sv
af

Daniel Keller

Introduction to the Dialog Machine

Bericht Nr. 5 / November 1989

Projekt-Zentrum IDA
Eidgenössische Technische Hochschule Zürich

Ke 060
2nd ed. Ke02SE

Daniel Keller

Introduction to the Dialog Machine

Bericht Nr. 5 / November 1989

Projekt-Zentrum IDA
Eidgenössische Technische Hochschule Zürich

Adresse des Autors:

Daniel Keller
Projekt-Zentrum IDA
ETH-Zentrum
CH-8092 Zürich

© 1989 Projekt-Zentrum IDA, ETH Zürich

Contents

Preface

The Dialog Machine is a library of Modula-2 routines for the easy programming of user-friendly interfaces on the Macintosh and the PC. It consists of about 300 procedures for the creation and handling of menu bars, windows, graphical and textual output, and input via the mouse and dialog boxes.

The Dialog Machine (later called the DM) was created at the ETH in the years 86/87 for the easier programming of the Apple Macintosh in Modula-2. The library contains practically everything needed to write programs which fully use the capabilities of the Macintosh (see the screen shots in this paper). However it is clearly a tool to *simplify* the programming of a graphical user interface and therefore it is restricted in its functionality. As a benefit the code of programs using the DM can be very short - considering their appearance and ease of use. The appendix lists seven sample programs, all of which count less than 200 lines.

In 1988/89 the DM was ported to the MS-DOS/PC. Apart from a few omissions and limitations, a Modula-2 program using the DM can easily be ported from the Macintosh to the PC. One only needs to re-compile and link with the DM/PC library. The resulting application then runs under GEM and makes full use of the mouse, drop-down menus, and windows, thereby giving the same user-friendly look and feel as on the Macintosh.

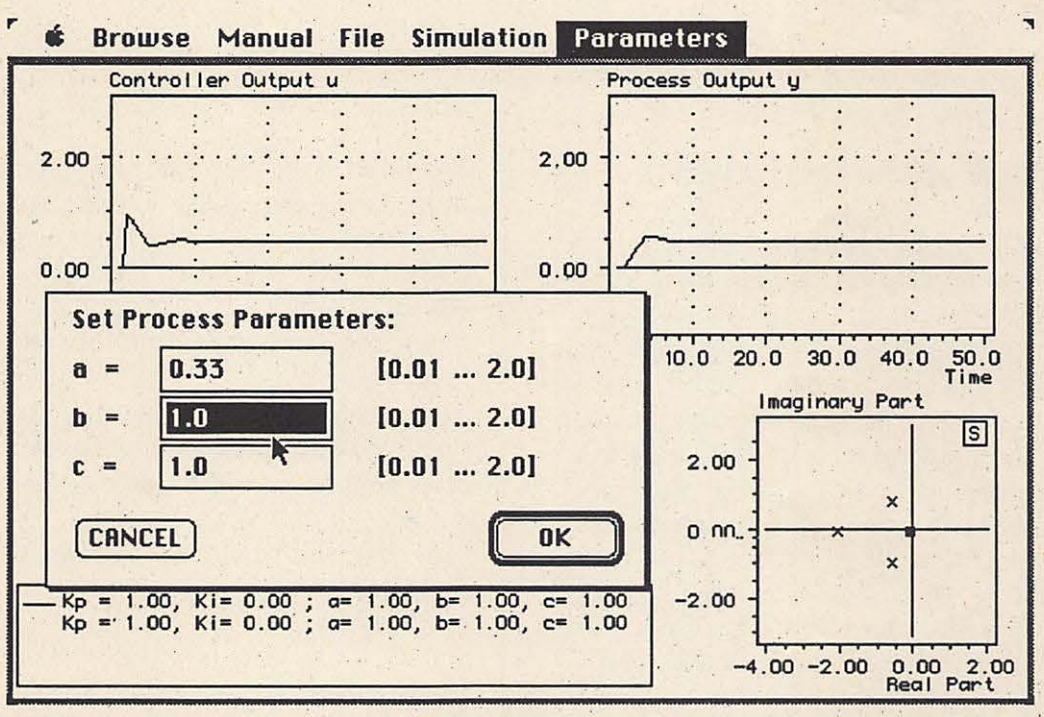


Fig. 1: sample screen of a program using the DM with menus, windows, graphical output, and a dialog box.

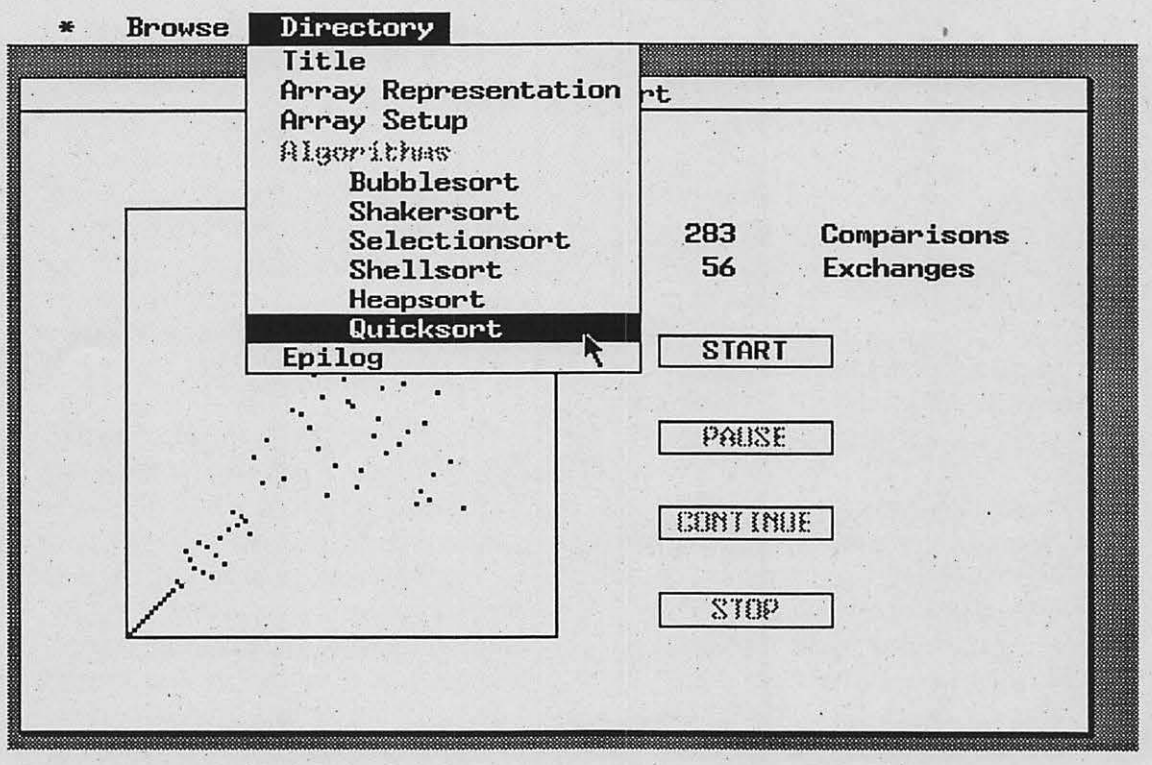


Fig. 2: PC-screen dump of a program which was originally written for the Macintosh and then ported to the PC with only a few hours work.

Scope of this Introduction

This paper is intended as a first introduction to the DM for programmers who are already fluent in Modula–2 . No aid is given in explaining the language Modula-2. It is also assumed that you are familiar with the user interface concepts of the Apple Macintosh (or the GEM Desktop on the PC). Before attempting to read this, you should already have worked with either a Macintosh - or another window-based system with a mouse - for more than a few days. It is recommended to read the chapter on User Interface Guidelines in Volume 1 of *Inside Macintosh.*

The paper is divided into two parts: the first one explaining the 30 most used commands of the DM, and the second one giving an outlook to the more advanced topics, i.e. the over 250 remaining commands which are used less frequently.
This short introduction does not relieve you from the burden of reading the comments in the definition modules of the DM. It only gives you a brief overview, so that you get to know the most important concepts before you start programming.

The Concept of Active Elements

The DM takes full control over the user´s input, i.e. mouse clicks, keystrokes, and menu selections. Some of the user´s input the DM handles all by itself, e.g. dragging a window to another location requires no code to be written, all is done by the DM automatically. Some other input must be handled by the program, e.g. the DM must know what to do when the user selects a menu entry.

The concept to handle such user actions is simple: the DM provides two sorts of active elements: *menu commands* (also called "menu entries") and *push buttons*. All the programmer has to do is to tell the DM which procedure to execute when the user clicks onto one of the active elements: *each active element must have a procedure attached.* The rest (i.e. event handling loop, menu inversion, etc.) is taken care of by the DM.

The following code fragment illustrates this concept. This fragment installs one menu with two command entries. The title of the menu is "Simulation" and the text of the two menu entries is "Open and read data file" and "Draw the curve".

```
InstallMenu( SimMenu, "Simulation",    ... );
InstallCommand( SimMenu, ... "Open and read data file",  ReadSimData  ... );
InstallCommand( SimMenu, ... "Draw the curve",  PlotSimCurve  ... );


PROCEDURE ReadSimData;
BEGIN
   GetExistingFile( ...FileIsOpen);
THEN
   IF FileIsOpen THEN
      WHILE NOT EOF(  )  DO
         ...
      END;
   END;
END ReadSimData;
```

```
PROCEDURE PlotSimCurve;
BEGIN
   IF NOT WindowExists(CurveWindow)

      CreateWindow( .... );
   END;
   SelectForOutput(CurveWindow);
   ...
END PlotSimCurve;
```

Fig. 3: The menu bar created by the above code fragment

One parameter of the procedure `InstallCommand` is *the name of another procedure* (here `ReadSimData` and `PlotSimCurve`), namely the procedure to be executed when this menu entry is selected. The DM now handles all user inputs: whenever the user selects one of these two entries, the corresponding procedure (`ReadSimData` or `PlotSimCurve` in the above example) is executed by the DM.

The typical DM program first creates a menu bar by calling `InstallMenu` and `InstallCommand`, thereby attaching a procedure to each menu entry. This procedure must be without parameters and must be located at the top level of the module. After having written the parameterless procedures and after having built the menu bar, the program passes the control over to the DM with `RunDialogMachine`. This means:

"Look, my dear Dialog Machine, I have constructed the menu bar with the attached procedures. Now YOU take over and monitor all the user's actions with the mouse. You know now which procedures to execute when the user selects a menu entry."

