```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
/* X11R2 modifications and fixes made from sources by Crispin Goswell,
 * Barry Shein of BU, and Jean Marie Diaz of MIT.  The resulting
 * melange, along with other fixes for color workstations, is entirely
 * my fault.
 *
 * David Jedlinsky
 * MIT Project Athena Watchmaker
 * opus@athena.mit.edu
 *
 */
#include "main.h"
#include "graphics.h"
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>
#include <stdio.h>

#define ISWIN   1
#define ISBITMAP 2

typedef int ColorType;
#define ISPIXEL         1
#define ISHALFTONE      2

/* Also change #define in font.c */
#define UpdateControl(h, on) {}

struct screen
{
  float val;
  int sx, sy;
  struct hardware *shade;
} *screen = NULL;

static int screen_size;

#define TRANSFER_SIZE   256

static int transfer [TRANSFER_SIZE];

extern int RUNSYNC, TESTING;

int pixels_per_inch;

float hppp, vppp;

int single_rop [] =
{
  ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
  ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
  ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
  ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE
  };

void InitTransfer (ppi)
     int ppi;
{
  int i;

  pixels_per_inch = ppi;
  for (i = 0; i < TRANSFER_SIZE; i++)
    transfer [i] = i;
}

static void Punt(str)
     char *str;
{
  fprintf(stderr, "%s\n", str);
  exit(1);
}

static Display *dpy;
static int SCREEN;
static struct hardware *NewHardware ();
struct hardware *GraySync ();

extern HardPoint DeviceToHard ();

typedef struct _HardwareRec {
  Drawable w;
  GC gc;
  union last_color {
    unsigned long pix;
    Pixmap halftone;
  } last_color;
  int last_used;
} HardwareRec, *Hardware;

/*
 * This file describes the interface that PostScript requires to the graphics
 * system at Version 1.4.
 *
 * ''Hardware'' in this context refers to a pointer to windows and/or bitmaps
 * and is the lowest level of access that PostScript is interested in. Any
 * Hardware parameter may be expected to be NULL.
 */

/********************* CREATION OF WINDOWS AND BITMAPS *******************/

#define MIN(x, y)       (((x) < (y)) ? (x) : (y))

static GC GlobalGC = NULL;
static int GlobalMask = 0;
static GC fillgc[16];
static GC BitmapGC[16];
static GC whitegc;
void ModifyGC ();

/*
 * InitHardware () returns a default device which PostScript may use
 * immediately (or NULL if not appropriate).  Its size and shape are
 * not defined. Most typically the user will want to start up another
 * device before it is used anyway. No attempt will be made by
 * PostScript to Destroy the resulting device.
 */
```



```c
struct hardware *InitHardware ()
{
  XGCValues values;
  int i;
  DevicePoint p;

  if ((dpy = XOpenDisplay("")) == NULL)
    Punt("Could not open display");
  SCREEN = DefaultScreen(dpy);

  /* DCJ:
   * This is the real ppi.  However, we might be using a screen that
   * is less than 11 inches tall, which makes life difficult.  So we
   * cheat a bit.
   */

  if (DisplayHeightMM(dpy, SCREEN) < 280) {
    hppp = (float) DisplayWidth(dpy,SCREEN) / ((279.4 / (float) DisplayHeightMM(dpy, SCRE
    vppp = (float) DisplayHeight(dpy, SCREEN) / 792.0;
    InitTransfer(DisplayHeight(dpy, SCREEN) / 11);
  } else {
    hppp = (float) DisplayWidth(dpy, SCREEN) / ((float) DisplayWidthMM(dpy, SCREEN) * 2.8
    vppp = (float) DisplayHeight(dpy, SCREEN) / ((float) DisplayHeightMM(dpy, SCREEN) * 2
    InitTransfer((int) ((float) DisplayHeight(dpy, SCREEN) * (float) 25.4 / (float) Displ
  }

  values.foreground = BlackPixel(dpy, SCREEN);
  values.background = WhitePixel(dpy, SCREEN);

  for (i=0 ; i<16 ; i++) {
    BitmapGC[i] = NULL;
    values.function = i;
    fillgc[i] = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                          GCFunction | GCForeground | GCBackground,
                          &values);
  }

  p.dx = DisplayWidth(dpy, SCREEN);
  p.dy = DisplayHeight(dpy, SCREEN);
  return NewHardware(p);
}

static struct hardware *NewHardware(width, height)
     int width, height;
{
  struct hardware *to;
  Hardware hard;

  to = (struct hardware *) malloc(sizeof(struct hardware));
  hard = (Hardware) malloc(sizeof(HardwareRec));
  to->hard.addr = (char *) hard;
  to->flags = 0;
  to->aux = to->clip = (struct hardware *) NULL;
  to->extent = NewDevicePoint(width, height);
  hard->w = (Drawable) NULL;
  hard->gc = (GC) NULL;
  to->touched = FALSE;
  to->changed = TRUE;
  hard->last_used = 0;
  return to;
}

/*
 * NewBitmapHardware () is expected to create a new bitmap. Only one plane
 * will be needed.
 *
 * NewWindowHardware () is expected to create a window on the screen. On a
 * colour system this will be expected to support full colour.
 */
struct hardware *NewBitmapHardware (width, height)
     int width, height;
{
  XGCValues values;
  struct hardware *to = NewHardware(width, height);
  Hardware hard = (Hardware) to->hard.addr;
  int i;

  to->flags = ISBITMAP;

  if (width == 0 || height == 0) {
    return NULL;
  }
  hard->w = XCreatePixmap(dpy, RootWindow(dpy, SCREEN), width,
                          height, 1);

  if (! BitmapGC[GXcopy]) {
    values.foreground = 1;
    values.background = 0;

    for (i=0; i<16; i++) {
      values.function = i;
      BitmapGC[i] = XCreateGC(dpy, hard->w,
                              GCFunction|GCForeground|GCBackground,
                              &values);
    }
  }
  XFillRectangle(dpy, hard->w, BitmapGC[GXclear], 0, 0,
                 width, height);

  return to;
}

struct hardware *NewWindowHardware (width, height)
     int width, height;
{
  struct hardware *to = NewHardware(width, height);
  Hardware hard = (Hardware) to->hard.addr;
  XEvent event;
  unsigned long vmask;
  XSetWindowAttributes xswa;
  XSizeHints xshints;

  to->flags = ISWIN;
  vmask = CWBackPixel|CWBorderPixel|CWBackingStore|
    CWBackingPlanes;
  xswa.background_pixel = WhitePixel(dpy,SCREEN);
  xswa.border_pixel = BlackPixel(dpy,SCREEN);
  xswa.backing_store = Always;
  xswa.backing_planes = AllPlanes;
  xshints.x = DisplayWidth(dpy, SCREEN) - width;
  xshints.y = 0;
  xshints.width = width;
  xshints.height = height;
  xshints.flags = USPosition|USSize;

  if (RUNSYNC) {
    fprintf(stderr, "Running synchronized.\n"); fflush(stderr);
    XSynchronize(dpy, TRUE);
```

```c
  }
  hard->w = XCreateWindow(dpy, RootWindow(dpy,SCREEN),
                          xshints.x, xshints.y, width, height,
                          1, DefaultDepth(dpy,SCREEN),
                          InputOutput, DefaultVisual(dpy,SCREEN),
                          vmask, &xswa);

  XChangeProperty(dpy,hard->w,XA_WM_NAME,XA_STRING,8,
                  PropModeReplace,"POSTSCRIPT",10);
  if (TESTING)
    XSetNormalHints(dpy, hard->w, &xshints);

  XSelectInput(dpy, hard->w, ExposureMask);
  XMapWindow(dpy, hard->w);
  XSync(dpy, FALSE);
  XNextEvent(dpy, &event);
  XSelectInput(dpy, hard->w, 0);
  return to;
}

/*
 * IsWindowHardware () should return TRUE if the hardware is a window, FALSE
 * otherwise.
 *
 * IsBitmapHardware () should return TRUE if the hardware is a bitmap,
 * FALSE otherwise.
 */

int IsWindowHardware (h)
     struct hardware *h;
{
  return h->flags & ISWIN;
}

#define IsWindowHardware(h) ((h)->flags & ISWIN)

#define IsBitmapHardware(h) ((h)->flags & ISBITMAP)

/*
 *
 * DestroyHardware () should release the resources required by the hardware,
 * bitmap or window.  This should cause a window device to vanish. NULL is not
 * an error (does nothing).
 */

void DestroyHardware (h)
     struct hardware *h;
{
  if (h) {
    Hardware hard = (Hardware) h->hard.addr;
    if (IsWindowHardware(h))
      XDestroyWindow(dpy, hard->w);
    else
      if (IsBitmapHardware(h))
        XFreePixmap(dpy, hard->w);
    Free(h);
  }
}

/*
 *
 * DeviceMatrix () should return a matrix appropriate to a device of the given
 * height and width.  For a typical display with a graphics origin at the top
 * left of a window, an appropriate definition would be:
 *
 * Matrix DeviceMatrix (width, height)
 * int width, height;
 * {
 *      Matrix m;
 *      NewMatrix (m, PIXELS_PER_INCH / 72.0, 0.0, 0.0,
 *                        -PIXELS_PER_INCH / 72.0, 0.0, (float) height);
 *      return m;
 * }
 */

Matrix DeviceMatrix (width, height)
     int width, height;
{
  Matrix m;
  NewMatrix(m, pixels_per_inch / 72.0, 0.0, 0.0,
            -pixels_per_inch / 72.0, 0.0, (float) height);
  return m;
}

/*
 * HardwareExtent () returns a DevicePoint describing the width and height of
 * the argument.  NULL has extent NewDevicePoint (0, 0).
 */

DevicePoint HardwareExtent (h)
     struct hardware *h;
{
  if (h)
    return h->extent;
  else
    return NewDevicePoint (0, 0);
}

static NeedAux (h)
     struct hardware *h;
{
  DevicePoint p;

  if (h->aux)
    return;
  p = HardwareExtent(h);
  h->aux = NewBitmapHardware (p.dx, p.dy);
}

/*************************** OUTPUT PRIMITIVES ******************************/
/*
 *
 * BitBlt () is a full function RasterOp. The 'rop' argument will have values
 * as described in the header file hard.h. If the from argument is NULL it is
 * taken to be a bitmap full of ones the shape of the fromPoint and extent. If
 * the to argument is NULL, this is a no-op.
 *
 * Paint () is an addition to BitBlt. Bits that are set in the source are
 * Painted into the destination in the given colour with a copying rasterop so
 * that they replace pixels previously there. If the machine does not support
 * colour windows, half-toning should be performed.  Colour values have hue,
 * saturation and brightness components. On a black and white or greyscale
 * system the brightness value will be a FP value between 0.0 (black) and 1.0
 * (white), which can be used as a grey level.
 *
 * Paint is expected to mask with the clip mask. BitBlt is not,
 */
```



```c
void BitBlt (from, to, fromPoint, toPoint, extent, rop)
     struct hardware *from, *to;
     DevicePoint toPoint, fromPoint, extent;
     int rop;
{
  Hardware fromhard, tohard;
  static int count = 0;
  GC gc = NULL;

  if (to == NULL) return;
  /*
   * Copying from full color window to monochrome bitmap is the
   * wrong idea.
   */
  if (IsWindowHardware(from) && !IsWindowHardware(to)) return;

  tohard = (Hardware) to->hard.addr;
  if (IsWindowHardware(to)) {
    if (from == NULL) {
      XFillRectangle(dpy, tohard->w, fillgc[rop], toPoint.dx,
                     toPoint.dy, extent.dx, extent.dy);
    } else {
      fromhard = (Hardware) from->hard.addr;
      if (IsWindowHardware(from)) {
        XCopyArea(dpy, fromhard->w, tohard->w, fillgc[rop],
                  fromPoint.dx, fromPoint.dy,
                  extent.dx, extent.dy,
                  toPoint.dx, toPoint.dy);
      } else
        if (IsBitmapHardware (from)) {
          XCopyPlane(dpy, fromhard->w, tohard->w, fillgc[15-rop],
                     fromPoint.dx, fromPoint.dy,
                     extent.dx, extent.dy,
                     toPoint.dx, toPoint.dy, 1);
        }
    }
  } else
    if (IsBitmapHardware(to)) {
      if (from == NULL) {
        XFillRectangle(dpy, tohard->w, BitmapGC[rop], toPoint.dx,
                       toPoint.dy, extent.dx, extent.dy);
      } else {
        if (IsBitmapHardware(from)) {
          fromhard = (Hardware) from->hard.addr;
          XCopyArea(dpy, fromhard->w, tohard->w, BitmapGC[rop],
                    fromPoint.dx, fromPoint.dy,
                    extent.dx, extent.dy,
                    toPoint.dx, toPoint.dy);
        }
      }
    }

  to->touched = TRUE;
  to->changed = TRUE;

  if (gc != NULL)
    XFreeGC(dpy, gc);
  if (count++ % 50 == 0) XSync(dpy, FALSE);
}

void Paint (from, to, fromPoint, toPoint, extent, colour)
     struct hardware *from, *to;
     DevicePoint fromPoint, toPoint, extent;
     Colour colour;
{
  Hardware hard, grayhard, tohard;
  static int count = 0;

  if (!IsBitmapHardware(from) && from != NULL)
    Punt("Paint() called with non-bitmap source.\n");


  if (to == NULL)
    return;

  if (!from->touched)
    return;

  ModifyGC (colour, from, to, fromPoint, toPoint, extent);

  tohard = (Hardware) to->hard.addr;

  XFillRectangle(dpy, tohard->w, GlobalGC, toPoint.dx, toPoint.dy,
                 extent.dx, extent.dy);

  to->touched = TRUE;
  to->changed = TRUE;

  if (++count % 100 == 0) {
    XSync(dpy, FALSE);
  }
}

int FillPath(to, path, rule)
     struct hardware *to;
     Path path;
     int rule;
{
  register Path p;
  static XPoint *vertices = NULL;
  int i = 0;
  Hardware hard;

  if (!vertices) {
    if ((vertices = (XPoint *) malloc(MAXPATHELEMENTS * sizeof(XPoint)))
        == NULL)
      Punt("Not enough memory for filling.\n");
  }

  if (EmptyPath (path))
    return TRUE;

  path = FlattenPath (path);

  if (!CloseAll (path)) {
    PathFree (path);

    return FALSE;
  }

  ModifyGC (gstate->colour, (struct hardware *)NULL, to,
            NewDevicePoint(0,0), NewDevicePoint(0,0),
            NewDevicePoint(0,0));

  hard = (Hardware) to->hard.addr;

  for (p = path->next; p != path; p = p->next)
    switch (p->ptype) {
```

```
    case EMove:
      if (i > 1)
        XFillPolygon(dpy, hard->w, GlobalGC, vertices, i,
                     Complex, CoordModeOrigin);
      vertices[0].x = (short) p->pe.point.hx;
      vertices[0].y = (short) p->pe.point.hy;
      i = 1;
      break;
    case ELine:
      vertices[i].x = (short) p->pe.point.hx;
      vertices[i++].y = (short) p->pe.point.hy;
      break;
    case EClose:
      XFillPolygon(dpy, hard->w, GlobalGC, vertices, i,
                   Complex, CoordModeOrigin);
      i = 0;
      break;
    }

  PathFree (path);

  to->touched = TRUE;
  to->changed = TRUE;

  return TRUE;
}

static short Xvalue (ax, ay, bx, by, cy)
     int ax, ay, bx, by, cy;
{
  if (ay - by == 0) {
    printf("Xvalue: ax=%d, ay=%d, bx=%d, by=%d, cy=%d\n",
           ax, ay, bx, by, cy);
    return bx + (cy - by) * (ax - bx);
  }
  return bx + (cy - by) * (ax - bx) / (float) (ay - by);
}

/*
 * BitBltTrapezoid () and PaintTrapezoid () render a complete trapezoidal
 * shape.  The corners of the trapezoid may lie far outside the range of
 * interesting scan-lines, but the slope of the line should be clipped by the
 * top and bottom. The coordinates are half-open.
