnames pathname "FEP:>↔.mus.newest"))
  ;; get the FEP file opened.
  (with-open-file (file pathname :direction :block
                        :if-exists :overwrite
                        :if-does-not-exist :error)
    ;; These numbers were picked after much experimentation and tuning.
    (let* ((npages 40.) (nbuffers 8))
      (audio:with-audio ()
        (with-multi-disk-buffering (npages nbuffers) (buffers buffer-size)
          ;; allocate a flag for each buffer for synchronization.
          (let* ((flags (audio:reserve-audio-flags nbuffers))
                 (start (audio:audio-index)))
            ;; build the audio program.  Push each buffer as an array of
            ;; samples and then cause the flag associated with the
            ;; buffer to be zeroed.
            (loop for buffer below nbuffers
                  do (audio:push-array-of-audio-samples (aref buffers buffer)
                       0 buffer-size) (audio:push-audio-zero-flag
                       (+ flags buffer)))
```

```
;; Loop back to the beginning.  To play new data (if we are
;; fast enough, there /will/ be new data in the buffers).
(audio:push-audio-jump start)
;; n-queued is the number of buffers filled with valid data
;; that the microcode can use.  (The microcode will use
;; all of them, but if we are fast enough we can keep them full.)
;; We fill up all the buffers and then start the audio facility.
;; This is done by an interaction with need-to-start and n-queued.
;; (There is also provision for small files.)  When all the buffers
;; are queued, we need to wait for the microcode to finish
;; the next one before we can do disk dma into it.
(loop with n-queued = 0
      with need-to-start = t
      with n-file-blocks = (sys:ceiling (send file :length) 1152.)
      with current-file-block = 0
      initially (format t "~&~F seconds~%"
                   (// (* n-file-blocks 288.) audio:*sample-rate*))
      as blocks-this-whack =
         ;; This is the number of blocks to do this time
         ;; around.  It is at most the number of pages of
         ;; buffering.  It is also at most the number of
         ;; blocks remaining in the file.
         (min npages (- n-file-blocks current-file-block))
      for buffer-number =
          ;; This is the current buffer number we are going
          ;; to try to fill.  It is gets incremented modulo
          ;; the number of buffers.
          0 then (\ (1+ buffer-number) nbuffers)
      as flag-index = (+ flags buffer-number)
      do ;; If all the buffers are queued, or if the end of
         ;; the file has been reached, wait for the
         ;; microcode to finish the buffer and then count it
         ;; as dequeued.
         (when (or (= n-queued nbuffers) (zerop blocks-this-whack))
           (audio:wait-for-audio-flag flag-index "Play disk file")
           (decf n-queued))
         ;; If we have some blocks to queue, make sure the
         ;; flag for this buffer is reset, read in the
         ;; blocks from the FEP file, increment the block
         ;; pointer into the file, and count another buffer
         ;; as queued.
         (when (not (zerop blocks-this-whack))
           (audio:reset-audio-flag flag-index)
           (send file :block-in current-file-block blocks-this-whack
                 (aref buffers buffer-number))
           (incf current-file-block blocks-this-whack)
           (incf n-queued))
```

```
        ;; If the audio facility hasn't been started and
        ;; all buffers are filled, start the audio facility
        ;; (and remember we did start it).
        (when (and need-to-start
                   (or (= n-queued nbuffers)
                       (≥ current-file-block n-file-blocks)))
          (audio:audio-start start)
          (setq need-to-start nil))
   until
     ;; We are finished when nothing is queued and we are
     ;; at the end of the file.
     (and (zerop n-queued)
          (≥ current-file-block n-file-blocks))))))))
```

**Polyphony Example**

This is a simple muse. It uses roughly the same multibuffering strategy as the disk example, so that portion will not be commented as heavily. (See the section "Playing Large Pieces Example".) The muse muses some number of voices (user specified) between 1 and 6. All voices start at DO (C). Each step (approximately every 1/4 second) causes each voice to wander randomly between 2 diatonic tones below the previous value and 2 diatonic tones above the previous value.

```
;;; Figure out how large wave tables are in this release.