A Small Example

The following sample program allows to open a window, draw some lines into it, clear the window contents, and exit the program.

Note that the program does not prevent the user to open the window a second time or to draw into the window when it is not open. Enforcing a certain sequence of commands or disabling unapplicable commands can be achieved by enabling and disabling the appropriate menu entries so that they cannot be selected when they should not. This is not done here. The purpose of this example is to demonstrate the typical structure of a DM program.

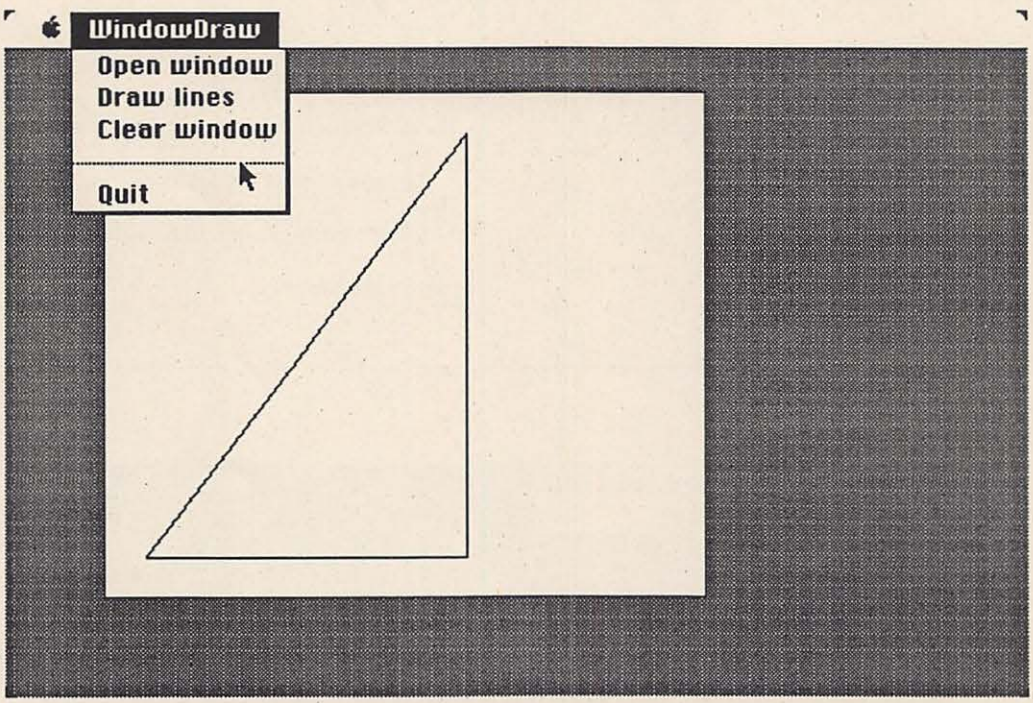


Fig. 4: A screen dump from the program "First" (the Quit command in the menu is automatically added by the DM)

```
MODULE First;                    (* Example of a small Dialog Machine program *)


FROM DMMaster     IMPORT RunDialogMachine;

FROM DMMenus      IMPORT Menu, Command, Separator, AccessStatus, Marking,
                         InstallMenu, InstallCommand;

FROM DMWindows    IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                         CloseAttr, ZoomAttr, WindowFrame, RectArea,
                         CreateWindow, AutoRestoreProc;

FROM DMWindowIO   IMPORT EraseContent, SetPen, LineTo;


VAR
   FirstMenu:       Menu;
   MakeWindowComm:  Command;
   DrawLinesComm:   Command;
   ClearComm:       Command;

   TheWindow:       Window;


PROCEDURE MakeWindow;
VAR
   BigFrame: WindowFrame;
BEGIN
   WITH BigFrame DO
      x :=  50; y :=  50;            (* coordinates of lower left corner *)
      w := 300; h := 250;            (* width and height of the window   *)
   END;
   CreateWindow( TheWindow, FixedLocation,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, BigFrame, "This is not a title", AutoRestoreProc );
END MakeWindow;


PROCEDURE DrawLines;
BEGIN
   SetPen(  20,  20 );                 (* coordinates within the window *)
   LineTo( 180,  20 );
   LineTo( 180, 230 );
   LineTo(  20,  20 );
END DrawLines;


BEGIN
   InstallMenu( FirstMenu, "WindowDraw", enabled );
   InstallCommand( FirstMenu, MakeWindowComm, "Open window",
                   MakeWindow, enabled, unchecked );
   InstallCommand( FirstMenu, DrawLinesComm, "Draw lines",
                   DrawLines, enabled, unchecked );
   InstallCommand( FirstMenu, ClearComm, "Clear window",
                   EraseContent, enabled, unchecked );
   RunDialogMachine;
END First.
```

Building up a Menu Bar

Menu entries and separators

A menu bar is built up from left to right. The first `InstallMenu` command creates the first menu after the ◆-menu. With the procedure `InstallCommand` a single menu entry can be added to an existing menu. The new command is added at the bottom of the list of commands already in the menu.

Note: With the DM it is not possible to insert either a menu between two menus, or a menu command between two commands. Menus and commands are always added to the right of the menu bar, or the bottom of a menu resp.

A separator (a dotted line or a blank line) can be added to a menu by using the `InstallSeparator` command. This is useful for separating one group of related commands from another.

The Quit Command

Each menu-driven application must have a QUIT command in order to be able to exit properly. With the procedure `InstallQuitCommand` a separator and a menu entry is added to the leftmost menu. The DM terminates when the user selects this entry. If you never issue a call to `InstallQuitCommand` before `RunDialogMachine` is called, the DM automatically adds a QUIT entry in the leftmost menu.

Enabling and Disabling Menu Entries

Menu entries can be disabled (their text appears in a light grey as opposed to full black) so that they cannot be selected. This is a very useful feature to ensure that certain commands cannot be issued at certain times, e.g. the command "Read data file" should be disabled when no data file is open (see Fig. 5); after having opened the file, the command "Read data file" can be enabled. Turning a command on and off is done with calls to `DisableCommand` and `EnableCommand`.

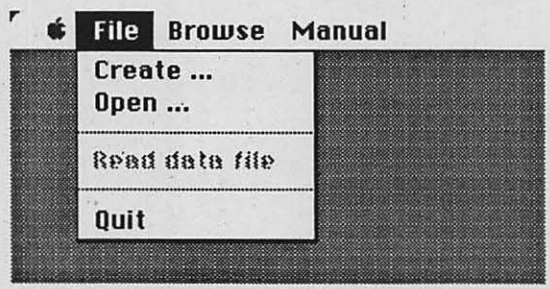


Fig 5: A menu with separators and a disabled command.

Using Windows

Creating a Window

Windows come in different shapes: they can be small or large, they can have a title bar or not, they can be moveable or sizeable or both, etc. All this is controlled with the parameters to the `CreateWindow` command.

There are four basic types of windows provided by the DM (see Fig. 6):

1) a very simple window which cannot be re-sized, nor moved (dragged) to another location: FixedLocation
2) one that cannot be resized, but moved around: FixedSize
3) the same as 2) but with a title bar
4) a window which can be dragged around and re-sized

The `WindowFrame` parameter for `CreateWindow` specifies the size of the usable area (without the title and scroll bars) within the window in pixels. The X and Y coordinates specify the location of the lower left corner.

To make the life of programmers easy, the DM provides automatic mechanisms for the restoring and scrolling of windows.

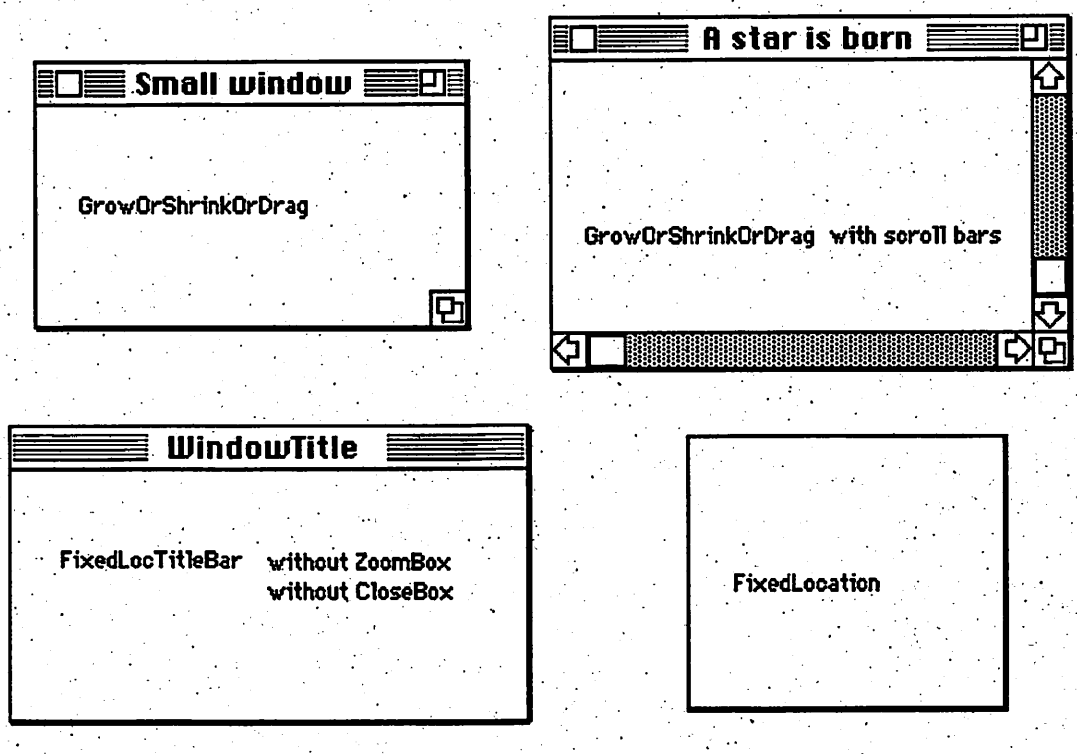


Fig. 6: Window types provided by the DM

The Concept of the Current Output Window

The DM always knows a *current output window*. All output takes place in this window. A newly opened window automatically becomes the current output window. With multiple windows it is possible to direct the output to the desired window by using the procedure `SelectForOutput`. This defines the new current output window. Note that the current output window needs not be the top (front) window, they can be different (see also the definition modules of DMWindow and DMWindowIO)

Cleaning and Closing a Window

The contents of the current output window can be blanked (cleaned) by using the command `EraseContent`. Closing and removing a window from the screen can be done with `RemoveWindow`.

