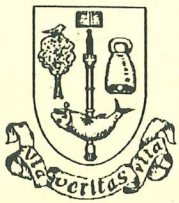
University of Glasgow

Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ



University of St Andrews

Department of Computational Science
North Haugh
St Andrews KY16 9SS

User Interface Tools in PS-algol

Cooper, R.L., MacFarlane D.K. & S. Ahmed

Persistent Programming
Research Report 56
March 1988

G. KIRBY

User Interface Tools in PS-algol

Cooper, R.L., MacFarlane, D.K. and S. Ahmed

Preface

An expanding range of tools are available to PS-algol application programmers. The three pieces of work documented in this report assist primarily in the construction of user interfaces.

Richard Cooper describes the organisation and content of a procedure library. Among the tools are editors for strings and integers, a forms interface and a chooser.

Douglas MacFarlane describes two packages for the construction of screen-oriented editors. These have been used to build a multi-window text editor.

Shahad Ahmed discusses two software toolboxes. In particular he describes his implementations of panel items, which include light buttons, radio buttons and scrollbars.

The software is available through either of the contact addresses given in the bibliography at the end of the report.

Paul Philbrow

Contents

A Utility Library for PS-algol

Richard Cooper

Abstract

When developing application programs in the Persistent Programming Language PS-algol, the programmer makes considerable use of the availability of first-class procedures in order to develop his program in a modular fashion. There is little support within the language, however, for providing a consistent environment within which to develop program modules. This document describes a systematically organised library of utility procedures which should assist in the faster development of PS-algol procedures. Both the organisation technique and the procedures in the library are described.

Most of the procedures in the library are for user interface management, including procedures for supplying help information, error messages, form and menu interfaces and various textual displays. There are also procedures for doing deep copies of objects, for providing a smooth path to the run-time compiler and for building libraries just like this one.

Introduction

The availability of first-class procedures is a great inducement to the PS-algol programmer to program in a modular style. Typically a design module of the program becomes one or more PS-algol procedures, which are implemented as one source file. This source file will contain two or three parts: possibly the retrieval of some objects from the Persistent Store (usually lower-level procedures); the source of the procedure(s); and then the export of the procedure(s) to the Persistent Store, from where they can be retrieved by higher-level procedures.

In the standard PS-algol system, little or no support is given for this method of working. Procedures are pushed into the Persistent Store somewhere and can only be retrieved by programmers who can remember where they are. This reduces software sharing and re-use and makes the structure of programs less accessible than need be. It is also true that storage and retrieval of procedures takes a significant amount of code, which should at least be systematised to reduce program overheads.

This document describes a systematic method of creating and storing PS-algol procedures, which is a first attempt to overcome these problems. It also overcomes the following binding problem. Consider two procedures of an application, *A* and *B* say, which are created in two separate modules with *B* calling *A*. If *A* is modified and re-entered into the library, *B* will still be bound to the old copy of *A*. The module which contains *B* must be re-run (but not re-compiled), in order to update the binding between *B* and *A*. Note that this problem is a consequence of providing a binding mechanism whose links cannot be broken by an outside agency. This feature of the Persistent Store is a vital aspect of the value of the notion of persistence.

To overcome these problems a utilities database has been set up to contain useful procedures. This database has a coherent structure and some programs which manipulate it have been written. These do the following:

- for each procedure which has been inserted in the database, maintain a list of all the other procedures in the library which call this one, a short description of the function of the program, and the date and time of its insertion into the library;
- any time a new version of a procedure is entered, remind the user of any calling procedures which must be rebound to the new version;
- display a list of the procedures in the database, with their information - in the form illustrated in Appendix C.

The document will proceed by describing the organisation of the database which holds the library; the programs which maintain the library; and the procedures which are in the library.

1/ The structure of the library

The library is created in the form of a PS-algol table. There is one entry in the table for each procedure in the library, keyed by the procedure name. When a library is created, the table is set up with two library system procedures, *prcget* and *prcput*, which retrieve and store procedures, respectively (the function of these procedures will be described in section 2). All the procedures, except *prcget*, are reached through a structure of the form:

```
structure intermed(
   pntr procpaki;
      ! points to the procedure packaged as below
   *string depended;
      ! a list of the procs which call this one
   string datestamp;    ! time of insertion
   string descriptor)   ! short description from prcput
```

A more sophisticated library might use a richer intermediary. The pntr field *procpaki* points to an object with one field, containing the packaged procedure. The structure of this packaging is conventionally of the form:

```
structure procpak(
   proc(...parameter types specific to proc...)xproc)
```

so that the structure name and field name is common to all the procedures and only the argument and result types vary. This packaging strategy has been used to simplify the storing and retrieving programs.

It is envisaged that any large scale application will use two sets of procedures, one a multi-user utilities library, which will be shared between applications, and one of procedures specific to the application, which will be either a separate database or a table in its own database. The structure outlined above is a general one, which can be used both for the utilities library and for an

application-specific library. The latter can be created using the *makelib* procedure described below. The utilities library is the database, "rutilities", password "friend".

2/ Software support for these structures

a) Retrieving and storing procedures - prcget and prcput

When the database is set up by *pdbmaker* (see below) it contains two procedures: *prcget*, which retrieves procedures from the database, and *prcput*, which stores them in the database. Similarly when *makelib* (see 3n below) is called it returns a table with specially named versions of these as their only entries. The function of these two procedures will now be described.

prcget is procedure which retrieves procedures. As it needs to be retrieved before any other procedures can be, it cannot retrieve itself and so is stored directly as a *procpak* structure and not via an *intermed* structure. It is retrieved by the following code fragment, which looks it up and unpackages it:

```
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record do
{write "No utilities database - do pdbmaker first'n"; abort}
let prcget=
begin
   structure procpak(proc(string->pntr)xproc)
   s.lookup("prcget",procsdb)(xproc)
end
```

The procedure is then used by code fragments like:

```
let prcput=
begin
   structure procpak(
      proc(string,pntr,*string,string)xproc)
   prcget("prcput")(xproc)
end
```

which retrieves *prcput*, described below. The first point to be noted about this clause is that from procedure to procedure, the only parts which vary are: the procedure variable name; the key to the procedure in the library; and the type description of the procedure. The first two will conventionally be the same as each other, but the fact that the type will vary from procedure to procedure means that the procedures are actually being stored as different *procpak* structures. This would normally mean that in order for more than one retrieval to occur in the same module, we would actually have to name *procpak* and *xproc* differently for each procedure type. This would complicate matters, so instead we have used **begin** and **end** (or more compactly but synonymously the curly brackets "{" and "}") to surround a block in which the structuring information appears. By the scope rules of PS-algol, this packaging disappears at the **end** (or "}") and so we may follow the retrieval of *prcput* by a similar clause which retrieves another, differently typed, procedure.

The function of *prcget* is quite simple: it looks up the procedure name in the library, dereferences the *procpaki* field, and returns the container it finds

there. The procedure itself is then referenced by looking up the *xproc* field as shown above for *prcput*.

prcput itself is a somewhat more complex procedure. It is retrieved, as above, in the same way as any of the utilities in the library. It is then used to store procedures as in the following fragment:

```
{structure procpak(proc(string,int->string) xproc)
prcput("fillstring",procpak(fillstring), vector 0::0 of "",
   "Fill out a string with spaces")}
```

This uses a similar technique of blocking the storage with "{" and "}" in order to localise the packaging which uses the same name for different structures. It contains a call to *prcput* with the following parameters:

string *procname*	- the name of the procedure;
pntr *procpntr*	- a pointer to the procedure packaged in a structure conventionally called *procpak* with a single field called *xproc*;
***string** *dependson*	- a list of the procedures this one calls;
string *desc*	- a short description of the procedure

The procedure *prcput* stores the procedure by the following steps:

(i) An empty set of dependencies (procedures which call this one) is set up - this will be filled by the subsequent insertion of procedures which call it.

(ii) Check if the procedure already exists - if not just enter it.

(iii) If it does exist, check that it is the same type - if not, the user is given the option of not entering the new procedure, as a procedure may be destroyed unintentionally - if it is of the same type, it is assumed that this is a genuine update.

(iv) The new version of the procedure is entered.

(v) A list of all the dependent procedures is printed, with a recommendation to update them as well.

(vi) The list of procedures this one calls is scanned and the name of the procedure is entered into their lists of dependencies.

(vii) Changes are committed to the persistent store.

b) The initialising program - dbmaker
The program is called with the command:

```
psr  pdbmaker.out
```

It proceeds as follows (refer to the program listing in Appendix A):

i) A database named, "rutilities", is constructed as *thedb*. A check is made to verify that the database does not already exist. If it does exist, the user is given the option of either abandoning the process or of removing the old database.

ii) A new database is created.

iii) The procedure which enters procedures, *prcput*, is created.

iv) The *prcget* procedure is created.

v) *prcget* and *prcput* are entered in the library - *prcget* as a simple packaged procedure while *prcput* is packaged in the *intermed* structure.

c) The library lister - dblister
Again the program expects to be called with the command:

```
psr  pdblister.out
```

It proceeds as follows (refer to the program listing in Appendix B):

i) It opens the database and gets out the *prcget* procedure from "rutilities". It also gets out the procedures *fillstring, columnator, putfile* and *putline*, which are described below. It cannot run until these procedures have been inserted by modules *pdbtext* and *pdbfileio*.

ii) There follow three procedures which process the class identifier of PS-algol **pntrs**:

someof - reduces the class identifier of the packaged procedure, which is likely to be something like: `"procpak(proc( string,int ) xproc'n)"` to `"(string,int)"` by replacing the first 14 and the last 10 characters with opening and closing round brackets.

dsomeof - strips characters from the date.

gsomeof - handles a structure of global variables, a special entry keyed by `id++"globals"` for some identifier *id*, and containing the global variables of the program - if you don't use this, ignore it.

iii) Then follows, *pdbscan*, the procedure which scans the database. For most of the procedures, it employs *columnator* to provide one column for the name and class identifier, one for the date and one for the description.

iv) The main program consists of header and footer generation and a single call of the *pdbscan* procedure. It sends its results both to the screen and to a file called *dblist*.

3/ The procedures available

A full list of the procedures available is given as Appendix C. There now follows a brief description of them, grouped according to the module that they are in. Those modules that require previously installed modules have the latter listed against them. This is a new set which replaces any previously described set.

The most recent significant modification to the set was performed in Autumn 1987 and was designed to remove shared local state variables from the procedures. For instance, the *pdbfonts* module used to maintain a "current font", modifiable by some procedures and used by others. This variable lived in the utilities database, which meant that to use it, the database had to be opened in write-mode which in turn meant the database couldn't be shared. Now none of these procedures maintain shared local state.

a) pdbvecs

These procedures are left in for historical reasons. They provide functions which extend and reduce PS-algol vectors.

addivec - takes a vector of images, of size N, and a new image and returns a vector of size N+1 with the new image added.

addpvec and *addsvec* - do the same for pointers and strings.

rempvec - removes a pointer from a vector of pointers, identified either by an integer index or, if this is zero, by a pointer to the object to be removed.

link - adds a pointer into a linked list. This must have the structure:

```
structure linklist(pntr this,next)
```

unlink - removes a pointer from a list.

b) pdbtext

A set of fairly trivial text manipulation procedures.

nothing - does nothing at all - fills the gap left by the fact that **nullproc** cannot be executed.

sright - returns the rightmost part of a string, given the string and a length.

fillstring - fills out a string to a given length with spaces.

columnator - takes in a vector of strings to be displayed in columns. It also requires a vector of column widths, a vector of indentation widths for the second and subsequent lines and a vector of inter-column spaces.

c) pdbstlc

Two useful functions for providing case-insensitive operations.

stlc - takes in a string and returns the string with its letters turned to lower case.

stuc - takes a string and returns the string with its letters turned to upper case.

d) pdbfont

This is an attempt to improve on the standard function *string.to.tile* (the **print** statement now largely obviates the use of *string.to.tile*). It contains:

ssttmaker - slow string to tile maker. This procedure takes in the name of a font and returns a *string.to.tile* procedure, i.e. a procedure which takes in a string and returns an image containing that string in that font. The returned procedure is faster than the conventional *string.to.tile* as it does not need to keep looking up the database. Its internal method of working is also slightly faster.

fsttmaker - fast string to tile maker. This procedure functions in a similar manner to *ssttmaker*, but makes the additional assumption that the font is of constant width and consequently the returned *string.to.tile* function is even faster.

Note that after *pdbchooser* has been run, both of these procedures are enhanced so that if given the parameter value "?", they allow the user to dynamically choose which font to use by menu from all the fonts in the font database.

e) pdbscrio (pdbfonts)

These are a set of screen i/o procedures. The *show.box...* set of procedures give a consistent method of displaying images, strings and text in boxes.

clsall - clear all of the screen.

clsome - clear a rectangular area of the screen, given origin and dimensions.

show.box.image - display an image on the screen. A fancy procedure with the following parameters:

- the image to be displayed;
- the origin of the area to be used, if (-1,-1) is given then make the area use the centre of the screen;
- the dimensions of the area to be used, only used if the figures given are bigger than the dimensions of the image;
- a boolean which should be **true** if a box is to be drawn around the image - the box will consist of a band of 23 pixels of white surrounded by a band of two pixels of black;
- another boolean which should be **true** if the procedure is to wait for a mouse click and then restore the original screen contents.

