An Event–Driven Software Architecture

Quintin Cutts and Graham Kirby

University of St.Andrews
North Haugh
St.Andrews
Scotland
KY16 9SS

Edition 1
20th October 1987

Contents :

1. Notifiers

1.1 Description

A Notifier can be thought of as an event distribution procedure which holds the thread of control. An event, which may be a button press, mouse movement or key depression, is passed to a Notifier, which then determines whether the thread of control should be passed to any one of a number of registered applications. This concept is fundamental to the design of the architecture.

For the system to function correctly, applications should be written in an object oriented style. Commonly, a program reads input by continually polling the keyboard and mouse until an event is received. In a Notifier system, however, applications are passive, in that they are called by the Notifier when an event occurs which concerns them. If all applications are written in this style, they will never be trapped in busy loops — this is especially important in a multi-threaded environment. As all applications are input–driven, a radically different style of programming is required. For example, procedures that get some input, process it, and return a result should not be written in this paradigm.

Any application wishing to run in the system first registers with a Notifier by passing it two procedures. The first determines whether an event is relevant to the application ; the second is the action required when such an event occurs. These two procedures constitute a notification. Subsequently, the Notifier will pass the thread of control to the second procedure if an event important to the application is detected by the first procedure. When the two procedures are submitted a third procedure is returned which, when called, will remove the notification from the Notifier.

1.2 Notifier Interface

A Notifier is an instance of the following structure class :

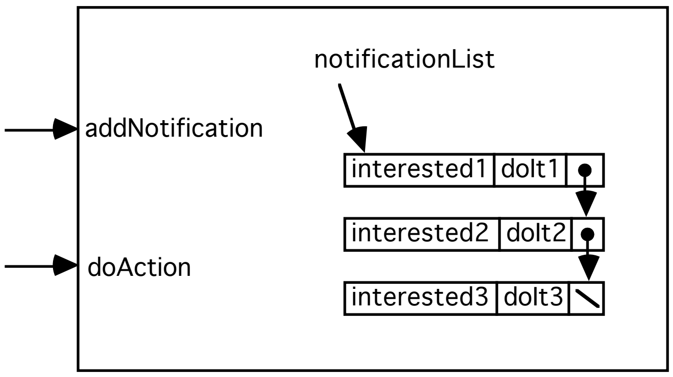
structure NotifierPack(**pr**

proc(proc(pntr –> bool), **proc(pntr**) –> **proc()**)
addNotification)

The **pntr** in all three cases represents an external event that will be passed by the Notifier when calling these procedures. An external event can take two forms :

1. **structure** mouse(**cint** X.pos, Y.pos ; **cbool** selected ; **c*cbool** the.buttons), which is the structure returned by the PS–algol locator function ; and
2. **structure** Chars(**string** chars), used to encapsulate keyboard events.

<u>Structure of a Notifier :</u>



A Notifier package consists of two procedures operating on a linked list encapsulated within their closure. The list contains instances of the following structure class :

structure NotificationItem(**proc(pntr –> bool**) interested ; **proc(pntr**) doIt ; **pntr** next)

where the two procedures constitute a notification.

An application registers with a Notifier by calling the **addNotification** procedure of that Notifier, passing its own **interested** and **doIt** procedures. **addNotification** places this pair at the top of the linked list pointed to by **notificationList**. The procedure returned by **addNotification** removes the pair from the list.

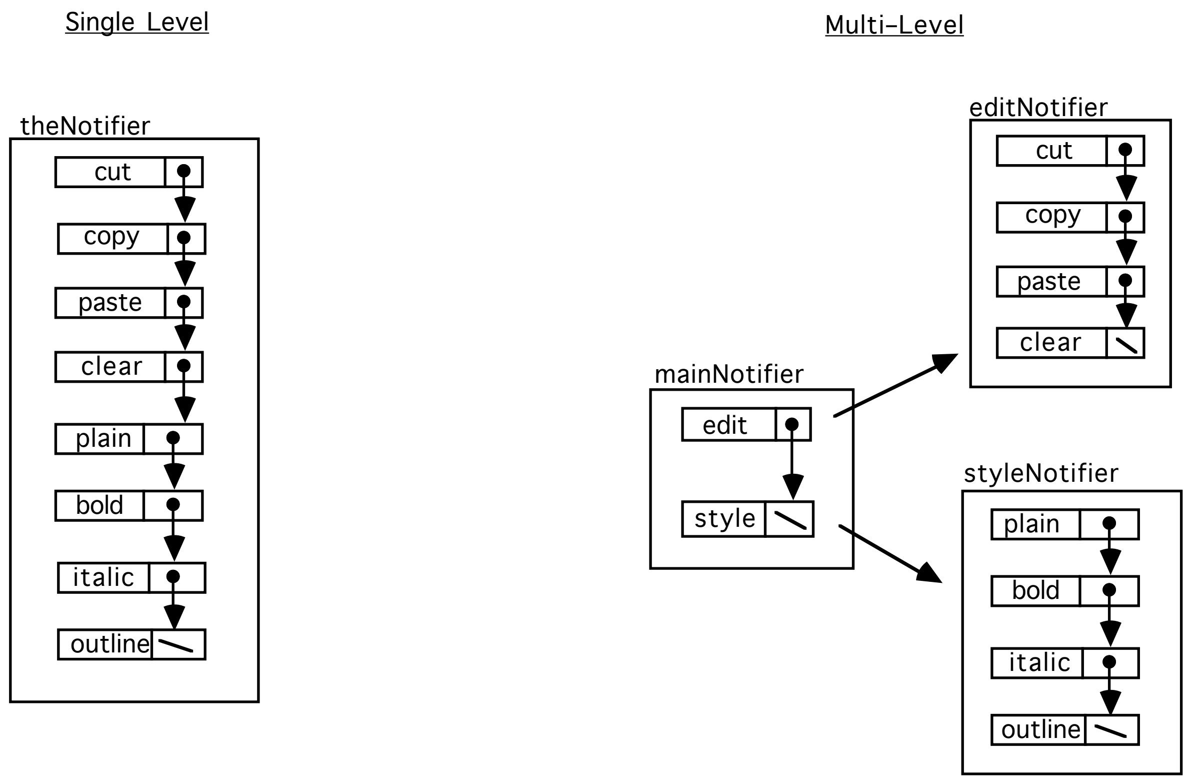
Having been called with the current event as a parameter, **doAction** chains down **notificationList**, calling each **interested** procedure, with the event as a parameter. When one of them returns **true**, the associated **doIt** procedure is called with the event as its parameter. The event is then considered to have been consumed. If none of the **interested** procedures returns **true**, the event is discarded.

1.3 Notifier Hierarchy

Since the signatures of **doAction** and **doIt** are the same, it is possible to substitute one for the other. This allows the construction of a hierarchy of notifiers. This can lead to an increase in speed because there is a reduction in the average number of tests needed to distribute each event. A sub–Notifier is invoked by setting the doIt procedure of a list element in the higher Notifier to be the doAction procedure of the sub–Notifier.

An example of such a system is given in section **1.5**.

This is illustrated below.



theNotifier represents a one–level system which tests for four different edit and four different style events. In the worst case it will perform eight tests.

mainNotifier is the top–level Notifier in a hierarchical system which first differentiates between edit and style events and then passes control to the appropriate sub–Notifier. This performs six tests in the worst case.

1.4 Event Monitor

An Event Monitor is a simple loop which takes all events and passes them to the doAction procedure of the Notifier at the top level of the tree. It is the fixed point in a system, from which the Notifier tree is built. An Event Monitor is generated by **eventMonitorGen**, which is taken from the database. This procedure takes two parameters : the first is a procedure which returns a boolean value, determining whether the Event Monitor should terminate ; the second is the **doAction** procedure of a top–level Notifier.