Output to a Window

The cursor can be placed anywhere in the window by using `SetPen` (pixel coordinates X, Y with fixpoint = lower left corner) or `SetPos` (character cell coordinates Line/Column with fixpoint = upper left corner). Both procedures move the same pen position. All drawing, writing, or painting now takes place starting from this point. All output routines move the pen position.

Textual Output

The DM provides the usual routines for ASCII output, like: `WriteString`, `WriteInt`, `WriteReal`, `Write`, `WriteLn`, `etc.`, starting from the current pen position.
The current font with its caracteristics (default: plain Geneva 12 on a Mac, the System font on a PC) can be changed with `SetWindowFont`.

Drawing Primitives

Drawing of simple graphics can be achieved with the routines `Dot`, `LineTo` (from the current to the new location), `Circle`, and `Area` (paints a rectangular area). The pattern with which all drawing takes place can be changed with `SetPattern`.

Displaying MacPaint Pictures

To show more elaborate pictures it is recommended to paint them first with MacPaint or another painting program. After that the picture can be copied to the clipboard and from there into a resource file (use the program DMResMover to achieve this). With the call `DisplayPredefinedPicture` such a picture can be displayed in a window.

Input from the User

Modal and Modeless Dialogs

modal:
In a modal dialog the user is forced to terminate the dialog by using one of the exit buttons (usually labeled CANCEL and OK). No other action outside the dialog box can be taken, i.e. the user is forced into "a mode".

modeless:
The modeless dialog on the other hand offers the user the choice to terminate the dialog or to do something else, e.g. selecting another window or selecting a menu entry. The modeless dialog is not used as often as the modal dialog, although it is friendlier to the user because it leaves more choices open rather than forcing the user to do something particular.

An example for a modeless dialog is the "find string" box in a word processing program: you can enter a search string, search for the occurrences of the string, or just click into the text window and continue editing without having to terminate the search dialog by pressing OK.

Modal Dialog Boxes

The Module DMEntryForms contains the procedures to construct modal dialogs: `StringField`, `RealField`, `RadioButton`, `CheckBox`, etc. (see the examples FontStyleTest and PaintModes in the appendix). After having placed the fields, the program passes control to the DM by calling `UseEntryForm`. The DM opens a dialog box of the specified size and first completes the dialog by adding two buttons labeled CANCEL and OK. Then it handles all user input like clicking a check box, skipping to the next editable field with the TAB key, editing a number field, etc.

When the user presses one of the two exit buttons (CANCEL or OK) the procedure `UseEntryForm` returns and the contents of the fields and the state of the check boxes and radio buttons can be examined.

Modeless Dialogs

Modeless dialogs can be constructed by opening a window and placing EditFields into it (see the paragraph on EditFields in the section *Advanced Topics* and DMEditFields.DEF).

A `PushButton` (from DMEditFields) is the most often used element of a modeless dialog. Such a button can be placed into any window. Similar to a command entry in a menu the programmer attaches a procedure to this button. The DM monitors the user's action, and - since the button is an "active element" - calls the attached procedure when the user presses the button.

Summary of the 30 Most Used Commands

DMMaster	InitDialogMachine	-- only in the PC version
	RunDialogMachine	
DMMenus	InstallMenu	
	InstallCommand	
	InstallSeparator	
	InstallQuitCommand	
	DisableCommand	
	EnableCommand	
DMWindows	CreateWindow	
	RemoveWindow	
DMWindowIO	SetPen	
	SetPos	
	Dot	
	LineTo	
	WriteString	
	WriteInt	
	WriteReal	
	WriteLn	
	DisplayPredefinedPicture	
	EraseContent	
	SelectForOutput	
DMEntryForms	StringField	
	IntField	
	RealField	
	CheckBox	
	DefineRadioButtonSet	
	RadioButton	
	WriteLabel	
	UseEntryForm	
DMEditFields	PushButton	

Many programmers will only use the 30 Most Used Commands of the previous paragraphs (and will write bautiful programs!). However, to take full advantage of the graphical user interface à la Macintosh one might need the elements and commands described briefly in this second part of the introduction.

The description of the commands is not exhaustive. It is intended to show the further possibilities of the DM without teaching the details of the calls (which can be looked up in the .DEF files).

Two-dimensional Function Graphs

The module DM2DGraphs offers some sophisticated routines for showing two-dimensional graphs in a window. Scaling and labeling of the axes is done automatically.
`DefineGraph`, `DefineCurve` and `Plot` are (almost) the only commands needed to create gorgeous curves.

File Access

The DM supports the standard Macintosh (and GEM) mechanism for selecting and opening a file with a file selector box. The routines for reading and writing text (ASCII) files are provided (including `GetInteger`, `PutInteger`, `GetReal`, etc.).

Controlling the Cursor Shape

A change of the cursor to the wristwatch symbol is done with a call to `ShowWaitSymbol`. The user now gets the visual feedback that something lengthy is going on, e.g. reading a file or a long calculation. A call to `HideWaitSymbol` reverts to the normal arrow cursor.

Watching the Mouse

`GetCurMousePos` delivers the current mouse position. `RectClicked` and `PointClicked` allow to monitor mouse clicks at specified locations within a window. The program MouseTest in the appendix demonstrates the use of these procedures.

Dragging

It is one of the nice things on the Macintosh that the user can pick up an object and drag it somewhere else. The DM supports this feature in a very easy way: just call the procedure `Drag` with two other procedures as parameters, one to be called repeatedly during dragging to draw the outline of the dragged object, the other to erase the object at its old location and re-draw it a the new location. Everything else is taken care of by the DM (see the sample program DragZoom).

Keyboard Equivalents for Menu Entries

It is good practice to provide keyboard equivalents for often-used commands (like the standard abbreviations ⌘-C, X, V for Copy, Cut, and Paste, ⌘-S and P for Save and Print, and ⌘-Q for Quit). The procedure `InstallAliasChar` allows to add a keyboard equivalent for the specified menu entry.

Edit Fields

The module DMEditFields provides all the routines for modeless dialogs. EditFields can be placed into any open window with the routines `StringField`, `CharField`, `IntField`, `RealField`, `PushButton`, etc.. The DM automatically watches for clicks into any of these fields and lets the user edit the (editable) fields. Whenever the program needs the values, it can test and fetch them with the routines `GetString`, `IsChar`, `IsReal`, `IsInteger`, `GetRadioButtonSet`, etc.

EditFields can be enabled, disabled, preset with a value, and removed from the window.

Scrolling

The window contents can be scrolled either automatically or by explicitely asking for the scrollbar positions and moving the contents. It is recommended to use the AutoScroll mechanism only (see the sample program ScrollTest).

Clipping

All output is automatically clipped by the DM to the work area of the current window. Under programm control the painting area can be further restricted by specifying a clipping rectangle and calling the routine `SetClipping`.

Restoring Window Contents

Whenever windows overlap, the contents of a window can be destroyed. After uncovering, the previously hidden parts must be restored. This can be done automatically by the DM's autorestore mechanism. Here each output command is executed twice, first it draws into the window, and afterwards it draws a second time the same into a hidden bitmap. Restoring destroyed parts of a window now just means copying from the hidden bitmap into the window.

Autorestoring is simple, but takes a lot of memory for the hidden bitmap (up to 22K per window on a Mac Plus, up to 200K with color monitors and larger screens). Drawing is also slower when autorestoring is in effect because each output routine is executed twice. The restoring itself (copying from the hidden bitmap into the window) is very fast.

Each window has a restore procedure attached (the `Repaint` parameter to `CreateWindow`). If it is the `AutoRestoreProc`, the above mechanism is in effect. Specifying the `DummyRestoreProc` means that no restoring takes place (except re-drawing EditFields in this window).

The programmer can specify his/her own routine which will then be called whenever the window needs restoring. So the programmer can pack all drawing calls into one procedure and pass the name of this procedure as a parameter to the `CreateWindow` call.

Passing Control to the Dialog Machine

Usually the control is with the DM. The DM monitors the user events (keyboard, mouse clicks) and calls the "attached" procedures provided by the programmer. In some cases the execution of the pro-

cedure takes a long time or the program executes a loop within this procedure. Now temporal control can be passed back to the DM, just for one split second (around 20ms) by calling DialogMachineTask. This means:

"Dear Dialog Machine, please check if anything of importance has happened since you last had control, e.g. the user has selected a menu entry. If so, quickly execute the attached procedure, but then return control back to me as soon as possible. Thanks."

Hardware & Software Environments

Apple Macintosh

Any Macintosh with at least 512K Bytes of memory and with the new ROMs can be used for programming with the DM. A hard disk is recommended, although it is possible to develop applications with only two 800K diskette drives.

The DM Version 1.0 for the Mac requires either MacMETH 2.4 (does not run on the SE/30 and the IIcx) or MacMETH 2.5.2 to compile and link the programs. MacMETH can be obtained from the Stabsstelle Software, Dept. Informatik, ETH, CH-8092 Zürich.

A separate editor is highly recommended, since the MacMETH Editor does not adhere to the usual Macintosh interface standards. Good choices: MEdit, an editor with user-definable macros, written by Matthias Aebi, and QUED/M, a powerful editor with macros and regular expression capabilities.

MS-DOS/PC

The DM Version 1.0 for the MS-DOS/PC requires an IBM-compatible computer with at least 512K memory, a mouse (Microsoft, Genius, LOGITECH, etc.), a hard disk drive, and a floppy disk drive. The compiler to use in connection with the DM is JPI's TopSpeed Modula-2 compiler (I had first used LOGITECH's, but after five months of development I switched to JPI - I never regretted it, TopSpeed is a smart, lean, and fast environment. The only thing I miss is mouse support in the editor).

GEM is the graphical interface with which the DM works. The GEM Desktop is all it takes, the developer's kit is not needed, except for those who want to market the programs written with the DM (licence and copyrights of DRI Inc.).

In order to run the programs - which often use graphics heavily - sufficiently fast, a computer of the AT class (Norton SI rating > 7) or at least an XT Turbo (Norton SI rating > 2) is recommended. An ordinary PC-XT with 4.77 MHz (Norton SI reference rating = 1) is too slow for most DM applications.

It is also recommended that the computer have the full 640K of memory, since DOS and GEM eat up close to 200K already. After the smallest possible DM program is loaded, a heap area of about 300K (of 640K) remains free.

The screen must have a resolution better than 512 x 342 pixels (most screens do, only the oldest ones used to have 640 x 200 which is not enough). The screen can be color or monochrome, although the DM only draws in black-and-white.