(defconst *samples-per-polyphonic-wave-table*
          (expt 2 (byte-size sys:%%audio-increment-integer)))

;; This is the wave-array for the muse.
;; It is big enough to ensure that there will be at least
;; *samples-per-polyphonic-wave-table* consecutive wired words.

(defvar *muse-wave-array*
        (make-array (+ *samples-per-polyphonic-wave-table* sys:page-size -1)
                              :initial-value 0 :area audio:audio-area))
```

```
(defun polyphonic-muse (&optional (n-voices 4) &aux address wired)
  (check-arg n-voices (and (fixp n-voices)
        (≤ 1 n-voices audio:*number-of-polyphonic-voices*))
             "an integer between 1 and 6")
  (audio:with-audio ()
    (unwind-protect
      (let ((offset-to-page
              ;; This is how one gets to the number of Qs
              ;;  to the beginning of a page boundary
              (ldb sys:%%vma-word-offset
                   (- sys:page-size
                      (ldb sys:%%vma-word-offset
                      (%pointer (locf (aref *muse-wave-array* 0))))))))
        ;;; Wire words of the wave table, starting at
        ;;;  the location computed above.
        (setq address (locf (aref *muse-wave-array* offset-to-page)))
        (si:wire-consecutive-words
          address                               ;where
          *samples-per-polyphonic-wave-table*)  ;how many, one per word.
        (setq wired t)                          ;Set a reminder to unwire it...
```

```
;; Set up the muse wave array for a 1/6 (minus a bit) amplitude
;;  sinewave (sawtooth doesn't seem to sound good here).  1/6
;;  allows all six voices to proceed without overflow.  The
;;  "minus a bit" avoids clipping at 1.0.
(loop for index below *samples-per-polyphonic-wave-table*
      do (setf (aref *muse-wave-array* (+ index offset-to-page))
               (audio:float-polyphonic-wave-table-entry
                 (// (sin (// (* 2.0 si:pi index)
                              *samples-per-polyphonic-wave-table*)) 6.2))))
;; Initialize each voice to a reasonable value.  It is essential
;; that each voice gets a proper wave-array pointer and
;; increment value.  An increment value of 0 will cause the
;; pointer never to be incremented.  (This isn't strictly true,
;; since the voice number is stored in the low 3 bits, but this
;; advances the pointer very slowly.)
(let ((start (audio:audio-index)))
  (loop for voice below audio:*number-of-polyphonic-voices*
     do
  (audio:push-audio-load-voice voice *muse-wave-array* offset-to-page 0))
  (audio:push-audio-stop)
  (audio:audio-start start)
  ;; put the audio index back to the start
  (setf (audio:audio-index) start))
(loop with nbuffers = 4
      with n-queued = 0
      with need-to-start = t
      with flags = (audio:reserve-audio-flags nbuffers)
      with start = (audio:audio-index)
      with chords-per-whack =
        ;; Take the room remaining, divide by the level of
        ;; buffering and then divide by the sum of [2 locations
        ;; per voice for the push-audio-load-voice command, one
        ;; for the push-audio-polyphony command, and one for a
        ;; possible flag or jump].
        (// (audio:audio-room) nbuffers (+ (* n-voices 2) 1 1))
      with half-tone-offsets =
        ;; 0 (and the multiples of 12) are DO.  The other
        ;; numbers are offsets (from 0) to consecutive notes in
        ;; the diatonic scale.
        '(-25. -24. -22. -20. -19. -17. -15. -13.
               -12. -10. -08. -07. -05. -03. -01.
               000. +02. +04. +05. +07. +09. +11.
               +12. +14. +16. +17. +19. +21. +23.
               +24. +26. +28. +29. +31. +33. +35.)
      with half-tone-offsets-length = (length half-tone-offsets)
```