 */

void BitBltTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
                      top, bottom, rop)
     struct hardware *to;
     DevicePoint lefttop, leftbottom, righttop, rightbottom;
     int top, bottom, rop;
{
  int i, j, temp;
  static int left [1024], right [1024];

  int
    ltx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                  leftbottom.dy, top),
    rtx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                  rightbottom.dy, top),
    lbx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                  leftbottom.dy, bottom),
    rbx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                  rightbottom.dy, bottom);

  if (ltx == lbx && rtx == rbx) {
    if (rtx < ltx)
      temp = rtx, rtx = ltx, ltx = temp;

    BitBlt ((struct hardware *) NULL, to,
            NewDevicePoint (0, 0), NewDevicePoint (ltx, top),
            NewDevicePoint (rtx - ltx + 1, bottom - top + 1), rop);

    return;
  }

  for (i = top, j = 0; i <= bottom; i++, j++) {
    int
      lx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                   leftbottom.dy, i),
      rx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                   rightbottom.dy, i);

    if (rx < lx)
      temp = rx, rx = lx, lx = temp;

    left [j] = lx; right [j] = rx;
  }
  BitBltBlob (to, top, bottom - top, left, right, single_rop [rop]);

  to->touched = TRUE;
  to->changed = TRUE;
}

void PaintTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
                     top, bottom, colour)
     struct hardware *to;
     DevicePoint lefttop, leftbottom, righttop, rightbottom;
     int top, bottom;
     Colour colour;
{
  struct hardware *gray = NULL;
  Hardware hard, grayhard, tohard;
  int valuemask = 0, ok = FALSE;
  XGCValues values;
  GC gc = NULL;
  unsigned long pix;
  ColorType ret;
  XPoint coord[4];

  ret = RealColor(colour, &pix, &gray);

  /* This should be done better: Halftones aren't handled... */
  if (IsBitmapHardware(to)) {
    BitBltTrapezoid(to, lefttop, leftbottom, righttop, rightbottom,
                    top, bottom, (pix == WhitePixel(dpy, SCREEN)) ? ROP_FALSE : ROP_TRUE)
    return;
  }

  if (to == NULL)
    return;

  coord[0].x = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                       leftbottom.dy, bottom);
  coord[0].y = bottom;
  coord[1].x = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                       leftbottom.dy, top);
  coord[1].y = top;
  coord[2].x = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
```



```
                       rightbottom.dy, top);
  coord[2].y = top;
  coord[3].x = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                       rightbottom.dy, bottom);
  coord[3].y = bottom;

  UpdateControl (to, FALSE);

  if (ret == ISPIXEL) {
    values.function = GXcopy;
    values.foreground = pix;
    valuemask = valuemask|GCFunction|GCForeground;
    if (pix == BlackPixel(dpy, SCREEN))
      gc = fillgc[GXcopy];
    if (pix == WhitePixel(dpy, SCREEN))
      gc = whitegc;
    hard = (Hardware) to->hard.addr;
    if (hard->last_used == ISPIXEL && hard->last_color.pix == pix)
      ok = TRUE;
  } else if (ret == ISHALFTONE) {
    values.function = GXcopy;
    values.foreground = BlackPixel(dpy, SCREEN);
    values.background = WhitePixel(dpy, SCREEN);
    values.fill_style = FillOpaqueStippled;
    grayhard = (Hardware) gray->hard.addr;
    values.stipple = grayhard->w;
    valuemask = valuemask|GCFunction|GCForeground|GCBackground|GCFillStyle|GCStipple;
    gc = grayhard->gc;

    hard = (Hardware) to->hard.addr;
    if (hard->last_used == ISHALFTONE &&
        hard->last_color.halftone == grayhard->w)
      ok = TRUE;

    if (!gc)
      printf("HalfTone GC not set!\n");

  } else return;

  if (to->clip) {
    hard = (Hardware) to->clip->hard.addr;
    values.clip_mask = hard->w;
    valuemask = valuemask|GCClipMask;

    if (ok)
      gc = ((Hardware) to->hard.addr)->gc;

/*        if (to->clip->changed) {
               to->clip->changed = FALSE;
               printf("Clip changed.\n");
               gc = NULL;
          }
*/
    gc = NULL;
  }

  if (!gc) {
    gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask, &values);
  }
  tohard = (Hardware) to->hard.addr;

  XFillPolygon(dpy, tohard->w, gc, coord, 4, Complex, CoordModeOrigin);

  hard = (Hardware) to->hard.addr;
  hard->last_used = ret;
  hard->gc = gc;

/*     if (created)
          XFreeGC(dpy, gc);
          */

  UpdateControl (to, TRUE);

  to->touched = TRUE;
  to->changed = TRUE;
}

/*
 *
 *      BitBltLine () is expected to draw a line between the given points
 *      with the given RasterOp and colour masking.
 *      The line should be one pixel wide and half-open.
 *      [Thicker lines are done with BitBlt.]
 *
 *      PaintLine () is expected to Paint a line by analogy with Paint
 *      and BitBlt.
 */

void BitBltLine (h, fromPoint, toPoint, rop)
     struct hardware *h;
     DevicePoint fromPoint, toPoint;
     int rop;
{
  if (h) {
    Hardware hard = (Hardware) h->hard.addr;
    if (IsWindowHardware(h)) {
      XDrawLine(dpy, hard->w, fillgc[rop], fromPoint.dx, fromPoint.dy,
                toPoint.dx, toPoint.dy);
    } else
      if (IsBitmapHardware(h)) {
        XDrawLine(dpy, hard->w, BitmapGC[rop], fromPoint.dx, fromPoint.dy,
                  toPoint.dx, toPoint.dy);
      }

    h->touched = TRUE;
    h->changed = TRUE;
  }
}

void PaintLine (h, fromPoint, toPoint, colour)
     struct hardware *h;
     DevicePoint fromPoint, toPoint;
     Colour colour;
{
  int col;

  if (h == NULL)
    return;

  col = IsWindowHardware (h) ? HardColour (colour) : 0;

  UpdateControl (h, FALSE);

  if (h->clip == NULL && (col == 0 || col == TransferSize () - 1))
    BitBltLine (h, fromPoint, toPoint,
                (col == 0 ? ROP_TRUE : ROP_FALSE));
  else {
    int mx = Min (fromPoint.dx, toPoint.dx), my = Min (fromPoint.dy,
```

```
                                              toPoint.dy),
    Mx = Max (fromPoint.dx, toPoint.dx), My = Max (fromPoint.dy,
                                              toPoint.dy));
    DevicePoint orig, ex;

    orig = NewDevicePoint (mx, my);
    ex = NewDevicePoint (Mx - mx + 1, My - my + 1);

    NeedAux (h);
    BitBlt ((struct hardware *) NULL, h->aux, orig, orig, ex, ROP_FALSE);

    BitBltLine (h->aux, fromPoint, toPoint, ROP_TRUE);

    BitBlt (h->clip, h->aux, orig, orig, ex, ROP_AND);
    BitBlt (h->aux, h, orig, orig, ex, ROP_NOTAND);

    if (col != 0 && col != TransferSize () - 1) {
      struct hardware *gray = GraySync (col);
      RasterTile (gray, h->aux, orig, ex, ROP_AND);
    }
    BitBlt (h->aux, h, orig, orig, ex, ROP_OR);
  }
  UpdateControl (h, TRUE);

  h->touched = TRUE;
  h->changed = TRUE;
}

/*
 * BitBltBlob () takes a set of pixel coordinates and fills the trapezon
 * figure half open.
 */

void BitBltBlob (to, top, height, left, right, rop)
     struct hardware *to;
     int top, height, *left, *right, rop;
{
  int i;
  DevicePoint p1, p2;
  for (i=0 ; i<height ; i++) {
    p1 = NewDevicePoint (left[i], top + i);
    p2 = NewDevicePoint (right[i], top + i);
    BitBltLine(to, p1, p2, rop);
  }
}

/*
 * RasterTile () replicates the whole of ‘‘from’’ over ‘‘to’’, but clipped by
 * the rectangle bounded by ‘‘toPoint’’ and ‘‘extent’’.
 */
void RasterTile (from, to, toPoint, extent, rop)
     struct hardware *from, *to;
     DevicePoint toPoint, extent;
     int rop;
{
  Hardware fromhard, tohard;
  static GC wgc = NULL, bgc = NULL;
  XGCValues values;
  int valuemask;
  if (to == NULL) return;
  if (from == NULL || IsWindowHardware(from))
    Punt("Can only RasterTile from Bitmap.");

  fromhard = (Hardware) from->hard.addr;
  tohard = (Hardware) to->hard.addr;
  values.tile = fromhard->w;
  values.fill_style = FillTiled;
  values.function = rop;
  valuemask = GCFunction | GCTile | GCFillStyle;
  if (IsWindowHardware (to)) {
    if (wgc == NULL) {
      wgc = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                      valuemask, &values);
    } else {
      XChangeGC(dpy, wgc, valuemask, &values);
    }
    XFillRectangle(dpy, tohard->w, wgc, toPoint.dx, toPoint.dy,
                   extent.dx, extent.dy);
  } else
    if (IsBitmapHardware(to)){
      if (bgc == NULL) {
        bgc = XCreateGC(dpy, tohard->w, valuemask, &values);
      } else {
        XChangeGC(dpy, bgc, valuemask, &values);
      }
      XFillRectangle(dpy, tohard->w, bgc, toPoint.dx, toPoint.dy,
                     extent.dx, extent.dy);
    }
  to->touched = TRUE;
  to->changed = TRUE;
}

/******************* BRIGHTNESS TRANSFER FUNCTION ************************/
/*
 *
 * TransferSize () and SetTransfer () control the mapping function between
 * user brightnesses and device brightnesses. The interface is expected to
 * perform this mapping of brightnesses to a sufficient resolution.
 * SetTransfer takes a table of floating point numbers between 0 and 1. User
 * brightnesses are scaled to the size of this table and mapped through it.
 * The argument table given to SetTransfer () will be deleted after use.
 * TransferSize () simply enquires the required size of the table.
 *
 * It may be appropriate to half-tone on a grayscale or colour device to
 * improve rendering if it is not too expensive. TransferSize () returns the
 * size of the pattern table.
 */

void ModifyGC (color, src, dest, fromPoint, toPoint, extent)
     Colour color;
     struct hardware *src, *dest;
     DevicePoint fromPoint, toPoint, extent;
{
  int bright, ret;
  Hardware hard, grayhard, cliphard, temphard;
  struct hardware *halftone;
  unsigned long pixelvalue = BlackPixel(dpy, SCREEN);
  Pixmap gray;
  XGCValues values;
  int valuemask = 0;
  Drawable root = RootWindow(dpy, SCREEN);

  /* Handles only monochrome for now.  Fix for color later. */

  /*
   * This routine should take a color, a destination, and a source
   * hardware.  Simplest case:
   * Solid color, No clip, No from.
```



```
   *
   * Worst case:
   * Halftone, to clip, from non-null.
   */

  bright = transfer[(int)((TRANSFER_SIZE - 1) * color.brightness + .5)];

  if (bright == 0) {
    pixelvalue = BlackPixel(dpy, SCREEN);
    ret = ISPIXEL;
  } else if (bright == TRANSFER_SIZE - 1) {
    pixelvalue = WhitePixel(dpy, SCREEN);
    ret = ISPIXEL;
  } else {
    halftone = GraySync(bright);
    grayhard = (Hardware) halftone->hard.addr;
    ret = ISHALFTONE;
  }

  if (IsBitmapHardware(dest)) {
    root = ((Hardware) dest->hard.addr)->w;
    values.function = GXcopy;
    values.foreground = 1;
    values.background = 0;
    values.fill_style = FillSolid;
    values.clip_mask = None;
    valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCClipMask;
    if (ret == ISHALFTONE) {
      values.fill_style = FillTiled;
      values.tile = grayhard->w;
      values.ts_x_origin = 0;
      values.ts_y_origin = 0;
      valuemask = valuemask|GCFillStyle|GCTile|GCTileStipXOrigin|GCTileStipYOrigin;
    }

    if (src) {
      hard = (Hardware) src->hard.addr;
      if (dest->clip) {
        cliphard = (Hardware) dest->clip->hard.addr;
        if (ret == ISHALFTONE) {
          NeedAux(dest);
          BitBlt(dest->clip, dest->aux, toPoint,
                 NewDevicePoint(0, 0), extent, ROP_SOURCE);
          BitBlt(src, dest->aux, fromPoint,
                 NewDevicePoint(0, 0), extent, ROP_AND);

          hard = (Hardware) dest->aux->hard.addr;
          values.clip_mask = hard->w;
          values.clip_x_origin = toPoint.dx;
          values.clip_y_origin = toPoint.dy;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
        } else {
          values.clip_mask = cliphard->w;
          values.clip_x_origin = 0;
          values.clip_y_origin = 0;
          values.fill_style = FillStippled;
          values.stipple = hard->w;
          values.ts_x_origin = 0;
          values.ts_y_origin = 0;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin|GCFillStyle|GCStip
        }
      } else {
        if (ret == ISHALFTONE) {
          values.clip_mask = hard->w;
          values.clip_x_origin = toPoint.dx;
          values.clip_y_origin = toPoint.dy;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
        } else {
          /*
           * This should be fixed to use stipple, not
           * clip.