Appendix

A Bibliography

Apple Computer Inc.: Inside Macintosh, Volume I, Addison-Wesley, 1985

Fischlin, Andreas: "The 'Dialog Machine' for the Macintosh", Projekt-Zentrum IDA, ETH Zürich, 1988

Keller, Daniel: "Mac-PC compatibility" (a short listing of inconsistencies between the two versions of the Dialog Machine), Projekt-Zentrum IDA, ETH Zürich, 1989

Listings of the DEFINITION modules of the Dialog Machine (on disk).

B Seven Sample Programs

WindowTest
shows the use of several overlapping windows together with the AutoRestore mechanism

FontStyleTest
shows how different fonts and styles are set with `SetWindowFont` and the aid of a modal dialog box

PaintModes
all drawing primitives are demonstrated in conjunction with the four paint modes: replace, paint, XOR, erase

MouseTest
illustrates how to monitor mouse coordinates and watch for clicks into a rectangular area

ScrollTest
the AutoScroll mechanism is demonstrated

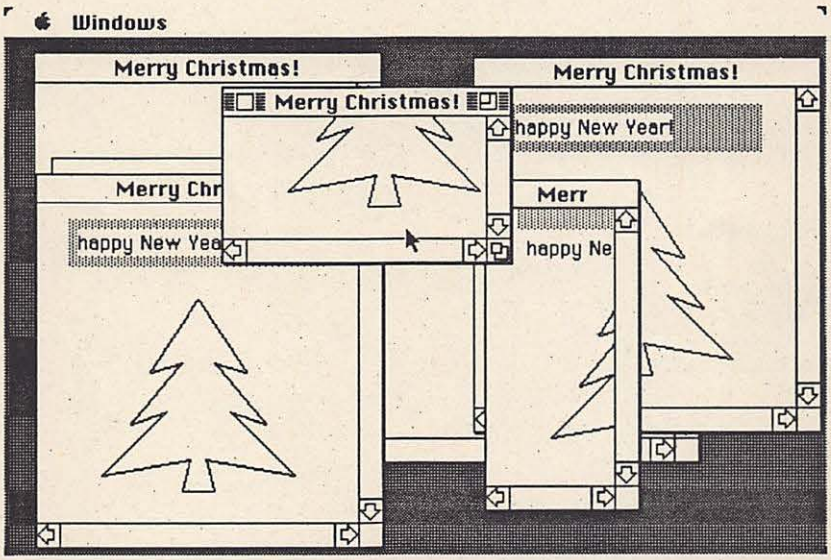
EFDemo
shows the use of EditFields (modeless dialog)

DragZoom
about dragging an object and drawing its outline, plus the use of `DisplayPredefinedPicture`.
(This sample program only works fine on the Macintosh. On the PC scaling a picture with `DisplayPredefinedPicture` does not work - only a 1:1 display of pictures).

4. Aug.89

```
MODULE WindowTest;

(*
 *  This program shows the use of several overlapping windows and their handling
 *  with the mouse. It is also a test for the AutoRestore mechanism.
 *)

FROM DMMaster   IMPORT RunDialogMachine;

FROM DMMenus    IMPORT Menu, Command, InstallMenu, InstallCommand,
                       Separator, InstallSeparator, AccessStatus, Marking,
                       InstallQuitCommand;

FROM DMWindows  IMPORT Window, CreateWindow, WindowKind, ScrollBars,
                       CloseAttr, ZoomAttr, WindowFrame, WFFixPoint,
                       RectArea, RemoveWindow, RemoveAllWindows,
                       FrontWindow, AutoRestoreProc;

FROM DMWindowIO IMPORT Dot, LineTo, SetPen, GreyContent, pat, Area,
                       SetPos, WriteString, SelectForOutput;

VAR
   FirstMenu:            Menu;
   OpenWindowComm:       Command;
   DrawComm:             Command;
   CloseAllWindowsComm:  Command;
   CloseFrontWindowComm: Command;

   WindowTable:          ARRAY[ 1..20 ] OF Window;
   NrOfWindows:          INTEGER;

PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   ReallyQuit := TRUE;
END JustQuit;

PROCEDURE Draw;
VAR
   GreyBox: RectArea;
BEGIN
   SelectForOutput( FrontWindow() );
   Dot( 90, 20 );
   LineTo( 110,  20 );
   LineTo( 105,  40 );
   LineTo( 160,  30 );
   LineTo( 120,  70 );
   LineTo( 142,  62 );
   LineTo( 110, 100 );
   LineTo( 130,  94 );
   LineTo( 100, 140 );
   LineTo(  70,  94 );
   LineTo(  90, 100 );
   LineTo(  58,  62 );
   LineTo(  80,  70 );
   LineTo(  40,  30 );
   LineTo(  95,  40 );
   LineTo(  90,  20 );

   GreyBox.x := 20; GreyBox.y := 160; GreyBox.w := 160; GreyBox.h := 30;
   Area( GreyBox, pat[ lightGrey ] );
   SetPos( 2, 4 ); WriteString( "happy New Year!" );
END Draw;

PROCEDURE MakeWindow;
VAR
   Frame: WindowFrame;
BEGIN
```

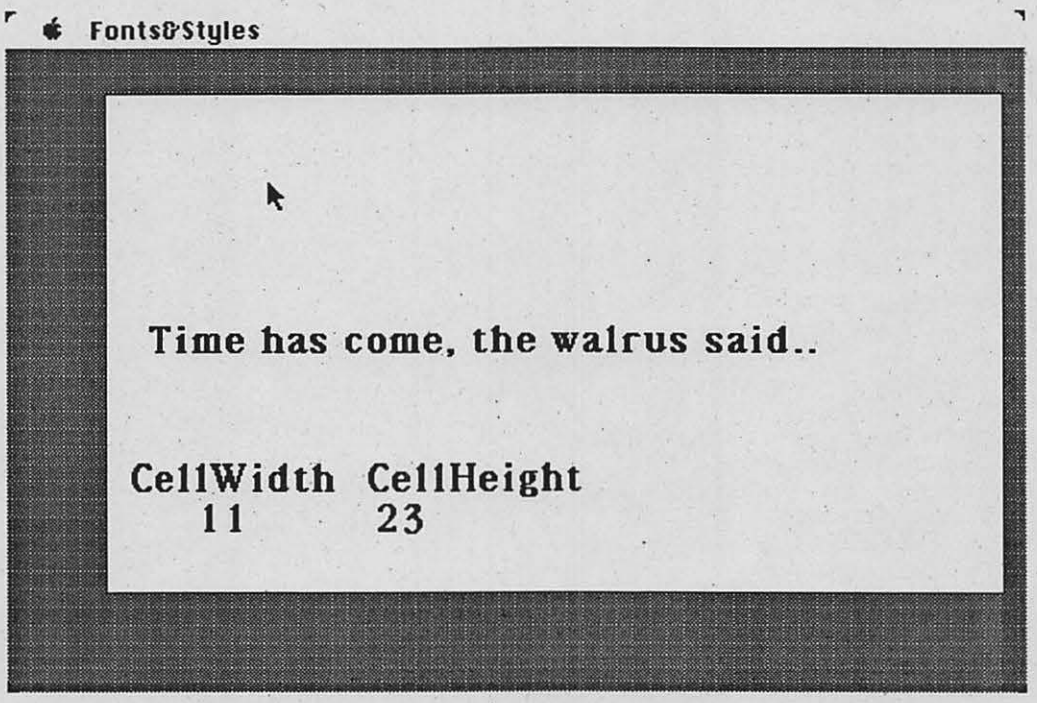
```
   WITH Frame DO
      x :=  20 + NrOfWindows*10; y :=  20 + NrOfWindows*10;
      w := 200; h := 200;
   END;

   INC( NrOfWindows );
   CreateWindow( WindowTable[ NrOfWindows ], GrowOrShrinkOrDrag,
                 WithBothScrollBars, WithCloseBox, WithZoomBox,
                 bottomLeft, Frame, "Merry Christmas!", AutoRestoreProc );
END MakeWindow;

PROCEDURE DeleteFrontWindow;
VAR
   w: Window;
BEGIN
   w := FrontWindow();
   RemoveWindow( w );
END DeleteFrontWindow;

PROCEDURE DeleteAllWindows;
BEGIN
   RemoveAllWindows;
   NrOfWindows := 0;
END DeleteAllWindows;

PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "Windows", enabled );
   InstallCommand( FirstMenu, OpenWindowComm, "Open new window",
                   MakeWindow, enabled, unchecked );
   InstallSeparator( FirstMenu, line );
   InstallCommand( FirstMenu, DrawComm, "Draw into FrontWindow",
                   Draw, enabled, unchecked );
   InstallCommand( FirstMenu, CloseFrontWindowComm, "Close FrontWindow",
                   DeleteFrontWindow, enabled, unchecked );
   InstallCommand( FirstMenu, CloseAllWindowsComm, "Close all windows",
                   DeleteAllWindows, enabled, unchecked );
   InstallQuitCommand( "Quit", JustQuit, 0C );
END MakeMenuBar;

BEGIN
   NrOfWindows := 0;
   MakeMenuBar;
   RunDialogMachine;
END WindowTest.
```