```
        with voice-indices =
          ;; A list, one element for each voice, starting at middle DO.
          (make-list n-voices
             :initial-value (find-position-in-list 000. half-tone-offsets))
        for buffer-number = 0 then (\ (1+ buffer-number) nbuffers)
        until (kbd-tyi-no-hang)   ; Stop when user hits a key
        do
    (when (≥ n-queued nbuffers)
      ;; this also resets the flag
      (audio:wait-for-audio-flag (+ flags buffer-number) "Muse")
      (decf n-queued))
    ;; If this is buffer zero, make sure we are back to the start.
    (when (zerop buffer-number)
      (setf (audio:audio-index) start))
    ;; setup the chords for this buffer
    (loop repeat chords-per-whack
          do ;; update each voice
             (loop for voice-indices-scan on voice-indices
                   as old-index = (car voice-indices-scan)
                   as new-index =
                          (let ((index (+ old-index (random 5) -2)))
                               ;; clip at the boundaries of the list
                               (cond ((< index 0) 1)
                                     ((≥ index half-tone-offsets-length)
                                             (- half-tone-offsets-length 2))
                                      (T index)))
                   do (setf (car voice-indices-scan) new-index))
             ;; And queue the new values to polyphony facility
             (loop for index in voice-indices
                   for voice-number upfrom 0
                   as half-tone-offset = (nth index half-tone-offsets)
                   as octave-offset = (// half-tone-offset 12.0)
                   as frequency-factor = (expt 2.0 octave-offset)
                   as frequency = (* 256.0 frequency-factor)
                   do (audio:push-audio-load-voice
                        voice-number *muse-wave-array* offset-to-page
                        (audio:frequency-polyphonic-increment frequency)))
             ;; Do polyphony for 1/4 second
             (audio:push-audio-polyphony (sys:round audio:*sample-rate* 4)))
    ;; synchronize this buffer
    (audio:push-audio-zero-flag (+ flags buffer-number))
    (incf n-queued)
    (when (and (≥ n-queued nbuffers) need-to-start)
      (audio:push-audio-jump start)
      (audio:audio-start start)
      (setq need-to-start nil))))
(when wired
  (si:unwire-words address *samples-per-polyphonic-wave-table*)))))
```

# Dates and Times

## Representation of Dates and Times

The **time** package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing a Symbolics machine's microsecond timer.

Times are represented in two different formats by the functions in the **time** package. One way is to represent a time by many numbers, indicating a year, a month, a date, an hour, a minute, and a second (plus, sometimes, a day of the week and time zone). The year is relative to 1900 (that is, if it is 1984, the *year* value would be **84**); however, the functions that take a year as an argument will accept either form. The month is 1 for January, 2 for February, and so on. The date is 1 for the first day of a month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A time zone is specified as the number of hours west of GMT; thus in Massachusetts the time zone is 5. Any adjustment for daylight saving time is separate from this.

This "decoded" format is convenient for printing out times in a readable notation, but it is inconvenient for programs to make sense of these numbers, and pass them around as arguments (since there are so many of them). So there is a second representation, called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This "encoded" format is easy to deal with inside programs, although it doesn't make much sense to look at (it looks like a huge integer). So both formats are provided; there are functions to convert between the two formats; and many functions exist in two forms, one for each format.

Symbolics hardware includes a timer that counts once every microsecond. It is controlled by a crystal and so is fairly accurate. The absolute value of this timer doesn't mean anything useful, since it is initialized randomly; what you do with the timer is to read it at the beginning and end of an interval, and subtract the two values to get the length of the interval in microseconds. These relative times allow you to time intervals of up to an hour (32 bits) with microsecond accuracy.

Symbolics keeps track of the time of day by maintaining a "timebase", using the microsecond clock to count off the seconds. When the machine first comes up, the timebase is initialized by querying hosts on the local network to find out the current time.

A similar timer counts in 60ths of a second rather than microseconds; it is useful for measuring intervals of a few seconds or minutes (or hours, which are longer than the microsecond timer's range) with less accuracy. Periodic housekeeping functions of the system are scheduled based on this timer.

## Getting and Setting the Time

Use these functions for getting and setting the time.

**get-decoded-time** Returns the current time in decoded time format.

**time:get-time** Gets the current time, in decoded form.

**get-universal-time** Returns the current time, in Universal Time form.

**time:set-local-time** &optional *new-time*
Set the local time to *new-time*.

### The 3600-family Calendar Clock

3600 family and XL400 machines have a calendar clock that operates independently of the other hardware timers. When you cold boot and the machine fails to get the time from the network, it asks you to type in the time. If the calendar clock has been set, it uses the calendar clock reading as the default for the time you specify. If the calendar clock has not been set, it offers to set it to the time you type in. See the function **time:initialize-timebase**.

You can also set the calendar clock yourself using **time:set-calendar-clock** and read it using **time:read-calendar-clock**.

Embedded systems use the clock on the embedding host as a calendar clock. Additionally, UX400 systems use the clock on the Ivory board as a backup for the clock on the embedding host.

### Elapsed Time in Seconds

Rather than calendrical date/times, the following functions deal with elapsed time in seconds, 60ths of seconds, or microseconds. These functions are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

**time**
**zl:time**
**time-lessp**
**time-difference**
**time-increment**
**time-elapsed-p**
**time:microsecond-time**
**time:fixnum-microsecond-time**

### Elapsed Time in Internal Time Units

One internal time unit is 1024 microseconds. These three functions use this internal time unit:

**get-internal-real-time**
**get-internal-run-time**
**internal-time-units-per-second**

### Printing Dates and Times

The functions in this section create printed representations of times and dates in various formats, and send the characters to a stream. To any of these functions, you may pass **nil** as the *stream* parameter, and the function will return a string containing the printed representation of the time, instead of printing the characters to any stream.

**time:print-current-time** &optional (*stream* **standard-output**)
Prints the current time, formatted as in **11/25/83 14:50:02**, to the specified stream.

**time:print-time** *seconds minutes hours day month year* &optional (*stream* **standard-output**)
Prints the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

**time:print-universal-time** *ut* &optional (*stream* **standard-output**) *timezone*
Prints the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

**time:print-current-date** &optional (*stream* **standard-output**)
Prints the current time, formatted as in **Friday the twenty-fifth of November, 1988; 3:50:41 pm**, to the specified stream.

**time:print-date** *seconds minutes hours day month year day-of-the-week* &optional (*stream* **standard-output**)
Prints the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

**time:print-universal-date** *ut* &optional (*stream* **standard-output**) *timezone*
Prints the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

**time:print-brief-universal-time** *ut* &optional (*stream* **standard-output**) (*ref-ut* **(zl-user:get-universal-time)**)
Like **time:print-universal-time** except that it omits seconds and only prints those parts of *ut* that differ from *ref-ut*, a universal time that defaults to the current time.

**zl:format** accepts some directives for printing dates and times.

### Reading Dates and Times

The functions listed in this section accept most reasonable printed representations of date and time and convert them to the standard internal forms. The following are representative formats that are accepted by the parser:

```
"March 15, 1960" "15 March 1960" "3//15//60" "3//15//1960"
"3-15-60" "3-15" "15-March-60" "15-Mar-60" "March-15-60"
"1960-3-15" "1960-March-15" "1960-Mar-15"
"1130." "11:30" "11:30:17" "11:30 pm" "11:30 am" "1130" "113000"
"11.30" "11.30.00" "11.3" "11 pm" "12 noon"
"midnight" "m" "Friday, March 15, 1980" "6:00 gmt" "3:00 pdt"
"15 March 60" "15 March 60 seconds"
"fifteen March 60" "the fifteenth of March, 1960;"
"one minute after March 3, 1960"
"two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"
"now" "today" "yesterday" "two days after tomorrow"
"one day before yesterday" "the day after tomorrow"
"five days ago"
```

The parsing functions accept date strings in ISO standard format. These strings are of the form `"yyyy-mm-dd"`, where:

| | |
|---|---|
| `yyyy` | Four digits representing the year |
| `mm` | The name of the month, an abbreviation for the month, or one or two digits representing the month |
| `dd` | One or two digits representing the day |

Following are some restrictions on strings to be parsed:

- You cannot represent any year before 1900.
- A four-digit number alone is interpreted as a time of day, not a year. For example, `"1954"` is the same as `"19:54:00"` or `"7:54 pm"`, not the year 1954.
- The parser does not recognize dates in European format. For example, `"3//4//85"` or `"3-4-85"` is always the same as `"March 4, 1985"`, never `"April 3, 1985"`. A string like `"15//3//85"` is an error. In such strings, the first integer is always parsed as the month and the second integer as the day.

Here are the functions:

**time:parse** *string* &optional (*start* **0**) *end* (*futurep* **t**) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* **t**)
Interprets *string* as a date and/or time, and return seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and relative-p.

**time:parse-universal-time** *string* &optional (*start* **0**) *end* (*futurep* **t**) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* **t**)
The same as **time:parse** except that it returns one integer, representing the time in Universal Time, and the *relative-p* value.

**time:parse-universal-time-relative** *date-spec reference-date-spec* &optional (*future-p* **t**)

Like **time:parse-universal-time**, except that *date-spec* is parsed relative to *reference-date-spec*.

**time:parse-present-based-universal-time** *time-being-parsed*

Like **time:parse-universal-time**, except that missing components in *time-being-parsed* are defaulted to the beginning of the smallest unsupplied unit of time.

### Reading and Printing Time Intervals

Several functions read and print time intervals. They convert between strings of the form "3 minutes 23 seconds" and integers representing numbers of seconds.

**si:parse-interval-or-never** *string*

Returns an integer if *string* represents an interval, or **nil** if *string* represents "never".

**time:print-interval-or-never** *interval* &optional (*stream* **zl:standard-output**)

Prints the representation of *interval* as a time interval onto *stream*.

**sys:read-interval-or-never** &optional *stream or-nil*

Reads a line of input from *stream* (using **zl:readline**) and calls **si:parse-interval-or-never** on the resulting string.

### Time Conversions

Use these functions and this variable to convert between universal time and decoded time.

**decode-universal-time** *universal-time* &optional *timezone*

Given a universal time, returns nine values for the corresponding decoded time: second (0-59); minute (0-59); hour (0-23); date (1-31); month (1-12); year (A.D.); day-of-week (0[Monday]-6[Sunday]); a flag (**t** or **nil**) indicating whether daylight savings time is in effect; and the timezone (hours west of GMT).

**time:decode-universal-time** *universal-time* &optional *timezone*

Converts *universal-time* into its decoded representation.

**encode-universal-time** *seconds minutes hours date month year* &optional *timezone*

Given a time in decoded time format, returns the corresponding universal time (plus or minus seconds since midnight, January 1, 1900 GMT).

**time:encode-universal-time** *seconds minutes hours day month year* &optional *timezone*

Converts the decoded time into Universal Time format and returns the Universal Time as an integer.