It returns a procedure, which, if it has been requested to wait, will be a dummy, but otherwise will restore the contents of the screen after the image is no longer required.

show.box.text - takes a vector of strings and converts this to an image, at least 50 pixels wide and then sends it to *show.box.image*. It has all the same parameters as *show.box.image*, except for having a vector of strings instead of an image. It uses the procedures from the font package.

show.box.string - takes in a single string and creates a vector of strings by breaking up the string wherever it finds new line characters and sends it on to *show.box.text*. Again it has all the same parameters.

error.message - this is a version of *show.box.string* which only provides the origin and the string as parameters. It provides (0,0) for the dimensions so that the box is as small as possible and does draw a box and does wait for a mouse click. It calls *show.box.string* and then throws away the dummy procedure returned.

more - is based on the UNIX *more* command. It takes in a vector of strings and an origin, and uses mouse clicks to go forward (right button), back (left button) and quit (top button).

errcheck - is a procedure for yes/no dialogue with the user. It takes in two strings, an origin and dimensions. It displays the two strings, the first of which is a statement and the second a question (replaced by "Do you wish to proceed?" if the empty string is given) and then awaits either "y" or "n" to be pressed. If other characters are pressed, it displays "Please press y or n". It returns **true** if "y" is pressed and **false** otherwise.

f) **pdbscrio2** (pdbfonts)
A second module containing screen i/o procedures. The *mousemenu* procedure is oriented towards the 4-button Perq mouse. The procedures *imtabsetup* and *mousemenu* depend on the old version of *pdbfont* and have not been updated. They are left in for future updating, if a four button mouse becomes available again.

doblank - takes in an origin and box size, clears the box and returns what was on the screen where the box now is.

hatch - hatches in a rectangular screen area. It has the following parameters:
- the gap between lines in pixels
- the size of the image
- the origin
- the style of the hatching - 1 means / lines, 2 means \ lines and 3 means cross hatching.

tilehatch - also hatches in a screen area. It modifies the area to be hatched so that the hatching exactly fits it.

triangle - has three integer parameters and returns an image of a specified size (the first two parameters) with either the lower left triangle black and the upper right triangle white, if the third parameter is 2, or the lower right triangle black and the upper left triangle white.

diamond - returns an image of a given size with a diamond drawn as large as can be with the centre at the centre of the image.

imtabsetup - takes in a vector of strings and a pointer to a font, in the same structure as assumed by *fsttmaker* etc. and returns a table of images corresponding to and keyed by the strings.

mousemenu - produces a diamond shaped mouse menu on the screen at a given position, with a vector of four strings given as the menu entries, a vector of 4 procedures as the actions and a pointer to a table (set up by *imtabsetup*) wherein the images corresponding to the entries may be found.

g) **pdbedit** (pdbscrio2)
max - returns the larger of two integer parameters.

min - returns the smaller of two integer parameters.

seditor - is a small single line string editor. It displays the string (second parameter) in a box whose origin and dimensions form the third to sixth parameters, with a title which is the first parameter. It responds to the following:
- a printing character is inserted at the current position
- a *del* rubs out the character to the left of the cursor
- an *oops* rubs out all the text to the left of the cursor
- a *return* exits with the string to the left of the cursor
- a *noscroll* exits with the whole string
- clicking over the bar on the left of the edit box brings in the characters to the left of those displayed, if there are any
- clicking over the right hand bar brings any characters to the right.

The editor uses the fonts "fix09" for the title, "fix13" for error messages, and "cou20" for the text.

ieditor - is a similar editor for integers. Note that it is syntax directed and will only accept characters which make the string so far the beginning of a legal integer.

h) **pdbhelp (print)**
help - displays text from a help table, in a box at the centre of the screen. It takes in a string key and a pointer to a table and displays help in the entry for that key. The structure of the entry is expected to be:

```
structure help.pack(*string the.help)
```

i) **pdbvmenu**
var.menu - This procedure is an extension of the standard *menu* function, which takes the same parameters and produces a procedure with three parameters - the origin as before and a vector of booleans, one for each entry. The menu actually displayed consists of only those entries whose

boolean is true. Thus a program can dynamically change which options of a menu are available. A version could be written which greys unavailable options.

j) pdbtimer

This provides one procedure *timerADT*, which returns an ADT to aid debugging. The ADT is returned as the following structure:

```
structure timerPack(
   proc() TMclear;
   proc(string) TMstore;
   proc() TMshow)
```

The three operations control a list of message, time pairs. Respectively, they clear the list; add a message to the list at the current time; and write out the list.

k) pdbformADT (print)

This contains two procedures which provide support for a form made of light buttons. The first generates a form as an Abstract Data Type, while the second, *form.null*, provides a parallel procedure for putting passive data on the screen in the same form as the light buttons.

The main procedure is *form.generate*, which returns an ADT of the following structure:

```
structure form.package(
   proc(pntr) Form.show;
   proc() Form.all.show;
   proc(string,int,int,int,int,bool,
      proc(),pntr->pntr) Form.add;
   proc(pntr) Form.remove;
   proc(string,pntr) Form.update;
   proc() Form.clear;
   proc(->pntr) Form.mouse;
   proc() fender;
   proc() Form.monitor)
```

The light buttons are held in a vector of the following data structure:

```
structure AREA(
      #pixel under;
         ! stores what was on the screen before the light
         ! button was displayed
      string strip;
         ! contains the message displayed in the light button
      int axo,ayo,axh,ayh;
         ! the origin and size of the light button
      bool redisplay;
         ! if the light button should be redisplayed after its
         ! action has been executed - i.e. if the action
         ! affects that part of the screen
      proc() action;
         ! the procedure activated by clicking over the button
      pntr afont)
         ! the font that the message is displayed in
```


The fields of the ADT have the following functions:

Form.show - display a light button, given a pointer to it.

Form.all.show - display all the light buttons.

Form.add - add a form element. It takes as parameters, the string to be displayed in the form box, the origin and size of the box, a boolean, a procedure and a pointer to the font it is to be displayed in - i.e. values for all the fields of the *AREA* structure, except the *under* field. It automatically calls *Form.show* to display the new element.

Form.clear - clear all the form elements from the display, by displaying their *under* images. NB this only clears the display not the vector.

Form.remove - remove an element from the form, given a pointer to it. This removes the light button both from the display and from the vector.

Form.update - update the text associated with a light button, given the new string and a pointer to it.

Form.mouse - return a pointer to the element over which the mouse was clicked.

Form.monitor - wait until the mouse is clicked over a form element and then execute the procedure associated with it. Continue until a mouse button associated with *fender* is called.

fender - This procedure must be associated with one of the light buttons to terminate the call of *form.monitor*.

There is also a procedure:

form.null - which acts a bit like *form.add*, taking all the same parameters except for the procedure and the boolean. It is used for parts of a form which are not light buttons.

The most common method of working with the Form Package consists of four steps: the form is generated by a call to *form.generate*; for each light button required, a call to the *Form.add* procedure is made; a final call to *Form.add* is made, associating *fender* with a button labelled "quit", say; *Form.monitor* is called to provide the functionality of the Form.

l) pdbcomp

This provides an interface to the callable compiler with the following procedures:

compile - takes a source program as a string and a procedure holder and returns the compiled procedure or a list of error messages. Under Unix PS-algol

implementations, it creates a file in the public scratch directory "/tmp", as a temporary buffer.

report.errors - displays the error messages returned by *compile.*

class.id.to.struc - turns the class identifier into the equivalent structure definition.

class.id.to.sname - returns the structure class name from the class identifier.

class.id.to.names - returns a vector of the field names.

class.id.to.types - returns a vector of the field types. These are returned as a vector of the following list structure:

```
structure type.list(string S,R;pntr N)
```

where the type is broken up into constituent "*"s, "c"'s and base types. The *S* field contains the current part, *R* contains the rest of the type and *N* points to the next part. Thus "*cint" would look like:
("*","cint",)->("c","int",)->("int","",nil)
This is very useful when writing programs which recursively compile structures and is used, for instance, by the deep object copy, object print and deep equality test procedures.

m) **pdbchoose** (pdbscrio, pdbfonts)
This is the Chooser module, initially written by Stephen Blott, and now extensively revised. The Chooser is a procedure which creates a menu for selecting from a set of things. The module also includes the procedures:

table.to.text - which takes in a table and produces a vector of the string keys;

some.table.to.text - which takes in a table and a procedure from string to boolean, *exclude*, and produces a vector of the string keys, but restricting it to those for which *exclude* returns **false**;

sort.strings - which takes a vector of strings and returns it alphabetically ordered;

as well as new versions of *fsttmaker* and *ssttmaker*, which allow the string parameter to have the value "?", in which case the user is given a menu of all the fonts in the database to select from, before the string-to-tile is created. This enhancement belongs here, since it uses the Chooser. Conversely, the original sttmakers had to be provided first as the Chooser uses them.

However, the module mainly consists of:

set.up.choose - which takes in a vector of strings and returns an ADT of operations in the following structure:

structure *chooser.pack*(
proc(string,int,int->string)*do.choose* which displays an alphabetically ordered menu of the first n keys in the list, where n is set to 15. The user can then either

- select one of the entry tiles and return the string
- select the "forward tile" and get the next n entries (if not at end)
- select the "backward tile" and get the last n entries (if not at start)
- select the "quit with null" tile to exit with an empty string
- press a keyboard key and get the menu of keys beginning with that character - subsequent keys restrict the menu further
- press *delete* to take one character off the selection string
- press *oops* to clear the selection string

proc(string)*add.choose* which adds a new string to the menu
proc(string)*remove.choose* which deletes a string from the menu
proc(int,int)*list.choose*) which produces an alphabetically ordered list via *more* at a given position on the screen.

The method of working with the Chooser may be illustrated in a system where a set of objects in table, *T*, is manipulated by a program, which adds and deletes objects in the table. First of all, a Chooser is created by calling *set.up.choose*(*table.to.text*(*T*)). Then, whenever an object is to be selected the *do.choose* procedure is called. Whenever objects are added or deleted, the *add.choose* and *remove.choose* procedures are called - to re-call *do.choose* every time would cost too much time. If the whole set needs to be displayed for any reason, the *list.choose* procedure can be called.

n) **pdbmakelib** (pdbvecs)
This provides a facility for setting up your own procedure library structured just like this one. This is done by the procedure:

make.lib which takes in a string as an identifier, *ID* say, and returns a table which initially contains the two procedures:

- *ID.prcget* which retrieves a procedure,
- *ID.prcput* which stores a procedure.

Having called *make.lib*, the table can then be stored in your database wherever you like.

o) **pdballobj** (pdbcomp, **print**)
This is a set of three utilities for recursively manipulating complex objects. There is one procedure to print a complex object; one to test two objects for deep equality; and one to make a complete copy. The procedures handle most kinds of objects, but as with the Browser, they cannot penetrate into the closure of a procedure. The procedures maintain lists of objects they have encountered so that cyclic objects can be dealt with.

All three procedures work in a similar manner to the Browser. That is they create procedures to handle objects of different classes as they encounter them, storing them in a table. However, so that users can each have access to this procedure, the table cannot reside in the utilities database, because this would constitute a shared local state variable, which we wish to avoid as this implies only one user at a time. Therefore, the use of these procedures necessitates the

user creating and keeping a table in an owned database, opened in write mode. Note that the procedures keep this table up-to-date.

Oprint takes in two pointers, one to the object to be printed and one to the table of print procedures, and a string which contains an indentation appended to the start of every line. The objects are printed with added indentation for vectors and sub-structures. Graphical and procedural sub-objects are not printed.

Oequal takes in three pointers, two to the objects being compared and one to the table of equality procedures. It returns a boolean which is true if the objects have the same structure with the same data values.

Ocopy takes in pointers to an object and a table of copy procedures and returns an exact replica of the object. This supports an alternative to editing *in situ*.

To illustrate the method of working, we may have one procedure to print any of our own objects as follows:

```
let ourOprint=
begin
   let T=table()
   proc(pntr O) ; Oprint(O,T,"")
end
```

p) **Toolbox**

ToolBoxGen yields a toolbox of controls (light buttons, choice buttons and sliders) in their non-notifier form. It is described elsewhere in this report (Shahad Ahmed, "PS-algol Toolboxes").

q) **Editors**

EditMaker and *fetchEditorProcs* are tools for the construction of screen-based text editors. They are described elsewhere in this report (Douglas MacFarlane, "PS-algol Text Editors").