The use of an Event Monitor is illustrated by the following code fragment :

```
! Declare structures
structure Chars( string chars )
structure NotifierPack(                                                                    pr
                       proc( proc( pntr -> bool ), proc( pntr ) -> proc() )
addNotification )
! Initialise top–level Notifier, using notiferGen, taken from the database
! topLevelNotifier is a pointer to an instance of NotifierPack
let topLevelNotifier = notifierGen()

! register some notifications using topLevelNotifier( addNotification )

! Event Monitor
let eventMonitorGen =proc( proc( -> bool ) finishedYet ;
                       proc( pntr ) aNotifiersDoAction -> proc() )
      proc()
            while ~finishedYet()
            do aNotifiersDoAction( if input.pending() then Chars( read() ) else
locator() )

let finished = proc( -> bool ) ; false  !  make the system go on for ever
let theEventMonitor = eventMonitorGen( finished, topLevelNotifier( doAction ) )
theEventMonitor()   ! initiate the event monitor
```

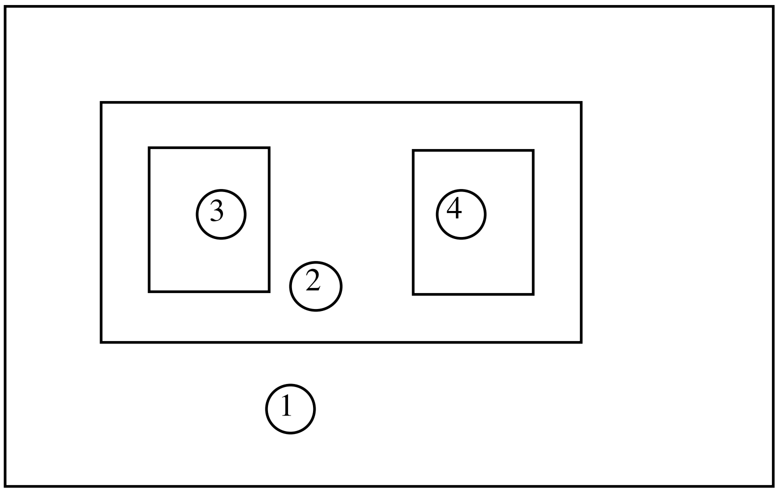
Once a top–level Notifier has been initialised, and some applications have been registered with it using its **addNotification** procedure, the Event Monitor passes any input to it using its **doAction** procedure. Subsequently, any registered applications, when called, may add extra elements to the list within the Notifier, or remove themselves from it.

1.5 Examples

The following program writes out any character typed at the keyboard, until a return is typed. This is an example of a one–level notifier system, with only one application registered.

```
   structure NotifierPack( . . . )
   structure Chars( . . . )          ! As defined in previous example

   let thisNotifier = notifierGen()   !  Generating function for a new Notifier ; from
database
   let finished := false
   let removePrint := proc() ; nullproc
   let interestedPrint = proc( pntr input –> bool )
            input is Chars

   let Print = proc( pntr input )
            if input( chars ) = "'n"  then { removePrint() ; finished := true }
                                      else write input( chars )

   removePrint := thisNotifier( addNotification )( interestedPrint, Print )

   ! event monitor
   let theEventMonitor = eventMonitorGen( proc( –> bool ) ; finished, thisNotifier(
doAction ) )
   theEventMonitor()
```

The next program illustrates the use of a Notifier hierarchy. The top–level Notifier distinguishes between mouse clicks in region 1 and clicks within region 2 (containing regions 3 and 4). If a click occurs within the second region, control is passed to a sub–Notifier which then calls the appropriate procedure for region 3, 4 or region 2 outside the two sub–regions.



```
structure notifierPack( . . . )
structure Chars( . . . )
let thisNotifier = NotifierGen()
let doIt1 = proc( pntr location ) ; write "region 1"
let interested1 = proc( pntr location –> bool ) ; . . . ! button pressed and mouse
in region 1
let removeNotification1 := thisNotifier( addNotification )( interested1, doIt1 )
let removeNotificationRegion2 := proc() ; nullproc
let doItRegion2 =
begin
        let subNotifier = NotifierGen()
        let removeNotification2 := proc() ; nullproc
        let doIt2 = proc( pntr location ) ; write "region 2"
        let interested2 = proc( pntr location –> bool ) ; true
        ! this will be on the bottom of the sub–Notifier list and so will be the default
        removeNotification2 := subNotifier( addNotification )( interested2, doIt2 )
        let removeNotification3 := proc() ; nullproc
        let doIt3 = proc( pntr location ) ; write "region 3"
        let interested3 = proc( pntr location –> bool ) ; . . . ! mouse in region 3
        removeNotification3 := subNotifier( addNotification )( interested3, doIt3 )
        let removeNotification4 := proc() ; nullproc      ! as above
        removeNotification4 := subNotifier( addNotification )( interested4, doIt4 )
        proc( pntr location ) ; subNotifier( doAction )
  end

  let interestedRegion2 = proc( pntr location –> bool ) ; . . . ! button pressed in region 2
  removeNotificationRegion2 := thisNotifier( addNotification )( interestedRegion2, doItRegion2 )

  ! event monitor
  let theEventMonitor = eventMonitorGen( proc( –> bool ) ; false, thisNotifier( doAction ) )
  theEventMonitor()
```

2. Window Manager

2.1 Window Manager Overview

The window manager is responsible for controlling an overlapping window system. When the window manager is initialised, a package of procedures is returned which create and manipulate windows within the system. An interactive window manager is also available, which provides a convenient user interface to the window system.

2.2 Modules In The System

The system as a whole is constructed from a number of procedures stored in the database. So that a procedure can access others, a number of pointers are passed into it. These point to structures containing procedures ; the procedures are read from the database by the top-level program and then passed down to the procedures which need them.

The structure classes used are given in section **6**.

2.3 Window Manager Interface

The window manager is of the following type :

```
windowManagerGen : proc( pntr,pntr,pntr,pntr,pntr -> pntr )
! windowManagerGen : proc(
                              higherWindow,aNotifier,defaultsPack,utilitiesPack
,
                              makeBorderPack -> WindowManagerPack )
```

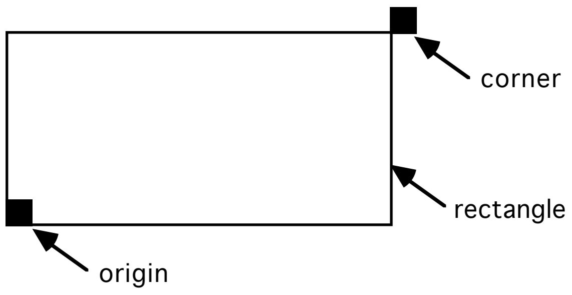
The procedure is passed a pointer to the window in which the window manager is to operate. This allows nesting of window managers. In a program which uses a window manager at top level, a dummy instance of a **window** structure (described below) should be set up and passed to the generating procedure. This window would contain dummy procedures in all its fields except those concerned with drawing onto the window.