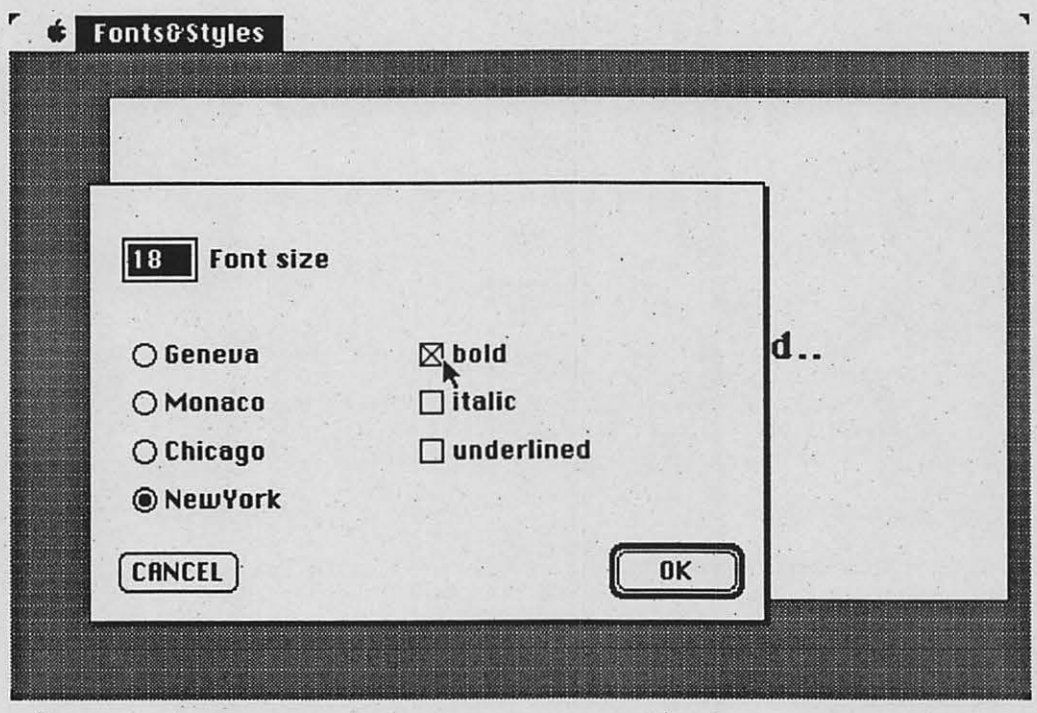


```
MODULE FontStyleTest;

(*
 *  This sample Dialog Machine program shows the use of a dialog
 *  box to change font characteristics.
 *
 *  Exercise:
 *     Watch out for the font name (Geneva, NewYork...) and why it is *NOT*
 *     kept in the dialog box:  whenever you select "Change text attributes"
 *     the font name is back to Chicago.  Try to make it "remember" the font
 *     name you had selected last.
 *)

FROM DMMaster     IMPORT RunDialogMachine;

FROM DMMenus      IMPORT Menu, Command, Separator, AccessStatus, Marking,
                         InstallMenu, InstallCommand, InstallQuitCommand;

FROM DMWindows    IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                         CloseAttr, ZoomAttr, WindowFrame, RectArea,
                         CreateWindow, AutoRestoreProc;

FROM DMEntryForms IMPORT FormFrame, DefltUse,
                         WriteLabel, IntField, CheckBox, UseEntryForm,
                         RadioButtonID, RadioButton, DefineRadioButtonSet;

FROM DMWindowIO   IMPORT EraseContent, SetPen, WriteString, WriteInt,
                         WindowFont, FontStyles, FontStyle, SetWindowFont,
                         CellWidth, CellHeight;

CONST
   TheText = "Time has come, the walrus said..";



VAR
   FirstMenu:     Menu;
   WriteComm:     Command;
   ChangeComm:    Command;

   TheWindow:     Window;

   CharHeight:                INTEGER;
   Bold, Italic, Underlined: BOOLEAN;

PROCEDURE WriteText;
VAR
   i, j: INTEGER;
BEGIN
   EraseContent;
   SetPen( 20, 120 ); WriteString( TheText );
   SetPen( 10,  50 ); WriteString( "CellWidth  CellHeight" );
   SetPen( 10,  30 ); WriteInt( CellWidth(), 7 ); WriteInt( CellHeight(), 11 );
END WriteText;

PROCEDURE SetTextAttributes;
VAR
   ff:          FormFrame;
   ok:          BOOLEAN;
   FontID:      WindowFont;
   FontButtons: RadioButtonID;
   Style:       FontStyle;
   GenevaRB:    RadioButtonID;
   MonacoRB:    RadioButtonID;
   ChicagoRB:   RadioButtonID;
   NewYorkRB:   RadioButtonID;
```

```
BEGIN
   IntField(   2, 2, 4, CharHeight, useAsDeflt, 0, 30000 );
   WriteLabel( 2, 7, "Font size" );

   DefineRadioButtonSet( FontButtons );
   RadioButton( GenevaRB,  4, 2, "Geneva" );
   RadioButton( MonacoRB,  5, 2, "Monaco" );
   RadioButton( ChicagoRB, 6, 2, "Chicago" );
   RadioButton( NewYorkRB, 7, 2, "NewYork" );
   FontButtons := ChicagoRB;

   CheckBox( 4, 20, "bold", Bold );
   CheckBox( 5, 20, "italic", Italic );
   CheckBox( 6, 20, "underlined", Underlined );

   ff.x := 40; ff.y := 40; ff.lines := 9; ff.columns := 40;
   UseEntryForm( ff, ok );
   IF ok THEN
      Style := FontStyle{};
      IF Bold THEN
         INCL( Style, bold );
      END;
      IF Italic THEN
         Style := Style + FontStyle{ italic };          (* alternative way *)
      END;
      IF Underlined THEN
         INCL( Style, underline );
      END;
      IF FontButtons = GenevaRB THEN
         FontID := Geneva;
      ELSIF FontButtons = MonacoRB THEN
         FontID := Monaco;
      ELSIF FontButtons = ChicagoRB THEN
         FontID := Chicago;
      ELSIF FontButtons = NewYorkRB THEN
         FontID := NewYork;
      END;
      SetWindowFont( FontID, CharHeight, Style );
   END;
   WriteText;
END SetTextAttributes;

PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   ReallyQuit := TRUE;
END JustQuit;

PROCEDURE MakeWindow;
VAR
   Frame: WindowFrame;
BEGIN
   WITH Frame DO
      x :=  50; y :=  50;
      w := 450; h := 250;
   END;
   CreateWindow( TheWindow, FixedLocation,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, Frame, "Font Test", AutoRestoreProc );
END MakeWindow;

PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "Fonts&Styles", enabled );
   InstallCommand( FirstMenu, WriteComm, "Write text",
                   WriteText, enabled, unchecked );
   InstallCommand( FirstMenu, ChangeComm, "Change text attributes ...",
                   SetTextAttributes, enabled, unchecked );
```

```
   InstallQuitCommand( "Quit", JustQuit, 0C );
END MakeMenuBar;


BEGIN
   CharHeight := 12;
   Bold := FALSE; Italic := FALSE; Underlined := FALSE;

   MakeMenuBar;
   MakeWindow;
   WriteText;
   RunDialogMachine;
END FontStyleTest.
```