Appendix A / Procedure library creation program

```
%title "General Procedures Database Setup Program"
%lines=62

! General Procedures Database Setup Program.
! ------------------------------------------

! Mark 2.1 - 21/06/85
! -------------------

! 1/ Check the database.
!    --------------------
      ! Do it only if
      !     a) There is no database at present
      !  or b) The user ok's a removal of the current database which
      !              is then succesful.
let dbid:=""
let opts = options()
case upb(opts) - lwb(opts) + 1 of
     3  :  dbid:=opts(3)              ! All options specified
     2  : {write "Give database id: "
           dbid:=read.a.line()}       ! No options specified
default : {write "wrong number of options specified'n"; abort}
let thedb=dbid++"procs"
let dbcheck:=false
let procsdb:=open.database(thedb,"friend","write")
let quest1="Database already exists'n"
let quest2="  Do you want to clear it and start afresh? (y/n): "
if procsdb is error.record
     then dbcheck:=true
     else
       begin
        write quest1,quest2
        let ans:=read()
        while ans~="y" and ans~="n" do
         begin
          write "Please answer y or n: "
          ans:=read()
         end
        if ans="y" do
          begin
           dbcheck:=true
           if remove.database(thedb,"friend")=nil
                  then write "removed o.k.'n"
                  else {write "removal error'n"; abort}
          end
       end


! 2/ Create the database.
!    --------------------
if dbcheck do
 begin
  procsdb:=create.database(thedb,"friend")
  structure intermed(pntr procpaki; *string depended;
                     string datestamp, descriptor)
  let addsvec=proc(*string oldvec;string newelt -> *string)
   begin
```

```
    let ne=upb(oldvec)
    let newvec:=vector 1::ne+1 of ""
    if ne>0 do for i=1 to ne do newvec(i+1):=oldvec(i)
    newvec(1):=newelt
    newvec
   end


! 3/ The put procedure.
!    ------------------
!     The put procedure is now (v 1.3) much more sophisticated
!and has the following parameters:
!     procname  - the name of the proc, which will be its key
!     procpntr  - the procedure packaged in a structure whose one
!        field is of type proc(appropriate typelist) named xproc
!     dependson  - a list of the names of the procedures it calls
!     desc       - a description of the procedure's function
!
!     All procedures (except get which must be accessed directly and
!therefore as simply as possible from the user programs) are stored
!as intermed structures.  These have the following four fields:
!     procpaki    - a structure of type 'procpak', which is a pntr to
!         a structure containing just one field of type proc(typelist)
!         and name xproc, where typelist is the typelist appropriate
!         to the current procedure
!     depended    - this contains a list of the names of procedures
!         which call this one (initially set to empty apart from
!         prcput which is set to "everything" as all things need this)
!     datestamp   - this contains the system date at insertion time
!     descriptor - this contains the desc input parameter
!
!     Put checks to see if the procedure is already in the database.
!If not it just enters the procedures and reports procedure created.
!If it is there, there is first a check to see if the revised version
!has parameters of the same type as the old version.  If not, the user
!is interrogated to see if he wishes the amendment to go ahead.
!When a procedure is updated in this way, a list of all the procedures
!in the database which call it is displayed with the advice to re-enter
!these in the database.  Finally put looks up all the procedures in the
!database which it calls (i.e. are in dependson) and adds procname to
!their depended list.

  let prcput=proc(string procname; pntr procpntr;
                  *string dependson; string desc)
   begin
    let procsdb:=open.database(thedb,"friend","write")
    if procsdb is error.record
      do {write "No utilities database - do prcdbmaker first'n"; abort}
    let initdeps:=vector 0::0 of ""
    if procname="prcput" do initdeps:=vector 1::1 of "everything"
    let okmess:=""
    let isithere=s.lookup(procname,procsdb)
    if isithere=nil then ! new procedure - just enter it
     begin
      s.enter(procname,procsdb,intermed(procpntr,initdeps,date(),desc))
      okmess:=" created"
     end
                    else ! old proc - check type and dependents
     begin
      if class.identifier(isithere(procpaki))~=
         class.identifier(procpntr) do
```



```
       begin                ! new proc is of changed type
        write "warning - type mismatch on changing ",procname,"'n"
        write "do you want to do it? (y/n): "
        if read.a.line()(1|1)~="y" do {write "ok - finishing'n"; abort}
       end
      let thedeps=isithere(depended)
      s.enter(procname,procsdb,intermed(procpntr,thedeps,date(),desc))
      okmess:=" updated"
      if upb(thedeps)>0 do for i=1 to upb(thedeps) do
          write "now update ",thedeps(i),
                " as well so that it uses this version'n"
     end
    if upb(dependson)>0 do for i=1 to upb(dependson) do
     begin                  ! mark procs this one depends on
      let depper=s.lookup(dependson(i),procsdb)
      if depper~=nil do
       begin
        let already:=false
        let theothers=depper(depended)
        if upb(theothers)>0 do for j=1 to upb(theothers) do
            if procname=theothers(j) do already:=true
        if ~already do depper(depended):=addsvec(theothers,procname)
       end
     end
    if commit()=nil then write procname,okmess," o.k.'n"
                    else {write "commit failed'n"; abort}
   end


! 4/ The get procedure.
!    ------------------
!     Get is much simpler - it merely looks up the intermed structure
!named by parameter, procname, and unpacks procpaki from it.
  let prcget=proc(string procname -> pntr)
   begin
    let procsdb:=open.database(thedb,"friend","write")
    if procsdb is error.record
      do {write "No utilities database - do prcdbmaker first'n"; abort}
    let got=s.lookup(procname,procsdb)
    if got=nil do {write "procedure ",procname," not found in the database'n"; abort}
    got(procpaki)
   end



! 5/ Store put and get and commit.
!    -----------------------------
! Nb get is stored as a simple procpak, while put is stored
!as an intermed structure like all the other procedures.  The
!different storage of get permits easier retrieval in user
!programs.
  {structure procpak(proc(string -> pntr) xproc)
   s.enter(dbid++"get",procsdb,procpak(prcget))}
  {structure procpak(proc(string,pntr,*string,string) xproc)
   prcput(dbid++"put",procpak(prcput), vector 0::0 of "",
          "Put procedures into the database")}

  if commit()=nil then write "new database created'n"
                  else {write "creation fails - aborted'n"; abort}
 end
```

Appendix B/ Procedure library lister

```
! Procedure Database Lister.
!---------------------------

! Mark 2.1 - 21/06/85
! -------------------

! 1/ Get the required utilities.
!    ---------------------------
let dbid:=""
let opts = options()
case upb(opts) - lwb(opts) + 1 of
     3  :  dbid:=opts(3)              ! All options specified
     2  : {write "Give database id: "
           dbid:=read.a.line()}       ! No options specified
default : {write "wrong number of options specified'n"; abort}
let thedb=dbid++"procs"
let thisdb:=open.database(thedb,"friend","read")
if thisdb is error.record
  do {write "No database - do dbmaker first'n"; abort}
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do pdbmaker first'n"; abort}
let prcget=
 begin
  structure procpak(proc(string -> pntr) xproc)
  s.lookup("prcget",procsdb)(xproc)
 end
  structure intermed(pntr procpaki; *string depended;
   string datestamp, descriptor)

let fillstring={structure procpak(proc(string,int -> string) xproc)
                prcget("fillstring")(xproc)}
let columnator={
     structure procpak(proc(*string,*int,*int,*int -> string) xproc)
     prcget("columnator")(xproc)}
let putfile={structure procpak(proc(string) xproc)
             prcget("putfile")(xproc)}
let putline={structure procpak(proc(string) xproc)
             prcget("putline")(xproc)}


! 2/ Some more utilities.
!    --------------------

! 2a/ Process the class structure.
!     ----------------------------
let someof=proc(string inp -> string)
 begin
  let out:="()"
  if length(inp)>23 do out:="("++inp(15 | length(inp)-24)++")"
  out
 end

! 2b/ Process the class structure.
!     ----------------------------
let dsomeof=proc(string inp -> string)
  inp(5|3)++inp(9|2)++"/"++inp(23|2)++inp(11|6)
```



```
! 2c/ Process global variables.
!     -------------------------
let gsomeof=proc(pntr inp -> string)
 begin
  let out:="()"
  let all=class.identifier(inp); let n=length(all)
  let f:=false; let j:=0
  for i= 1  to n do if all(i|1)="("  and ~f do {f:=true; j:=i}
  out:=all(1 | j)
  f:=false; let k:=0
  for i=j+1 to n do if all(i|1)=" "  and ~f do {f:=true; k:=i}
  let curtype:=all(j+1 | k-j-1)
  out:=out++curtype
  let ended:=false
  while ~ended do
   begin
    f:=false
    for i=k+1 to n do if all(i|1)="'n" and ~f do {f:=true; j:=i}
    out:=out++all(k | j-k)
    if all(j+1 | 1)=")"
      then
        ended:=true
      else
       begin
        f:=false
        for i=j+1 to n do if all(i|1)=" "  and ~f do {f:=true; k:=i}
        if curtype~=all(j+1 | k-j-1)
         then {curtype:=all(j+1 | k-j-1); out:=out++"; "++curtype}
         else out:=out++","
       end
   end
  out++")"
 end


! 2d/ Process one entry.  (Scan function).
!     -------------------
let globline:=""
let pdbscan=proc(string procname; pntr interproc -> bool)
 begin
  if procname=dbid++"get" then
    putline(fillstring(procname++someof(class.identifier(interproc)),99)++"|'n")
                              else
  if procname=dbid++"globals" then
    globline:=columnator(
     @1 of string ["global variables",
                   gsomeof(interproc)],
     @1 of int [35,59],
     @1 of int [12,10],
     @1 of int [3,2])
                                  else
    putline(columnator(
     @1 of string [procname++someof(class.identifier(interproc(procpaki))),
           dsomeof(interproc(datestamp)),
           interproc(descriptor)],
     @1 of int [35,14,42],
     @1 of int [12,5,6],
     @1 of int [3,3,2]))
  true
 end
```

```
! 3/ The main program.
!    ------------------
putline(fillstring("Procedure",40)++fillstring("Date",20)++"Description'n")
putline("-------------------------------------------------------------'n")
let n:=s.scan(thisdb,pdbscan)
if globline~="" do n:=n-1

putline(fillstring("------------------------------",99)++"|'n")
putline(fillstring(iformat(n)++" procedures in the database",99)++"|'n")
if globline~="" do {putline(fillstring("------------------------------",99)++"|'n")
                    putline(globline)}
putline("-------------------------------------------------------------'n")

putfile("dblists/"++dbid++"dblist")
putfile("screen")
?
```


Appendix C/ The current list of procedures

Procedure	Date	Description	dbmaker
proget(string -> pntr)			
prcput(string,pntr,*string,string)	Nov19/87 17:25	Put procedures into the database	
addivec(*#pixel,#pixel -> *#pixel)	Nov19/87 17:25	Add image to vector	(a) pdbvecs
addpvec(*pntr,pntr -> *pntr)	Nov19/87 17:25	Add pointer to vector	
addsvec(*string,string -> *string)	Nov19/87 17:25	Add string to vector	
link(pntr,pntr -> pntr)	Nov19/87 17:25	Add pointer into linked list	
rempvec(*pntr,pntr,int -> *pntr)	Nov19/87 17:25	Remove pointer from vector	
unlink(pntr,pntr -> pntr)	Nov19/87 17:25	Remove pointer from linked list	
columnator(*string,*int,*int,*int -> string)	Nov19/87 17:25	Prepare columns of text for display	(b) pdbtext
fillstring(string,int -> string)	Nov19/87 17:25	Fill out a string with spaces	
nothing()	Nov19/87 17:25	Do nothing at all	
sright(string,int -> string)	Nov19/87 17:25	Return the right hand end of a string	
stlc(string -> string)	Nov20/87 17:37	Make string lower case	(c) pdbstlc
stuc(string -> string)	Nov20/87 17:37	Make string upper case	
fsttmaker(string -> proc(cstring -> #pixel))	Apr 1/88 14:40	Constant width string.to.tile maker	(d) pdbfont
sttmaker(string -> proc(cstring -> #pixel))	Apr 1/88 14:40	Variable width string.to.tile maker	
clsall()	Nov19/87 17:25	Clear the whole screen	(e) pdbscrio
clscme(int,int,int,int)	Nov19/87 17:25	Clear a window of the screen	
errcheck(string,string,int,int,int,int -> bool)	Nov19/87 17:25	Error message with yes/no interrogation	
error.message(string,int,int)	Nov19/87 17:25	Display a message in a screen window	
more(*string,int,int)	Nov19/87 17:25	More text display	
show.box.image(#pixel,int,int,int,int, bool,bool -> proc())	Nov19/87 17:25	Display an image with optional box and wait	
show.box.string(string,int,int,int,int, bool,bool,proc(string -> #pixel) -> proc())	Nov19/87 17:25	Display an image with optional box and wait	
show.box.text(*string,int,int,int,int, bool,bool,proc(string -> #pixel) -> proc())	Nov19/87 17:25	Display an image with optional box and wait	

diamond(int,int -> #pixel)	Nov19/87 17:26	Returns a diamond image	|f) **pdbscrio2**
doblank(int,int,int,int -> #pixel)	Nov19/87 17:26	Makes a screen box and returns what is beneath	|
hatch(int,int,int,int,int,int)	Nov19/87 17:26	Cross hatches an image	|
imtabsetup(*string,string -> pntr)	Nov19/87 17:26	Sets up a dictionary for the mouse menu	|
mousemenu(int,int,*string,*proc(),pntr)	Nov19/87 17:26	Gives a mouse menu	|
tilehatch(int,int,int,int,int,int,int, int)	Nov19/87 17:26	Cross hatches part of an image	|
triangle(int,int,int -> #pixel)	Nov19/87 17:26	Returns a triangular image	|
ieditor(string,int,int,int,int,int -> int)	Nov19/87 17:26	Integer editing in a screen window	|g) **pdbedit**
max(int,int -> int)	Nov19/87 17:26	Returns larger of two integers	|
min(int,int -> int)	Nov19/87 17:26	Returns smaller of two integers	|
seditor(string,string,int,int,int,int -> string)	Nov19/87 17:26	Text editing in a screen window	|
help(cstring,pntr)	Nov19/87 17:26	Helper	|h) **pdbhelp**
var.menu(c#pixel,c*#pixel,cbool,c*proc(c#pixel,cint) -> proc(cint,cint, c*bool -> bool))	Nov19/87 17:25	Variable menu procedure	|i) **pdbvmenu** |
timerADT(-> pntr)	Nov19/87 17:25	An ADT for timing	|j) **pdbtimer**
form.generate(-> pntr)	Feb 7/88 04:20	Generate a form ADT	|k) **pdbformADT**
form.null(string,int,int,int,int,pntr)	Feb 7/88 04:20	Consistent functionless form element	|
class.id.to.names(string -> *string)	Nov19/87 17:25	Returns Field Names	|l) **pdbcomp**
class.id.to.sname(string -> string)	Nov19/87 17:25	Returns The Structure Name	|
class.id.to.struc(string -> string)	Nov19/87 17:25	Make A Structure Definition	|
class.id.to.types(string -> *pntr)	Nov19/87 17:25	Returns Field Types	|
compile(cstring,cpntr -> pntr)	Nov19/87 17:25	Compile a string	|
report.errors(pntr)	Nov19/87 17:25	Report Compiler Errors	|
set.up.choose(*string -> pntr)	Apr 1/88 14:40	Create a chooser ADT	|m) **pdbchoose**
some.table.to.text(pntr,proc(string -> bool) -> *string)	Apr 1/88 14:40	Make a string vector of the keys of a table with exclusions	|
sort.strings(*string -> *string)	Apr 1/88 14:40	Sort a string vector alphabetically	|
table.to.text(pntr -> *string)	Apr 1/88 14:40	Make a string vector of the keys of a table	|
make.lib(string -> pntr)	Nov19/87 17:26	Make your own library	|n) **pdbmakelib**

Ocopy(pntr,pntr,pntr -> pntr)	May 1/88 14:56	Copy any object	|o) **pdballobj**
Oequal(pntr,pntr,pntr -> bool)	May 1/88 14:56	Deep equality between two objects	|
Oprint(pntr,pntr,string)	May 1/88 14:56	Print any object	|
ToolBoxGen(-> pntr)	Nov19/87 17:26	Buttons, radio buttons, scrollbars See PPRR56	|p) **Toolbox**
EditMaker(#pixel,pntr -> pntr)	May23/88 15:30	Generate simple visual editors See PPRR56	|q) **Editors**
fetchEditorProcs(-> pntr)	May23/88 15:35	Screen editor package See PPRR56	|
57 procedures in the database			|

PS-algol Text Editors

Douglas MacFarlane

Introduction

For a long time the need has been felt for a callable screen editor for use in PS-algol programs. One was developed under the ICL Perq/PNX implementation of the language, and this eventually produced a reasonable editor. Rather than simply port this program to the Sun/UNIX version the opportunity was taken to improve on some design decisions.

The original program had been a single procedure which took a string (along with some other parameters) and returned a string as its result. During its operation it created an image on the screen similar in appearance to the Rutherford Appleton Laboratory's *spy* editor [2]. The program interacted with the keyboard and the mouse to perform editing functions on the text, and returned the result of these operations. The program was quite large which made development of the program quite time consuming on the Perq. An attempt was made to break the program down into pieces of a more manageable size but this slowed the resulting editor down so much that it was unuseable.