A pointer to a Notifier is also passed, along with pointers to the packages containing the procedures needed by the window manager generator.

windowManagerGen returns an instance of

```
structureWindowManagerPack(          proc( pntr, int, bool -> pntr )
createWindow;
                               proc ( pntr )                              bri

                               openWindow,zapWindow,makeCurrent;
                               proc( pntr -> int ) findPriority;
                               proc( pntr -> pntr ) getWindow;
                               proc( pntr, pntr ) resize,putIconPos,resizeProc
)
```

createWindow creates a window. The first parameter specifies the size and position, and is a pointer to an instance of **structure** Rect(**pntr** origin, corner), where *origin* and *corner* are pointers to instances of **structure** point.strc(**int** point.x, point.y), as defined in the PS-algol prelude. This represents a 'half open' rectangle : the origin point is included, but not the corner :



This facilitates tests for overlapping rectangles, and rastering operations. The second parameter is an integer representing a window style. There are eight of these :

1. A window with a title bar, a go–away box, and a resize box. This resize box is on

2. As above, but without a resize box.

3. A window with a border all round.

4. A plain window.

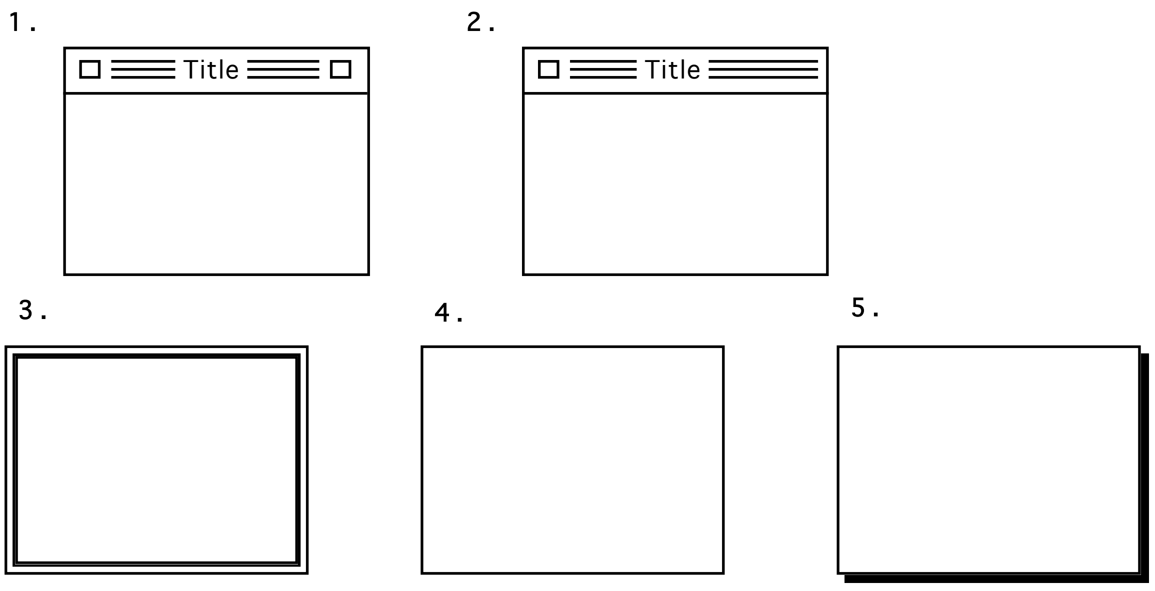
5. A window with a shadow below and to the right.

6. A window with no border.

7. As 1, but without the horizontal lines on the title bar.

8. As 2, but without the lines.

The last two styles are used when a window of one of the first two styles is not current : see section **2.5**.



The third parameter determines whether the window takes all mouse input, or just those events occurring within it. This will be used when it is necessary for one application to receive input from the user before any other application — for example, a confirmation box before quitting an application.

bringToFront redraws the specified window in front of all other windows.

putToBack redraws the specified window behind all other windows.

closeWindow removes the specified window from the screen and displays its icon at the position of the window. Each window has an associated icon, as described in section **2.4**.

openWindow removes an icon from the screen and draws its associated window at its previous position and level with respect to other windows.

zapWindow destroys a window ; the application running in it is lost.

makeCurrent makes the given window the current window ; the notification for its application is put on the Notifier stack, and the previous one is taken off.

findPriority takes a window and returns its priority *i.e.* its level with respect to the other windows. The levels are numbered from 1 for the top window, increasing towards the background window.

getWindow takes a screen position in world coordinates and returns a pointer to the upper–most window at that position. It will return **nil** if there is no such window.

resize takes a window and a screen rectangle and redraws the window within that rectangle. If the size of the the window has been changed, a new blank image is created and the contents of the old window are copied onto it at the bottom left corner ; if the new window is smaller, then the appropriate part of the old image, again starting at the bottom left corner, is copied. In all cases, a procedure, provided by the application running in the window, is called to allow the application to adjust the window's contents if this becomes necessary.

putIconPos sets the position of the icon.

resizeProc is the procedure described above which is called when a window is resized.

2.4 Data Structure

The window manager keeps a doubly–linked list of all the windows currently visible in the system, in order from front to back. Each item in the list has a pointer to a visible–list containing just the parts of the window which are currently unobscured. Every time a new window is created, it is placed on top of the main list, with its visible–list consisting of one segment only, the whole window. Then every other visible segment in the whole structure is checked against this one, and if overlapping occurs, then the segment is replaced by a sub–list of its remaining visible sections.

The check for two windows' overlapping can be achieved in eight comparisons — four to check whether they overlap at all, and another four to show in what manner they overlap.

Each item in the list also has pointers to

1. a **Layer** structure (defined below).

2. the window in front of the window.

3. the window behind the window.

A **Layer** structure contains:

1. the image of the contents of the window ; this is a limit of image **2**;

2. the image of the whole window, with border and contents;

3. the image of the icon for the window;

4. a **Window** structure for the window;

5. a **Rect** structure (as defined in section **2.3**) for the window;

6. a **Rect** structure for the icon;

7. a **Rect** structure for the border;

8. a boolean which specifies whether the window is currently open or closed (*i.e.* whether

9. a boolean which specifies whether the window takes all mouse input or just those events

structure Layer(**#pi**

pntr winProcs, winRect, borderRect, iconRect;

bool windowOpen, takesAllInput)