           *
           * values.fill_style = FillStippled;
           * values.stipple = hard->w;
           * valuemask = valuemask|GCFillStyle|GCStipple;
           * gc = MakeGC;
           */
          NeedAux(dest);
          BitBlt(src, dest->aux, fromPoint,
                 NewDevicePoint(0, 0),
                 extent, ROP_SOURCE);
          hard = (Hardware) dest->aux->hard.addr;
          values.clip_mask = hard->w;
          values.clip_x_origin = toPoint.dx;
          values.clip_y_origin = toPoint.dy;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
        }
      }
    } else { /* (!src) */
      if (dest->clip) {
        cliphard = (Hardware) dest->clip->hard.addr;
        values.clip_mask = cliphard->w;
        values.clip_x_origin = 0;
        values.clip_y_origin = 0;
        valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
      }
    }
  } else { /* (IsWindowHardware(dest)) */
    root = RootWindow (dpy, SCREEN);
    values.function = GXcopy;
    values.foreground = pixelvalue;
    values.background = WhitePixel (dpy, SCREEN);
    values.fill_style = FillSolid;
    values.clip_mask = None;
    valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCClipMask;
    if (ret == ISHALFTONE) {
      values.fill_style = FillOpaqueStippled;
      values.stipple = grayhard->w;
      values.ts_x_origin = 0;
      values.ts_y_origin = 0;
      valuemask = valuemask|GCFillStyle|GCStipple|GCTileStipXOrigin|GCTileStipYOrigin;
    }

    if (src) {
      hard = (Hardware) src->hard.addr;
      if (dest->clip) {
        cliphard = (Hardware) dest->clip->hard.addr;
        if (ret == ISHALFTONE) {
          NeedAux(dest);
          BitBlt(dest->clip, dest->aux, toPoint,
                 NewDevicePoint(0, 0), extent, ROP_SOURCE);
          BitBlt(src, dest->aux, fromPoint,
                 NewDevicePoint(0,0), extent, ROP_AND);

          hard = (Hardware) dest->aux->hard.addr;
          values.clip_mask = hard->w;
          values.clip_x_origin = toPoint.dx;
          values.clip_y_origin = toPoint.dy;
```

```
          values.fill_style = FillOpaqueStippled;
          values.stipple = grayhard->w;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin|GCFillStyle|GCStip
        } else { /* ISPIXEL */
          if (IsWindowHardware(src)) {
            values.clip_mask = cliphard->w;
            values.fill_style = FillTiled;
            values.tile = hard->w;
            valuemask = valuemask|GCClipMask|GCFillStyle|GCTile;
          } else {
            /* should also handle fromPoint here,
             * but this would have to be saved in
             * order for the cache to work properly.
             */
            values.clip_mask = cliphard->w;
            values.fill_style = FillStippled;
            values.stipple = hard->w;
            valuemask = valuemask|GCClipMask|GCFillStyle|GCStipple;
          }
        }
      } else { /* (!dest->clip) */
        if (ret == ISHALFTONE) {
          values.clip_mask = hard->w;
          values.clip_x_origin = 0;
          values.clip_y_origin = 0;
          valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
        } else { /* ISPIXEL */
          if (IsWindowHardware(src)) {
            hard = (Hardware) src->hard.addr;
            values.fill_style = FillTiled;
            values.tile = hard->w;
            values.ts_x_origin = 0;
            values.ts_y_origin = 0;
            valuemask = valuemask|GCFillStyle|GCTile|GCTileStipXOrigin|GCTileStipYOrigin;
          } else {
            values.fill_style = FillStippled;
            values.stipple = hard->w;
            values.ts_x_origin = toPoint.dx-fromPoint.dx;
            values.ts_y_origin = toPoint.dy-fromPoint.dy;
            valuemask = valuemask|GCFillStyle|GCStipple|GCTileStipXOrigin|GCTileStipYOrig
          }
        }
      }
    } else { /* (!src) */
      hard = (Hardware) dest->hard.addr;
      if (dest->clip) {
        cliphard = (Hardware) dest->clip->hard.addr;
        if (ret == ISHALFTONE) {
/*                      if ((hard->last_used != ISHALFTONE) ||
                            (hard->last_color.halftone != grayhard->w)) {*/
          values.clip_mask = cliphard->w;
          values.fill_style = FillOpaqueStippled;
          values.stipple = grayhard->w;
          valuemask = valuemask|GCClipMask|GCFillStyle|GCStipple;
/*                           hard->gc = MakeGC;
                             hard->last_used = ISHALFTONE;
                             hard->last_color.halftone = grayhard->w;
                        }
                        gc = hard->gc;
*/
        } else { /* ISPIXEL */
          values.clip_mask = cliphard->w;
          valuemask = valuemask|GCClipMask;
        }
      } else { /* (!dest->clip) */
        if (ret == ISHALFTONE) {
          values.fill_style = FillOpaqueStippled;
          values.stipple = grayhard->w;
          valuemask = valuemask|GCFillStyle|GCStipple;
/*                  } else {
                         if (pixelvalue == BlackPixel(dpy, SCREEN))
                              gc = fillgc[GXcopy];
                         else if (pixelvalue == WhitePixel(dpy, SCREEN))
                              gc = whitegc;
                         else
                              gc = MakeGC;
                         }
*/
        }
      }
    }
  }

  if (!GlobalGC) {
    GlobalGC = XCreateGC(dpy, root, valuemask, &values);
    GlobalMask = valuemask;
  } else {
    XChangeGC(dpy, GlobalGC, valuemask, &values);
    GlobalMask = valuemask;
  }
}

ColorType RealColor (color, pixelvalue, halftone)
     Colour color;
     unsigned long *pixelvalue;
     struct hardware **halftone;
{
  int bright;
  XGCValues values;

  bright = transfer[(int)((TRANSFER_SIZE - 1) * color.brightness + .5)];

  if (bright == 0) {
    *pixelvalue = BlackPixel(dpy, SCREEN);
    return ISPIXEL;
  } else if (bright == TRANSFER_SIZE - 1) {
    *pixelvalue = WhitePixel(dpy, SCREEN);
    if (!whitegc) {
      values.function = GXcopy;
      values.foreground = WhitePixel(dpy, SCREEN);
      values.background = BlackPixel(dpy, SCREEN);
      whitegc = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                          GCFunction|GCForeground|GCBackground,
                          &values);
    }
    return ISPIXEL;
  }

  *halftone = GraySync(bright);
  return ISHALFTONE;
}

struct hardware *GraySync (col)
     int col;
{
  col = col * (float) screen_size / TRANSFER_SIZE + 0.5;

  return screen[col].shade;
```



```
}

int HardColour (colour)
     Colour colour;
{
  /* DCJ:
   * This is a hack.  This is only a hack.  If this had been real
   * code, you would have been notified of it through email.
   */
  if (BlackPixel(dpy, SCREEN) == 0)
    return (TRANSFER_SIZE - 1 - transfer [(int) ((TRANSFER_SIZE - 1) * colour.brightness
  else
    return transfer [(int) ((TRANSFER_SIZE - 1) * colour.brightness + .5)];
}

int TransferSize ()
{
  return TRANSFER_SIZE;
}

#define TransferSize() TRANSFER_SIZE

void SetTransfer (tran)
     float *tran;
{
  int i;

  for (i = 0; i < TRANSFER_SIZE; i++)
    transfer [i] = (TRANSFER_SIZE - 1) * tran[i] + .5;
}

/********************** BITMAP CONVERSION ********************************/
/*
 *
 * StringFromHardware () produces a string from its argument which describes
 * the bitmap.  The bitmap is returned in row-major order with the leftmost
 * bit of each byte in the most significant position. Rows are padded to byte
 * boundaries. Only single plane bitmaps are used.
 *
 * HardwareFromString () performs the inverse mapping, generating a bitmap
 * from a set of bits, given a width and height. Only single plane bitmaps are
 * used.
 */

/*char *StringFromHardware (h)
struct hardware *h;
{
    XImage *image;
    Hardware hard;
    char *result, *ptr, c;
    int x, y, i, hdx, hdy;
    if (h == NULL) return NULL;
    hard = (Hardware) h->hard.addr;
    hdx = h->extent.dx;
    hdy = h->extent.dy;
    image = XGetImage(dpy, hard->w, 0, 0, hdx, hdy, AllPlanes, ZPixmap);
    result = Malloc(((hdx + 7) / 8) * hdy);
    ptr = result;
    for (y=0 ; y<hdy ; y++) {
        for (x=0 ; x<hdx ; x+=8) {
            c = 0;
            for (i=0 ; i<8 ; i++) {
                c = c << 1;
                if (x+i < hdx)
                    c |= XGetPixel(image, x+i, y);
            }
        }
        *ptr++ = c;
    }
    Free((char *) image);
    return (char *) result;
}

struct hardware *HardwareFromString (s, width, height)
char *s;
int width, height;
{
    struct hardware *h = NewBitmapHardware(width, height);
    Hardware hard = (Hardware) h->hard.addr;
    XImage *image;
    if (s == NULL) Punt("HardwareFromString called with NULL string!");
    image = XCreateImage(dpy, DefaultVisual(dpy, SCREEN),
                         DefaultDepth(dpy, SCREEN), ZPixmap, 0, s,
                         h->extent.dx, h->extent.dy, 8, 0);
    image->bitmap_bit_order = MSBFirst;
    XPutImage(dpy, hard->w, fillgc[GXcopy], image, 0, 0, 0, 0,
              h->extent.dx, h->extent.dy);
    free((char *) image);
    return h;
}
*/
unsigned char *StringFromHardware (h)
     struct hardware *h;
{
  XImage *image;
  Hardware hard;
  unsigned char *result, *ptr, c;
  int x, y, i, hdx, hdy;

  printf("StringFromHardware called.\n");

  if (h == NULL || !IsBitmapHardware(h)) return NULL;

  printf("and is getting the image.\n");

  hard = (Hardware) h->hard.addr;
  hdx = h->extent.dx;
  hdy = h->extent.dy;
  image = XGetImage(dpy, hard->w, 0, 0, hdx, hdy, AllPlanes, XYPixmap);
  result = (unsigned char *) Malloc(((hdx + 7) / 8) * hdy);
  ptr = result;
  for (y=0 ; y<hdy ; y++) {
    for (x=0 ; x<hdx ; x+=8) {
      c = 0;
      for (i=0 ; i<8 ; i++) {
        c = c << 1;
        if (x+i < hdx)
          c |= XGetPixel(image, x+i, y);
      }
    }
    *ptr++ = c;
  }
  Free((char *) image);
  return result;
}

struct hardware *HardwareFromString (s, width, height)
     unsigned char *s;
```

```
     int width, height;
{
  struct hardware *h = NewBitmapHardware(width, height);
  Hardware hard = (Hardware) h->hard.addr;
  XImage *image;

  printf("HardwareFromString called.\n");

  if (s == NULL) Punt("HardwareFromString called with NULL string!");
  image = XCreateImage(dpy, DefaultVisual(dpy, SCREEN),
                       1, XYBitmap, 0, s,
                       h->extent.dx, h->extent.dy, 8, 0);
  image->bitmap_bit_order = MSBFirst;
  XPutImage(dpy, hard->w, BitmapGC[GXcopy], image, 0, 0, 0, 0,
            h->extent.dx, h->extent.dy);
  free((char *) image);
  h->touched = TRUE;
  h->changed = TRUE;
  return h;
}

/************************* HALF-TONE SCREEN *******************************/
/*
 * ScreenSize () allows PostScript to determine how large an array of sample
 * points to expect.  It should return the length of the side of the sample
 * square.
 *
 * BuildScreen () returns a set of sampling coordinates to PostScript to hand
 * to the users spot-function
 *
 * SetScreen () allows PostScript to set the thresholds for each sample point
 * so that half-tone bitmaps can be made.
 */

static int FreqSize (freq)
     float freq;
{
  int i = pixels_per_inch / freq + 0.5;

  if (i < 2)
    return 2;
  return i;
}

int ScreenSize (freq, rot)
     float freq, rot;
{
  int size = FreqSize (freq);

  return size * size;
}

void BuildScreen (freq, rotation, x, y)
     float freq, rotation, *x, *y;
{
  int size = FreqSize (freq);
  int i, j;

  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      *x++ = (2 * i - size + 1) / (float) size,
      *y++ = (2 * j - size + 1) / (float) size;
}
```

```
#ifdef notdef
static int sgn (a)
     float a;
{
  if (a == 0)
    return 0;
  else if (a < 0)
    return -1;
  else
    return 1;
}

static int screen_cmp (a, b)
     char *a, *b;
{
  struct screen *aa = (struct screen *) a, *bb = (struct screen *) b;
  static int count = 0;

  return sgn (aa->val - bb->val);
}
#endif

static int screen_cmp (a, b)
     struct screen *a, *b;
{
  float diff = a->val - b->val;
  if(diff > 0.0) return 1;
  if(diff < 0.0) return -1;
  return 0;
}

void SetScreen (freq, rotation, thresh)
     float freq, rotation, *thresh;
{
  struct hardware *temp;
  int i, j, size = FreqSize (freq);
  struct screen *p;
  int valuemask;
  XGCValues values;
  Hardware hard;

  if (screen) {
    for (i = 0; i < screen_size; i++)
      DestroyHardware (screen [i].shade);
    free ((char *) screen);
  }
  p = screen = (struct screen *) Malloc ((unsigned) (((screen_size = size * size) + 1) *
/*     screen_side = size;*/
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++) {
      p->val = *thresh++;
      p->sx = i;
      p->sy = j;
      ++p;
    }
  qsort ((char *) screen, screen_size, sizeof (struct screen), screen_cmp);
  temp = NewBitmapHardware (size, size);
  BitBlt ((struct hardware *) NULL, temp, NewDevicePoint (0, 0),
          NewDevicePoint (0, 0), NewDevicePoint (size, size), ROP_TRUE);

  for (i = 0; i < screen_size; i++) {
    screen [i].shade = NewBitmapHardware (size, size);
    BitBlt (temp, screen[i].shade,
            NewDevicePoint (0, 0), NewDevicePoint (0, 0),
```



```
            NewDevicePoint (size, size), ROP_SOURCE);
    BitBlt ((struct hardware *) NULL, temp,
            NewDevicePoint (0, 0),
            NewDevicePoint (screen[i].sx, screen[i].sy),
            NewDevicePoint (1, 1), ROP_FALSE);
    values.function = GXcopy;
    values.foreground = BlackPixel(dpy, SCREEN);
    values.background = WhitePixel(dpy, SCREEN);
    values.fill_style = FillOpaqueStippled;
    hard = (Hardware) screen[i].shade->hard.addr;
    values.stipple = hard->w;
    valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCStipple;
    hard->gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask,
                         &values);
    screen[i].shade->touched = TRUE;
    screen[i].shade->changed = TRUE;
  }
  screen[screen_size].shade = temp;
  values.function = GXcopy;
  values.foreground = BlackPixel(dpy, SCREEN);
  values.background = WhitePixel(dpy, SCREEN);
  values.fill_style = FillOpaqueStippled;
  hard = (Hardware) temp->hard.addr;
  values.stipple = hard->w;
  valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCStipple;
  hard->gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask,
                       &values);
  screen[screen_size].shade->touched = TRUE;
  screen[screen_size].shade->changed = TRUE;
}

/************************* CLIPPING ******************************************/
/*
 *
 * SetClipHardware sets hardware which is a clip mask for BitBlt. This mask
 * should be ANDed with any output operation. If clip is NULL, masking will
 * not be needed.