As a result of all these factors a specification for an editor was produced which would separate out the interaction with the user from the manipulation and display of the text. The result of this was a specification for an editor-generating program (EditMaker) which would generate a package of procedures which could be called to perform editing functions on a piece of text.

A program which meets the original specification for EditMaker was written and exercised. An example multi-window demonstration editor was produced.

More recently the editing functions developed have been repackaged to produce a slight variation in usage which may be beneficial in some circumstances. Instead of producing a procedure generator which returns a unique set of procedures for each editor, a single set of procedures is supplied instead, one of which will generate a structure for each editor required. This structure is then passed to the appropriate procedure along with any other parameters and the operation is carried out. This method of operation should make generation of multiple editors cheaper although they may operate more slowly. The original multi-window editor demonstration program has been reworked to make use of the alternative arrangement.

What follows is firstly a description of EditMaker followed by a description of the new package.

There has been some work by other people on PS-algol text editors. Richard Cooper has produced a simple string editor as part of his procedure library which looks very useful for many applications. A text editor also formed

part of the PS-algol toolbox work reported in [1]. This offered a more sophisticated editor supporting, for example, mixed fonts.

1. EditMaker

EditMaker is a procedure which generates simple screen editors. The idea is to produce a general purpose editor that can be used on its own for simple jobs, but can also be built up to produce quite sophisticated screen editors. For example a multi-window editor could be built using an EditMaker editor for each window on the screen.

1.1 Programmer's interface

An editor is generated by the application of a procedure, whose skeleton is:

```
let EditMaker:=
   proc(#pixel TheWindow ; pntr TheFont->pntr) ; nullproc
```

The generator takes an image to be used by the editor. Text will appear on the image in the font passed at generation time. A font is a PS-algol structure containing a vector of images [3]. The procedure returns a handle to the new editor, this is the procedural interface to the editor operations. It is a structure of class *EditorPac*:

```
structure EditorPac(
   proc()                    EditorRefresh ;
   proc(#pixel)              EditorNewWindow ;
   proc(->bool)              EditorScrollUp,
                             EditorScrollDown,
                             EditorScrollLeft,
                             EditorScrollRight ;
   proc(pntr)                EditorInsert,
                             EditorSetFont ;
   proc(->pntr)              EditorDeleteAll,
                             EditorDeleteSelected,
                             EditorReturnAll,
                             EditorReturnSelected ;
   proc(string)              EditorProcessChar ;
   proc(int,int)             EditorPutTextMarker,
                             EditorSelectText ;
   proc(pntr,bool->bool)     EditorFindText ;
   proc(->pntr)              EditorGetStatus)
```

Two structures pass through this procedural interface. Text (a string) is passed to and from the editor in text structures defined as

```
structure EditorText(string TheText ; pntr Prev,Next)
```

Status is returned by the editor in a status structure defined as

```
structure EditorStatus(
   int LineCount,LinesVisible,FirstVisibleLine)
```

There is the notion of a "text marker". This is where character operations will take place on the text. There may also be a "selected" section of text that is used by some operations.

1.2 **Operations**
1.2.1 **Screen control**
EditorRefresh Refresh the editor's window.

EditorNewWindow Replace the image used by the editor as its window. The old image will not be altered by the editor, and the new one will only be written to as changes are made. A call to the refresh procedure EditorRefresh is needed to put an initial picture up.

1.2.2 **Scroll operations**
Each of these operations yields a **bool** to indicate success (**true**) or failure. A font will have constant height, but characters within a font can differ in their width. Vertical scrolling is always related to the height of the font. Horizontal scrolling for variable-width fonts will be by one pixel, otherwise by one character width.

EditorScrollUp Scroll the display upwards by one line of text.

EditorScrollDown Scroll the display downwards by one line of text.

EditorScrollLeft Scroll the text left by one position.

EditorScrollRight Scroll the text to the right by one position.

1.2.3 **Text operations**
EditorInsert This procedure takes a text structure and inserts it into the edited text at the current text marker position.

EditorSetFont Change the font used by the editor to display the text. The change will take place immediately, there will be a refresh of the editor's window to reflect the change.

EditorDeleteAll Delete all the text in the editor's text buffer and return it in a text structure.

EditorDeleteSelected Delete the current selection and return it in a text structure.

EditorReturnAll Return all the text in the text buffer in a text structure.

EditorReturnSelected Return the current selection in a text structure.

EditorProcessChar This procedure passes a character to the editor for processing. The editor itself will deal with DEL, TAB, "oops" etc.

1.2.4 **Cursor control**
EditorPutTextMarker Place the editor's text marker at some position in the text. The procedure is passed x and y coordinates in its window space which it may use in some way to decide where to put the marker. It will cancel any existing text selection.

EditorSelectText This procedure is used to indicate to the editor a section of text to be "selected". The procedure is passed x and y coordinates in its window space which it may use in some way to decide on the new selection. The selection will probably be from the current position of the text marker to the new position specified in the procedure call. The text marker will be at the start of the selected text, if a selection was made. The selected text will probably be highlighted in some way e.g. underlined.

1.2.5 **Searching**
EditorFindText Search the text buffer from the text marker, in the direction indicated by the **bool**, for a match of the contents of the text structure. If the text is found it becomes the current selection and the operation yields **true**, otherwise the result is **false**.

1.2.6 **Status**
EditorGetStatus This procedure returns a structure of class *EditorStatus*. This gives the number of lines in the editor's text buffer, the number of lines currently visible, and the number of the first visible line. Its purpose is to support the host program in supplying information to the user.

2. **EditMaker2**
EditMaker2 is a revision of EditMaker. The most obvious difference is in the use of a single common set of (persistent) editing procedures, instead of a set of procedures for each editor. One of these procedures is used to create a data structure that simply holds the state of a new "editor". This structure is then passed to the appropriate editing procedure along with any required parameters. The editing procedures are basically the same as those generated by EditMaker, although each procedure takes an extra parameter (the editor state the procedure is to operate on). There is also the aforementioned extra procedure which generates new editor structures.

2.1 **Programmer's interface**
The package of editing procedures is stored in the database in a structure of the following class:

```
structure EditorProcPac(
   proc(pntr)                          EditorRefresh ;
   proc(pntr,#pixel)                   EditorNewWindow ;
   proc(pntr->bool)                    EditorScrollUp,
                                       EditorScrollDown,
                                       EditorScrollLeft,
                                       EditorScrollRight ;
   proc(pntr,pntr)                     EditorInsert,
                                       EditorSetFont ;
   proc(pntr->pntr)                    EditorDeleteAll,
                                       EditorDeleteSelected,
                                       EditorReturnAll,
                                       EditorReturnSelected ;
   proc(pntr,string)                   EditorProcessChar ;
   proc(pntr,int,int)                  EditorPutTextMarker,
                                       EditorSelectText ;
   proc(pntr,pntr,bool->bool)          EditorFindText ;
   proc(pntr->pntr)                    EditorGetStatus ;
   proc(#pixel,pntr->pntr)             EditorCreateNew)
```

Text is passed to and from the editor, and status returned, in the same structures as used by the EditMaker package, namely

```
structure EditorText(string TheText ; pntr Prev,Next)

structure EditorStatus(
   int LineCount,LinesVisible,FirstVisibleLine)
```

The state of a particular "editor" is accessible through a **pntr**. The host program passes this value around but doesn't need to see inside. For the curious, it is defined as follows:

```
structure EditorPac(
   #pixel     EditWindow,EdRealWindow,EdCursor ;
   pntr       EdCurrentFont ;
   *#pixel    EdFontChars ;
   int        EdFontHeight,EdFontWidth,EdFontDescender,EdGap ;
   *#pixel    EdRowImages ;
   *pntr      EdEnhancedImages ;
   int        EdHeight,EdWidth ;
   pntr       EdText,EdTopLine ;
   int        EdLines,EdUnseenLines,EdUnseenChars,
              EdUnseenBits,EdCursLine,EdCursChar,
              EdSelLine,EdSelChar ;
   bool       EdSelected,EdFound ;
   string     EdDelKey)
```

2.2 Operations

The editor operations are identical to their counterparts produced by EditMaker (except for the extra parameter). There is one addition:

2.2.1 Creation

EditorCreateNew This procedure takes two parameters. The first is the image the editor is to use, and the second is the font. The procedure returns a data structure representing an editor state. (This is the *EditorPac* structure described above.)

3. Future developments

Both these packages offer a basic set of editing operations. There are a number of possible extensions to these basic operations, some of which are given below. Future extensions will be made on the basis of feedback from users of the existing packages.

3.1 Style

EditorSetStyle A procedure taking a "style structure" representing the text style to be used from the current position.

EditorGetStyle A procedure returning a structure representing the style at the text marker's current position.

EditorChangeStyle A procedure taking a "style structure" representing the text style to be used for the current selection.

3.2 Selection

EditorSelectWord The "word" containing the text marker will become the selected text. Any current selection will be cancelled.

EditorSelectLine The "line" containing the text marker will become the selected text. Any current selection will be cancelled.

3.3 Cursor control

EditorCursorUp,
EditorCursorDown,
EditorCursorLeft,
EditorCursorRight These four procedures will move the text cursor in the appropriate direction one character or line. Any current selection will be cancelled. If the cursor is moved to a place where there is no text, for example to the right when it is at the end of a line, then it will be moved to a "sensible" place in the text, e.g. the start of the next line. This would allow a program to make use of the "arrow" keys available on some terminals.

EditorCursorPut This procedure takes a line number and a character number, and tries to place the text cursor at that position in the text. If there was a selection it will be cancelled. A boolean result indicates success or failure.

4. Where to find them

Both editors currently form part of the procedure library described elsewhere in this report (Richard Cooper, 'A Utility Library for PS-algol'). In the following segments of PS-algol code we use the library utility *prcget* to obtain EditMaker and EditMaker2 respectively:

```
let editmaker=
begin
   structure procpak(proc(#pixel,pntr->pntr) xproc)
   prcget("EditMaker")(xproc) !Screen editor generator
end

let editorprocs=
begin
   structure procpak(proc(->pntr) xproc)
   prcget("fetchEditorProcs")(xproc)() !->EditorProcPac
end
```

The multi-window demonstration editor is available as the PS-code file 'editor.out'.

References

[1] Cutts, Q. and G. Kirby. An event-driven software architecture. Department of Computing Science, Glasgow University and Department of Computational Science, St Andrews University, Persistent Programming Research Report 48, 1987

[2] *ICL PERQ Guide to PNX Version 5*, Chap. 4, Technical Publication R10251/01, ICL 1985

[3] Persistent Programming Research Group. PS-algol Reference Manual, Fourth Edition. Department of Computing Science, Glasgow University and Department of Computational Science, St Andrews University, Persistent Programming Research Report 12, 1987

PS-algol Toolboxes

Shahad Ahmed

Introduction

This paper describes two toolboxes which PS-algol programmers can draw on to build user interfaces. The first (section one) is suited to a conventional sequential program style. The second uses the event-driven notifier architecture described by Quintin Cutts and Graham Kirby [2]. In the second section, the notifier system is reviewed in the light of experience, as a preamble to an exposition of *panels* from the notifier-based toolbox (section three). The paper concludes with mention of an application (an icon editor) that was built using this toolbox, and with details of where the toolboxes might be found.

1. The non-notifier toolbox

This section describes the interface to a set of light buttons, radio buttons and scrollbars that are available as library procedures to PS-algol application programmers.

The implementation of these objects, known here as *controls*, is comparable to that found in the Macintosh toolbox control manager [1]. Readers familiar with this will find the use of the controls described here broadly similar. It is noted that the Macintosh toolbox is not the only or best method of implementing such a system. Section 3 describes a substantially different implementation based on a notifier system [2]. Implementation details are given where they give a better insight into the programming interface.

The discussion is subdivided as follows:

- Toolbox definitions
- The control manager
- Defining controls (light buttons, radio buttons, scrollbars)
- Generic functions operating on controls

1.1 Toolbox definitions

The non-notifier toolbox is a bundle of procedures. For convenient reference we set down here the PS-algol structure definitions for the toolbox itself, and for its ancillary structures. In subsequent subsections the toolbox procedures and structures are discussed in detail.

```
!A box for the tools
structure toolbox(
   proc(pntr,pntr->int) trackControl ;
   proc(pntr,pntr->pntr) newControl ;
   proc(pntr,pntr->pntr) disposeControl ;
   proc(pntr,pntr->pntr) findControl ;
   proc(pntr) showControl ;
   proc(pntr) drawControls ;
   proc(pntr) draw.all.Controls ;
   proc(pntr,int) hiliteControl ;
   proc(pntr,#pixel) setCTitle ;
   proc(pntr->#pixel) getCTitle ;
```



```
   proc(pntr,pntr->int) testControl ;
   proc(pntr,pntr) moveControl ;
   proc(pntr,int,int) sizeControl ;
   proc(pntr) hideControl ;
   proc(pntr,int) setCtlValue ;
   proc(pntr->int) getCtlValue ;
   proc(pntr,int) setCtlMin ;
   proc(pntr->int) getCtlMin ;
   proc(pntr,int) setCtlMax ;
   proc(pntr->int) getCtlMax ;
   proc(pntr->pntr) selectedRadio ;
   proc(int,int,int,int,#pixel->pntr) create.Button ;
   proc(->pntr) create.Radio.List ;
   proc(pntr,int,int,int,int,#pixel->pntr) create.Radio ;
   proc(int,int,int,int->pntr) create.VScrollbar ;
   proc(int,int,int,int->pntr) create.HScrollbar)

!Representation of a control
structure control(pntr next,in.rect ; bool visible ; int hilite,
   value,max,min ; proc(pntr,int,pntr->int) control.def ;
   #pixel title ; pntr user.misc)

!Partcode packaged with the control it is part of
structure control.code(pntr the.control ; int partcode)

!Represents a cartesian point
structure point(int x,y)

!Lower left of rectangle is (x1,y1) upper right is (x2,y2)
structure rect(int x1,y1,x2,y2)

!Header of a radio list
structure head(pntr radio.head)

!Radio list
structure radio(pntr radio.name,next.radio ; #pixel rad.off,
   rad.hi,rad.on,rad.hion,rad.inactive)

!Image container for a button
structure button.pic(#pixel but.on,but.off,but.inactive)

!Scrollbar information
structure sbar(int orientation ; pntr slop ; #pixel up.arrow,
   down.arrow,box,out.box,bar.inactive ; int pos,but.len)
```

1.2 Control manager

1.2.1 Controls

A control is an object that the user may alter causing some side-effect. Usually such an object is visible on the screen and is manipulated via a locator device such as a mouse or graphics tablet. Down at the program level a control is represented by a PS-algol object. The toolbox procedures typically perform operations involving control objects.