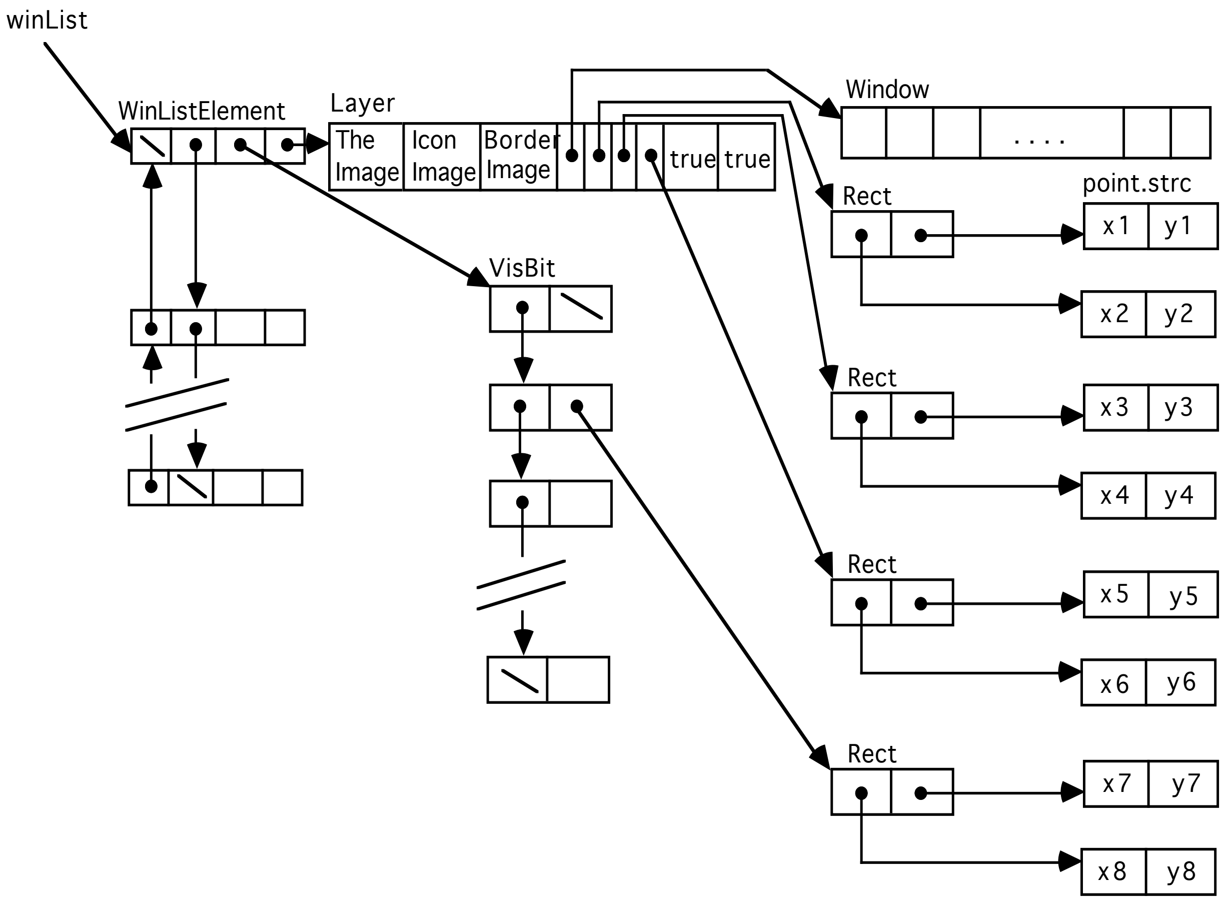
structure WinListElement(**pntr** layer, visList, fWin, bWin)

structure VisBit(**pntr** visBit, nextBit)

Instances of VisBit are used in the visible list of each window, to hold Rect structures representing visible rectangles on the screen.

On initialisation, a single background layer is put on the window list.

Here is the data structure described above :



The result of the window creation procedure is a pointer to a window structure, as follows :

```
structure Window(                                                                       pr
                    proc( string ) putTitle ;
                    proc( –> pntr ) getPos, getSize, getIconPos ;
                    proc( –> proc( pntr ) ) getApplication ;
                    proc( proc( pntr ) ) putApplication ;
                    proc( –> proc( pntr,pntr ) ) getResizeProc ;
                    proc( proc( pntr,pntr ) ) putResizeProc ;
                    proc( –> #pixel ) getContent, getCursor, getIcon ;
                    proc( #pixel ) putContent, putCursor, putIcon ;
                    proc( #pixel,int,int,int,int,int ) rasterTo ;
                    proc( int,int,int,int,int ) rasterLine ;
                    proc( int ) putCursorRule ;
                    proc( –> int ) getStyle ;
                    proc( –> bool ) windowIsOpen )
```

getTitle	returns the title of the window.
putTitle	sets the title of the window.
getPos	returns the current origin point of the window - this may
be negative if	the window is partially off the screen.
getSize	returns the current dimensions of the window.
getIconPos	returns the position of the icon.
getApplication	returns the application associated with this window.
putApplication	sets the application associated with this window.
getResizeProc	gets the application's procedure for adjusting the
window's contents	when a window resize has occurred, as described in the
previous	section.
putResizeProc	sets the above procedure.
getContent	returns the image containing the window's contents.
putContent	sets the window's contents to be the specified image (this
should	normally be called with the image returned by
getWinContent — this	prevents a new image from being generated where not
required).	
getCursor	returns the image containing the cursor's contents.
putCursor	sets the cursor's contents to be the specified image (this
should be	called with the image returned by getWinCursor).
getIcon	returns the image containing the icon of the window.
putIcon	sets the window's icon to be the specified image (this
should be called	with the image returned by getIcon).
rasterTo	rasters the given image onto the contents of the window
at the given	position ; without this procedure the entire window
contents would	have to be copied in order to update one pixel. The last
parameter	specifies the raster operation to be used, as follows :

0	**copy**	4	**rand**
1	**nor**	5	**nand**

2	**xor**	6	**not**
3	**ror**	7	**xnor**

rasterLine	draws a line between two given points onto the contents
of the	window, using the given drawing rule (**rand**, **ror**, or **xor**
).	
putCursorRule	specifies how the cursor image should be combined with
the	window's image. (This is not implemented in the current
system : the	cursor is always **xor**'d onto the screen.)
getStyle	returns the window's style.
windowIsOpen	returns a boolean specifying whether the window is open
or closed.	

The window manager places a default application — an empty procedure — for the window, which may be changed later using **putApplication**. The system has a notion of a current window, which will normally take all character input, and only the application therein will be active. The only exception to this is when a window has been created to take all mouse input, as described in section **2.3**. The current window is the only one whose application has an entry in the Notifier ; when another window is made current, the previous entry is removed from the Notifier, and the new application registered. This prevents the Notifier list from becoming too long.

2.5 Interactive Window Manager

The interactive window manager provides an interface to the procedures made available by the window manager package.

A window is made current by clicking in it. This operation does not bring the window to the 'front' of the screen.

For opening windows, and altering them when partially obscured, a background menu is provided. This contains the following options :

Create — the user clicks at one corner of the window, moves to the opposite corner, and clicks again.

Delete removes the window clicked on, and its associated application.

Resize — click near the appropriate corner or edge, move to the new position, and click again.

Move — click within the window, move to the new position, and click again.

Push / Pop brings a window to the front if it is not there already ; if it is then it is put to the back.

Iconise closes the window clicked on and displays its icon.

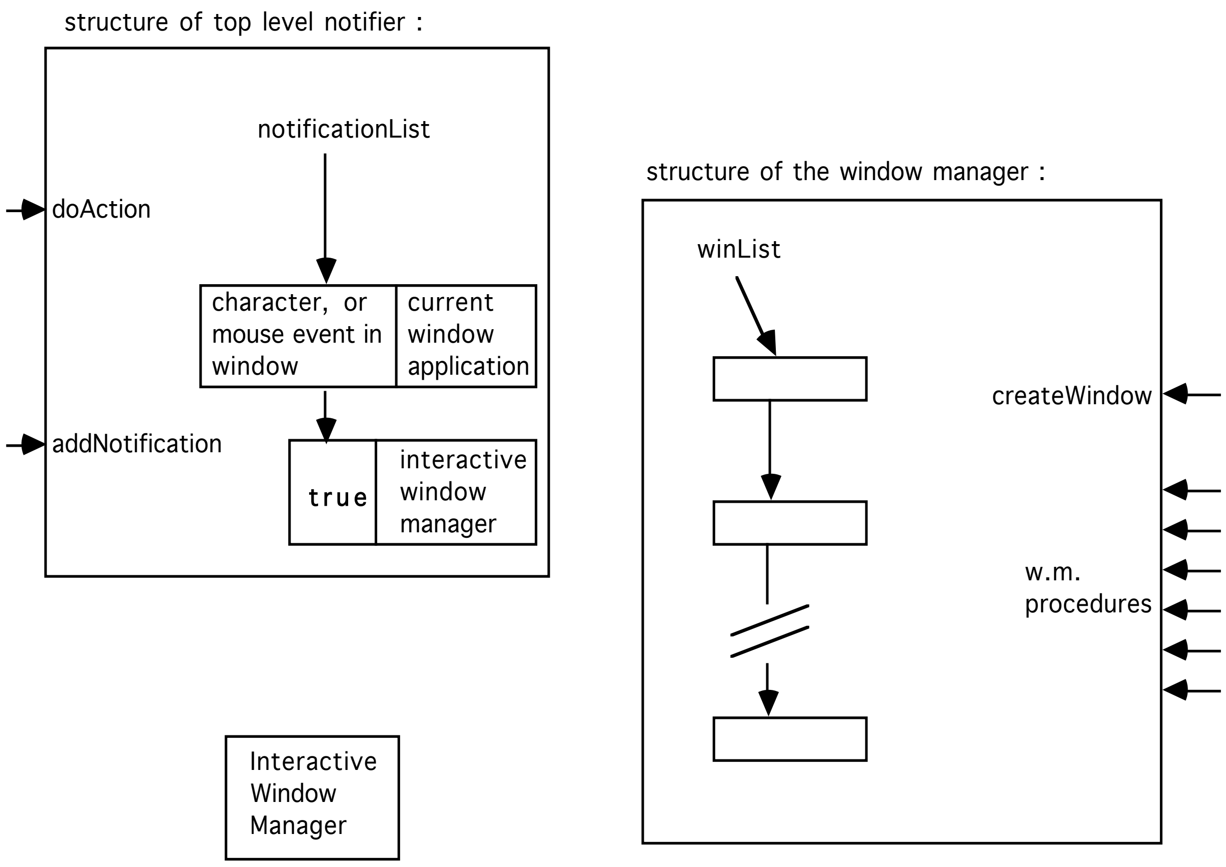
Quit quits the interactive window manager.