**time:*timezone*** The time zone in which this Symbolics machine resides, expressed in terms of the number of hours west of GMT this time zone is.

### Internal Time Functions

These functions provide support for functions that deal with dates and time. Some user programs might need to call them directly, so they are listed here.

**time:initialize-timebase** &optional *ut* (*use-network* **t**)
Initializes the timebase.

**time:daylight-savings-p**
Returns **t** if daylight savings time is currently in effect.

**time:month-length** *month year*
Returns the number of days in *month*; you must supply a *year* in case the month is February.

**time:leap-year-p** *year*
Returns **t** if *year* is a leap year.

**time:verify-date** *day month year day-of-the-week*
Returns **nil** if the day of the week of the date specified by *day*, *month*, and *year* is the same as *day-of-the-week*; otherwise, returns a string that contains a suitable error message.

**time:day-of-the-week-string** *day-of-the-week* &optional (*mode* **':long**)
Returns a string representing the day of the week.

**time:month-string** *month* &optional (*mode* **':long**)
Returns a string representing the month of the year.

**time:timezone-string** &optional (*timezone* **time:*timezone***) (*daylight-savings-p* **(time:daylight-savings-p time:timezone)**) *force-numeric-p punctuate*
Returns the printed representation of a timezone; the default timezone is the current one for the user's site.

For more information on functions that deal with dates and times:

See the section "Getting and Setting the Time".
See the section "Elapsed Time in Seconds".
See the section "Elapsed Time in Internal Time Units".
See the section "Printing Dates and Times".
See the section "Reading Dates and Times".
See the section "Reading and Printing Time Intervals".
See the section "Time Conversions".

### Zwei Internals

### Introduction to Zwei Internals

Zmacs, the Genera editor, is built on a large and powerful system of text-manipulation functions and data structures, called *Zwei*.

Zwei is not an editor itself, but rather a system on which other text editors are implemented. For example, in addition to Zmacs, the Zmail mail reading system also uses Zwei functions to allow editing of a mail message as it is being composed or after it has been received. The subsystems that are established upon Zwei are:

- Zmacs, the editor that manipulates text in files
- Dired, the editor that manipulates directories represented as text in files
- Zmail, the editor that manipulates text in mailboxes
- Converse, the editor that manipulates text in messages

Since these subsystems share Zwei in the dynamically linked Lisp environment, many of the commands available as Zmacs commands are available in other editing contexts as well.

### Stream facility for editor buffers

**zwei:with-editor-stream** opens a stream to an editor buffer; it is analogous to **with-open-file** for files. **zwei:open-editor-stream** also opens a stream to an editor buffer; it is analogous to **open** for files.

### The zwei:with-editor-stream macro

**zwei:with-editor-stream** takes the same keyword options as **zwei:open-editor-stream**.

On exit, it sends a **:force-redisplay** message to the stream, which causes the editor to do any necessary redisplay. See the section "Keyword Options".

### The zwei:open-editor-stream function

**zwei:open-editor-stream** takes the same keyword options as **zwei:with-editor-stream**.

You can send a **:force-redisplay** message at any time while the stream is open. See the section "Keyword Options".

### Keyword Options

**zwei:with-editor-stream** and **zwei:open-editor-stream** both recognize the same set of keyword options. Some of the options are mutually exclusive and some are interdependent.

You specify where to find the text by using one of the following keywords, whichever is appropriate to the situation. The keywords appear here in priority or-

der. When the options specify several of these, one from the top of the list overrides one from further down in the list, regardless of what order the keywords appear in the options list.

**:interval**
**:buffer-name**
**:pathname**
**:window**
**:start**

The options refer to an object called a *bp*. This is a Zwei data structure for representing a particular position in a buffer.

*Option* — *Values and meaning*

**:buffer-name**
The full name of a buffer to use for the stream.