The toolbox can be used without knowing the definition of a control. But for the purpose of exposition, and for the curious, here follows a partial description. A control object is defined as:

```
structure control(pntr next,in.rect ;
   bool visible ;
   int hilite,value,max,min ;
   proc(pntr,int,pntr->int) control.def ;
   #pixel title ;
```

```
        pntr user.misc)
```

WHERE *visible* is set **true** if the control is drawn, else **false**
AND *hilite* is the partcode of the currently highlighted region of the control
AND *value* is the current value of the control e.g. 1 or 0 for a radio button (not meaningful for a simple button)
AND *max* is the maximum possible setting of *value*
AND *min* is the minimum possible setting of *value*
AND *title* is used to label the control.

The significance of some of these terms will become clear.

1.2.2 **Control size**
The size of each control is defined by the size of the rectangle in which the control is defined. The toolbox represents rectangles internally as:

```
structure rect(int x1,y1,x2,y2)
```

WHERE *x1,y1* are the coordinates of the bottom left corner of the rectangle with respect to the screen origin
AND *x2,y2* define the top right of the rectangle.

1.2.3 **Partcodes**
A control may be simple such as a button or compound such as a scrollbar. It is therefore useful to split a control into parts (regions). Each part is assigned an integer called its *partcode*. A button has only one region inside its rectangle and this has the value 1, whereas a scrollbar has different regions (scroll buttons, thumb etc.) each of which is identified by a different integer. This convention enables the control manager to tell the programmer what part of a control was clicked in by simply returning the partcode.

1.2.4 **Control list**
All controls such as buttons and radio buttons are connected in a *control list*. The control list is a pointer to the head of a linked list of all the controls that are currently in use. A new control list is set to **nil** to signify that it is empty. New controls are added to the head of the control list using the function:

newControl **proc(pntr** *control.list,cntl*->**pntr)**
WHERE *control.list* is a list of *controls*
AND *cntl* is a *control*.
PURPOSE is to augment *control.list* with *cntl*.
RETURNS the augmented control list.

To remove an item from the control list use the function:

disposeControl **proc(pntr** *control.list,cntl*->**pntr)**
WHERE *control.list* is a list of *controls*
AND *cntl* is a *control*.
PURPOSE is to remove *cntl* from *control.list*.
RETURNS the amended control list.

The control manager routines are used by the programmer in response to mouse actions. When such an event occurs i.e. a mouse button-down is detected *findControl* should be called:

findControl **proc(pntr** *control.list,the.point*->**pntr)**
WHERE *control.list* is a list of *controls*
AND *the.point* is a *point* containing a pair of *screen* coordinates.
PURPOSE is to match a screen position with a control. If *the.point* lies within a member of *control.list* then a *control.code* structure is passed back with the *the.control* field set to that control and the *partcode* field set appropriately. If *the.point* does not lie in any of those controls then *the.control* is **nil** and the partcode returned is zero.
RETURNS a *control.code*.

The *point* and *control.code* structures are defined in section 1.1. If an active control was hit then the function *trackControl* is provided:

trackControl **proc(pntr** *cntl,the.point*->**int)**
WHERE *cntl* is a *control*
AND *the.point* is a *point* containing a pair of *screen* coordinates (the position at which the control was originally hit).
PURPOSE is to handle dragging of the mouse while the button is down. It takes care of actions such as highlighting buttons, moving scrollbar thumbs etc. When the mouse is released the partcode of the region at that position is returned. If the position was outside the control this is zero. This function also performs any alterations to the value of a control e.g. the scrollbar value, or highlighting a new radio button etc.
RETURNS the partcode of the region indicated at mouse-up.

1.3 **Defining controls**
1.3.1 **Buttons**
Graphically a light button consists of a rectangle that has a border and which has a program-supplied image placed within it. When a mouse-down is detected within a button it is highlighted. If the mouse is depressed inside a button, a call of *findControl* will return the button's *control* structure, together with the partcode for the inside of a button, i.e. 1. At this stage *trackControl* should be called. If the mouse is now released inside the button *trackControl* will return the partcode 1, indicating that the button has been selected. However if the user moves the cursor outside the light button with the mouse button still down, the button will un-highlight. A released button at this stage will cause *trackControl* to return with a partcode of zero; but if the cursor is returned without releasing the button, the light button will highlight again. Fig. 1 shows a typical button.

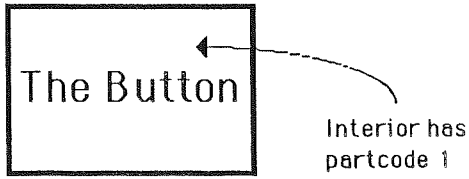


Fig. 1

The function *create.Button* is used to set up a single light button:

create.Button **proc(int** *left,bottom,right,top* **; #pixel** *button.name*->**pntr)**
WHERE *left,bottom,right,top* are the edges of the enclosing rectangle
AND *button.name* is an image to be placed within the button.
PURPOSE is to set up the fields of the button *control* structure and return it ready to be placed on a control list. The difference between *top* and *bottom*, and between *left* and *right* must be at least 20. Thus the smallest button obtainable is a 20 by 20 square.
RETURNS a new *control* representing a button.

1.3.2 Radio buttons

Radio buttons are a particular form of choice buttons. They consist of a set of buttons only one of which can be on (selected) at any given time. Fig. 2 shows a set of radio buttons; the currently selected button has its centre highlighted.

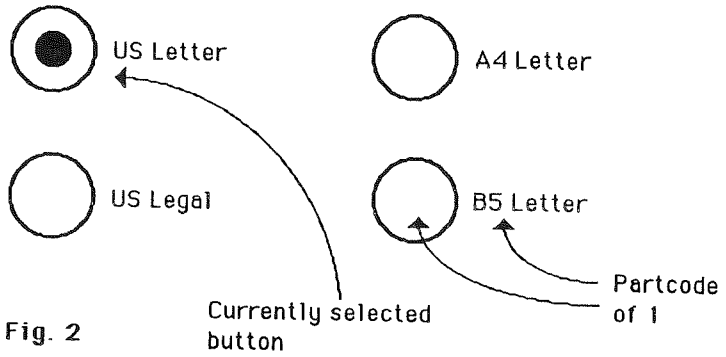


Fig. 2

Each individual radio button works in exactly the same way as a simple button and it should be activated in the same way. As with the button the interior has a partcode 1. However there are two specific regions: one is inside the circle, the second is inside the associated label. The reason for not having different partcodes for these separate regions is that the same action is performed regardless of which region is pressed.

A set of radio buttons is defined as a linked list of radio buttons, so a radio list is simply a pointer to the head of this list. Such a radio list is created using the following function:

create.Radio.List **proc(->pntr)**
PURPOSE is to create a new radio list. The list has a header node, defined in section 1.1. The header contains a pointer to the first element of the radio list; it is initially **nil**.
RETURNS a *head* structure.

Each radio button within a set must be placed on the control list after it has been created. We populate a radio list with new buttons:

create.Radio **proc(pntr** *radio.list* **; int** *left,bottom,right,top* **; #pixel** *the.name* **->pntr)**
WHERE *radio.list* is the *head* of a list of radio buttons

AND *left,bottom,right,top* are the edges of the enclosing rectangle
AND *the.name* is the label to be placed beside the button.
PURPOSE is to create a new radio button, drawing it on the screen and adding it to *radio.list*. For a radio button the difference between *left* and *right* must be at least 100 pixels and the difference between *top* and *bottom* must be at least 20.
RETURNS a *control* representing the new radio button.

The following function is used to see which button in a set is currently selected:

selectedRadio **proc(pntr** *radio.list*->**pntr)**
WHERE *radio.list* is the *head* of a list of radio buttons
PURPOSE The selected button has a *value* of one while that of the others is zero. Find the first radio button in the list with *value* one.
RETURNS the *control* representing the selected button.

Care is needed when disposing of radio buttons since they are part of two data structures, namely a control list and a radio list. A button that is not selected can be removed straightforwardly. If a button is selected then as it is removed the next button in the list becomes the current selection.

1.3.3 Scrollbars

The scrollbar is a graphical form of valuator and in this implementation returns an integer value within a programmer-defined range. Fig. 3 shows a scrollbar with the partcodes of various regions labelled.

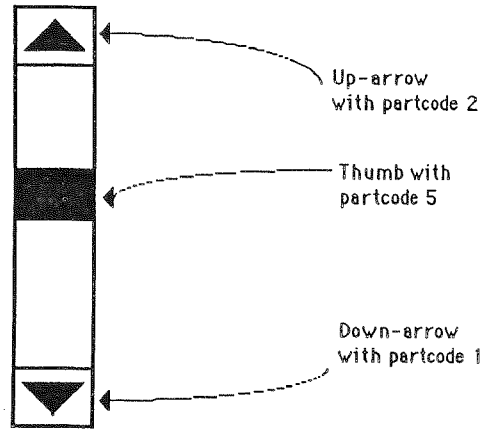


Fig. 3

The down-arrow button (partcode 1) decreases the value of the scrollbar by one. The up-arrow button (partcode 2) increases the value of the scrollbar by one. The thumb (partcode 5) is moveable by the user using the mouse and causes the value of the scrollbar to alter according to the relative placing of the thumb.

The up and down buttons are activated by simply clicking within them with the mouse. The thumb is moved by depressing the mouse button within the thumb and, while keeping the mouse button pressed, moving the mouse along the major axis of the scrollbar. This causes an outline of the thumb to be

dragged along the scrollbar. When the mouse is released the thumb moves to the position of the outline and the value of the scrollbar is changed accordingly. While the mouse is held down the cursor may be moved outside the scrollbar. If the cursor remains within the 'slop' rectangle the outline of the thumb remains visible, however if the cursor is moved outwith the slop rectangle the outline disappears. If the mouse is now released the value of the scrollbar will be unaltered and the *trackControl* function will return with partcode zero. If however the mouse is moved back inside the slop rectangle with the button still held down, the outline reappears and will drag as normal.

The maximum and minimum values that the scrollbar can take are set using the toolbox functions *setCtlMax* and *setCtlMin* (see below).

There are two forms of scrollbar: a vertical scrollbar where the axis of movement is vertical (as in fig. 3) and a horizontal scrollbar where the axis is horizontal. These are created respectively with:

create.VScrollbar **proc(int** *left,bottom,right,top*->**pntr)**
WHERE *left,bottom,right,top* define the enclosing rectangle.
PURPOSE Produces a vertical scrollbar, drawing it on the screen.
RETURNS A *control* representing a vertical scrollbar.

create.HScrollbar **proc(int** *left,bottom,right,top*->**pntr)**
WHERE *left,bottom,right,top* define the enclosing rectangle.
PURPOSE Produces a horizontal scrollbar, drawing it on the screen.
RETURNS A *control* representing a horizontal scrollbar.

For the horizontal scrollbar *right* less *left* must be at least 60 pixels and *top* less *bottom* at least 6. Similarly for the vertical scrollbar: *top* less *bottom* must be at least 60 pixels and *right* less *left* at least 6. For the horizontal scrollbar the left button corresponds to the down button and the right button corresponds to the up button (see fig. 3).

1.4 **Generic functions**
1.4.1 **Drawing controls**
showControl **proc(pntr** *cntl*)
WHERE *cntl* is a *control*.
PURPOSE Draws the control, making it visible if it was previously invisible.

drawControls **proc(pntr** *control.list*)
WHERE *control.list* is a list of *controls*.
PURPOSE Draws all the visible controls in *control.list*.

draw.all.Controls **proc(pntr** *control.list*)
WHERE *control.list* is a list of *controls*.
PURPOSE Draws all the controls in *control.list* making invisible controls visible.

hideControl **proc(pntr** *cntl*)
WHERE *cntl* is a *control*.
PURPOSE Sets the control invisible and removes its representation from the screen.

hiliteControl **proc(pntr** *cntl* ; **int** *hilitestate*)
WHERE *cntl* is a *control*
AND *hilitestate* is an integer partcode.
PURPOSE Draws the control having set the *hilite* field to *hilitestate*. If *hilitestate*=0 the whole control is drawn. If *hilitestate*=255 the control is made inactive (crossed out).

1.4.2 **Setting titles**
setCTitle **proc(pntr** *cntl* ; **#pixel** *name*)
WHERE *cntl* is a *control*
AND *name* is an image.
PURPOSE Sets the *title* field of *cntl* to the image *name*. The control is not re-drawn.

getCTitle **proc(pntr** *cntl*->**#pixel)**
WHERE *cntl* is a *control*.
PURPOSE Get *cntl*'s current title (the image drawn inside a button).
RETURNS an image.

1.4.3 **Point testing**
testControl **proc(pntr** *cntl,the.point*->**int)**
WHERE *cntl* is a *control*
AND *the.point* is a *point* containing a pair of *screen* coordinates.
PURPOSE Get the partcode of the control's region that *the.point* is in.
RETURNS an integer partcode. (0 if *the.point* is outside the control.)

1.4.4 **Rectangle manipulation**
sizeControl **proc(pntr** *cntl* ; **int** *width,height*)
WHERE *cntl* is a *control*
AND *width,height* represent numbers of pixels.
PURPOSE Change the top left of *cntl*'s defining rectangle by *width* and *height* pixels along the x and y axes respectively and then re-draw the control.

moveControl **proc(pntr** *cntl,the.point*)
WHERE *cntl* is a *control*.
AND *the.point* is a *point* containing a pair of *screen* coordinates.
PURPOSE Moves the entire control rectangle so that *the.point* is the new bottom left. The dimensions of the rectangle remain unchanged and the control is re-drawn.