When an option has been chosen the cursor will change appropriately. This menu facility is provided in addition to the drag bar, grow, and go–away box.

The system will function with only one mouse button through the use of the background menu. Some of the operations can also be performed without bringing up a menu : to push/pop, click button 1 on the border ; to iconise, click button 1 on the left–hand box in the title bar ; and to resize, click button 1 on the right–hand box. If there is more than one mouse button, then moving can be done by clicking button 2 on the border of a window.

When a window has been iconised, the icon can be reopened by clicking button 1 over it. If there is a button 2, the icon can be moved by clicking button 2 on it, moving, and clicking button 2 again.

The relation of the window manager to the top–level Notifier is shown overleaf.


structure of top level notifier :
notificationList
doAction
character, or mouse event in window
current window application
addNotification
true
interactive window manager
Interactive Window Manager
structure of the window manager :
winList
createWindow
w.m. procedures

2.6 Coordinate Spaces

The window manager is required to do the mapping from world to local coordinate space. When a window is supplied with an Application via the **putApplication** procedure it wraps the procedure up in another procedure which does coordinate translation. The **grow** and **move** operations in section **2.4** above may also change this enclosing procedure. In this way all the applications may be written without reference to world coordinates.

3. Tiling Window Manager

In addition to the overlapping–window manager there is also a requirement for a tiling manager which manages non–overlapping areas. This is needed in the context of panel positioning (*e.g.* text subwindows, buttons, sliders etc.), and is similar to the main window manager except that it ensures that there are no overlapping areas. The manager operates on the same **window** structures as does the window manager. Unlike the window manager, there is no current tile. When a tile is created, its application is registered with the tiling manager's Notifier, and it then remains registered until the tile is destroyed.

3.1 Interface

The tiling manager has the following interface :

tileManagerGen : **proc**(**pntr**,**pntr**,**pntr**,**pntr**,**pntr** -> **pntr**)

! tileManagerGen : proc(*hig*

utilitiesPack –> TileManagerPack)

The parameters to **tileManagerGen** are pointers to the window in which the tile manager is to operate, to a Notifier for that window, and to the necessary procedure packs.

tileManagerGen returns an instance of

structure TileManagerPack(**pr**

proc(**pntr**) closeTile,openTile,zapTile;

proc(**pntr** –> **pntr**) getTile;

p r o c (p n t r , p n t r)
resizeTile,putTileIconPos,resizeTileProc)

If a tile can not be created at the given point without overlapping another, **nil** is returned. The other procedures operate in the same way as the equivalent ones in the **WindowManagerPack**, except that calls which request an illegal operation (such as moving a tile on top of another) have no effect.

3.2 Data Structure

The data structure used is similar to that inside the window manager, but as no tiles can overlap, there is no need for lists of the visible parts.

The PS-algol structures are :

```
structure TileInfo(                                                    #pi
                  pntr tileProcs,tileRect,borderRect,iconRect;
                  bool tileOpen,takesAllInput;
                  proc() removeTileNotification )
structure TileListElement( pntr tileInfo,fTile,bTile )
```

Because there may be more than one tile application registered with the tiling Notifier at a given time, the procedure which removes a tile's notification has to be stored with the other information about the tile, rather than held in a single variable as is the case with the window manager. The other fields in **TileInfo** correspond to the equivalent ones in **Layer**.

The tile information is held as a linked list of instances of **TileListElement**, although the order is now arbitrary, as none of the tiles overlap.

4. Panels

A number of panel items are available as predefined applications.

4.1 Types of Panel Item

4.1.1 Buttons

A light–button associates an area of a window with a procedure. When a light–button is selected the user receives visual feedback *e.g.* the button is highlighted, and the associated procedure is executed.

4.1.2 Choices

Choices allow the user to select one item from a list of choices. The choice may be presented as radio buttons, cycle items, knobs *etc*.

4.1.3 Sliders

These provide a choice with a continuous range of values. This could be presented as a scroll bar or a knob with no discrete states.

4.2 Interface

The general interface for each of these types is shown below. There will be a number of procedures available for various styles within each group.

lightButton : **proc**(**pntr**,**pntr**,**proc**() –> **pntr**)

! lightButton : ***proc****(stringIm,position,buttonProc –> PanelStatus)*

choice : **proc**(***pntr**,**pntr**,**proc**(**pntr**) –> **pntr**)

! choice : ***proc****(stringIm,position,processNewState –> PanelStatus)*

slider : **proc**(**pntr**,**int**,**int**,**real**,**real**,**proc**(**pntr**) –> **pntr**)

! slider : proc(position,xSize,ySize,min,max,processNewState –> PanelStatus)

There are generating functions with the type :

lightButtonGen : **proc**(**pntr**,**pntr**,**pntr**,**pntr** -> **proc**(**pntr**,**pntr**,**proc**() -> **pntr**))

! lightButtonGen : ***proc****(tileManager,thisNotifier,defaultsPack,utilitiesPack –> lightButton)*

choiceGen : **proc**(**pntr**,**pntr**,**pntr**,**pntr** –> **proc**(***pntr**,**pntr**,**proc**(**pntr**) –> **pntr**))

! choiceGen : ***proc****(tileManager,thisNotifier,defaultsPack,utilitiesPack –> choice)*

sliderGen : **proc**(**pntr**,**pntr**,**pntr** –> **proc**(**pntr**,**int**,**int**,**real**,**real**,**proc**(**pntr**) –> **pntr**))

! sliderGen : ***proc****(tileManager,thisNotifier,defaultsPack –> slider)*

All the generating procedures take pointers to a tile manager and its Notifier, and the procedures which actually make the panel items take a position within the tile manager's parent window. Each procedure creates a new tile for the item.

Each of the procedures returns a pointer to a structure containing information about the placing of the item : whether it was placed successfully ; if so the position and size of the item ; a procedure which will remove the panel item ; and a procedure to set the value of the item. This set procedure is not necessary for light–buttons, so in their case a null procedure is returned. The structure is :

structure PanelStatus(**bo**
proc() removeItem ; **proc**(**pntr**) setItem)

The **setItem** procedure takes a pointer to **structure** IntHolder(**int** theInt)
when the item is a choice, and a pointer to **structure** RealHolder(**real** theReal)
when the item is a slider.