```
(zwei:with-editor-stream
  (foo ':buffer-name (send zwei:*interval* ':name))
  ...)
```

The buffer does not need to exist (see **:create-p**). The following example creates a Zmacs buffer named temp and opens the stream **foo** to it.

```
(zwei:with-editor-stream (foo "temp")
  ...)
```

**:create-p**
Specifies what to do when the buffer does not exist. This applies only in conjunction with **:buffer-name** or **:pathname** with **:load-p**.

| *Value* | *Meaning* |
|---|---|
| **:ask** | Queries the user before creating the buffer. |
| **:error** | Signals an error and provides proceed types for creating it or supplying an alternate. |
| **t** | Creates the buffer. |
| **:warn** | Notifies the user that a buffer is being created (the default). |

**:defaults**
Specifies the pathname defaults against which a **:pathname** option would be merged. These are necessary in case reprompting needs to occur. The default is **nil**, meaning to use the default defaults. This option applies only in conjunction with **:pathname**.

**:end**
Specifies the conditions for terminating the stream (the "end of file" condition).

| *Value* | *Meaning* |
|---|---|
| bp | Stops when this buffer bp is reached. |
| **:end** | Stops at the end of the buffer (the default). This applies only if **:start** was also a bp. |

| | | |
|---|---|---|
| | **:mark** | Stops when it reaches the mark. This option requires that you use the **:window** option as well. |
| | **:point** | Stops when it reaches point. This option requires that you use the **:window** option as well. |

**:hack-fonts** Specifies how to treat font shifts in the buffer.

| *Value* | *Meaning* |
|---|---|
| **nil** | Ignores font shifts (the default). |
| **t** | Provides full font support. Encodes font shifts on both input and output using epsilons, as would go to a file. |

**:interval** Specifies a Zwei interval to use for the stream.

**:kill** Specifies what to do with the buffer before using it as a stream.

| *Value* | *Meaning* |
|---|---|
| **nil** | No action (the default) |
| **t** | Deletes all the text currently in the designated part of the buffer. |

**:load-p** Specifies whether to read the file specified by **:pathname** into the editor before using the buffer as a stream. (This is analogous to Find File in Zmacs.) This works only from within Zmacs.

| *Value* | *Meaning* |
|---|---|
| **nil** | No action (the default) |
| **t** | Loads the file into the editor. |

**:no-redisplay** Suppresses the redisplay of any windows associated with the interval being written into.