1.4.5 **Control values**
setCtlValue **proc(pntr** *cntl* ; **int** *val*)
WHERE *cntl* is a *control*
AND *val* is the value to which the control is to be set.
PURPOSE Sets the *value* field of *cntl* to *val* unless it is outwith the range of the previously specified minimum and maximum. If *val* exceeds the range *value* becomes the maximum, if *val* is below the range then *value* is set to the minimum.

getCtlValue **proc(pntr** *cntl*->**int)**

WHERE *cntl* is a *control*.
RETURNS an integer representing the value of the scrollbar.

setCtlMax **proc(pntr** *cntl* **; int** *val***)**
WHERE *cntl* is a *control*
AND *val* is the maximum value to which the control can be set.
PURPOSE Sets the maximum value the control can represent to *val*. If *val* is less than the minimum allowed then it is set to the minimum instead.

getCtlMax **proc(pntr** *cntl*->**int)**
WHERE *cntl* is a *control*.
RETURNS the maximum possible value of the scrollbar.

setCtlMin **proc(pntr** *cntl* **; int** *val***)**
WHERE *cntl* is a *control*
AND *val* is the minimum value to which the control can be set.
PURPOSE Sets the minimum value the control can represent to *val*. If *val* is greater than the maximum allowed then it is set to the maximum instead.

getCtlMin **proc(pntr** *cntl*->**int)**
WHERE *cntl* is a *control*.
RETURNS the minimum possible value of the scrollbar.

2. Writing applications that use the notifier system

Before reading this section the reader should already be familiar with the basic ideas behind the notifier toolbox as discussed in the PS-algol toolbox specifications by Quintin Cutts and Graham Kirby [2]. A brief refresher of the main points is given below.

2.1 Basic ideas

2.1.1 Notifiers

The fundamental idea in the above specification is the use of passive programming. All applications are written in an object-oriented style. Conventionally an application reads input by continually polling the keyboard and mouse until an event is received. In a notifier system applications are passive and are only called when an event occurs which concerns them.

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 (henceforth referred to as *'interested'* and *'doIt'* respectively) constitute a notification. The *doAction* procedure (see the definitions in 2.2) of the notifier traverses its registered notifications, interrogating the *interested* procedure of each until a notification indicates that the given event is indeed of interest; the paired *doIt* procedure is then applied.

2.1.2 Event monitor

An event monitor is a loop which takes all events and passes them to the *doAction* procedure of a notifier. This is usually some special notifier designated by the main application as a top level notifier in a hierarchy. Since the signatures

of the *doAction* and *doIt* procedures are the same, it is possible to substitute one for the other. This allows for a hierarchy of notifiers.

2.2 Some notifier definitions

To expedite the discussion we include here the definitions of those PS-algol structures, mentioned in the text but not otherwise defined, that are used in the notifier system:

```
structure NotifierPack(proc(pntr) doAction ;
   proc(proc(pntr->bool), proc(pntr)->proc()) addNotification)

structure Rect(pntr origin, corner)
structure Dimensions(int sizeX, sizeY)
structure StringHolder(string theString)
structure RealHolder(real theReal)
structure ImageHolder(#pixel theImage)

structure WindowManagerPack(
   proc(pntr, int, bool->pntr) createWindow;
   proc(pntr) bringToFront, putToBack, closeWindow,
      openWindow, zapWindow, makeCurrent ;
   proc(pntr->int) findPriority ;
   proc(pntr->pntr) getWindow ;
   proc(pntr, pntr) resize, putIconPos, resizeProc)

structure Window(
   proc(->string) getTitle ;
   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) rasterTo ;
   proc(int, int, int, int, int) rasterLine ;
   proc(int) putCursorRule ;
   proc(->int) getStyle ;
   proc(->bool) windowIsOpen)

structure DefaultsPack(
   int IconSize, CursorSize, TitleBarHeight, BorderThickness,
      NoBorderStyle, MinWinWidth, MinWinHeight ;
   pntr TheFont ;
   #pixel BackgroundPicture, DefaultIcon)

structure TileManagerPack(
   proc(pntr, int, bool->pntr) createTile ;
   proc(pntr) closeTile, openTile, zapTile ;
   proc(pntr->pntr) getTile ;
   proc(pntr, pntr) resizeTile, putTileIconPos, resizeTileProc)

structure SystemPack(
   proc(->pntr) NotifierGen ;
   proc(pntr, pntr, pntr, pntr, pntr->pntr)
      WindowManagerGen, 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->pntr) EditorGen)

structure UtilitiesPack(
```

```
    proc(pntr,pntr->proc(int->pntr)) BorderOffsetGen ;
    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)

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

structure PanelItemPack(
    proc(pntr,pntr,pntr,pntr
      ->proc(pntr,pntr,proc()->pntr)) LightbuttonGen ;
    proc(pntr,pntr,pntr,pntr,pntr->
      proc(pntr,*pntr,proc(pntr)->pntr)) ChoiceGen ;
    proc(pntr,pntr,pntr->proc(pntr,bool,int,int,real,real,
      real,proc(pntr)->pntr)) ClickSliderGen ;
    proc(pntr,pntr,pntr->proc(pntr,bool,int,int,real,real,
      real,proc(pntr)->pntr)) ValuatorGen ;
    proc(pntr,pntr,pntr->proc(pntr,bool,int,int,real,real,
      real,real,proc(pntr)->pntr)) ScrollBarGen)
```

2.3 Producing applications

2.3.1 Window managers

The following discussion also applies to tile managers since a tile is only a special case of a window.

When creating a window manager or tile manager it is necessary to specify a parent window in which the manager is to reside. However this begs the question: in which window do you specify the first window manager? Obviously in order to create the window manager you require a window, but in order to create this window you require a window manager (recognise a chicken and egg situation?). The answer is to cheat. Unfortunately it is necessary to actually set up the first window by specifying each field of a *Window* structure yourself. This is tedious so it is recommended that you keep a file that has this window-setting code in it together with any structure definitions required, and use this file as a header when writing applications for the toolbox.

2.3.2 Creating windows

2.3.2.1 Borders

When creating a window or a tile it is necessary to provide a *Rect* structure specifying the position and limits within which the window is drawn, together with a border style for the window. However when the window is drawn only the window interior, i.e. the part of the window in which the user may write and draw, conforms to this rectangle. Depending on the border thickness of the style of window chosen, the actual image of the window on the screen may exceed the dimensions of the rectangle. This causes windows to slightly overlap sometimes if they are close together. However this is only of aesthetic significance and should not cause any problems with an application.

2.3.2.2 Notification

When a window is created a notification is placed on the window manager's notifier. The *doIt* of the notification is empty (i.e. it is a procedure that does nothing). The *interested* procedure is set in one of two ways. The call of *createWindow* can include an indication that the window will only accept events

generated from within its borders: *interested* will only return true if this is satisfied. Otherwise *interested* unconditionally returns true, which means that the window will accept events generated anywhere on the screen.

2.3.2.3 Coordinate translation

As mentioned above, when a window or tile is created a notification is placed on the notifier for that window. If the window restricts acceptable events to those occurring within its boundary then the *interested* procedure is responsible for guaranteeing this. The *interested* will be 'wrapped' in an enclosing procedure: its purpose is coordinate translation.

Whenever a mouse event occurs the coordinates of the mouse location are in global form, i.e. with respect to the PS-algol *screen* origin. When a notification for a window is entered the window manager adds a procedure that causes the event coordinates to be changed from global to local. Fig. 4 illustrates this point.

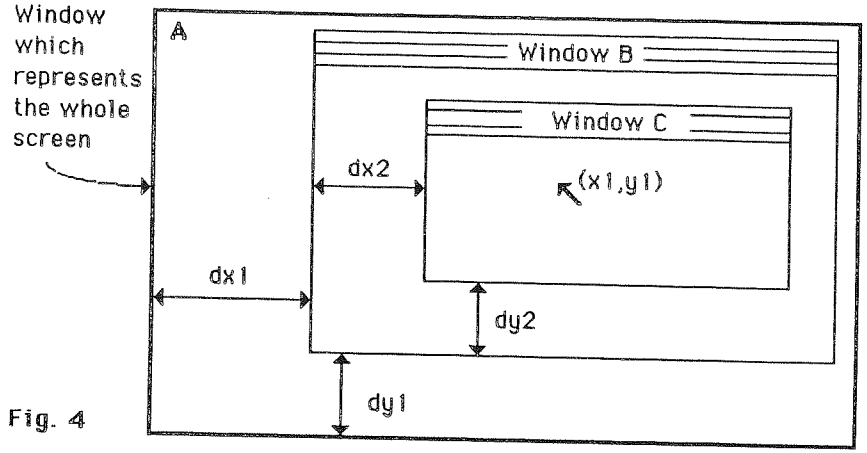


Fig. 4

If a window manager is defined within B and creates a window C then, if an event occurs within C, the following translations occur:

- event occurs at x1,y1
- event within window B translates x1,y1 -> x1-dx1,y1-dy1 = x2,y2
- event within window C translates x2,y2 -> x2-dx2,y2-dy2 = x3,y3

The main upshot of this translation is that the application programmer, when writing an application procedure (*doIt*) for a window, may assume that any event coordinates passed to the window application are in the coordinate system of that window. Caution is required in certain cases where window applications themselves add notifications to the notifier. In these cases the addition of the notification is not under the control of the window manager and so no coordinate translation will be performed when this new notification is given control. This is illustrated in the example that follows.

Below is a *doIt* application to detect a mouse-down within a window (*theWindow*) and to subsequently detect the mouse button's release. On the button being released inside the window the procedure *actionSuccess* is

performed otherwise *actionFail* is called. The set up is illustrated in fig. 5. The 'action' procedures are assumed to have been declared previously, as are *theWindow* and *theNotifier* (a *Window* and *NotifierPack* respectively).

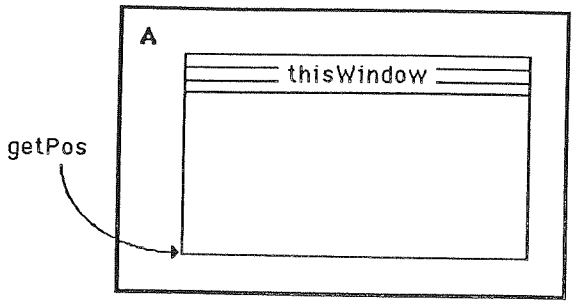


Fig. 5

```
let doIt=proc(pntr event)
if event is mouse do
begin
   let remove.mouseDown:=proc() ; nullproc

   let interested.mouseDown=proc(pntr event->bool) ; true

   let doIt.mouseDown=proc(pntr event)
   if event is mouse and ~event(the.buttons)(1) do
   begin !the mouse button is up
      let thisOrigin=thisWindow(getPos)() !origin of window
      let size=thisWindow(getSize)() !x and y size of window

      let insideWin= !true if mouse in window
         event(X.pos)>=thisOrigin(point.x) and
         event(X.pos)<=thisOrigin(point.x)+size(sizeX) and
         event(Y.pos)>=thisOrigin(point.y) and
         event(Y.pos)<=thisOrigin(point.y)+size(sizeY)

      if insideWin then actionSuccess() else actionFail()

      remove.mouseDown() !from the notifier
   end

   if event(the.buttons)(1) do !mouse button pressed
      remove.mouseDown:=thisNotifier(addNotification)(
         interested.mouseDown,doIt.mouseDown)
end
```

The above procedure is added as an application to the window *thisWindow* thus:

```
thisWindow(putApplication)(doIt)
```

This application is only called by the notifier *thisNotifier* when the mouse is inside the window *thisWindow*. Thus when *doIt* is called it can be asserted that the mouse is already within *thisWindow*. The *doIt* procedure performs a check to see if the mouse button is down, and if so the application places the 'mouseDown' notification on *thisNotifier* and terminates.

The mouseDown notification is added to the top of *thisNotifier* with its *interested* procedure set to yield **true**. This ensures that its *doIt* procedure will

always be called every time *thisNotifier*(*doAction*) is called. This is a very useful technique for applications required to perform some action such as polling the mouse for a new event.

There are two important points to note here. Firstly, the notification was put on the notifier by the notification rather than the window manager and is not therefore wrapped in a coordinate transformation procedure. This means that when *doIt.mouseDown* is called it is called with the coordinate system of window A which is the parent window of the window manager and this is why in the example above the boolean *insideWin* is computed. Secondly, when the mouseDown notification is placed on the notifier it is placed at the top and with *interested* set to always yield **true**. This means that all the other applications on the notifier are in effect disabled since the notifier will always call the first procedure on the notifier. Therefore in the *doIt.mouseDown* application the notification is removed from the notifier by the application itself when a mouse button-up is detected.

The changes in the notifier during the execution of the application are shown in fig. 6.

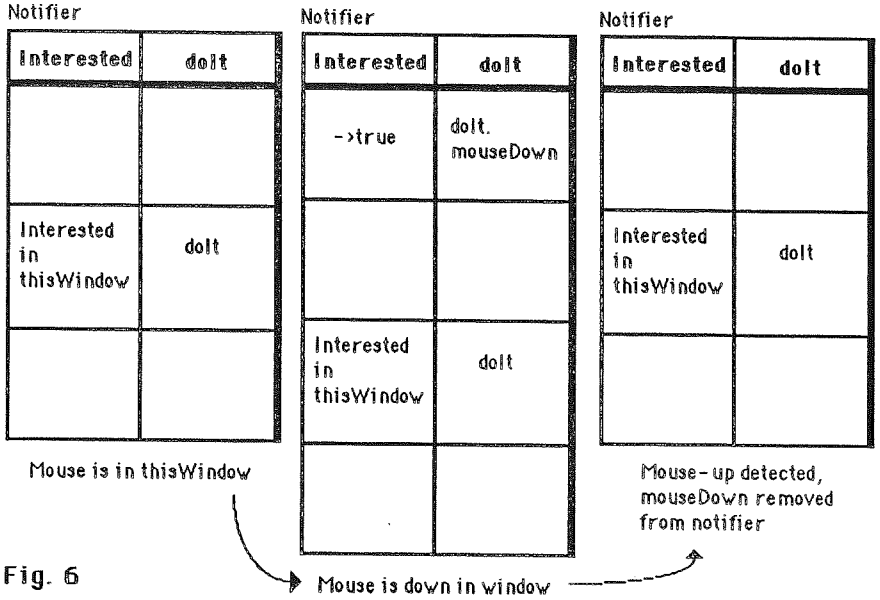


Fig. 6

2.3.3 Dialogue windows

We have seen in previous sections that when a window is created one option is to allow the window to accept all events, whether they originated inside the window or not.