The **stringIm** parameter points to a string or an image which is to be displayed as the item label ; in the case of the **choice** procedure, this is a vector of such pointers, one for each possible choice.

The **lightButton** procedures take a procedure which is executed whenever the button is 'pressed'. The **slider** and **choice** procedures take a procedure which is executed whenever a change is made to the state *i.e.* when the slider is moved to a different position or a different choice item is selected. This procedure is passed the new state wrapped up in an instance of **IntHolder** or **RealHolder** as described above.

An application which sets up a slider or a choice item has no way of determining the current state of the panel item. Rather than polling the item for its state and then taking some action on the basis of that state, the application specifies what that action should be in the **processNewState** parameter passed when setting up the panel item. The application may then set the current value of the panel item, but otherwise it can 'forget about' the item's existence.

The **slider** procedures also take parameters which determine the size of the slider, and the range of values the slider can take. If the width of the slider tile is greater than or equal to its height, the slider will move horizontally ; otherwise it will move vertically. If the minimum value given is greater than the maximum, the slider will be reversed so that the low values will be to the right or the top as appropriate. If the minimum and maximum values specified are equal, no slider will be created.

There is also a menu–generating procedure with the type :

menuGen : **proc**(**pntr**,**pntr**,**pntr**,**pntr**,**pntr**,**pntr** –> **proc**(**pntr**,***pntr**,***proc**() –> **proc**(**pntr**)))

*! menuGen : **proc**(windowManager,defaultsPack,makeBorderPack,*
panelPack,systemPack,utilitiesPack –> menu)

The procedure which is returned takes a string or image (held in a **structure** StringHolder or **structure** ImageHolder as described in the final section) for the title, a vector of strings or images for the entries, a vector of procedures, and returns a procedure which draws the menu at the specified position.

5. Text Editing Package

5.1 Text Editing Interface

The text editing functions are accessed by passing the necessary procedure packages to an editor generating procedure.

The interface is shown below.

editorGen : **proc**(**pntr**,**pntr**,**pntr**,**pntr**,**pntr**,**pntr** –> **pntr**)
*! editorGen : **proc**(* *def*
windowDefaultsPack,windowUtilitiesPack,utilitiesPack –>
EditorPack)

editorGen returns an instance of
structure EditorPack(**pr**
proc(**pntr**) putRecord,cutText,pasteText,copyText;
proc(**string**,**pntr**) insert;
proc(**pntr**,**pntr**) select;
proc() new;
proc(**string**) search;
proc(**int**) seek;
proc(–> **string**) readChar,readLine,peekChar;
proc(–> **bool**) endOfText;
proc(**int** –> **pntr**) getMarker;
proc(**pntr** –> **pntr**) makeWindowEditor)

makeEditor returns a pointer to a new instance of **EditorPack** — with
its procedures operating on an initially empty text data structure.

getRecord returns a pointer to the data structure operated on by the
current incarnation of the editor.

putRecord makes the editor work on the data structure indicated by the
given pointer.

cutText removes the selected text and puts it in the given buffer.
This buffer is another instance of **editorPack**. Any text already in the
buffer is removed.

pasteText removes any text which is selected and then inserts the text
in the given buffer at that point.

copyText copies the selected text into the given buffer.

insert	takes a string and pointer to a font and inserts the string in that font after the current selection.
select	selects the text in the given range, specified by two **Markers** — as described in section **5.3**.
new	erases the text in the current document.
search	searches for the given string. If it is found, the first occurrence becomes the selection.
seek	moves the selection so that no text is selected and new text will be inserted before the given position (specified in number of characters from the beginning of the text).
readChar	returns the first character after the current selection. If the end of the text has been reached the null string is returned. The insertion point is moved to after the character returned.
readLine	returns a string containing the text from the end of the currentselectionup to, but not including, the next newline character. The selection is moved to just after the newline character (with no text selected).
peekChar	returns the same result as **readChar**, but the selection is not changed.
endOfText	returns true if there is no more text after the current selection.
getMarker	returns a pointer to a **Marker** structure corresponding to the character at the given position in the text.

makeWindowEditor ret

```
structure WindowEditorPack(    proc( pntr ) winCutText,winPasteText;
                          proc( string,pntr ) winInsert;
                          proc( pntr,pntr ) winInvert;
                          proc( int ) winScroll,winSeek,winSplitView;
                          proc()                                        wi
                          proc( string ) winSearch;
                          proc( pntr -> pntr ) winFindMarker )
```

winScroll scrolls the text in the window up by the given number of lines — which can be negative for downwards scrolling.

winSplitView splits the window (or the bottom sub–window if it is already split) in two horizontally at the given y–coordinate and puts a copy of the current document in the new sub–window. Changes made to text in one sub–window will be reflected in others.

winUnSplitView un

winRedisplay redraws the contents of the window.

winFindMarker returns a **Marker** to the character nearest to the given window position (given as an instance of **point**.**strc**).

The other procedures do exactly the same as their counterparts in **EditorPack**, except that changes are also reflected in the text displayed in the window. The procedures for splitting and combining sub–windows have not yet been implemented.

5.2 Data Structures

There are two distinct data structures — one representing the text of the entire document, and the other holding the information about the current contents of a particular window.

The text is stored as a doubly–linked list of strings, each string being one text line *i.e.* all the characters between two carriage returns.

The display information is stored in a vector of pointers to window–lines, one element for each line in the window. For each of these lines there is a marker pointing to the first character of that line within the text data structure (a marker will be a pointer to a text line with an index into that line).

The information for an instantiation of the text–editor is held in an instance of **TextRecord.** This has fields pointing to the text structure, to the positions of the beginning and end of the currently selected text, and to the current font. Originally, it was planned that the editor would be capable of using multiple fonts ; however, a less ambitious version using a single fixed–width font has been implemented. The **current font** field is a hangover from the original specification, and can be ignored.

5.3 Editor Structures

<u>Text Structures</u>

structure Font(**int** Font.height,Descender ; ***#pixel** The.chars ; **string** Info)

! copy of font structure but with variable fields so can find width of characters without

*! calling **variable.image***

structure TextLine(**pntr** previousLine,nextLine,fontInfo ; **string** theLine)

! previousLine,nextLine : TextLine ; fontInfo : now obsolete

<u>Window Structure</u>

```
structure WindowRecord(                                                    pn
                    *pntr windowLines ; bool insPointInWindow )
```

General Structures

structure TextRecord(**pntr** currentFont,firstSelection,lastSelection,firstLine)

! currentFont : font ; firstSelection,lastSelection : Marker ; firstLine : TextLine

structure Marker(**pntr** line ; **int** offset)

! line : TextLine

Points to a character within the text.

structure EditHandle(**pntr** textRecord)

structure DrawStringInfo(**int** charsMissed,pixelsDrawn ; **bool** lastCharDrawn)

structure WindowEditHandle(**pntr** windowRecord)

6. PS-algol Structures

```
structure CursorPack( #pixel                                              Wi
                             Create,Delete,MenuPointer,Quit,TheEnd,PushPop,Ic
onise )

structure DefaultsPack(  int                                              Ico
                           NoBorderStyle,MinWinWidth,MinWinHeight;
                         pntr TheFont;
                         #pixel BackgroundPicture,DefaultIcon )

structure Dimensions( int sizeX,sizeY )

structure MakeBorderPack(     proc(   pntr -> #pixel ) MakeNoBorder;

                            proc( pntr,pntr -> #pixel )                   Ma

                                               MakeShadowBorder;

                            proc( pntr,pntr,pntr,string ->

                                  #pixel ) MakeTitleResizeQuitBarsBorder,

                                           MakeTitleQuitBarsBorder,

                                           MakeTitleResizeQuitBorder,
```