 */

void SetClipHardware (h, clip)
     struct hardware *h, *clip;
{
  if (h) {
    h->clip = clip;
    if (h->clip)
      h->clip->changed = TRUE;
  }
}

/************************ UPDATE CONTROLS **********************************/
/*
 * HardUpdate is a hook to allow devices which do output buffering to flush
 * that buffering when appropriate.  This allows an interactive user to see
 * completed graphics between prompts (it is called as a side-effect of the
 * PostScript flush operator). Typically is is a no-op.
 */

void HardUpdate ()
{
  XFlush(dpy, 0);
}

/*
 * This call can be used to enable batching of output operations.
```

```
 * UpdateControl (h, FALSE) means ‘‘start of batching’’ UpdateControl (h,
 * TRUE) means ‘‘end of batching’’. It is used to improve performance on
 * machines where screen updates have a high locking overhead. It may be a
 * no-op.  The operation should nest if batching is already in progress: FALSE
 * increments a counter, TRUE decrements a counter. Display changes are
 * allowed when the counter is non-zero.
 */

#ifndef UpdateControl
void UpdateControl (h, on)
     struct hardware *h;
     int on;
{}
#endif

DevicePoint ConvertToDevicePoint (x, y)
     int x, y;
{
  DevicePoint res;

  res.dx = (int) ((float) x * (float) pixels_per_inch / 72.0);
  res.dy = (int) ((float) y * (float) pixels_per_inch / 72.0);

  return res;
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static Object OpForAll;

static int PLength (), PArray (), PRBracket (), PAload (), PAstore ();

static int forAll (), ForAll (), PReadOnly (), PExecOnly (), PrCheck ();
static int PwCheck (), PutInterval (), GetInterval (), Put (), Get ();
static int Length (), Copy (), Eq ();

int ExecArray ();
Object Marker;

static int autobind = 0;

static int PSetAutoBind (), PGetAutoBind ();

InitArray ()
{
  TypeInstallOp (Mark,  "eq", EqTrue, 2, 1, 0, 0, Mark, Mark);

  MakeObject (Marker, Mark);

  OpForAll = MakeOp ("(forallarray)", forAll, 0, 0, 3, 5);

  Install ("[",         Marker);
  Install ("null",      Nil);

  InstallOp ("array",   PArray,         1, 1, 0, 0, Integer);
  InstallOp ("]",       PRBracket,      0, 1, 0, 0);
  InstallOp ("aload",   PAload,         1, 0, 0, 0, Array);
  InstallOp ("astore",  PAstore,        1, 1, 0, 0, Array);

  InstallOp ("setautobind",     PSetAutoBind,   1, 0, 0, 0, Bool);
  InstallOp ("getautobind",     PGetAutoBind,   0, 1, 0, 0);

  TypeInstallOp (Array, "exec",         ExecArray,      1, 1, 0, 2, Array);
  TypeInstallOp (Array, "eq",           Eq,             2, 1, 0, 0, Array,
                Array);
  TypeInstallOp (Array, "put",          Put,            3, 0, 0, 0, Array,
                Integer, Poly);
  TypeInstallOp (Array, "get",          Get,            2, 1, 0, 0, Array,
                Integer);
  TypeInstallOp (Array, "putinterval",  PutInterval,    3, 0, 0, 0, Array,
                Integer, Array);
  TypeInstallOp (Array, "getinterval",  GetInterval,    3, 1, 0, 0, Array,
                Integer, Integer);
  TypeInstallOp (Array, "length",       PLength,        1, 1, 0, 0, Array);
  TypeInstallOp (Array, "copy",         Copy,           2, 0, 0, 0, Array,
                Array);
  TypeInstallOp (Array, "forall",       ForAll,         2, 0, 0, 4, Array,
                Array);
  TypeInstallOp (Array, "executeonly",  PExecOnly,      1, 1, 0, 0, Array);
  TypeInstallOp (Array, "readonly",     PReadOnly,      1, 1, 0, 0, Array);
  TypeInstallOp (Array, "rcheck",       PrCheck,        1, 1, 0, 0, Array);
  TypeInstallOp (Array, "wcheck",       PwCheck,        1, 1, 0, 0, Array);
}


Object MakeArray (p, length)
     Object *p;
     int length;
{
  Object res;

  MakeObject (res, Array);
  res.u.Array = p;
  res.Length = length;

  return res;
}


static Object Make (p, length)
     Object *p;
     int length;
{
  Object res;

  MakeObject (res, Array);
  res.u.Array = p;
  res.Length = length;

  return res;
}

#define Body(a) ((a).u.Array)

#ifndef Body
static Object *Body (item)
     Object item;
{
  return item.u.Array;
}
#endif

static Object ArrayStore (n)
     int n;
{
  Object *body = (Object *) Malloc ((unsigned) sizeof (Object) * n);
  int i, h = Height (OpStack);

  for (i = h - n; i < h; i++)
    body[i-h+n] = OpStack->stack_body[i];
  OpStack->stack_fill -= n;
  return Make (body, n);
}

/*VARARGS
Object ArrayFrom (length, args)
 int length; Object args;
 {
        Object *argv = &args, *array = (Object *) Malloc ((unsigned) (sizeof (Object) * l
        int i;
```



```
        for (i = 0; i < length; i++)
                array[i] = argv[i];
        return Cvx (Make (array, length));
 }
*/

static int PLength (object)
     Object object;
{
  return Push (OpStack, MakeInteger (lengthArray(object)));
}

int ExecArray (item)
     Object item;
{
  int l = lengthArray (item);
  register Object *rp;
  Type t;

  if (l == 0) return TRUE;

  /*
   * BZS - try some open-coding here
   * TMD - try some real open-coding here...
   Push (ExecStack, SameFlags (item, Make (Body (item) + 1, l - 1)));
   */
  if(ExecStack->stack_fill != ExecStack->stack_size) {
    rp = &ExecStack->stack_body[ExecStack->stack_fill++];
    rp->type = Array;
    rp->u.Integer = 0;
    rp->u.Array = item.u.Array + 1;
    rp->Length = l - 1;
    rp->flags = item.flags;
  }

  rp = Body(item);
  t = TypeOf(*rp);
  if(t == Name || t == Operator) Push (ExecStack, *rp);
  else                           Push (OpStack, *rp);

  return TRUE;
}

static int Get (object, key)
     Object object, key;
{
  int index;

  if (TypeOf (key) == Integer)
    index = BodyInteger (key);
  else if (TypeOf (key) == Real)
    index = (int) BodyReal (key);
  else
    return Error (PTypeCheck);
  if (index >= 0 && index < lengthArray(object)) {
    Object t;

    t = Body (object) [index]; /* VAX Compiler Broken */
    return Push (OpStack, t);
  } else
    return Error (PRangeCheck);
}

Object getArray (o, index)
     Object o;
     int index;
{
  return Body (o)[index];
}

static int GetInterval (object, begin, length)
     Object object, begin, length;
{
  int b = BodyInteger (begin), l = BodyInteger (length);

  if (l < 0 || b < 0 || b + l > lengthArray(object))
    return Error (PRangeCheck);
  return Push (OpStack, Make (Body (object) + b, l));
}

Object getIArray (object, begin, length)
     Object object;
     int begin, length;
{
  return Make (Body (object) + begin, length);
}

static int Put (object, key, value)
     Object object, key, value;
{
  int index = BodyInteger (key);

  if (index < 0 || index >= lengthArray(object))
    return Error (PRangeCheck);
  Body (object) [index] = value;
  return TRUE;
}

int putArray (object, index, value)
     Object object, value;
     int index;
{
  if (index < 0 || index >= lengthArray(object))
    return Error (PRangeCheck);
  Body (object) [index] = value;
  return TRUE;
}

static int PutInterval (object1, begin, object2)
     Object object1, begin, object2;
{
  int i, l1 = lengthArray(object1), l2 = lengthArray (object2),
  b = BodyInteger (begin);

  if (b < 0 || l2 + b > l1)
    return Error (PRangeCheck);
  for (i = b; i < b + l2; i++)
    Body (object1) [i] = Body (object2) [i - b];
  return TRUE;
}

static int Copy (object1, object2)
     Object object1, object2;
{
  if (lengthArray (object1) <= lengthArray (object2))
    return PutInterval (object2, MakeInteger (0), object1)
      && Push (OpStack, Make (Body (object2),
                              lengthArray (object1)));
```

```c
  else
    return Error (PRangeCheck);
}

static Eq (a, b)
     Object a, b;
{
  return Push (OpStack,
               MakeBool (lengthArray (a) == lengthArray (b) &&
                            (lengthArray (a) == 0 || Body (a) == Body (b))));
}

static int ForAll (array, proc)
     Object array, proc;
{
  return Push (ExecStack, array)
    && Push (ExecStack, MakeInteger (0))
      && Push (ExecStack, proc)
        && Push (ExecStack, OpForAll);
}

static forAll ()
{
  Object array, index, proc;

  proc  = Pop (ExecStack);
  index = Pop (ExecStack);
  array = Pop (ExecStack);

  if (BodyInteger (index) >= lengthArray (array))
    return TRUE;
  return Push (ExecStack, array)
    && Push (ExecStack, MakeInteger (BodyInteger (index) + 1))
      && Push (ExecStack, proc)
        && Push (ExecStack, OpForAll)
          && Push (ExecStack, proc)
            && Push (OpStack, Body (array) [BodyInteger (index)]);
}

static int PSetAutoBind (bool)
     Object bool;
{
  autobind = BodyBool (bool);

  return TRUE;
}

static int PGetAutoBind ()
{
  return Push (OpStack, MakeBool (autobind));
}

Object ParseArray (o)
     Object o;
{
  Object res, *array;
  int i, length = 0;
  struct acc
    {
      Object item;
      struct acc *prev;
    } *p = NULL;

  for (;;) {
    struct acc *item = (struct acc *) Malloc (sizeof (struct acc));
    res = Parse (o);

    if (TypeOf (res) == Condition)
      return Absent;
    if (TypeOf (res) == Bool)
      if (BodyBool (res))
        break;
      else
        return Absent;
    else if (TypeOf (res) == Null)
      return res;

    if (autobind && TypeOf (res) == Name && xCheck (res)) {
      Object binding;

      binding = Load (res);

      if (TypeOf (binding) == Operator)
        res = binding;
    }
    item->item = res;
    item->prev = p;
    p = item;
    ++length;
  }
  array = (Object *) Malloc ((unsigned) sizeof (Object) * length);

  for (i = length - 1; i >= 0; i--) {
    struct acc *prev = p->prev;

    array[i] = p->item;
    Free ((char *)p);
    p = prev;
  }
  PanicIf (p != NULL, "Something badly wrong in Array literal parser\n");
  return Make (array, length);
}

static int PArray (size)
     Object size;
{
  if (BodyInteger (size) < 0)
    return Error (PRangeCheck);
  else
    {
      int i, s = BodyInteger (size);
      Object *body = (Object *) Malloc ((unsigned) sizeof (Object) * s);

      for (i = 0; i < s; i++)
        body[i] = Nil;
      return Push (OpStack, Make (body, s));
    }
}

static int PRBracket () /* [ a1 ... an --- array */
{
  int n = CountTo (Mark, OpStack);

  if (n < 0)
    return Error (PUnMatched);
  else {
    Object res;
```



```c
    res = ArrayStore (n);
    VOID Pop (OpStack);
    return Push (OpStack, res);
  }
}

static int PAload (array)
     Object array;
{
  Object *body = Body (array);
  int i, l = lengthArray (array);

  if (MaxStack (OpStack) - Height (OpStack) < l + 1)
    return Error (POpOverflow);
  else {
    for (i = 0; i < l; i++)
      if (!Push (OpStack, body[i]))
        break;
    return Push (OpStack, array);
  }
}

static int PAstore (array)
     Object array;
{
  if (lengthArray (array) > Height (OpStack))
    return Error (POpUnderflow);
  else {
    Object *body = Body (array);
    int i, l = lengthArray (array);

    for (i = l-1; i >= 0; i--)
      body[i] = Pop (OpStack);
    return Push (OpStack, array);
  }
}

static int PExecOnly (item)
     Object item;
{
  return Push (OpStack, ExecOnly (item));
}

static int PReadOnly (item)
     Object item;
{
  return Push (OpStack, ReadOnly (item));
}

static int PrCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (rCheck (v)));
}

static int PwCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (wCheck (v)));
}
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

Object True, False;
static Object OpNot;
static int Cvs (), And (), Or (), Not (), Xor ();
static int PEq (), PNeq (), EqBool ();

InitBool ()
{
  MakeObject (True, Bool);
  MakeObject (False, Bool);

  True.u.Bool = TRUE;
  False.u.Bool = FALSE;

  Install ("true",      True);
  Install ("false",     False);

  TypeInstallOp (Bool, "cvs",   Cvs,            2, 1, 0, 0, Bool, String);
  TypeInstallOp (Bool, "and",   And,            2, 1, 0, 0, Bool, Bool);
  TypeInstallOp (Bool, "or",    Or,             2, 1, 0, 0, Bool, Bool);
  TypeInstallOp (Bool, "xor",   Xor,            2, 1, 0, 0, Bool, Bool);
  TypeInstallOp (Bool, "not",   Not,            1, 1, 0, 0, Bool);
  TypeInstallOp (Bool, "eq",    EqBool,         2, 1, 0, 0, Bool, Bool);

  OpNot = Lookup (Bool, NameFrom ("not"));

  InstallOp ("eq",      PEq,            2, 1, 0, 0, Poly, Poly);
  InstallOp ("ne",      PNeq,           2, 1, 0, 0, Poly, Poly);
  InstallOp ("ge",      PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("gt",      PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("le",      PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("lt",      PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("not",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("and",     PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("or",      PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("xor",     PolyPair,       2, 2, 0, 0, Poly, Poly);
  InstallOp ("bitshift",        PolyPair,       2, 2, 0, 0, Poly, Poly);
}

Object MakeBool (b)
     int b;
{
  if (b)
    return True;
  else
    return False;
}

int BodyBool (b)
     Object b;
{
  return b.u.Bool;
}

static int Cvs (v, string)
     Object v, string;
{
  char *choice = BodyBool (v) ? "true" : "false";
  int length = strlen (choice);

  if (lengthString (string) < length)
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), choice, length);
  return Push (OpStack, getIString (string, 0, length));
}

static int PEq (a, b)
     Object a, b;       /* any any --- Bool */
{
  if (!rCheck (a) || !rCheck (b))
    return Error (PInvAccess);
  if (TypeOf (a) == Name && TypeOf (b) == String)
    a = StringName (a);