This is particularly useful for text input. Whereas normally the cursor has to be inside the window before a keyboard event can successfully cause the *doIt* to be called, this method allows text to be directed at a given window. In particular dialogue boxes like those of the Macintosh can be implemented. This is possible

since if a window which takes all input is put at the top of a notifier it has the effect of disabling all the other windows that are within that notifier. Dialogues can be used when applications require user input or when an error message needs to be displayed e.g. like the Macintosh system uses to display system error messages. Dialogue windows are used in the icon editor demonstration application (see section 4). They are used to get user input when saving and loading icons to and from the database and are also used to display any errors that occur.

2.3.4 Removing windows

When removing a window using *zapWindow* from the *WindowManagerPack* the programmer should always make sure that all toolbox systems connected with the window have been removed. For example any windows created in window managers created within the window to be 'zapped' should be removed first. If this is not done then when a child window tries to re-draw itself within its parent, it will not be able to do so (since the parent has been removed) and an error will occur. This also applies to applications such as panel items (section 3) that are declared within the window. If these are not removed then the panel item application remains on the notifier and so the button will remain active though it will not be visible. Thus if a mouse-down occurs within the region of the button it will activate, even though it is not displayed, and may cause 'unusual' effects.

2.3.5 Drawing in a window

Every window has an image associated with it onto which all graphics and text appearing in the window is drawn. This image is returned by a *Window's getContent*.procedure. However any changes made to this image will not be shown on the screen until an update of the window occurs. The user can however cause such an update, e.g. to the window *thisWindow*, by:

```
thisWindow(putContent)(thisWindow(getContent)())
```

Another point to note is that when a window is reduced in size the image associated with it remains the same size, but only the region of it which corresponds to the size of the window with respect to the origin is visible in the window. The part not displayed is not lost, so when the window is enlarged again the hidden part of the image is still there. Fig. 7 illustrates this.

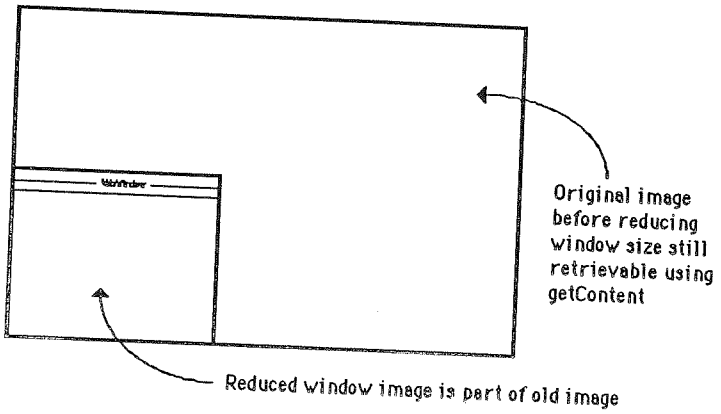


Fig. 7

3. Panel items from the notifier-based toolbox

In [2] a notifier-based user interface toolbox is described. The notifier system was reviewed in the previous section. Here we describe the use of panel items, a subset of the notifier toolbox.

3.1 Types of panel item

3.1.1 Buttons

A light button is an area of the screen which has an associated procedure. When a light button is selected the user receives visual feedback and the associated procedure is executed.

3.1.2 Choices

Choices allow the user to select one item from a list of choices. In this implementation these consist of radio buttons.

3.1.3 Sliders

Sliders are graphical valuators that provide an analogue interface (i.e. a continuous range of values). A good example is a scrollbar.

3.2 Interfaces

The general interface for each of these panel types consists of a generator procedure that returns a procedure which creates a panel item. These generator procedures are held within a *PanelItemPack* structure. All the generator procedures are parameterised by a tile manager and it is within the parent window of this tile manager that the panel item will be placed when the item's procedure is called. Each of the procedures returned by the generators return a pointer to a structure (*PanelStatus*) containing information about the placement of the item: whether it was placed successfully (*placedOK*), if so the position and the size of the item within the parent window of the tile manager that was used in the generator procedure (in the *positionRect* field as a *Rect*), and a procedure that will remove the panel item (*removeItem*). The structure is:

```
structure PanelStatus(
   bool placedOK ; pntr positionRect ; proc() removeItem)
```

If the item is placed successfully then *placedOK* is set to **true** otherwise **false**.

3.2.1 Buttons

The generator procedure for the light button implementation is shown below.

LightbuttonGen **proc(pntr** *tileManager,notifier,defaultsPack,utilitiesPack*
->proc(pntr,pntr,proc()->pntr))

WHERE *tileManager* is a *TileManagerPack*. This structure is returned by a call of the *TileManagerGen* procedure and represents the tile manager and associated window into which all panel items created with the procedure returned by the generator will be placed

AND *notifier* is a pointer to the notifier onto which the button application will be placed. This will normally be the notifier with which the tile manager was generated

AND *defaultsPack* is a *DefaultsPack* containing default settings for various procedures such as window style, border thickness of tiles etc.
AND *utilitiesPack* is a *UtilitiesPack* containing utility procedures such as a string-to-tiler, used within the light button procedure.
RETURNS a procedure that creates a button within the parent window of the tile manager.

Here is that procedure returned by *LightbuttonGen*:

proc(pntr *stringIm,position* **; proc()***buttonProc*->**pntr)**
WHERE *stringIm* is a container for a string or image. This is the label that is placed within the button
AND *position* is a *point.strc* (defined in the PS-algol prelude) specifying the position at which the origin (bottom left corner) of the button will be placed. This point is given in the coordinate system of the parent window
AND *buttonProc* is the application called when the button is activated.
RETURNS a *PanelStatus*, described earlier in 3.2.

Fig. 8 shows a typical button:

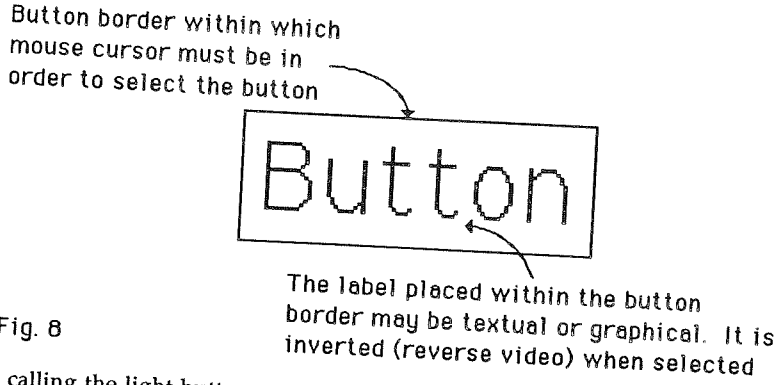


Fig. 8

On calling the light button procedure a tile is created into which the button image is placed. The size of this tile is decided by the size of the image or string held by *stringIm*. The button application is then placed on the notifier associated with the tile manager. If the tile could not be created (e.g. because it overlaps with another tile) then the *placedOK* field of the *PanelStatus* will be set **false**. Otherwise it is **true**.

In order to activate a button it is necessary to press and release the mouse button within the border of the button. If the mouse is pressed within the light button and the cursor is moved outside, with the mouse button still held down, then the button is deactivated. If the mouse button is now released the light button remains inactive. But if the cursor is returned before the button's release, the light button will activate. When the mouse button is pressed inside a light button and the cursor dragged outside the button onto another button, this new button will not activate unless the mouse button is released and pressed again on the new light button.

3.2.2 Choice
The generator procedure for the choice implementation is shown below.

ChoiceGen **proc(pntr** *tileManager,defaultsPack,systemPack,utilitiesPack, makeBorderPack*->**proc(pntr,*pntr,proc(pntr)->pntr))**
WHERE *tileManager,defaultsPack,utilitiesPack* are the same as for buttons
AND *systemPack* is a *SystemPack* containing procedures to create new window managers, notifiers etc.
AND *makeBorderPack* is a *MakeBorderPack* containing procedures for the creation of a variety of borders around tiles.
RETURNS a procedure which creates a set of choice buttons within the parent window of the tile manager.

Here is the procedure returned by *ChoiceGen*:

proc(pntr *position* **; *pntr** *theImages* **; proc(pntr)***processNewState*->**pntr)**
WHERE *position* is as described for the light button procedure
AND *theImages* is a vector of *StringHolders* or *ImageHolders*. The size of the vector gives the number of choice buttons to be created. The strings or images are used as the labels that are placed beside each choice button
AND *ProcessNewState* is a procedure called when an item is selected from the choice set. Its parameter is the button's label, i.e. the structure passed in *theImages*.
RETURNS a *PanelStatus*.

When the generated choice procedure is called it sets up a tile containing all the choice buttons. The size of this tile is determined by the number of choice elements. The choice buttons are activated in a similar way to ordinary buttons and are shown in fig. 9.

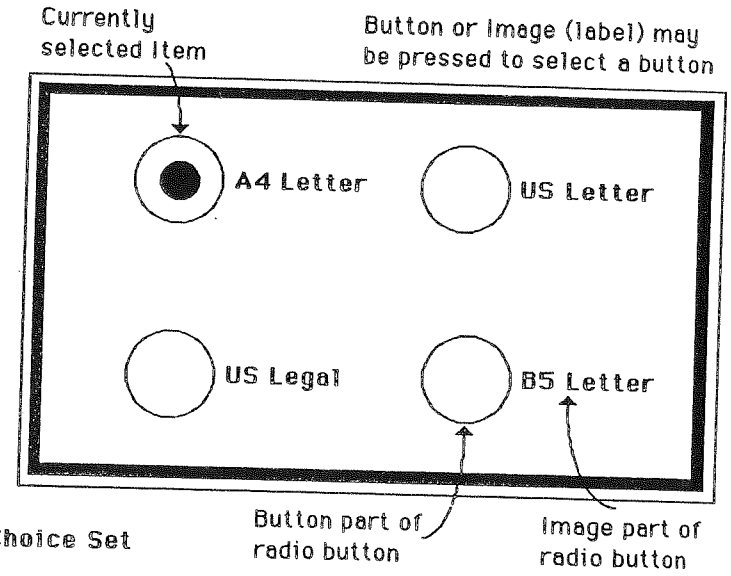


Fig.9 Choice Set

A button is activated by pressing the label or the button itself. The current choice is identified by its highlighted centre.

3.2.3 Sliders

Three kinds of slider have been implemented and are obtained through the following generator procedures:

ValuatorGen **proc(pntr** *tileManager,notifier,defaultsPack*
->proc(pntr,bool,int,int,real,real,real,proc(pntr)->pntr))
ClickSliderGen **proc(pntr** *tileManager,notifier,defaultsPack*
->proc(pntr,bool,int,int,real,real,real,proc(pntr)->pntr))
ScrollBarGen **proc(pntr** *tileManager,notifier,defaultsPack*
->proc(pntr,bool,int,int,real,real,real,real,proc(pntr)->pntr))
WHERE *tileManager,notifier,defaultsPack* are as for *LightbuttonGen*.

These three generator procedures return the following procedures respectively:

proc(pntr *position* **; bool** *horizontal* **; int** *xSize,ySize* **; real** *min,max,initial* **;**
proc(pntr)*processNewState*->**pntr)**
proc(pntr *position* **; bool** *horizontal* **; int** *xSize,ySize* **; real** *min,max,initial* **;**
proc(pntr)*processNewState*->**pntr)**
proc(pntr *position* **; bool** *horizontal* **; int** *xSize,ySize* **; real** *min,max,initial,jump* **;**
proc(pntr)*processNewState*->**pntr)**
WHERE *position* is a *point.strc* specifying the position at which the origin (bottom left corner) of the button will be placed. This is in the coordinate system of the parent window
AND *horizontal* specifies whether the bar lies horizontally (**true**) or vertically
AND *xSize* gives the size of the bar in pixels along the horizontal axis
AND *ySize* gives the size of the bar in pixels along the vertical axis
AND *min* specifies the minimum value that the slider can represent
AND *max* specifies the maximum value that the slider can represent
AND *initial* is the value to which the slider is set when first created. This also determines the position of the sliding 'thumb'
AND (FOR SCROLLBARS ONLY) *jump* (described below)
AND *processNewState* is a procedure that is called after the thumb has been moved. It is passed a *RealHolder* containing the value of the slider following the move.
RETURNS a *PanelStatus*.

The following restrictions apply:

- If *horizontal* is **true** then *xSize* must be at least 20 pixels (60 if it is a scrollbar). Similarly *ySize* of vertical sliders must be at least 20 (60 if it is a scrollbar). 20 is the size of the thumb (60 is the size of the thumb plus the two end buttons of a scrollbar).
- *ySize* of horizontal sliders and *xSize* of vertical sliders must exceed zero.
- The relationship *min<=initial<=max* where *min<max* must hold.
- No tiles may overlap.

If any of the above conditions are broken then the slider will not be created.

Fig. 10 shows a valuator. It consists of a thumb which may be moved along the length of a slider. The thumb is moved by placing the cursor over the thumb and, while holding down the mouse button, dragging the thumb along the slider. When the mouse is released the *processNewState* procedure is applied to the new value of the slider. If the cursor is moved outside the slider while dragging then the slider is deactivated; the thumb returns to the position it was at before being dragged. In this case *processNewState* is not called.

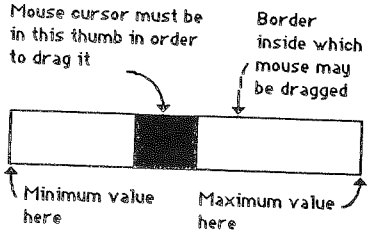


Fig.10 Valuator

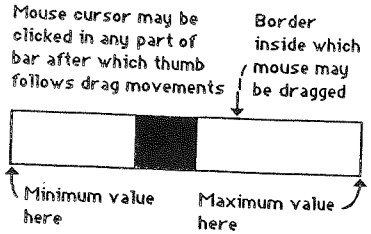


Fig.11 ClickSlider

Fig. 11 shows a click-slider. This is similar to a valuator except that it is not necessary to click directly on the thumb to move it. Instead, when the mouse is clicked within the bar the thumb moves directly to that location and is then dragged in the normal manner.

Fig. 12 shows a scrollbar. The arrow buttons at either end allow for fine movement of the thumb in either direction by single pixel steps. In addition, if the area between the thumb and the left-arrow (or right-arrow) button is pressed, the thumb moves left (or right) the distance *jump* represents. The thumb is moved like that in a valuator, i.e. it is clicked and dragged.