MakeTitleQuitBorder)

```
   structure EditorProcsPack(      proc( pntr -> proc( -> pntr ) ) GetRecord;
                              proc( pntr -> proc( pntr ) ) PutRecord;
                              proc( pntr,proc( pntr ) -> proc( pntr ) ) CutText;
                              proc( pntr,proc( pntr ),proc( string,pntr ) ->
                                   proc( pntr ) ) PasteText;
                              proc( pntr,proc( -> string ),proc( -> string ) ->
                                   proc( pntr ) ) CopyText;
                              proc( pntr,pntr -> proc( string,pntr ) ) Insert;
                              proc( pntr -> proc( pntr,pntr ) ) Select;
                              proc( pntr,pntr -> proc() ) New;
                              proc( pntr,proc( -> string ) -> proc( string ) )
Search;
                              proc( pntr -> proc( int ) ) Seek;
                              proc( pntr -> proc( -> string ) )
ReadChar,ReadLine,PeekChar;
                              proc( pntr -> proc( -> bool ) ) EndOfText;
                              proc( pntr -> proc( int -> pntr ) ) GetMarker;
                              proc( pntr,pntr,pntr,pntr,pntr,proc( pntr ),proc(
pntr ),
                                   proc( pntr ),proc( string,pntr ),proc(
pntr,pntr ),
                                   proc( int ),proc(),proc( string ) ->
                                   proc( pntr -> pntr ) ) MakeWindowEditor )

   structure ImageHolder( #pixel theImage )



   structure PanelPack(       proc(                                              pn
                              proc( pntr,pntr,proc() -> pntr ) ) LightButtonGen;
                         proc( pntr,pntr,pntr,pntr,pntr,pntr ->
                              proc( pntr,*pntr,*proc() -> proc( pntr ) ) )
MenuGen;
                         proc( pntr,pntr,pntr ->
                              proc( pntr,int,int,real,real,proc( pntr ) -> pntr ) )
SliderGen;
                         proc( pntr,pntr,pntr,pntr ->
                               proc( *pntr,pntr,proc( pntr ) -> pntr ) )
ChoiceGen )
```

```
   structure StringHolder( string theString )

   structure UtilitiesPack(        proc(                                                                pn
                              proc( pntr,pntr,pntr ->
                                    proc( pntr,int,string ->    #pixel )    )
MakeBorderGen;
                              proc( pntr  -> proc( pntr,pntr,int -> int ) )
FindPositionInWinGen;
                              proc( pntr  -> proc( pntr,int  -> pntr )    )
ImageToBorderSizeGen;
                              proc( pntr,string -> #pixel ) StringToTiler )
```

```
  structuSystemPack(      proc(                        –>    p n t r   )
NotifierGen;
                     proc( pntr,pntr,pntr,pntr,pntr –> pntr )                Wi
                                                       TileManagerGen;
                     proc(
                     pntr,pntr,pntr,pntr,pntr,pntr,pntr,pntr,pntr,pntr –>
                          proc() ) InteractiveWindowManagerGen;
                     proc( proc( –> bool ),proc( pntr )  –>  proc() )
EventMonitorGen;
                     proc( pntr,pntr,pntr,pntr,pntr,pntr –> p n t r  )
EditorGen )


  structure WindowDefaultsPack(                                              pn
                            int   InterLineSpace,   CharacterWidth,
CharacterHeight )

  structure WindowEditorProcsPack(proc( pntr,pntr,pntr,pntr,proc( pntr ) –>
                                       proc( pntr ) ) WinCutText;
                                  proc( pntr,proc( pntr ) –>
                                       proc( pntr ) ) WinPasteText;
                                  p r o c(  p n t r ,p n t r ,p n t r ,p n t r ,p r o c(
string,pntr ),proc( int ) –>
                                       proc( string,pntr ) ) WinInsert;
                                  proc( pntr,pntr,pntr –> proc( pntr,pntr )
) WinInvert;
                                  proc( pntr,proc( int ),proc( pntr,pntr ) –>
                                       proc( int ) ) WinSeek;
                                  proc( pntr,pntr,pntr,pntr –> proc( int ) )
WinScroll;
                                  proc( pntr –> proc( int ) ) WinSplitView;
                                  proc( pntr,proc() –> proc() ) WinNew;
                                  proc( pntr,pntr,pntr,pntr –>  proc()  )
WinRedisplay;
                                  proc( pntr –> proc() ) WinUnSplitView;
                                  proc( pntr,proc( string ),proc( pntr,pntr )
–>
                                       proc( string ) ) WinSearch;
```

```
                                                  proc( pntr,pntr,pntr ->
                                                       p r o c ( p n t r   -> p n t r )    )
WinFindMarker )

  structure WindowUtilitiesPack(       proc(                                              pn

                                       proc( pntr,pntr,string,int,int -> pntr ) )

DrawStringGen;

                                  proc( pntr,pntr,pntr,pntr ) MoveArea;

                                  proc( string -> bool ) WordBreak;

                                  proc( pntr,pntr,pntr ) BlankOut;

                                  proc( pntr,pntr,pntr,pntr ->

                                       proc( pntr,pntr,int,int ) ) DrawText )
```

Appendix : Setting Up The System

This section describes how to use the facilities provided by the architecture, and how to set up the system initially.

1. Setting Up the Window–Manager

The following files must be present:

backgroundImageGen
borderProcs
choiceGen
createDb
cursorImageGen
defaults
eventMonitorGen
findPositionInWinGen
iconImageGen
imageToBorderSizeGen
interactiveWindowManagerGen
lightButtonGen
makeBorderGen
makeSubWindowManagerGen
menuGen
notifierGen
sliderGen
stringToTile
tileManagerGen

backgroundCursor
borderCursor
createCursor
deleteCursor
iconiseCursor
menuPointerCursor
moveCursor
pushPopCursor
quitCursor
resizeCursor
theEndCursor
windowInteriorCursor

INIT
COMP

interactive

To compile the appropriate set–up programs, type	*COMP*
To run them, type	*INIT*
To run the interactive window–manager, type	*psr interactive.out*

2. Using the Window–Manager

The interactive window–manager is set up, as a default, to run a system in which the user is presented with a choice of demonstration applications after opening a window.

The user–interface of the interactive window–manager is described in section **2.5** of the specification document.

An application can be run in a window using the **putApplication** procedure of that window. There are some examples in the directory called **demos.**

Running Simple Applications

An example is given in the file **plainFlashDemo**.

This program creates a window and sets its application to be a procedure which makes the interior of the window flash on and off whenever the left–hand mouse button is held down while the cursor is within the window. The program terminates when a character is typed at the keyboard.

As explained in the specification document, the window–manager runs in a notifier–based system. This means that applications are passive and input–driven ; they are called whenever there is input relevant to them. A program in this style has a top–level notifier which distributes input events to the appropriate applications. An application may in turn incorporate a sub–notifier.

There are procedures stored in the database which generate notifiers, event–monitors (an event–monitor is a procedure which repeatedly scans for input and passes any received to a top–level notifier. One event–monitor is needed in each program.), window–managers, tile–managers, *etc*..

The demonstration program reads in these procedures and then calls them to generate a notifier and a window–manager. The window–manager generating function is passed a dummy window which occupies the whole screen, and whose only procedures defined are those concerned with updating the contents of the window. This is needed because both window–managers and tile–managers must be given a window of a higher–level window/tile–manager to run in ; this enables nesting of window–managers.