  else if (TypeOf (b) == Name && TypeOf (a) == String)
    b = StringName (b);
  else if (TypeOf (a) == Real && TypeOf (b) == Integer)
    b = RealInteger (b);
  else if (TypeOf (a) == Integer && TypeOf (b) == Real)
    a = RealInteger (a);
  if (TypeOf (a) == TypeOf (b)) {
    VOID Push (OpStack, a);
    VOID Push (OpStack, b);
    return Apply (TypeOf (a));
  }
  else
    return Push (OpStack, False);
}

static int PNeq (a, b)
     Object a, b;               /* any any --- Bool */
{
  if (!rCheck (a) || !rCheck (b))
    return Error (PInvAccess);
  if (TypeOf (a) == Name && TypeOf (b) == String)
    a = StringName (a);
  else if (TypeOf (b) == Name && TypeOf (a) == String)
    b = StringName (b);
  else if (TypeOf (a) == Real && TypeOf (b) == Integer)
    b = RealInteger (b);
  else if (TypeOf (a) == Integer && TypeOf (b) == Real)
    a = RealInteger (a);
  if (TypeOf (a) == TypeOf (b)) {
    VOID Push (OpStack, a);
    VOID Push (OpStack, b);
    VOID Push (ExecStack, OpNot);
    Self = NameFrom ("eq");
    return Apply (TypeOf (a));
  } else
    return Push (OpStack, True);
}

static int EqBool (a, b)
     Object a, b;
{
```



```c
  return Push (OpStack, MakeBool (BodyBool (a) == BodyBool (b)));
}

static int Not (bool)
     Object bool;
{
  return Push (OpStack, MakeBool (!BodyBool (bool)));
}

static int And (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (BodyBool (a) && BodyBool (b)));
}

static int Or (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (BodyBool (a) || BodyBool (b)));
}

static int Xor (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (BodyBool (a) != BodyBool (b)));
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

static struct cache
{
  struct cache *cache_next, *cache_prev;
  int fid;
  Matrix mat;                           /* character -> device */
  int width, height, swidth, sheight;   /* device coords */
} cache_header = { &cache_header, &cache_header },
  *cache_ring = &cache_header;

static struct char_table {
  Object                char_key;
  struct cache  *char_cache;

  struct device         *char_device;
  UserPoint             char_width;                 /* character coords */
} *CharTable;

static int bsize = 0, bmax = 0;
static int msize = 0, mmax = 0;
static int csize = 0, cmax = 0;

static int maxbits = 0;

static int CacheStatus ();
static int PSetCharWidth ();
static int SetCacheLimit ();
static int PCurrentCharWidth ();
static int PSetCacheDevice ();
static int SetCache ();
static int SaveCurrentFont ();
static DestroyCache ();
static CacheInsert ();


int SetCacheDevice ();
static struct cache *NewCache (), *CacheRemove ();

InitCache ()
{
  InstallOp ("cachestatus",     CacheStatus,            0, 7, 0, 0);
  InstallOp ("setcachedevice",  PSetCacheDevice,        6, 0, 0, 0,
             Float, Float, Float, Float, Float, Float);
  InstallOp ("setcharwidth",    PSetCharWidth,          2, 0, 0, 0,
             Float, Float);
  InstallOp (".currentcharwidth",       PCurrentCharWidth,      0, 2, 0, 0);
  InstallOp ("setcachelimit",   SetCacheLimit,          1, 0, 0, 0,
             Integer);
  InstallOp ("setcache",                SetCache,               3, 0, 0, 0,
             Integer, Integer, Integer);
  InstallOp ("savecurrentfont", SaveCurrentFont,        0, 0, 0, 0);
}

static InitCharTable ()
{
  int i;

  CharTable = (struct char_table *) Malloc ((unsigned) (sizeof (struct char_table) * cmax
  for (i = 0; i < cmax; i++)
    CharTable[i].char_cache = NULL;
}

static struct char_table *CharAddition (table, cache, key, size)
     struct char_table *table;
     struct cache *cache;
     Object key;
     int size;
{
  int i, h = (BodyInteger (key) + (int) cache) % size;

  for (i = h;;) {
    if (table[i].char_cache == NULL) {
      table[i].char_key = key;
      table[i].char_cache = cache;

      return &table[i];
    }
    if (++i == size)
      i = 0;
    if (i == h)
      return NULL;
  }
}

CharStore (key, width, dev)
     Object key;
     UserPoint width;
     struct device *dev;
{
  struct char_table *p;

  while (csize == cmax)
    DestroyCache (CacheRemove (cache_ring->cache_prev));

  if (p = CharAddition (CharTable, gstate->show->ccache, key, cmax)) {
    p->char_width = width;
    p->char_device = dev;
    ++csize;
  } else
    Panic ("dictionary full in CharStore");
}

static struct char_table *CharFind (table, cache, key, size)
     struct char_table *table;
     struct cache *cache;
     Object key;
     int size;
{
  int i, h = (BodyInteger (key) + (int) cache) % size;

  for (i = h;;) {
    if (table[i].char_cache == NULL) return NULL;
    if (table[i].char_cache == cache && DictEqual (key, table[i].char_key))
```



```
      return &table[i];
    if (++i == size) i = 0;
    if (i == h)  break;
  }
  return NULL;
}

static FlushChars (p)
     struct cache *p;
{
  int i;

  for (i = 0; i < cmax; i++)
    if (CharTable[i].char_cache == p) {
      CharTable[i].char_cache = NULL;
      --csize;
    }
}

#ifdef notdef
/*
 * Converts a bounding box to equivalent one in device coordinates.
 */
DeviceBBox (left, right, top, bottom, llx, urx, ury, lly)
     float *left, *right, *top, *bottom, llx, urx, ury, lly;
{
  HardPoint ll;

  ll = ExtToInt (NewUserPoint (llx, lly));

  *left = *right = ll.hx; *top = *bottom = ll.hy;

  Bound (left, right, top, bottom, ExtToInt (NewUserPoint (urx, lly)));
  Bound (left, right, top, bottom, ExtToInt (NewUserPoint (llx, ury)));
  Bound (left, right, top, bottom, ExtToInt (NewUserPoint (urx, ury)));
}
#endif

/*
 * Efficiently converts a bounding box to equivalent one in device coordinates.
 */
DeviceBBox (left, right, top, bottom, llx, urx, ury, lly)
     float *left, *right, *top, *bottom, llx, urx, ury, lly;
{
  register float a,b,c,d,x1,y1,x2,y2;

  /* Compute left and right from x values of transformed points */
  a = gstate->CTM.A; c = gstate->CTM.C;
  x1 = llx*a + lly*c;
  x2 = urx*a + lly*c;
  if(x1 < x2) { *left = x1; *right = x2; }
  else        { *left = x2; *right = x1; }

  x1 = llx*a + ury*c;
  x2 = urx*a + ury*c;
  if(x2 < x1) { y1 = x1; x1 = x2; x2 = y1; }
  if(*left > x1) *left = x1;
  if(*right < x2) *right = x2;

  /* Compute top and bottom from y values of transformed points */
  b = gstate->CTM.B; d = gstate->CTM.D;
  y1 = llx*b + lly*d;
  y2 = urx*b + lly*d;
  if(y1 < y2) { *bottom = y1; *top = y2; }
  else        { *bottom = y2; *top = y1; }

  y1 = llx*b + ury*d;
  y2 = urx*b + ury*d;
  if(y2 < y1) { x1 = y1; y1 = y2; y2 = x1; }
  if(*bottom > y1) *bottom = y1;
  if(*top < y2) *top = y2;

  /* Translate to get final bounding box location */
  x1 = gstate->CTM.tx;  y1 = gstate->CTM.ty;
  *left += x1;   *right += x1;   *bottom += y1;   *top += y1;
}

int EqVector (a, b)
     Vector a, b;
{
  return (int) a.vx == (int) b.vx && (int) a.vy == (int) b.vy;
}

static int PCurrentCharWidth ()
{
  VOID Push (OpStack, MakeReal (gstate->show->Width.x));
  VOID Push (OpStack, MakeReal (gstate->show->Width.y));
  return TRUE;
}

static int PSetCharWidth (wx, wy)
     Object wx, wy;
{
  if (!gstate->show->InShow) return Error (PUndefined);
  gstate->show->Width.x = BodyReal(wx);
  gstate->show->Width.y = BodyReal(wy);
  return TRUE;
}

static int PSetCacheDevice (wx, wy, llx, lly, urx, ury)
     Object wx, wy, llx, lly, urx, ury;
{
  if (!gstate->show->InShow)
    return Error (PUndefined);
  return SetCacheDevice (gstate->show->CharName,
                         NewUserPoint (BodyReal (wx), BodyReal (wy)),
                         BodyReal (llx), BodyReal (lly),
                         BodyReal (urx), BodyReal (ury));
}

int SetCacheDevice (name, char_width, llx, lly, urx, ury)
     Object name;
     UserPoint char_width;
     float llx, lly, urx, ury;
{
  float left, right, top, bottom, width, height;
  struct device *new_char;
  struct cache *ccache;
  HardPoint P;

  gstate->show->Width = char_width;
  if (!(ccache = gstate->show->ccache)) return TRUE;

  DeviceBBox (&left, &right, &top, &bottom, llx, urx, ury, lly);
  width = right - left; height = top - bottom;

  /* Check if character is too big */
  if ((int) (width * height) > maxbits * 8) return TRUE;
```

```c
  new_char = NewCacheDevice (ccache->mat, ccache->width, ccache->height,
                             ccache->swidth, ccache->sheight);
  CharStore (name, char_width, LinkDevice (new_char));
  SetDevice (new_char);

  PathFree (gstate->clip);
  gstate->clip = NewClipPath (0.0, width, height, 0.0);
  P.hx = gstate->CTM.tx = ccache->swidth;
  P.hy = gstate->CTM.ty = ccache->sheight;
  VOID MoveTo (gstate->path, P);

  return TRUE;
}

printMatrix (m)
     Matrix m;
{
  printf ("[%g %g]\n", m.A, m.B);
  printf ("[%g %g]\n", m.C, m.D);
  printf ("[%g %g]\n", m.tx, m.ty);
}

extern float fontbboxCache[];
extern int fidCache;

struct cache *SearchCache (m, fid)
     Matrix m;
     int fid;
{
  struct cache *p;
  register float a,b,c,d;

  a = m.A;  b = m.B;  c = m.C;  d = m.D;
  for (p = cache_ring->cache_next; p != cache_ring; p = p->cache_next)
    if (p->fid == fid && p->mat.A == a && p->mat.B == b &&
        p->mat.C == c && p->mat.D == d) {
      if (cache_ring->cache_next != p)
        CacheInsert (cache_ring->cache_next,CacheRemove (p));
      return cache_ring->cache_next;
    }

  return NULL;
}

int FindCache ()
{
  float left, right, top, bottom;
  int width, height, swidth, sheight;

  if ((gstate->show->ccache = SearchCache (gstate->CTM,
                                           fidCache)) != NULL)
    return TRUE;
  Message ("building a new cache");
  DeviceBBox (&left, &right, &top, &bottom,
              fontbboxCache[BBOX_LEFT], fontbboxCache[BBOX_RIGHT],
              fontbboxCache[BBOX_BOTTOM], fontbboxCache[BBOX_TOP]);

  width = (int) (right - left); height = (int) (top - bottom);
  if (width*height > maxbits * 8) {
    Message ("character too big to cache");
    return TRUE;
  }
  if (msize == mmax)
    DestroyCache (CacheRemove (cache_ring->cache_prev));

  swidth = (int) (gstate->CTM.tx - left);
  sheight = (int) (gstate->CTM.ty - bottom);

  CacheInsert (cache_ring->cache_next,
               NewCache (fidCache,
                         gstate->CTM,
                         width, height,
                         swidth, sheight));
  gstate->show->ccache = cache_ring->cache_next;

  /* It takes longer to load them from the disk than to create them! */
#ifdef notdef
  LoadDiskCache (gstate->CTM, fontbboxCache);
#endif

  return TRUE;
}

int CacheShow (name, cp)
     Object name;
     HardPoint cp;
{
  struct char_table *b;
  struct cache *ccache = gstate->show->ccache;
  DevicePoint from, to, extent;

  if (!ccache) return FALSE;
  gstate->show->CharName = name;

  if((b = CharFind (CharTable, ccache, name, cmax)) == NULL) return FALSE;

  gstate->show->Width = b->char_width;

  from = NewDevicePoint(0, 0);
  to = NewDevicePoint(((int) cp.hx - ccache->swidth),
                      ((int) cp.hy - ccache->sheight));
  extent = NewDevicePoint((int) ccache->width, ccache->height);

  Paint (b->char_device->dev, gstate->device->dev, from, to, extent,
         gstate->colour);

  return TRUE;
}

static struct cache *NewCache (fid, m, width, height, swidth, sheight)
     int fid;
     Matrix m;
     int width, height, swidth, sheight;
{
  struct cache *res = (struct cache *) Malloc (sizeof (struct cache));

  m.tx = m.ty = 0.0;
  res->fid      = fid;
  res->mat      = m;
  res->width    = width;
  res->height   = height;
  res->swidth   = swidth;
  res->sheight  = sheight;

  return res;
}
```



```c
static DestroyCache (p)
     struct cache *p;
{
  FlushChars (p);
  Free ((char *) p);
}

static CacheInsert (where, cache)
     struct cache *where, *cache;
{
  cache->cache_next = where; cache->cache_prev = where->cache_prev;
  where->cache_prev->cache_next = cache;
  where->cache_prev = cache;
  ++msize;
}

static struct cache *CacheRemove (item)
     struct cache *item;
{
  item->cache_prev->cache_next = item->cache_next;
  item->cache_next->cache_prev = item->cache_prev;
  --msize;

  return item;
}

static int SetCacheLimit (mb)
     Object mb;
{
  int b = BodyInteger (mb);
  if (b < 0)
    return Error (PRangeCheck);
  maxbits = b;
  return TRUE;
}

static int SetCache (bitmap, fonts, chars)
     Object bitmap, fonts, chars;
{
  int b = BodyInteger (bitmap);
  int f = BodyInteger (fonts);
  int c = BodyInteger (chars);

  if (b < 0 || f < 0 || c < 0)
    return Error (PRangeCheck);
  bmax = b; mmax = f; cmax = c;
  InitCharTable ();
  return TRUE;
}

static int CacheStatus ()
{
  VOID Push (OpStack, MakeInteger (bsize));
  VOID Push (OpStack, MakeInteger (bmax));
  VOID Push (OpStack, MakeInteger (msize));
  VOID Push (OpStack, MakeInteger (mmax));
  VOID Push (OpStack, MakeInteger (csize));
  VOID Push (OpStack, MakeInteger (cmax));
  VOID Push (OpStack, MakeInteger (maxbits));

  return TRUE;
}

LoadDiskCache (m, bbox)
     Matrix m;
     float *bbox;
{
  Object font_name;
  FILE *fp;
  Vector origin, right, bottom;
  char disk_name [BUFSIZE], *bits, mess[BUFSIZ];
  /*    struct hardware *screen = gstate->device->dev; */

  Message ("Attempting to find a cache on disk");
  font_name = DictLoad (gstate->font, FontName);
  if (TypeOf (font_name) != Name)
    return;
  origin = Transform (NewVector (bbox [BBOX_LEFT], bbox [BBOX_TOP], 1.0),
                      m);
  right  = DiffVector (Transform (NewVector (bbox [BBOX_RIGHT],
                                             bbox [BBOX_TOP], 1.0), m),