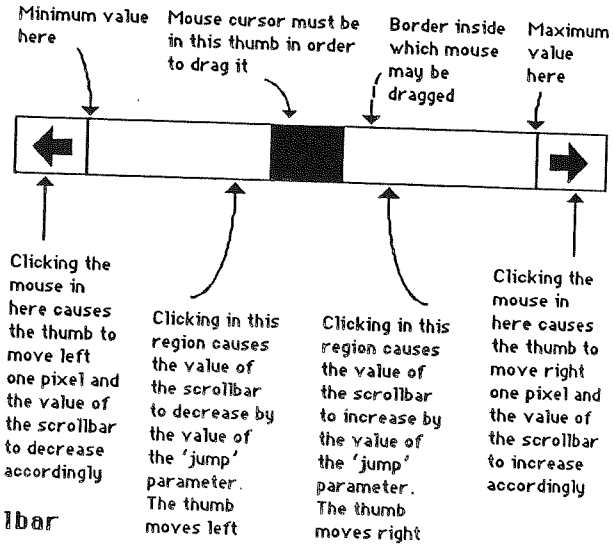


Fig.12 Scrollbar

4. An icon editor

The 'IconEdit' demonstration program is built from the notifier toolbox. It illustrates in particular the use of light buttons, radio buttons and scrollbars.

The icons used by the notifier-based window manager are objects in the persistent store. IconEdit is a tool for editing these objects.

Fig. 13 shows IconEdit running in a window maintained by the window manager. Across the top is the title bar of the window. In the top-right quadrant is a cluster of light buttons. These support a number of top-level operations (*clear*, *quit*, *load*, *save* and *resize*). In the figure, *load* has been selected and a dialogue box has appeared in the lower right quadrant.

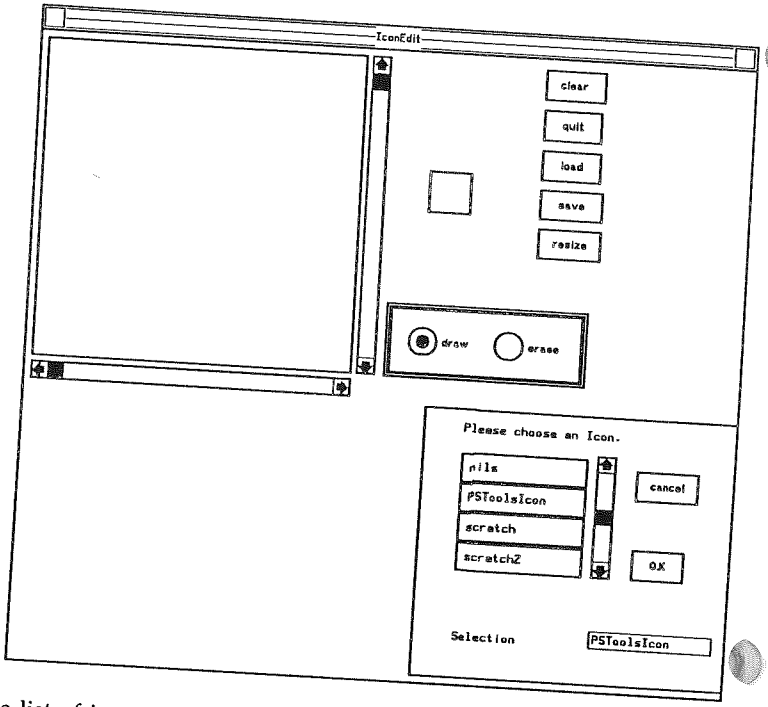


Fig. 13

Part of a list of icon names is visible in the dialogue box. The scrollbar to their right is for scrolling through the remainder of the list. A box in the lower right shows the current icon is 'PSToolsIcon'; this is revised by clicking on one of the visible icon names (they are themselves implemented as panel items). The dialogue box is closed by selecting either of *cancel* or *O.K.* While the dialogue box is active input to all external panel items in the application is disabled. By clicking *O.K.* we proceed to fig. 14.

The top-left quadrant of the application's window is where editing operations take place. Modifications of the (magnified) icon is reflected in the smaller (actual size) box to the right. Each icon pixel appears in the editing area as a larger square block (the ratio is set using the *resize* button). The mouse cursor is used as a brush. A set of two radio buttons (beneath the actual-size box)

toggles the action of the brush (black paint, white paint). *clear* paints every pixel white. The icon is panned across using the vertical and horizontal scrollbars.

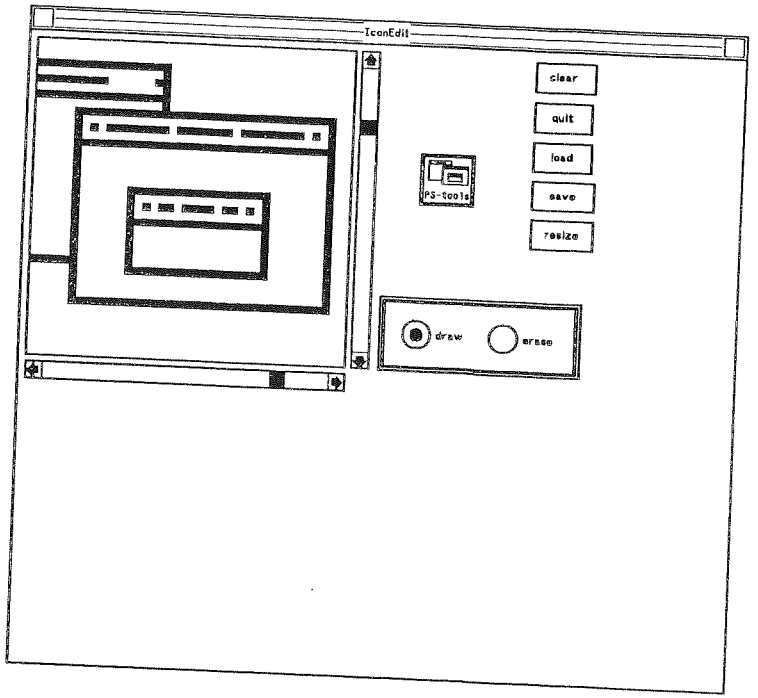


Fig. 14

5. How to find them

The non-notifier toolbox is part of the "rutilities" procedure library (see Richard Cooper, 'A Utility Library for PS-algol', elsewhere in this report). In the following segment of PS-algol code a *toolbox* structure (see section 1.1) is reached through the library utility *prcget*.

```
let thetoolbox=
begin
   structure procpak(proc(->pntr)xproc)
   prcget("ToolBoxGen")(xproc)() !->toolbox
end
```

The panel tools are in a database called "dbTools" with password "tools". The object keyed by "panelItemPack" is a *PanelItemPack* (see section 2.2).

IconEdit is available as the PS-code file 'IconEdit.out'.

References

[1] Apple Computer, Inc. *Inside Macintosh, Volume I*, Addison-Wesley (1985)

[2] Cutts, Q. and G. Kirby. An event-driven software architecture. Department of Computing Science, Glasgow University and Department of Computational Science, St Andrews University, Persistent Programming Research Report 48, 1987

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", *Pergammon Infotech State of the Art Report*, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Gray, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages", *Software, Practice and Experience*, 7, 5, 535-537, July 1977.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", *Proceedings of the DECUS conference*, Bath, 5-10, April 1978.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", *ACM SIGDA newsletter*, Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", *Proceedings of the 4th International Conference on Very Large Data Bases*, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", *Journal of Documentation*, 35, 1, 49-51, March 1979. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", *Information Processing Letters*, 9, 1, 1-4, July 1979.

Atkinson, M.P.
"Data management for interactive graphics", *Proceedings of the Infotech State of the Art Conference*, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", *Infotech State of the Art Report*, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", *Software Practice and Experience*, 12, 767-776, 1981.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17, 7, 24-31, July 1981. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in *Database, Pergammon Infotech State of the Art Report*, Series 9, No.8, 299-318, January 1982 - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", *Computer Bulletin* II/31, March 1982.

Morrison, R.
"The string as a simple data type", *ACM Sigplan Notices*, 17, 3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure".

Morrison, R.
"Towards simpler programming languages: S-algol", *IUCC Bulletin*, 4, 3, 130-133, October 1982.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the *Workshop on programming languages and database systems*, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983.

Atkinson, M.P.
"Data management", in *Encyclopedia of Computer Science and Engineering 2nd Edition*, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", *Software Practice and Experience*, 13, 3, 259-272, March 1983. Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", *Software Practice and Experience*, 13, 3, 273-285, March 1983. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the *DEC workshop on Programming Languages and Databases*, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", *The Computer Journal*, 26, 4, 360-365, 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", *10th Australian Computer Conference, Melbourne*, 70-79, September 1983.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", *Eurographics Conference Zagreb*, North Holland, 7-17, September 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", *Software Practice and Exerience*, 14, 1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in *Databases - Role and Structure*, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", *Foundations of Software Technology and Theoretical Computer Science* (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin, 1984.

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", *Proceedings of the third British National Conference on Databases*, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, July 1984.

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", *ACM TOPLAS*, 7, 4, 539-559, Oct. 1985.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", *8th International Conference on Software Engineering*, Imperial College, London, 166-172, August 1985.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988

Davie, A.J.T.
"Conditional declarations and pattern matching", *Proceedings of Data Types and Persistence Workshop*, Appin, 278-283, August 1985.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988

Buneman, O.P.
"Data types for data base programming", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988.

Norrie, M.C.
"PS-algol: A user perspective", *Proceedings of Data Types and Persistence Workshop*, Appin, 399-410, August 1985.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", *Proceedings of Data Types and Persistence Workshop*, Appin, August 1985, 423-438.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", *Software Practice and Experience*, 14, 3, 1986.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", *Proceedings of the 9th Australian Computing Science Conference*, January, 1986.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", *The Computer Journal*, 29, 1, 38-45, 1986.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages", *Proceedings ACM SIGMOD Conference 1986*, Washington, USA, May 1986.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", *Computer Graphics Forum*, 5, 2, 147-157, June 1986.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", *Proceedings of Information Processing 1986*, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", *University Computing*, 8, 2, Summer 1986.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", *Software - Practise and Experience*, 17, 3, 171-185, March 1987.

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", *Software Tool 87: Improving Tools, Advance Computing Series*, 8, 89-96, Online Publications, June 1987.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", *Presnted at the 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", *ICL Technical Journal*, 5, 3,477-491, May 1987.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Wai, F.
"Distribution and Persistence", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Guy, M.R.
"Persistent Store - Successor to Virtual Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment", *Presented at the 2nd Internaional Workshop on Persistent Object Stores*, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store", *Presented at the 2nd International Workshop on Persistent object Stores*, Appin, August 1987.

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment", *Proceedings of the Thirteenth Internaional Conference on Very Large Databases*, Brighton, 117-126, September 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", *Presented at the Workshop on Database Programming Languages, Roscoff*, September 1987.

Brown, A.L., Dearle, A. & Morrison, R.
"An Architecture for a Strongly Typed Persistent Object Store", *Proceedings of Workshop on Object Oriented Programming Systems, Languages and Applications at the OOPSLA Conference in Orlando*, Florida, 1987.

Brown, A.L.
"The Implementation of a Distributed Persistent Object Store", *Presented at the Workshop on Object-Oriented Databases*, Semantic Aspects, at the OOPSLA Conference in Orlando, Florida, 1987.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A. & Atkinson, M.P.
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object - Oriented Environment", *IEE & BCS Journal on Software Engineering*, Dec 1987.

Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Binding and Type-Checking in Database Programming Languages", *Computer Journal*, 31, 2, 99-109, 1988.

Dearle, A. & Brown, A.L.
"Safe Browsing in a Strongly Typed Persistent Environment", *Computer Journal*, 31, 2, 1988.

Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"Flexible Incremental Bindings in a Persistent Object Store", *ACM Sigplan Notices*, 1988.

Philbrow P., Armour I., Atkinson, M.P. and Livingstone J.
"PS-algol's Device-Independent Output Statement", *Computer Journal*, 31, 1988.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 6th May 1988. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-88	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - 2nd edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00

PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-87	An Introduction to PS-algol Programming (third edition) - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00
PPRR-44-87	Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£20.00
PPRR-45-87	Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R.	£1.00
PPRR-46-87	Transactions and Concurrency - Krablin, G.L.	£1.00
PPRR-47-87	Persistent Information Space Architecture - PISA Club Rules - Atkinson, M.P., Lucking, J.R., Morrison, R. and Pratten, G.D.	£1.00
PPRR-48-87	An Event-Driven Software Architecture - Cutts, Q. and Kirby, G.	£1.00
PPRR-49-87	An Implementation of Multiple Inheritance in a Persistent Environment - Benson, P.J., D'Souza, E.B., Rennie, I.S., Waddell, S.J.	£1.00
PPRR-50-87	A Distributed Stable Store - Brown, A.L.	£1.00
PPRR-51-87	Constructing Compilers in a Persistent Environment - Dearle, A.	£1.00
PPRR-52-87	Lgen, Pgen and Sgen - Language Development Tools for a Persistent Programming Environment - Blott, S.M. and Campin, J.	£1.00
PPRR-53-87	Polymorphic Names and Iterations - Atkinson, M.P. and Morrison, R	£1.00
PPRR-54-87	A Requirements Modelling Tool Built in PS-algol - Cooper, R.L. and Atkinson, M.P.	£1.00

PPRR-55-87	A Persistent Software Database with Version Control - Cooper, R.L. and Atkinson, M.P.	£1.00
PPRR-56-87	User Interface Tools in PS-algol - Cooper, R.L., McFarlane, D.K. and Ahmed, S.	£1.00
PPRR-57-88	On the integration of Object-Oriented and Process-Oriented computation in persistent environments Morrison, R., Brown, A.L., Carrick, R.,Connor, R. and Dearle, A.	£1.00
PPRR-58-88	The Napier Type-Checking Module Connor, R.	£1.00
PPRR-59-88	The Persistent Abstract Machine Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Morrison, R.	£1.00
PPRR-60-88	Distributed PS-algol Wai, F.	£1.00
PPRR-61-88	Universal Persistent Store - An Approach towards Distributing Persistent Data Wai, F.	£1.00
PPRR-62-88	Tense Logic and Nested Transactions Campin, J.	£1.00
PPRR-63-88	Language Constructs for Distributed Persistent Programming Campin, J.	£1.00