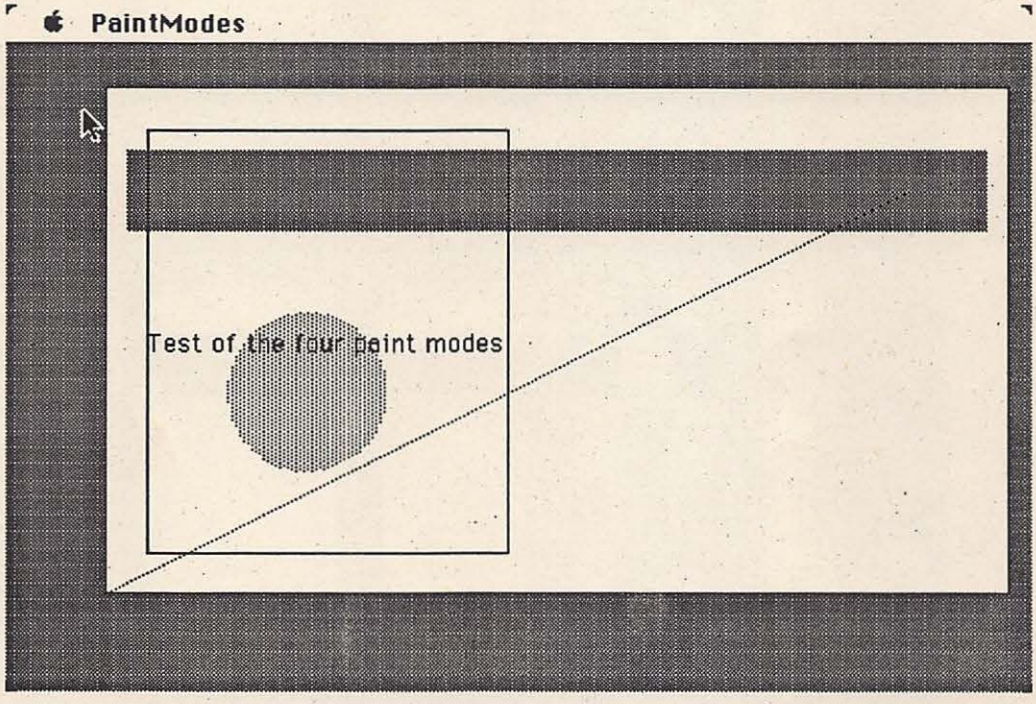


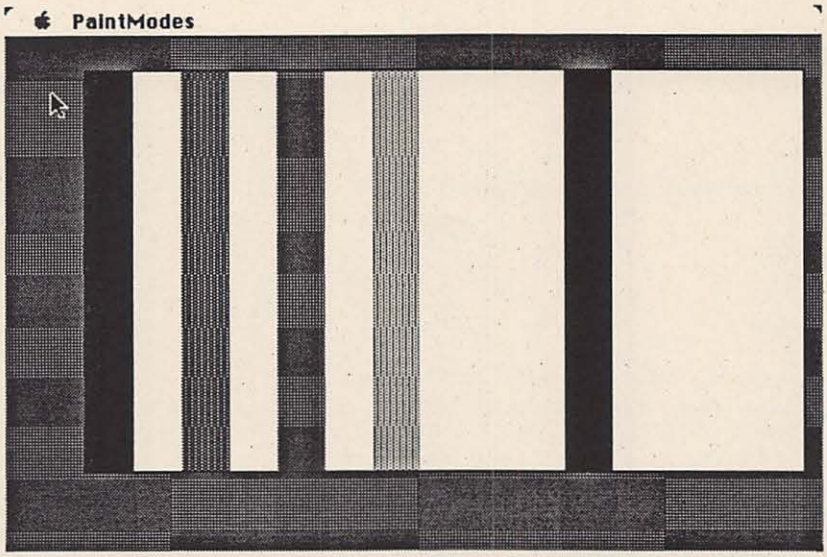


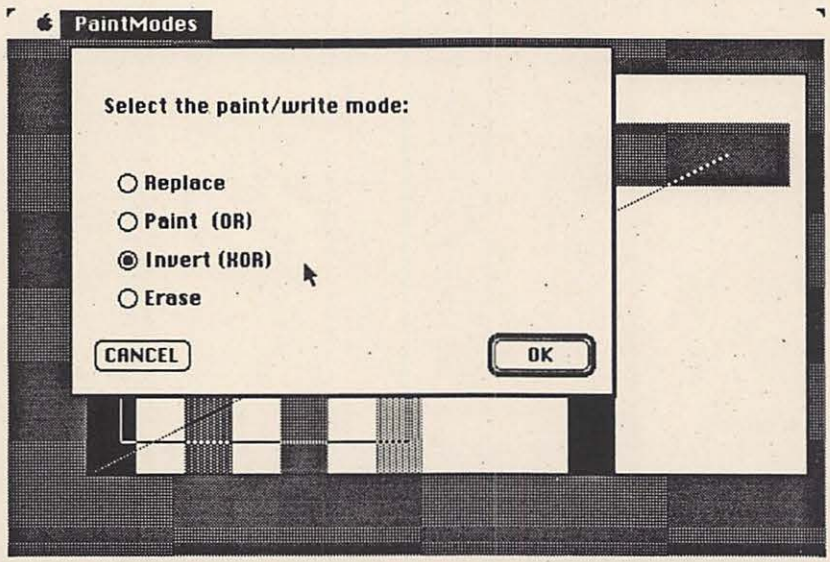
```
MODULE PaintModes;

(*
 *   Tests and shows the use of the paint modes provided by the Dialog Machine.
 *   Watch the interesting effects when you paint over existing patterns
 *   (paint the stripes first, then select "test paint...").
 *   Also try the combination of the "invert" mode with multiple painting of
 *   the same lines and patterns.
 *)

FROM DMMaster     IMPORT RunDialogMachine;

FROM DMMenus      IMPORT Menu, Command, Separator, AccessStatus, Marking,
                         InstallMenu, InstallCommand, InstallQuitCommand;

FROM DMEntryForms IMPORT FormFrame, UseEntryForm, WriteLabel,
                         RadioButtonID, RadioButton, DefineRadioButtonSet;

FROM DMWindows    IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                         CloseAttr, ZoomAttr, WindowFrame, RectArea,
                         CreateWindow, AutoRestoreProc;

FROM DMWindowIO   IMPORT GreyContent, Pattern, pat, PaintMode, SetMode,
                         EraseContent, SetPen, WriteString,
                         Dot, LineTo, Area, Circle;

FROM DMLanguage   IMPORT Language, SetLanguage;

VAR
   FirstMenu:   Menu;
   SetModeComm: Command;
   PaintComm:   Command;
   StripesComm: Command;
   ClearComm:   Command;

   TheWindow:        Window;
   BigFrame:         WindowFrame;
   CurrentPaintMode: PaintMode;

PROCEDURE SetPaintMode;
VAR
   ModeButtons: RadioButtonID;
   ReplaceRB:   RadioButtonID;
   PaintRB:     RadioButtonID;
   InvertRB:    RadioButtonID;
   EraseRB:     RadioButtonID;
   ff:          FormFrame;
   ok:          BOOLEAN;

BEGIN
   ff.x := 40; ff.y := 100; ff.lines := 9; ff.columns := 40;

   WriteLabel( 2, 2, "Select the paint/write mode:" );
   DefineRadioButtonSet( ModeButtons );
   RadioButton( ReplaceRB, 4, 3, "Replace" );
   RadioButton( PaintRB,   5, 3, "Paint  (OR)" );
   RadioButton( InvertRB,  6, 3, "Invert (XOR)" );
   RadioButton( EraseRB,   7, 3, "Erase" );
   CASE CurrentPaintMode OF
      replace: ModeButtons := ReplaceRB; |
      paint:   ModeButtons := PaintRB; |
      invert:  ModeButtons := InvertRB; |
      erase:   ModeButtons := EraseRB; |
   END;

   UseEntryForm( ff, ok );
```

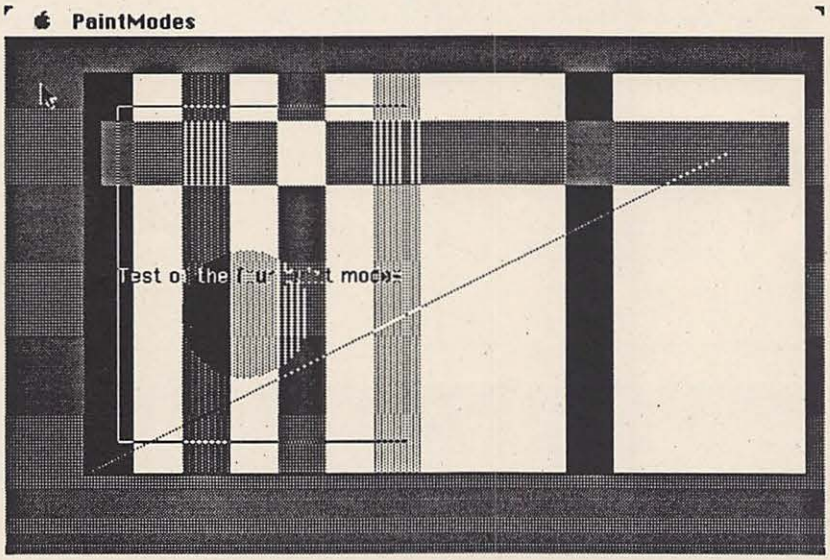
```
    IF ok THEN
       IF ModeButtons = ReplaceRB THEN
          CurrentPaintMode := replace;
       ELSIF ModeButtons = PaintRB THEN
          CurrentPaintMode := paint;
       ELSIF ModeButtons = InvertRB THEN
          CurrentPaintMode := invert;
       ELSIF ModeButtons = EraseRB THEN
          CurrentPaintMode := erase;
       END;
       SetMode( CurrentPaintMode );
    END;
END SetPaintMode;

PROCEDURE DrawStripes;
CONST
    StripWidth = 30;
VAR
    Strip: RectArea;
BEGIN
    WITH Strip DO
       x := 0; y := 0;
       w := StripWidth;
       h := BigFrame.h;
    END;
    Area( Strip, pat[ dark ] );
    Strip.x := Strip.x + 2 * StripWidth;
    Area( Strip, pat[ darkGrey ] );
    Strip.x := Strip.x + 2 * StripWidth;
    Area( Strip, pat[ grey ] );
    Strip.x := Strip.x + 2 * StripWidth;
    Area( Strip, pat[ lightGrey ] );
    Strip.x := Strip.x + 2 * StripWidth;
    Area( Strip, pat[ light ] );
    Strip.x := Strip.x + 2 * StripWidth;
    Area( Strip, pat[ dark ] );
END DrawStripes;

PROCEDURE Paint;
VAR
    i:       INTEGER;
    GreyBox: RectArea;
BEGIN
    SetPen( 20, 120 );
    WriteString( "Test of the four paint modes" );
    SetPen( 10, 10 );
    FOR i := 1 TO 200 DO
       Dot( 2*i, i );
    END;
    SetPen( 20, 20 );
    LineTo( 20, 230 );
    LineTo( 200, 230 );
    LineTo( 200, 20 );
    LineTo( 20, 20 );
    Circle( 100, 100, 40, TRUE, pat[ lightGrey ] );
    WITH GreyBox DO
      x := 10; y := 180;
      w := BigFrame.w - 20; h := 40;
    END;
    Area( GreyBox, pat[ grey ] );
END Paint;


PROCEDURE MakeWindow;
BEGIN
    WITH BigFrame DO
       x :=  50; y :=  50;
```

```
      w := 450; h := 250;
   END;
   CreateWindow( TheWindow, FixedLocation,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, BigFrame, "Test", AutoRestoreProc );
END MakeWindow;

PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   ReallyQuit := TRUE;
END JustQuit;

PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "PaintModes", enabled );
   InstallCommand( FirstMenu, SetModeComm, "Select paint mode ...",
                   SetPaintMode, enabled, unchecked );
   InstallCommand( FirstMenu, StripesComm, "Draw stripes",
                   DrawStripes, enabled, unchecked );
   InstallCommand( FirstMenu, PaintComm, "Test paint and write",
                   Paint, enabled, unchecked );
   InstallCommand( FirstMenu, ClearComm, "Clear window",
                   EraseContent, enabled, unchecked );
   InstallQuitCommand( "Quit", JustQuit, 0C );
END MakeMenuBar;

BEGIN
   CurrentPaintMode := replace;

   SetLanguage( English );
   MakeMenuBar;
   MakeWindow;
   RunDialogMachine;
END PaintModes.
```



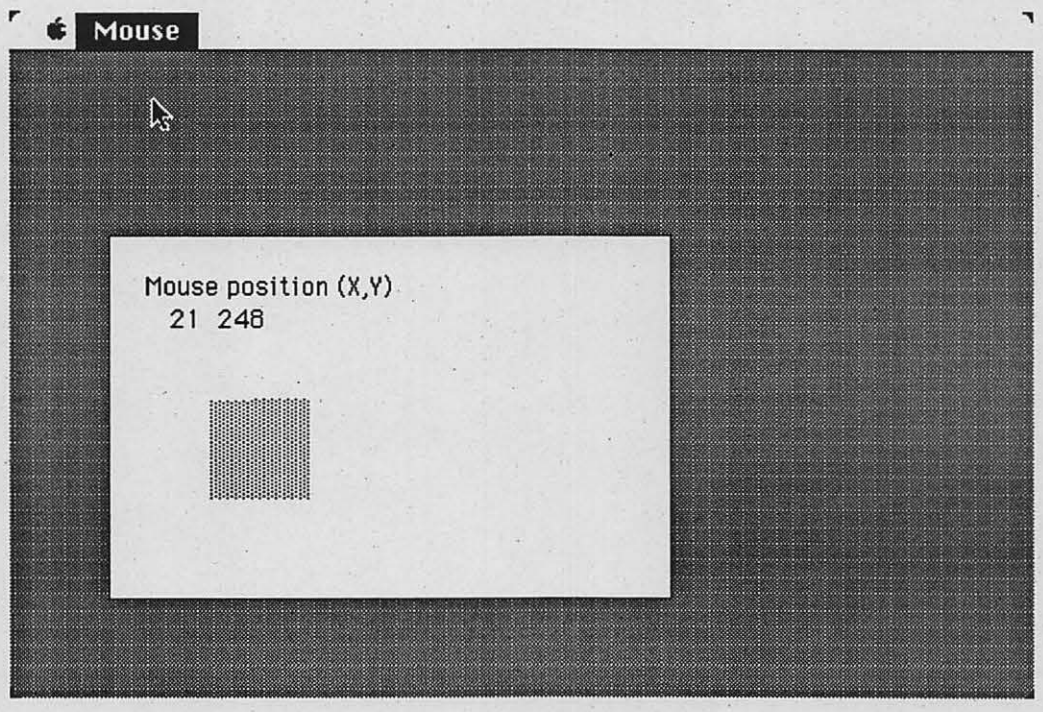

PaintModes


PaintModes
Select the paint/write mode:
Replace
Paint (OR)
Invert (XOR)
Erase
CANCEL
OK