                       origin);
  bottom = DiffVector (Transform (NewVector (bbox [BBOX_LEFT],
                                             bbox [BBOX_BOTTOM], 1.0),
                                  m), origin);
  VOID sprintf (disk_name,
               "%s/cache/%.*s/%3d%3d%3d%3d",
               library,
               lengthName (font_name), BodyName (font_name),
               (int) right.vx, (int) right.vy, (int) bottom.vx,
               (int) bottom.vy);
  Message (disk_name);
  puts (disk_name);
  if ((fp = fopen (disk_name, "r")) == NULL)
    return;
  Message ("found a disk cache");
  VOID GSave ();
  bits = Malloc ((unsigned) maxbits);

  sprintf (mess, "width = %d, height = %d\n",
           gstate->show->ccache->width, gstate->show->ccache->height);
  Message (mess);
  sprintf (mess, "computed length = %d",
           gstate->show->ccache->height * ((gstate->show->ccache->width + 7) / 8));
  Message (mess);

  while (!feof (fp)) {
    char char_name [BUFSIZE];
    struct hardware *newdev;
    int width, height, length;

    if (fscanf (fp, "%s %d %d %d\n",
               char_name,
               &width, &height,
               &length) == EOF)
      break;
    VOID fread (bits, (unsigned) length, 1, fp);
    VOID getc (fp);
    newdev = HardwareFromString (bits, gstate->show->ccache->width,
                                 gstate->show->ccache->height);
    /*          VOID BitBlt (newdev, screen,
     * NewDevicePoint (0, 0), NewDevicePoint (200, 200),
     * NewDevicePoint (gstate->show->ccache->width,
     * gstate->show->ccache->height),
     * ROP_SOURCE);
     */
    VOID SetCacheDevice (NameFrom (char_name),
```

```
                         NewUserPoint ((float) width, (float) height),
                         bbox [BBOX_LEFT],  bbox [BBOX_TOP],
                         bbox [BBOX_RIGHT], bbox [BBOX_BOTTOM]);

    VOID BitBlt (newdev, gstate->device->dev,
                 NewDevicePoint (0, 0), NewDevicePoint (0, 0),
                 NewDevicePoint (gstate->show->ccache->width,
                                 gstate->show->ccache->height),
                 ROP_SOURCE);
    DestroyHardware (newdev);
  }
  Free (bits);
  VOID GRestore ();
  VOID fclose (fp);
  Message ("load completed");
}

static int SaveCurrentFont ()
{
  Matrix fm, m;
  Vector origin, right, bottom;
  Object font_name;
  FILE *fp;
  char disk_name [BUFSIZE];
  int i;
  float bbox[4];
  struct cache *ccache;

  ExtractMatrix (&fm, DictLoad (gstate->font, FontMatrix));
  m = MatMult (fm, gstate->CTM);
  ExtractBBox (bbox, DictLoad (gstate->font, FontBBox));
  font_name = DictLoad (gstate->font, FontName);

  origin = Transform (NewVector (bbox [BBOX_LEFT], bbox [BBOX_TOP], 1.0),
                      m);
  right  = DiffVector (Transform (NewVector (bbox [BBOX_RIGHT],
                                             bbox [BBOX_TOP], 1.0), m),
                       origin);
  bottom = DiffVector (Transform (NewVector (bbox [BBOX_LEFT],
                                             bbox [BBOX_BOTTOM], 1.0),
                                  m), origin);

  if ((ccache = SearchCache (m, BodyFontID (DictLoad (gstate->font,
                                                      Fid)))) == NULL)
    return Error (PInvFont);

  Message (disk_name);
  VOID sprintf (disk_name,
                "%s/cache/%.*s/%3d%3d%3d%3d",
                library,
                lengthName (font_name), BodyName (font_name),
                (int) right.vx, (int) right.vy, (int) bottom.vx,
                (int) bottom.vy);

  if ((fp = fopen (disk_name, "w")) == NULL)
    return Error (PInvFileAccess);

  /* DCJ:
   * Do we want to set a length based on real coordinates or virtual?
   */
  for (i = 0; i < cmax; i++)
    if (CharTable[i].char_cache == ccache) {
      Object char_name;
      unsigned char *bits;
      DevicePoint extent;
      int length;

      char_name = CharTable[i].char_key;
      extent = HardwareExtent (CharTable[i].char_device->dev);
      length = (extent.dx + 7) / 8 * extent.dy;
      bits = StringFromHardware (CharTable[i].char_device->dev);
      VOID fprintf (fp,
                    "%.*s %d %d %d\n",
                    lengthName (char_name), BodyName (char_name),
                    (int) CharTable [i].char_width.x,
                    (int) CharTable [i].char_width.y,
                    length);
      VOID fwrite (bits, (unsigned) length, 1, fp);
      putc ('\n', fp);
      /*                Free (bits);*/

    }

  VOID fclose (fp);

  return TRUE;
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"


Colour NewGray (level)
     float level;
{
  return NewHSBColour (0.0, 0.0, level);
}

Colour NewColour (h, s, b)
     float h, s, b;
{
  return NewHSBColour (h, s, b);
}

Colour NewHSBColour (h, s, b)
     float h, s, b;
{
  Colour res;

  res.hue = h;
  res.saturation = s;
  res.brightness = b;

  return res;
}

Colour NewRGBColour (R, G, B)
     float R, G, B;
{
  float H, S, L, m, M, r, g, b;

  M = R > G ? R : G; M = M > B ? M : B;
  m = R < G ? R : G; m = m < B ? m : B;
  if (M != m) {
    r = (M - R) / (M - m);
    g = (M - G) / (M - m);
    b = (M - B) / (M - m);
  }
  L = (M + m) / 2;

  if (M == m)
    S = 0;
  else if (L <= 0.5)
    S = (M - m) / (M + m);
  else
    S = (M - m) / (2 - M - m);

  if (S == 0)
    H = 0;
  else if (R == M)
    H = 2 + b - g;
  else if (G == M)
    H = 4 + r - b;
  else
    H = 6 + g - r;

  H /= 6;

  return NewHSBColour (H, S, L);
}

void ColourHSB (colour, h, s, b)
     Colour colour;
     float *h, *s, *b;
{
  *h = colour.hue;
  *s = colour.saturation;
  *b = colour.brightness;
}

float Value (m, M, hue)
     float m, M, hue;
{
  if (hue < 0.0)
    hue += 2 * PI;
  if (hue < PI / 3)
    return m + (M - m) * hue / (PI / 3);
  else if (hue < PI)
    return M;
  else if (hue < 4 * PI / 3)
    return m + (M - m) * (4 * PI / 3 - hue) / (PI / 3);
  else
    return m;
}

void ColourRGB (colour, r, g, b)
     Colour colour;
     float *r, *g, *b;
{
  float H = colour.hue, S = colour.saturation, L = colour.brightness;
  float m, M;

  if (L <= .5)
    M = L * (1 + S);
  else
    M = (L + S) - (L * S);
  m = 2 * L - M;
  H *= 2 * PI;
  *r = Value (m, M, H);
  *g = Value (m, M, H - 2 * PI / 3);
  *b = Value (m, M, H - 4 * PI / 3);
}

float Brightness (colour)
     Colour colour;
{
  return colour.brightness;
}


/*
 * The RGB colour model and Hue Saturation Brightness/Lightness model
 * are derived from the paper:
```

```
 *
 *      "Colour Gamut Transform Pairs"
 *      by Alvy Ray Smith in Computer Graphics Volume 12 #3. August 1978
 *
 * PostScript uses the NTSC video colour weights.
 */

#define RED_WEIGHT      0.3333  /* .3   */
#define GREEN_WEIGHT    0.3333  /* .59  */
#define BLUE_WEIGHT     0.3333  /* .11  */

#define a0      120.0 * PI / 180                /* 156.58 * PI / 180 */
#define a1      120.0 * PI / 180                /* 115.68 * PI / 180 */

#define A0      0.0             /* -21.60 * PI / 180 */
#define A1      0.0             /* 14.98 * PI / 180 */
#define A2      0.0             /* 10.65 * PI / 180 */


/*
static int SetRGB (red, green, blue) Object red, green, blue;
 {
        float R = BodyReal (red), G = BodyReal (green), B = BodyReal (blue);
        float r, g, b, r_, g_, b_, d, x, rr, gg, bb, wr_, k0, k1, min, H, S, L;

        if (R < 0 || R > 1 || G < 0 || G > 1 || B < 0 || B > 1)
                return Error (PRangeCheck);
        L = R * RED_WEIGHT + G * GREEN_WEIGHT + B * BLUE_WEIGHT;
        r_ = R/L; g_ = G/L; b_ = B/L;
        r = RED_WEIGHT * r_; g = GREEN_WEIGHT * g_; b = BLUE_WEIGHT * b_;
        rr = r - RED_WEIGHT; gg = g - GREEN_WEIGHT; bb = b - BLUE_WEIGHT;
        min = (r_ < g_ ? r_ : g_); min = min < b_ ? min : b_;
        S = 1 - min;
        if (S != 0)
         {
                k0 = sqrt (rr*rr + gg*gg + bb*bb);
                wr_ = 1 - RED_WEIGHT;
                d = wr_ * rr - GREEN_WEIGHT * gg + BLUE_WEIGHT * bb;
                k1 = sqrt (wr_ * wr_ + GREEN_WEIGHT * GREEN_WEIGHT - BLUE_WEIGHT * BLUE_W
                x = d / (k0 * k1);
                H = PI / 2 - atan2 (x, sqrt (1 - x*x));
                if (b_ > g_)
                        H = 2 * PI - H;
                H /= 2 * PI;
         }
        if (H < 0 || H > 1 || S < 0 || S > 1 || L < 0 || L > 1)
                return Error (PRangeCheck);
        gstate->colour = NewHSLColour (H, S, L);
        return TRUE;
 }

static int GetRGB ()
 {
        float H = gstate->colour.hue, S = gstate->colour.saturation, L = gstate->colour.b
        float r, g, b, Wr = RED_WEIGHT, Wg = GREEN_WEIGHT, Wb = BLUE_WEIGHT;

        H *= 2 * PI;
        if (0 <= H && H <= a0)
         {
                H -= A0;
                b = Wb * (1 - S);
                r = Wr + Wb * S * cos (H) / cos (PI / 3 - H);
                g = 1 - (r + b);
         }
        else if (a0 <= H  && H <= (a0 + a1))
         {
                H -= a0 + A1;
                r = Wr * (1 - S);
                g = Wg + Wr * S * cos (H) / cos (PI / 3 - H);
                b = 1 - (r + g);
         }
        else
         {
                H -= a0 + a1 + A2;
                g = Wg * (1 - S);
                b = Wb + Wg * S * cos (H) / cos (PI / 3 - H);
                r = 1 - (g + b);
         }

        VOID Push (OpStack, MakeReal (L* r / Wr));
        VOID Push (OpStack, MakeReal (L* g / Wg));
        VOID Push (OpStack, MakeReal (L* b / Wb));
        return TRUE;
 }
*/
```



```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

typedef struct { float hue, saturation, brightness; } Colour;

extern Colour Black, White;

extern Colour NewColour (), NewHSBColour (), NewRGBColour (), NewGray ();
extern void ColourHSB (), ColourRGB ();
extern float Gray (), Brightness ();
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static int SizeArray = 53, SizeMark = 19, SizeBool = 53, SizeNull = 19;
static int SizeCondition = 19, SizeFile = 41, SizeString = 71, SizePoly = 19;
static int SizeInteger = 71, SizeReal = 71, SizeName = 19, SizeOperator = 19;
int SizeSysDict = 1001, SizeDictionary = 53, SizeFloat = 0, SizeFontID = 19;

Type Array, Mark, Bool, Dictionary, Condition, Null, File, Integer;
Type Real, Name, Operator, String, Poly, Float, FontID;

extern Object Absent, SysDict;
Object Nil;

#ifdef MACINTOSH
char default_library[] = "";
#else
char default_library[] = "/mit/sipb/lib/postscript";
#endif

Init ()
{
  MakeObject (Nil, (Type) 0);
  Null          = Nil.type = MakeType (SizeNull);
  EmptyDict ((Type) Null);      /* needed because of this recursion */

  Dictionary    = MakeType (SizeDictionary);
  Condition     = MakeType (SizeCondition);
  Name          = MakeType (SizeName);
  Operator      = MakeType (SizeOperator);
  Array         = MakeType (SizeArray);
  Mark          = MakeType (SizeMark);
  Bool          = MakeType (SizeBool);
  File          = MakeType (SizeFile);
  Integer       = MakeType (SizeInteger);
  Real          = MakeType (SizeReal);
  String        = MakeType (SizeString);
  Poly          = MakeType (SizePoly);
  Float         = MakeType (SizeFloat);
  FontID        = MakeType (SizeFontID);

  Message ("InitDictionary");   InitDictionary ();
  Message ("InitOperator");     InitOperator ();
  Message ("InitName");         InitName ();
  Message ("InitPoly");         InitPoly ();
  Message ("InitArray");        InitArray ();
  Message ("InitStack");        InitStack ();
  Message ("InitFile");         InitFile ();
  Message ("InitMisc");         InitMisc ();
  Message ("InitBool");         InitBool ();
  Message ("InitInteger");      InitInteger ();
  Message ("InitReal");         InitReal ();
  Message ("InitMath");         InitMath ();
  Message ("InitString");       InitString ();
  Message ("InitProperty");     InitProperty ();
  Message ("InitControl");      InitControl ();

  Message ("InitMatrix");       InitMatrix ();
  Message ("InitPath");         InitPath ();
  Message ("InitFill");         InitFill ();
  Message ("InitStroke");       InitStroke ();
  Message ("InitGSave");        InitGSave ();
  Message ("InitDevices");      InitDevices ();
  Message ("InitCache");        InitCache ();
  Message ("InitImage");        InitImage ();
  Message ("InitState");        InitState ();
  Message ("InitFont");         InitFont ();
  Message ("InitUnix");         InitUnix ();

  Install ("nulltype",          DictFrom (Null));
  Install ("dicttype",          DictFrom (Dictionary));
  Install ("conditiontype",     DictFrom (Condition));
  Install ("nametype",          DictFrom (Name));
  Install ("operatortype",      DictFrom (Operator));
  Install ("arraytype",         DictFrom (Array));
  Install ("marktype",          DictFrom (Mark));
  Install ("booleantype",       DictFrom (Bool));
  Install ("filetype",          DictFrom (File));
  Install ("integertype",       DictFrom (Integer));
  Install ("realtype",          DictFrom (Real));
  Install ("stringtype",        DictFrom (String));
  Install ("polytype",          DictFrom (Poly));
  Install ("fonttype",          DictFrom (FontID));

  Install ("version",           StringFrom ("Version 1.4"));
}
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.  It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include <signal.h>
#include "main.h"
#include "graphics.h"

/*
 * This file implements the flow control operators of PostScript
 *
 * All operations are implemented using the ExecStack and without recursion.