A window is created using the **createWindow** function of the window–manager, and the window is made the current window. The current window is the one which receives all character input ; mouse input is sent to the window containing the cursor, if any. Next, a boolean variable **finished** is initialised. A procedure which determines whether the event–monitor should terminate is passed to the event–monitor generator ; this procedure returns the value of this variable. By altering its value (that of the boolean variable) an application can control the termination of the event–monitor and hence of the program as a whole.

The application itself is declared next. The procedure takes a pointer, which will be to either a **mouse** structure for a mouse event, or a **Chars** structure for a keyboard event. If a keyboard event is received the procedure destroys the window by calling the **zapWindow** procedure of the window–manager, and then sets **finished** to **true**, so that the program will halt on the next cycle of the event–monitor. If a mouse event is received and the left button is pressed down, the contents of the window are inverted. This is done by getting the window image with **getContent**, inverting this image, and then calling **putContent** to propagate the change to the screen. The **putContent** procedure can be passed any image, but if the image is the one already held by the window–manager for that window, as in this case, the update will be quicker. This is because the image has only to be copied to the screen, and not onto the internal copy as well.

The application is then placed with the window using the **putApplication** procedure, and finally an event–monitor is generated and called to start the application.

Running Applications with the Interactive Window–Manager

The running of applications within windows created interactively is illustrated in the program **interactiveFlashDemo**. This is the same as the one which makes nested interactive window–managers, except that the application put into each window has been changed to the flash application used in the last example. The application itself is defined between lines 416 and 427 ; this is the only section which would have to be changed to run a different application. Running the program writes that version of the interactive window–manager into the database. To run the window–manager, run the program **interactive.out** in the main directory.

Another example is the program **bounceDemo** : this creates a bouncing ball within each window.

Nesting Window– and Tile–Managers

In the program **tileFlashDemo** a window is created within which a tile–manager is run. Three tiles are created using this, and within each of these the flash application is run.

The main window–manager, its notifier, and a window are created as before ; next the tile–manager is created. The procedure which generates this takes the window already created, and another notifier, as well as the necessary procedure packages as described in section 2.2 of the specification. Three tiles are created using the tile–manager, and an application given to each one. Finally the **doAction** procedure of the tile–manager's notifier (the details of notifiers are explained in section 1 of the specification) is passed to the window as its application.

This general pattern can be followed to give nesting of window/tile–managers to any depth. A tile–manager can be used in any situation where a window–manager can, and *vice–versa*, with the obvious constraint that the windows created with a tile–manager cannot overlap. (The terms **window** and **tile** are used interchangeably ; windows and tiles are instances of the same structure).

Keyboard Input with Tiles

There is no concept of current tile in the way that a window–manager has a current window. Because of this, there is no way to decide which tile should receive any keyboard input. Normally, then, keyboard input to a tile–manager is discarded, and tiles receive only mouse events within their boundaries. If a tile has been created to take all input, however, it will receive all mouse *and* keyboard events.

If an application in a tile which does not take all input needs to read keyboard input, it must place an entry on the notifier stack to enable it to do so. This is illustrated in the program **tileCharInputDemo**. This is similar to **tileFlashDemo**, except that the application in the bottom tile now reads a letter from the keyboard when activated. It does this by creating a new application. The **interested** procedure for the new application, *i.e.* the procedure which determines whether the application should be called with a given event, returns **true** if the event is a keyboard event. The application itself first removes its entry from the notifier stack, and then if the character passed to it is "q" it sets the **finished** variable to **true**, thus making the event–monitor terminate.

The applications in the top two tiles, then, function as before — the interior of the tile is flashed when the mouse button is held down with the cursor inside the tile. When the mouse is clicked in the bottom window the application places the notification on the notifier stack which will lead the application to be called with the next keystroke. If the character typed is a "q" the program halts.

Light–Buttons

Procedures are available to generate light–buttons, sliders and menus. The use of the light–button generator is demonstrated in **lightButtonDemo**.

First, a tile–manager is generated, and the light–button generator is called to make a generator specific to that tile–manager. The parameters of this specific generator are the image or string to be displayed as the button title, held in a structure (**structure** stringHolder(**string** theString) or **structure** imageHolder(**#pixel** theImage)) ; the position of the bottom–left corner of the button (in the coordinates of the tile–manager) ; and the procedure which is to be executed when the button is 'pressed'. When called the generator returns a pointer to a structure containing the location and size of the button, a boolean which specifies whether the button was created successfully (*e.g.* if the requested location for the button overlapped another, this would be **false**), a procedure which when called will remove the light–button, and another procedure which will be undefined. This last field in the structure is used by slider generators *etc.* for a procedure to set the value of the slider. A light–button has no associated value to set.

Three light–buttons are created. The first writes out 'button pressed' when pressed ; the second removes the first light–button, and the third makes the program terminate.

Sliders

Their use is illustrated in **sliderDemo**.

As in the last example, a tile–manager is created and a slider generator specific to it is made. This generator takes as parameters the position of the bottom–left corner of the slider, the dimensions of the slider, the minimum and maximum values of the slider (real values), and a procedure which will be called whenever the value of the slider changes. The procedure will be passed the new value. If the minimum value specified is greater than the maximum, the slider will be reversed so that the values increase towards the left. If the height specified is greater than the width, the slider will be drawn so that the bar moves vertically.

In the example, two sliders are defined, one lying along the bottom edge of the window, and one along the left edge. Together they control the position of a small square in the window. This is achieved by having the two procedures which are called when the slider values change access common data.

Menus

See **menuDemo**.

Menus operate within window–managers, so as before a window–manager and a menu generator specific to it are created. This generator's parameters are : a string or an image held in the appropriate structure, for the menu title ; a vector of such strings or images for the entries in the menu ; and a vector of procedures with an element for each menu entry, which will be executed when the entry is selected. The generator returns a procedure which takes a pointer to a window position and displays the menu at that point. The menu is drawn below and to either side of the specified point *i.e.* the point becomes the top–centre of the menu.

A menu with four entries is created, and finally a procedure is registered with the notifier which, when the mouse button is pressed at a given point, displays the menu at that point, and halts the program when a key is pressed.

3. Setting Up the Editor

The following files are necessary ; they should be present in the **editor** directory:

- blankOut
- copyText
- cutText
- drawStringGen
- editorGen
- endOfText
- getMarker
- getRecord
- insert
- makeWindowEditor
- moveArea
- new
- pasteText
- peekChar
- putRecord
- readChar
- readLine
- search
- seek
- select
- winCutText
- winFindMarker
- winInsert
- winInvert
- winNew
- winRedisplay
- winScroll
- winSearch
- winSeek
- windowDefaults
- wordBreak

To compile them, type *EditorCOMP*

To run them, type *EditorINIT*

4. Editor Functions

As described in the specification, the editor provides a package of procedures which operate on text containing various fonts. The procedures in the main package work independently of any window representation of the text. One of the procedures, however, generates a further package of functions which maintain a display of the text in a given window.

The window display functions for splitting and combining the display window have not yet been implemented.

The use of the non–window–based editor functions is illustrated in **editorDemo**.

A demonstration of those window functions implemented is given in **windowEditorDemo**.

After creating a window for the window editor to run in, a number of strings are inserted into the data structure. A window editor is generated, and the text is drawn in the window with the **winRedisplay** function. The insertion point at the end of the text is highlighted, and then an application is registered which will highlight text selected using the mouse. Text can be inserted at the insertion point by typing, and deleted using the *delete* key.