PaintModes

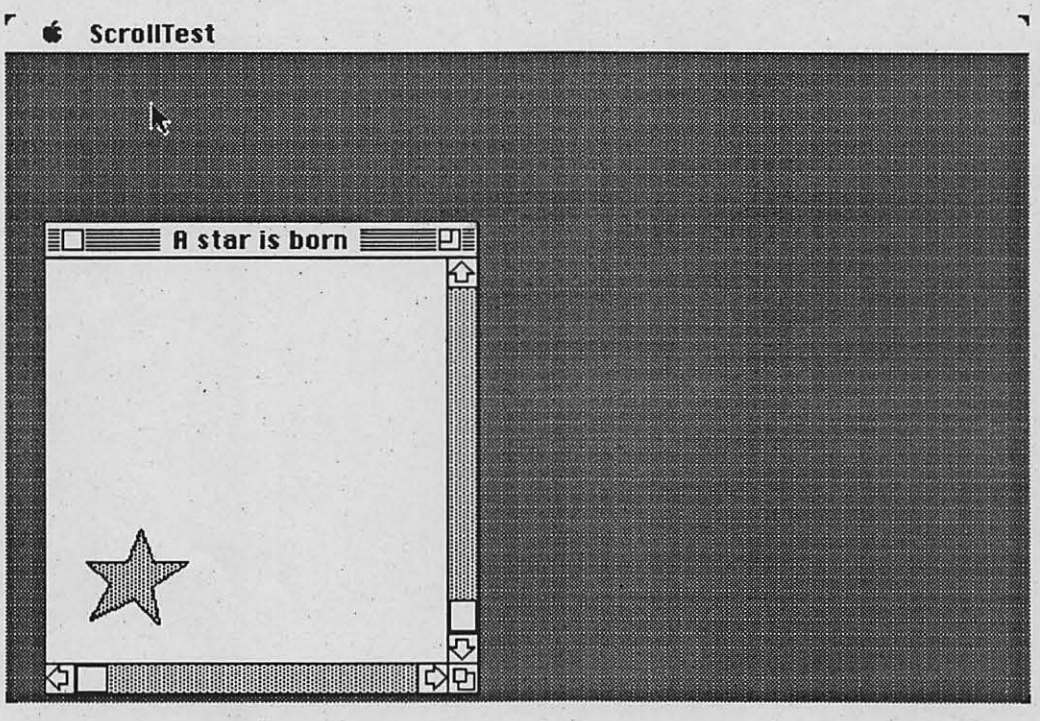
```
MODULE MouseTest;

(*
 *  Demonstrates and tests some basic mouse functions of the
 *  Dialog Machine.
 *)

FROM DMMaster   IMPORT RunDialogMachine, DialogMachineTask;
FROM DMMenus    IMPORT Menu, Command, InstallMenu, InstallCommand,
                       AccessStatus, Marking, InstallAliasChar,
                       ChangeCommandText, InstallQuitCommand;
FROM DMWindows  IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                       CloseAttr, ZoomAttr, WindowFrame, RectArea,
                       CreateWindow, DummyRestoreProc;
FROM DMWindowIO IMPORT GetCurMousePos, RectClicked, SetPos,
                       WriteString, WriteInt, GreyContent, pat, Area;

VAR
   FirstMenu:   Menu;
   StartComm:  Command;
   StopComm:   Command;

   TheWindow:    Window;
   GreyBox:      RectArea;

   MonitoringIsOn: BOOLEAN;

PROCEDURE MonitorMouse;
VAR
   MouseX, MouseY: INTEGER;
   Counter:        INTEGER;
BEGIN
   MonitoringIsOn := TRUE;
   Counter := 0;
   WHILE MonitoringIsOn DO
      GetCurMousePos( MouseX, MouseY );
      SetPos( 3, 3 ); WriteInt( MouseX, 5 ); WriteInt( MouseY, 5 );
      IF RectClicked( GreyBox ) THEN
         SetPos( 5, 3 ); WriteString( "Mouse click" );
         Counter := 20;
      ELSE
         IF Counter > 0 THEN
            DEC( Counter );
         ELSIF Counter = 0 THEN
            SetPos( 5, 3 ); WriteString( "              " );
         END;
      END;
      DialogMachineTask;
   END;
END MonitorMouse;

PROCEDURE StopMonitoring;
BEGIN
   MonitoringIsOn := FALSE;
END StopMonitoring;

PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   StopMonitoring;
   ReallyQuit := TRUE;
END JustQuit;

PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "Mouse", enabled );
   InstallCommand( FirstMenu, StartComm, "Start monitoring mouse",
                   MonitorMouse, enabled, unchecked );
```

```
   InstallCommand( FirstMenu, StopComm, "Stop monitoring mouse",
                   StopMonitoring, enabled, unchecked );
   InstallAliasChar( FirstMenu, StopComm, "S" );
   InstallQuitCommand( "Quit", JustQuit, "Q" );
END MakeMenuBar;


PROCEDURE MakeWindow;
VAR
   Frame: WindowFrame;
BEGIN
   WITH Frame DO
      x :=  50; y :=  50;
      w := 280; h := 180;
   END;
   CreateWindow( TheWindow, FixedLocation,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, Frame, "Test", DummyRestoreProc );
   SetPos( 2, 3 ); WriteString( "Mouse position (X,Y)" );
   WITH GreyBox DO
      x := 50; y := 50;
      w := 50; h := 50;
   END;
   Area( GreyBox, pat[ lightGrey ] );
END MakeWindow;


BEGIN
   MonitoringIsOn := FALSE;
   MakeMenuBar;
   MakeWindow;
   RunDialogMachine;
END MouseTest.
```



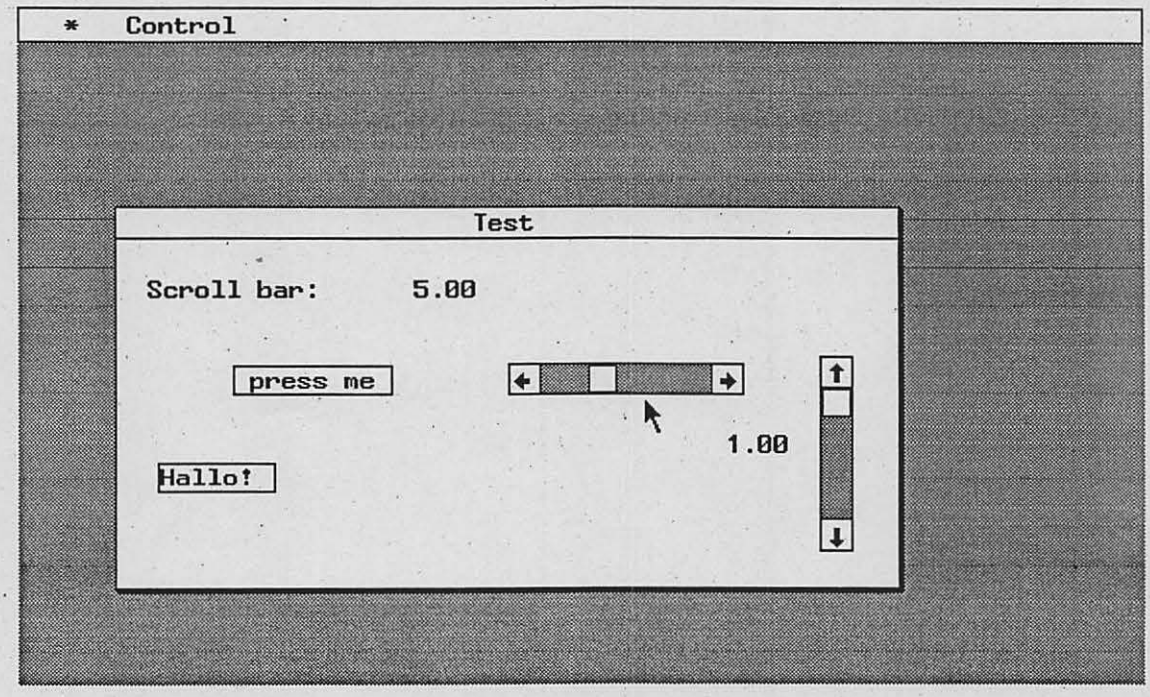
```
MODULE ScrollTest;

(*
 *  This program demonstrates the use of scroll bars
 *)

FROM DMMaster   IMPORT RunDialogMachine;

FROM DMMenus    IMPORT Menu, Command, InstallMenu, InstallCommand,
                       AccessStatus, Marking, InstallQuitCommand;

FROM DMWindows  IMPORT Window, CreateWindow, WindowKind, ScrollBars, RectArea,
                       CloseAttr, ZoomAttr, WindowFrame, WFFixPoint;

FROM DMWindowIO IMPORT SetPen, LineTo, GreyContent, pat, Area,
                       StartPolygon, CloseAndFillPolygon,
                       SetContSize, SetScrollStep, SetScrollProc,
                       AutoScrollProc;

VAR
   FirstMenu:        Menu;
   OpenWindowComm:   Command;

   StarWindow:       Window;

PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   ReallyQuit := TRUE;
END JustQuit;

PROCEDURE DrawStar( w: Window );
BEGIN
   SetPen( 22, 22 );
   StartPolygon;
   LineTo( 43, 32 );
   LineTo( 56, 20 );
   LineTo( 52, 38 );
   LineTo( 70, 50 );
   LineTo( 53, 50 );
   LineTo( 47, 66 );
   LineTo( 40, 50 );
   LineTo( 20, 50 );
   LineTo( 32, 40 );
   LineTo( 22, 22 );
   CloseAndFillPolygon( pat[ lightGrey ] );
END DrawStar;

PROCEDURE MakeWindow;
VAR
   Frame:       WindowFrame;
   ContentRect: RectArea;
BEGIN
   WITH Frame DO
      x :=  20; y :=  20;
      w := 200; h := 200;
   END;

   CreateWindow( StarWindow, GrowOrShrinkOrDrag,
                 WithBothScrollBars, WithCloseBox, WithZoomBox,
                 bottomLeft, Frame, "A star is born", DrawStar );

   WITH ContentRect DO
      x := 0; y := 0;
      w := 300; h := 300;
   END;
   SetContSize( StarWindow, ContentRect );
   SetScrollStep( StarWindow, 50, 10 );
```

```
   SetScrollProc( StarWindow, AutoScrollProc );
END MakeWindow;


PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "ScrollTest", enabled );
   InstallQuitCommand( "Quit", JustQuit, 0C );
END MakeMenuBar;


BEGIN
   MakeMenuBar;
   MakeWindow;
   RunDialogMachine;
END ScrollTest.
```



```
MODULE EFDemo;

(*
 *  Demonstrates the use of EditFields
 *
 *  Note that the Dialog Machine automatically adds a menu entry (the quit
 *  command) because without it the program could not be terminated.
 *)


FROM DMMaster     IMPORT InitDialogMachine, RunDialogMachine;
FROM DMWindows    IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                         CloseAttr, ZoomAttr, WindowFrame,
                         CreateWindow, DummyRestoreProc;
FROM DMEditFields IMPORT EditItem, StringField, PushButton, GetString,
                         ScrollBar, Direction, IsReal;
FROM DMWindIO     IMPORT SetPos, WriteString, WriteReal;



VAR
   TheWindow:     Window;
   StringEF:      EditItem;
   PressMeButton: EditItem;
   HoriScroller:  EditItem;
   VertiScroller: EditItem;



PROCEDURE DisplayEFContents;
VAR
   CurrentContents: ARRAY[ 0..21 ] OF CHAR;
BEGIN
   (*
    *  fetch the contents of the string field
    *)
   GetString( StringEF, CurrentContents );
   (*
    * erase what was previously there and display the current contents
    *)
   SetPos( 2, 3 ); WriteString( "                                                  " );
   SetPos( 2, 3 ); WriteString( "Field contents: '" );
   WriteString( CurrentContents ); WriteString( "'." );
END DisplayEFContents;


PROCEDURE DisplayHoriScrollValue;
VAR
   curVal: REAL;
BEGIN
   SetPos( 2, 3 ); WriteString( "                                                  " );
   SetPos( 2, 3 ); WriteString( "Scroll bar:" );
   IF IsReal( HoriScroller, curVal ) THEN
      WriteReal( curVal, 10, 2 );
   END;
END DisplayHoriScrollValue;


PROCEDURE DisplayVertiScrollValue;
VAR
   curVal: REAL;
BEGIN
   SetPos( 7, 38 );
   IF IsReal( VertiScroller, curVal ) THEN
      WriteReal( curVal, 6, 2 );
   END;
END DisplayVertiScrollValue;
```