 * This is to allow error recovery to operate cleanly.
 *
 * Actual control is acheived by stacking continuation operators for
 * the looping operators, declared thus:
 *
 */

static Object OpLoop, OpRepeat, OpStopped;

/*
 * "loop", for example, stacks the following:
 *      proc
 *      OpLoop
 *      proc
 *      Nil
 *
 * OpLoop is the continuer, proc is the body code, passed as a parameter.
 *
 * Nil is placed at the bottom by each of the looping constructs,
 * which are 'exit'able.
 * "exit" pops the exec stack until it encounters a Nil.
 *
 * A Marker is placed on the ExecStack to enclose a Stopped context.
 * A File is placed on the ExecStack to enclose a Run context.
 * A Nil is placed on the ExecStack to enclose a Looping construct.
 * "stop" will unwind the stack down to the Marker, ignoring any Nil
 * or File it might find.
 * "exit" will baulk if it finds a Marker or File before a Nil.
 *
 */

static int PExec (), PIf (), PIfElse (), PRepeat (), PFor (), PLoop ();
static int PExit (), PCountExecStack ();
static int PQuit (), PStart ();
int PStop (), PStopped ();
static int Repeat (), Loop (), Stopped ();

InitControl ()
{
  OpRepeat      = MakeOp ("(repeat)",   Repeat, 0, 0, 3, 2);
  OpLoop        = MakeOp ("(loop)",     Loop,   0, 0, 1, 2);
  OpStopped     = MakeOp ("(stopped)",  Stopped,0, 1, 1, 0);

  InstallOp ("exec",    PExec,          1, 1, 0, 1, Poly);
  InstallOp ("if",      PIf,            2, 0, 0, 1, Bool, Array);
  InstallOp ("ifelse",  PIfElse,        3, 0, 0, 1, Bool, Array, Array);
  InstallOp ("repeat",  PRepeat,        2, 0, 0, 4, Integer, Array);
  InstallOp ("for",     PFor,           4, 0, 0, 6, Poly, Poly, Poly, Array);
  InstallOp ("loop",    PLoop,          1, 0, 0, 4, Array);
  InstallOp ("exit",    PExit,          0, 0, 0, 0);
  InstallOp ("stop",    PStop,          0, 1, 0, 0);
  InstallOp ("stopped", PStopped,       1, 1, 0, 3, Array);
  InstallOp ("quit",    PQuit,          0, 0, 0, 0);
  InstallOp ("start",   PStart,         0, 0, 0, 0);
  InstallOp ("countexecstack",
            PCountExecStack,0, 1, 0, 0);
}

static int PExec (item)
     Object item;
{
  VOID Push (OpStack, item);
  if (xCheck (item)) {
    Object fn;

    fn = Lookup (TypeOf (item), Self);
    if (TypeOf (fn) != Condition)
      VOID Push (ExecStack, fn);
  }
  return TRUE;
}

static int PIf (bool, proc)
     Object bool, proc;
{
  if (!xCheck (proc))
    return Error (PTypeCheck);
  if (BodyBool (bool))
    VOID Push (ExecStack, proc);
  return TRUE;
}

static int PIfElse (bool, proc1, proc2)
     Object bool, proc1, proc2;
{
  if (!xCheck (proc1) || !xCheck (proc2))
    return Error (PTypeCheck);
  if (BodyBool (bool))
    VOID Push (ExecStack, proc1);
  else
    VOID Push (ExecStack, proc2);
  return TRUE;
}

static int PRepeat (count, proc)
     Object count, proc;
{
  if (!xCheck (proc))
    return Error (PTypeCheck);
  else if (BodyInteger (count) < 0)
    return Error (PRangeCheck);

  VOID Push (ExecStack, Nil);
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, count);
  VOID Push (ExecStack, OpRepeat);

  return TRUE;
```

```
}

static int Repeat ()
{
  int count;

  count = BodyInteger (Pop (ExecStack));

  if (count != 0) {
    Object proc;

    proc = Top (ExecStack);
    VOID Push (ExecStack, MakeInteger (count - 1));
    VOID Push (ExecStack, OpRepeat);
    VOID Push (ExecStack, proc);
  } else {
    VOID Pop (ExecStack);
    VOID Pop (ExecStack);
  }
  return TRUE;
}

static int PFor (initial, increment, limit, proc)
     Object initial, increment, limit, proc;
{
  int integers = 0, reals = 0, res;

  if (!xCheck (proc))
    return Error (PTypeCheck);

  if (TypeOf (initial) == Integer || TypeOf (initial) == Real)
    integers += TypeOf (initial) == Integer,
    reals += TypeOf (initial) == Real;

  if (TypeOf (increment) == Integer || TypeOf (increment) == Real)
    integers += TypeOf (increment) == Integer,
    reals += TypeOf (increment) == Real;

  if (TypeOf (limit) == Integer || TypeOf (limit) == Real)
    integers += TypeOf (limit) == Integer,
    reals += TypeOf (limit) == Real;

  if (integers + reals == 3 && integers != 3 && reals != 3) {
    if (TypeOf (initial) == Integer)
      initial = RealInteger (initial);
    if (TypeOf (increment) == Integer)
      increment = RealInteger (increment);
    if (TypeOf (limit) == Integer)
      limit = RealInteger (limit);
  }
  if (res = Apply (TypeOf (initial))) {
    VOID Push (OpStack, initial);
    VOID Push (OpStack, increment);
    VOID Push (OpStack, limit);
    VOID Push (OpStack, proc);
  }
  return res;
}

static int PLoop (proc)
     Object proc;
{
  if (!xCheck (proc))
    return Error (PTypeCheck);

  VOID Push (ExecStack, Nil);
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpLoop);
  VOID Push (ExecStack, proc);

  return TRUE;
}

static int Loop ()       /* proc --- */
{
  Object proc;

  proc = Top (ExecStack);
  VOID Push (ExecStack, OpLoop);
  VOID Push (ExecStack, proc);
  return TRUE;
}

static int PExit ()
{
  int n, m, p;

  if ((n = CountTo (Null, ExecStack)) < 0
      || (m = CountTo (Mark, ExecStack)) >= 0 && m < n
      /* don't cross Stopped Context */
      || (p = CountTo (File, ExecStack)) >= 0 && p < n)
    /* don't cross Run context */
    return Error (PInvExit);
  VOID ClearTo (Null, ExecStack);
  return TRUE;
}

int PStop ()    /* --- */
{
  Object item;

  for (;;) {
    if (Height (ExecStack) == 0) {
      if (interactive)
        return Push (ExecStack, Cvx (NameFrom ("executive")));
      else
        return TRUE;
    }
    item = Pop (ExecStack);
    if (item.type == Mark)
      return Push (OpStack, True);
  }
}

static int Stopped ()
{
  VOID Pop (ExecStack);
  VOID Push (OpStack, False);
  return TRUE;
}

int PStopped (proc)
     Object proc;
{
  VOID Push (ExecStack, Marker);
  VOID Push (ExecStack, OpStopped);
  VOID Push (ExecStack, proc);
  return TRUE;
```



```
}

static int PCountExecStack ()
{
  VOID Push (OpStack, MakeInteger (Height (ExecStack)));
  return TRUE;
}

/*
 * PExecStack ()
 *
 * This routine is in stack.c
 *
 */

static int PQuit ()
{
  UnlinkDevice (gstate->device);
  Cleanup ();
  exit (0);

  return TRUE; /* never will */
}

static int PStart ()
{
  return TRUE;
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not sold for profit
 * or incorporated into a product except under licence from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes to the source
 * made by persons other than the author should be marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

static int NullDevice ();
static int FrameDevice ();
static int GrabBits ();
static int PWindow ();

Colour Black = { 0.0, 0.0, 0.0 }, White = { 0.0, 0.0, 1.0 };

struct device *NewWindowDevice (), *NewBitmapDevice (), *NewCacheDevice ();
DevicePoint HardToDevice ();
HardPoint DeviceToHard ();
extern DevicePoint ConvertToDevicePoint ();
extern float hppp, vppp;

InitDevices ()
{
  struct hardware *h;
  DevicePoint extent;

  InstallOp ("framedevice",     FrameDevice,            4, 0, 0, 0,
             Array, Integer, Integer, Array);
  InstallOp ("nulldevice",      NullDevice,             0, 0, 0, 0);
  InstallOp ("grabbits",        GrabBits,               4, 0, 0, 0,
             Float, Float, Float, Float);
  InstallOp ("window",          PWindow,                5, 0, 0, 0,
             Float, Float, Float, Float, Integer);

  extent = HardwareExtent (h = InitHardware ());
  gstate->device = NULL; gstate->clipdevice = NULL;
  SetDevice (LinkDevice (DeviceFrom (h, DeviceMatrix (extent.dx,
                                                      extent.dy))));
}

static int GrabBits (x, y, w, h)
     Object x, y, w, h;
{
  DevicePoint or,ul,lr;
  HardPoint origin, corner;
  int width, height, size;
  unsigned char *s;
  struct hardware *dev;

  origin = ExtToInt (NewUserPoint (BodyReal (x), BodyReal (y)));
  corner = ExtToInt (NewUserPoint (BodyReal (x) + BodyReal (w),
                                   BodyReal (y) + BodyReal (h)));
  width = (int) (corner.hx - origin.hx);
  height = (int) (corner.hy - origin.hy);

  fprintf (stderr, "width = %d, height = %d\n", width, height);

  dev = NewBitmapHardware (width, height);
  or.dx = (int) origin.hx; or.dy = (int) origin.hy;
  ul.dx = 0;               ul.dy = 0;
  lr.dx = width;           lr.dy = height;

  BitBlt (gstate->device->dev, dev, or, ul, lr, ROP_SOURCE);
  s = StringFromHardware (dev);
  DestroyHardware (dev);
  size = (width + 7) / 8 * height;

  Push (OpStack, MakeString (s, size));
  Free (s);
  Push (OpStack, MakeInteger (width));
  Push (OpStack, MakeInteger (height));

  return TRUE;
}

static int NullDevice ()
{
  extern Matrix identity;

  SetDevice (NewDevice (
                       NewClipPath (0.0, 0.0, 0.0, 0.0),
                       identity,
                       (struct hardware *) NULL));
  return TRUE;
}

/*ARGSUSED*/
static int FrameDevice (mat, width, height, proc)
     Object mat, width, height, proc;
{
  Matrix m;

  if (lengthArray (mat) != 6 || !ExtractMatrix (&m, mat))
    return Error (PTypeCheck);
  if (BodyInteger (width) < 0 || BodyInteger (height) < 0)
    return Error (PRangeCheck);
  SetDevice (NewWindowDevice (BodyInteger (width) * 8, BodyInteger (height),
                              m));
  ErasePage ();

  return TRUE;
}

/*ARGSUSED*/
static int PWindow (x, y, wid, hgt, typ)
     Object x, y, wid, hgt, typ;
{
  int wd, ht;
  Matrix m;
  struct device *w;

  wd = (int) (BodyReal (wid) * hppp);
  ht = (int) (BodyReal (hgt) * vppp);

  NewMatrix(m, hppp, 0, 0, -vppp, 0, ht);
  w = NewWindowDevice (wd, ht, m);
  SetDevice (w);
  ErasePage ();

  return Push (OpStack, MakeInteger (w));
```

```
}

struct device *DeviceFrom (h, m)
     struct hardware *h; Matrix m;
{
  DevicePoint extent;

  extent = HardwareExtent (h);
  return NewDevice (NewClipPath (0.0,
                                 (float) extent.dx,
                                 (float) extent.dy, 0.0),
                    m, h);
}

struct device *NewWindowDevice (width, height, m)
     int width, height;
     Matrix m;
{
  return DeviceFrom (NewWindowHardware (width, height),
                     m);
}

struct device *NewBitmapDevice (width, height, m)
     int width, height;
     Matrix m;
{
  return DeviceFrom (NewBitmapHardware (width, height),
                     m);
}

struct device *NewCacheDevice (m, width, height, swidth, sheight)
     Matrix m; int width, height, swidth, sheight;
{
  m.tx = (float) swidth;
  m.ty = (float) (height - sheight);
  return NewDevice (NewClipPath (0.0, (float) width,
                                 (float) height, 0.0),
                    m,
                    NewBitmapHardware (width, height));
}

Path NewClipPath (left, right, top, bottom)
     float left, right, top, bottom;
{
  Path p = NewPath ();
  HardPoint cp;
  int cp_def = gstate->cp_defined;

  cp = gstate->cp;
  VOID MoveTo (p, NewHardPoint (left,  bottom));
  VOID LineTo (p, NewHardPoint (right, bottom));
  VOID LineTo (p, NewHardPoint (right, top));
  VOID LineTo (p, NewHardPoint (left,  top));
  ClosePath (p);

  gstate->cp = cp; gstate->cp_defined = cp_def;

  return p;
}

struct device *NewDevice (clip, m, dev)
     Path clip;
     Matrix m;
     struct hardware *dev;
{
  struct device *res = (struct device *) Malloc (sizeof (struct device));

  res->link_count = 0;
  res->default_clip = clip;
  res->default_matrix = m;
  res->dev = dev;

  return res;
}

SetDevice (d)
     struct device *d;
{
  UnlinkDevice (gstate->device);
  gstate->device = LinkDevice (d);
  gstate->CTM = d->default_matrix;
  InitClip ();
}

int IsCache (d)
     struct device *d;
{
  return !IsWindowHardware (d->dev);
}

struct device *LinkDevice (d)
     struct device *d;
{
  if (d)
    ++d->link_count;

  return d;
}

UnlinkDevice (d)
     struct device *d;
{
  if (d == NULL)
    return;
  if (--d->link_count != 0)
    return;
  if (d->dev != NULL)
    DestroyHardware (d->dev);
  Free ((char *) d);
}

struct device *UniqueDevice (d)
     struct device *d;
{
  struct device *res;
  DevicePoint ex;

  if (d && d->link_count == 1)
    return d;
  ex = HardwareExtent (gstate->device->dev);
  res = LinkDevice (NewBitmapDevice (ex.dx, ex.dy, gstate->CTM));
  /*if (d)
   * {
   * BitBlt (d->dev, res->dev, NewDevicePoint (0, 0), NewDevicePoint (0, 0),
   *         ex, ROP_SOURCE);
   * UnlinkDevice (d);
   * }
   * else
```



```
   */
  BitBlt ((struct hardware *)NULL, res->dev,
          NewDevicePoint (0, 0), NewDevicePoint (0, 0),
          ex, ROP_FALSE);

  return res;
}

HardPoint DeviceToHard (p)
     DevicePoint p;
{
  HardPoint res;

  res.hx = (float) p.dx;
  res.hy = (float) p.dy;

  return res;
}

DevicePoint NewDevicePoint (x, y)
     int x, y;
{
  DevicePoint res;

  res.dx = x;
  res.dy = y;

  return res;
}

/*
 * DevicePaintLine() gets called when a thin line (linewidth 0) is to
 * be drawn.