```
(zwei:with-editor-stream
  (standard-output :buffer-name "Herald" :no-redisplay t)
  (print-herald))
```

**:ordered-p** States whether **:start** and **:end** are guaranteed to be in forward order. The default is **nil**. This applies only when **:start** and **:end** are bps or **:point** and **:mark**.

**:pathname** Specifies a pathname to use for the stream. This can be a pathname object or any file spec that can be coerced to a pathname by **fs:parse-pathname**.

**:start** Specifies where to start the stream with respect to the buffer contents.

| *Value* | *Meaning* |
|---|---|
| **:append** | Starts at the end of the buffer. (Same as **:end**.) |
| **:beginning** | Starts at the beginning of the buffer. |
| bp | Starts with this bp. |
| **:end** | Starts at the end of the buffer (the default). (Same as **:append**.) |
| **:mark** | Starts at the mark, which does not move as a result. This requires a Zmacs window. |
| **:point** | Starts at point, which does not move as a result. This requires that you use the **:window** option as well. |
| **:region** | Starts at point and ends at mark (or vice versa, depending on the ordering). This requires that you use the **:window** option as well. It ignores any **:end** in this case. |

**:window** Specifies a Zmacs window as the stream source.

**zwei:with-editor-stream** does not currently interlock to prevent simultaneous access to a single buffer by multiple processes. Neither does anything else. Trying to access the same buffer with several processes simultaneously is not guaranteed to work.

### Making Standalone Editor Windows

You can create an editor window with the following properties:

- Is standalone, that is, has its own process.
- Does not necessarily have the buffer structure of Zmacs.
- Uses a pop-up window in place of the minibuffer.
- Uses its own command table.

To create such a window, follow this procedure:

Start with **zwei:standalone-editor-frame**. Send it an **:edit** message to make it edit. It does not have its own process by default; you can mix **tv:process-mixin** with it and make that process send the **:edit** message if you want it to have its own process.

For example:

```
;;; -*- Syntax: Zetalisp; Base: 10; Mode: LISP; Package: ZWEI -*-
```

```
(defun funny-edit (string)
  (let ((window (tv:make-window 'standalone-editor-window :label "Funny Little Editor"
                                :height 400.
                                ;; do this if you want its own command table
                                #+ignore #+ignore :*comtab* *some-comtab*)))
    (let ((string (catch 'abort-standalone-edit
                    (edstring string window))))
      string)))
```

Compile this, and do (zwei:funny-edit "any string"), and you will get a standalone editor, primed with the string. It has a pop-up/pop-down minibuffer for m-X commands and the like. Press END to exit and return the current text from the buffer, as a string.

A slightly more elaborated version of this is used as the patch-comment editor under the m-X Finish Patch interface. Some of the main points of that:

- As given, the only way to abort out of the standalone editor is with c-m-ABORT, which is way too big a hammer. You can create a comtab for the standalone editor, say, **zwei:*some-comtab***, and prime it with an existing Zwei command, **zwei:com-standalone-abort**. Of course, you can also add custom commands for the little editor:

```
(setq *some-comtab*
      (set-comtab "Funny Little Editor"
                  '(#\abort com-standalone-abort)))
(set-comtab-indirection *some-comtab* *standalone-comtab*)
```

- If this is going to be used repeatedly, you will want to use a window resource instead of consing a new window each time.

- You may want to define your own window flavor on **zwei:standalone-editor-window** and define special methods on it.

- All this is obvious in the source for **zwei:edit-patch-comment-immediately** and related stuff, in SYS:ZWEI;PATED.LISP.

Two other useful messages:

**:set-interval-string**
Inserts a string in the editor.

**:interval-string** Returns a string to the caller when **:edit** returns.

For providing a special comtab, you can initialize the instance variable **zwei:*comtab*** by using the **:*comtab*** keyword in the init plist.

You can exit from this kind of editor by using END.