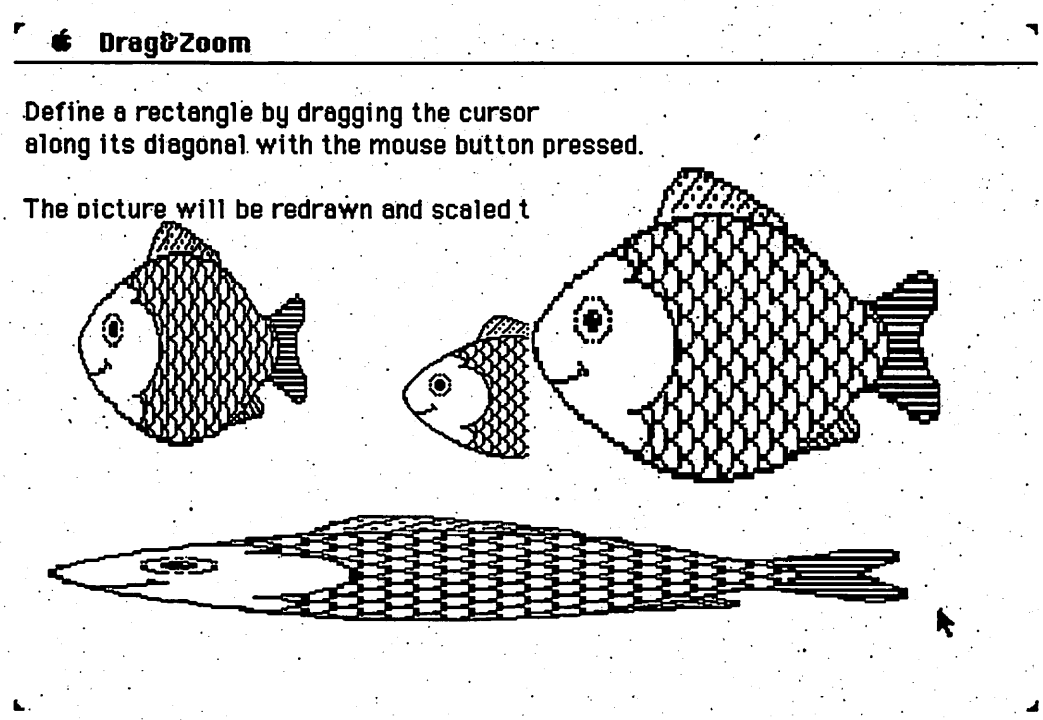
```
PROCEDURE AddEditFields;
BEGIN
   StringField( TheWindow, StringEF, 20, 50,  7, "Hallo!" );
   PushButton( TheWindow, PressMeButton, 60, 100, 10,
                          "press me", DisplayEFContents );
   ScrollBar( TheWindow, HoriScroller, 200, 100, 120, horizontal,
              0.0, 15.0, 0.2, 2.0, 5.0, DisplayHoriScrollValue );
   ScrollBar( TheWindow, VertiScroller, 360, 20, 100, vertical,
             -1.0, 1.0, 0.05, 0.5, 1.0, DisplayVertiScrollValue );
END AddEditFields;



PROCEDURE MakeWindow;
VAR
   Frame: WindowFrame;
BEGIN
   WITH Frame DO
      x :=  50; y :=  50;
      w := 400; h := 180;
   END;
   CreateWindow( TheWindow, FixedSize,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, Frame, "Test", DummyRestoreProc );
END MakeWindow;



BEGIN
   InitDialogMachine;
   MakeWindow;
   AddEditFields;
   RunDialogMachine;
END EFDemo.
```



```
MODULE DragZoom;

(*
 *  This sample Dialog Machine program shows the use of
 *  the dragging mechanism and how to display predefined pictures.
 *)


FROM DMMaster    IMPORT RunDialogMachine, InstallMouseHandler, MouseHandlers;

FROM DMSystem    IMPORT ScreenWidth, ScreenHeight, MenuBarHeight;

FROM DMMenus     IMPORT Menu, Command, AccessStatus, Marking,
                        InstallMenu, InstallCommand, InstallQuitCommand;

FROM DMWindows   IMPORT Window, WindowKind, ScrollBars, WFFixPoint,
                        CloseAttr, ZoomAttr, WindowFrame, RectArea,
                        CreateWindow, AutoRestoreProc;

FROM DMWindowIO  IMPORT DisplayPredefinedPicture, EraseContent, SetPen,  LineTo,
                        GetCurMousePos,Drag, SetPos, WriteString, WriteLn;


CONST
   PictWidth    = 125;    (* values provided by the DMResMover *)
   PictHeight   =  73;
   PictID       = 777;
   PictFileName = "Fish";



VAR
   FirstMenu:   Menu;
   ClearComm:   Command;
   ResetComm:   Command;

   TheWindow:   Window;

   StartX, StartY: INTEGER; (* start coordinates of the selection rectangle *)


PROCEDURE Min(a,b: INTEGER): INTEGER;
BEGIN
   IF a < b THEN
      RETURN a
   ELSE
      RETURN b
   END;
END Min;


PROCEDURE DrawRect( MouseX, MouseY: INTEGER);
BEGIN
   SetPen( StartX, StartY );
   LineTo( StartX, MouseY );
   LineTo( MouseX, MouseY );
   LineTo( MouseX, StartY );
   LineTo( StartX, StartY );
END DrawRect;


PROCEDURE DrawZoomedPicture( MouseX, MouseY: INTEGER );
VAR
   r: RectArea;
BEGIN
   WITH r DO
      x := Min( StartX, MouseX );          (* set (x,y) to lower left corner *)
```

```
      y := Min( StartY, MouseY);
      w := ABS( StartX - MouseX );          (* fit pict size to rectangle *)
      h := ABS( StartY - MouseY );
   END;
   DisplayPredefinedPicture( PictFileName, PictID, r);
END  DrawZoomedPicture;


PROCEDURE MakeSelection( w: Window );
BEGIN
   GetCurMousePos( StartX, StartY );
   Drag( DrawRect, DrawZoomedPicture );
END MakeSelection;


PROCEDURE WriteExplanations;
BEGIN
   SetPos( 2, 1 );
   WriteString( " Define a rectangle by dragging the cursor" ); WriteLn;
   WriteString( " along its diagonal with the mouse button pressed." ); WriteLn;
   WriteLn;
   WriteString( " The picture will be redrawn and scaled to fit the rectangle." );
END WriteExplanations;


PROCEDURE ResetPicture;
VAR
   r: RectArea;
BEGIN
   WITH r DO
      x := (ScreenWidth() - PictWidth) DIV 2;              (* center picture *)
      y := (ScreenHeight() - MenuBarHeight() - PictHeight) DIV 2;
      w := 0;                                                 (* no scaling *)
      h := 0;
   END;
   EraseContent;
   WriteExplanations;
   DisplayPredefinedPicture( PictFileName, PictID, r);
END  ResetPicture;


PROCEDURE JustQuit( VAR ReallyQuit: BOOLEAN );
BEGIN
   ReallyQuit := TRUE;
END JustQuit;


PROCEDURE MakeWindow;
VAR
   Frame: WindowFrame;
BEGIN
   WITH Frame DO
      x :=  0; y :=  0;
      w := ScreenWidth(); h := ScreenHeight() - MenuBarHeight();
   END;
   CreateWindow( TheWindow, FixedLocation,
                 WithoutScrollBars, WithoutCloseBox, WithoutZoomBox,
                 bottomLeft, Frame, "Drag & Zoom", AutoRestoreProc );
END MakeWindow;


PROCEDURE MakeMenuBar;
BEGIN
   InstallMenu( FirstMenu, "Drag&Zoom", enabled );
   InstallCommand( FirstMenu, ClearComm, "Clear window contents",
```

```
                    EraseContent, enabled, unchecked );
    InstallCommand( FirstMenu, ResetComm, "Redraw picture in original size",
                    ResetPicture, enabled, unchecked );
    InstallQuitCommand( "Quit", JustQuit, 0C );
END MakeMenuBar;


BEGIN
    MakeMenuBar;
    MakeWindow;
    ResetPicture;
    InstallMouseHandler( WindowContent, MakeSelection );
    RunDialogMachine;
END DragZoom.
```



DM 1.0
MW 1.0
M2 Topspeed
GEM