 */
DevicePaintLine (d, fromPoint, toPoint, colour)
     struct device *d;
     HardPoint fromPoint, toPoint;
     Colour colour;
{
  DevicePoint fp,tp;
  fp.dx = (int) fromPoint.hx;  fp.dy = (int) fromPoint.hy;
  tp.dx = (int) toPoint.hx;    tp.dy = (int) toPoint.hy;
  PaintLine (d->dev, fp, tp, colour);
}


#ifdef notdef
void ColourLine (h, fromPoint, toPoint, rop, colour)
     struct hardware *h;
     DevicePoint fromPoint, toPoint;
     int rop;
     Colour colour;
{
  int x1 = fromPoint.dx, y1 = fromPoint.dy, x2 = toPoint.dx, y2 = toPoint.dy;
  int col = IsWindowHardware (h) ? HardColour (colour) : HardBlack;

  if (col == HardBlack) {
    ddbm = h->bm;
    dd->d_line = rop_map [rop];
    line (x1, y1, LNMOVEABS);
    line (x2, y2, LNDRAWABS);
  }
  else {
    box b;

    b = boxbuild (Min (x1, x2), Min (y1, y2), Max (x1, x2), Max (y1, y2));

    NeedAux (h);
    GraySync (h, col);
    bmxcopy (h->bm, b, h->bm, b, WWXOR);

    ddbm = h->aux;
    dd->d_line = WWOR;
    line (x1, y1, LNMOVEABS);
    line (x2, y2, LNDRAWABS);

    bmxcopy (h->aux,  b, h->bm,  b, WWAND | WWNOT);
    bmxcopy (h->gray, b, h->aux, b, WWAND);
    bmxcopy (h->aux,  b, h->bm,  b, rop_map [rop]);
  }
}

void ColourBitBlt (from, to, fromPoint, toPoint, extent, rop, colour)
        struct hardware *from, *to;
        DevicePoint fromPoint, toPoint, extent;
        int rop;
        Colour colour;
 {
        box frombox, tobox;
        int col = IsWindowHardware (to) ? HardColour (colour) : HardBlack;
        int op = rop_map [rop];


        frombox = boxbuild (fromPoint.dx, fromPoint.dy,
                fromPoint.dx + extent.dx - 1, fromPoint.dy + extent.dy - 1);
        tobox = boxbuild (toPoint.dx, toPoint.dy,
                toPoint.dx + extent.dx - 1, toPoint.dy + extent.dy - 1);

        if (to == NULL)
                return;

        if (rop == ROP_FALSE)
                bmxcopy (to->bm, tobox, to->bm, tobox, WWXOR);
        else if (rop == ROP_TRUE)
                if (col == HardBlack)
                        bmxcopy (to->bm, tobox, to->bm, tobox, WWXOR | WWNOT);
                else
                 {
                        GraySync (to, col);
                        bmxcopy (to->gray, tobox, to->bm, tobox, WWCOPY);
                 }
        else if (col == HardBlack)
                bmxcopy (from->bm, frombox, to->bm, tobox, op);
        else
         {
                GraySync (to, col);
                NeedAux (to);
                bmxcopy (to->gray, tobox, to->aux, tobox, WWCOPY);
                bmxcopy (from->bm, frombox, to->aux, tobox, WWAND);
                bmxcopy (to->aux, tobox, to->bm, tobox, op);
         }
 }
#endif
```

```c
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#include "hard.h"

struct device
{
  Matrix default_matrix;
  Path default_clip;
  int link_count;
  struct hardware *dev;
};

extern struct device *NewDevice (), *NewCacheDevice (), *LinkDevice ();
extern struct device *DeviceFrom (), *UniqueDevice (), *NewBitmapDevice ();
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

extern int SizeUserDict, SizeSysDict;

static Object OpForDict;

Object Absent, SysDict;
extern Object Nil;

Object DictLookup (), MakeDict ();

static int LengthDict (), CopyDict (), forDict (), ForDict (), PutDict ();
static int GetDict (), PDict (), PBegin (), PEnd (), PDef (), PStore ();
static int PKnown (), PLoad (), PrCheck (), PwCheck (), PReadOnly ();
static int EqDict (), PWhere (), PMaxLength (), PCurrentDict ();

int hash_tries = 0, hash_collisions = 0, hash_attempts = 0;
static int PHashStats (), PDictHash ();

InitDictionary ()
{
  SysDict       = MakeDict (SizeSysDict);

  MakeObject (Absent, Condition);

  OpForDict     = MakeOp ("(foralldict)", forDict, 0, 2, 3, 5);

  Install ("systemdict",        SysDict);

  InstallOp ("dict",    PDict,          1, 1, 0, 0, Integer);
  InstallOp ("begin",   PBegin,         1, 0, 0, 0, Dictionary);
  InstallOp ("end",     PEnd,           0, 0, 0, 0);
  InstallOp ("def",     PDef,           2, 0, 0, 0, Poly, Poly);
  InstallOp ("store",   PStore,         2, 0, 0, 0, Poly, Poly);
  InstallOp ("known",   PKnown,         2, 1, 0, 0, Dictionary, Poly);
  InstallOp ("load",    PLoad,          1, 1, 0, 0, Poly);
  InstallOp ("where",   PWhere,         1, 2, 0, 0, Poly);
  InstallOp ("maxlength",       PMaxLength,     1, 1, 0, 0, Dictionary);
  InstallOp ("currentdict",PCurrentDict,       0, 1, 0, 0);
  InstallOp ("hashstats", PHashStats,   0, 0, 0, 0);
  InstallOp ("dicthash",        PDictHash,      1, 0, 0, 0, Dictionary);

  TypeInstallOp (Dictionary, "eq",      EqDict,         2, 1, 0, 0,
                Dictionary, Dictionary);
  TypeInstallOp (Dictionary, "put",     PutDict,        3, 0, 0, 0,
                Dictionary, Poly, Poly);
  TypeInstallOp (Dictionary, "get",     GetDict,        2, 1, 0, 0,
                Dictionary, Poly);
  TypeInstallOp (Dictionary, "length",  LengthDict,     1, 1, 0, 0,
                Dictionary);
  TypeInstallOp (Dictionary, "copy",    CopyDict,       2, 0, 0, 0,
                Dictionary, Dictionary);
  TypeInstallOp (Dictionary, "forall",  ForDict,        2, 0, 0, 4,
                Dictionary, Array);
  TypeInstallOp (Dictionary, "readonly",        PReadOnly,      1, 1, 0, 0,
                Dictionary);
  TypeInstallOp (Dictionary, "rcheck",  PrCheck,        1, 1, 0, 0,
                Dictionary);
  TypeInstallOp (Dictionary, "wcheck",  PwCheck,        1, 1, 0, 0,
                Dictionary);

  TypeInstallOp (Null, "eq", EqTrue, 2, 1, 0, 0, Null, Null);
}

Type MakeType (size)
     int size;
{
  Type dict;
  int i;
  struct dict_entry *hash;

  dict = (struct dict_struct *) Malloc ((unsigned) sizeof (struct dict_struct));
  dict->dict_size = size;
  dict->dict_fill = 0;
  dict->dict_flags = READABLE | WRITEABLE;
  dict->dict_body = hash = (struct dict_entry *) Malloc ((unsigned) sizeof (struct dict_e

  for (i = 0; i < size; i++)
    hash[i].entry_key = Nil;
  return dict;
}

void EmptyDict (dict)
     Type dict;
{
  int i, size = dict->dict_size;
  struct dict_entry *hash = dict->dict_body;

  for (i = 0; i < size; i++)
    hash[i].entry_key = Nil;
}

#ifdef notdef
Type TypeOf (item)
     Object item;
{
  return item.type;
}
#endif

Object MakeDict (size)
     int size;
{
  Object res;

  MakeObject (res, Dictionary);
  res.u.Dictionary = MakeType (size);
  return res;
}

Object DictFrom (dict)
     Type dict;
{
  Object res;
```

```
  MakeObject (res, Dictionary);
  res.u.Dictionary = dict;
  return res;
}

/*ARGSUSED*/
int EqTrue (a, b)
     Object a, b;
{
  return Push (OpStack, True);
}

#define Body(I) ((I).u.Dictionary)

#ifndef Body
static Type Body (item)
     Object item;
{
  return item.u.Dictionary;
}
#endif

static int PHashStats ()
{
  extern int name_tries, name_hits;

  printf ("Attempts = %d, Successs = %d, Collisions = %d\n",
          hash_attempts, hash_tries, hash_collisions);
  printf ("Name tries = %d, Name hits = %d\n", name_tries, name_hits);
  return TRUE;
}

static int PDictHash (dict)
     Object dict;
{
  struct dict_entry *hash = Body (dict)->dict_body;
  int i, l = Body (dict)->dict_size;

  for (i = 0; i < l; i++)
    if (TypeOf (hash[i].entry_key) == Null)
      putchar (' ');
    else
      putchar ('@');
  putchar ('\n');
  return TRUE;
}

static int EqDict (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) == Body (b)));
}

static DictReplace (hash, key, value, size, h)
     struct dict_entry *hash;
     Object key, value;
     int size, h;
{
  int i;

  for (i = h;;) {
    if (TypeOf (hash[i].entry_key) == Null)
      return FALSE;
    if (DictEqual (key, hash[i].entry_key)) {
      hash[i].entry_value = value;
      return TRUE;
    }
    if (++i == size)
      i = 0;
    if (i == h)
      return FALSE;
  }
}

static DictAddition (dict, hash, key, value, size, h)
     struct dict_struct *dict;
     struct dict_entry *hash;
     Object key, value;
     int size, h;
{
  int i;

  for (i = h;;) {
    if (TypeOf (hash[i].entry_key) == Null) {
      hash[i].entry_value = value;
      hash[i].entry_key = key;
      if (TypeOf (key) != Null)
        ++dict->dict_fill;
      return TRUE;
    }
    if (++i == size)
      i = 0;
    if (i == h)
      return FALSE;
  }
}

TypeInstall (dict, name, value)
     Type dict;
     char *name;
     Object value;
{
  struct dict_entry *hash = dict->dict_body;
  int h, size = dict->dict_size;
  Object key;

  key = NameFrom (name);

  PanicIf (size == 0, "dictionary full in Install");
  /*    h = BodyInteger (key) % size;   */
  h = (key.Length + BodyInteger (key) + (int) TypeOf (key)) % size;
  if (TypeOf (key) == String)
    key = Cvn (key);
  if (DictReplace (hash, key, value, size, h))
    return;
  if (DictAddition (dict, hash, key, value, size, h))
    return;
  Panic ("dictionary full in Install");
}

DictStore (dict, key, value)
     Object dict, key, value;
{
  struct dict_entry *hash = Body (dict)->dict_body;
  int h, size = Body (dict)->dict_size;

  PanicIf (size == 0, "dictionary full in DictStore");
  /*    h = BodyInteger (key) % size;   */
```



```
  h = (key.Length + BodyInteger (key) + (int) TypeOf (key)) % size;
  if (TypeOf (key) == String)
    key = Cvn (key);
  if (DictReplace (hash, key, value, size, h))
    return;
  if (DictAddition (Body (dict), hash, key, value, size, h))
    return;
  Panic ("dictionary full in DictStore");
}

Install (key, value)
     char *key;
     Object value;
{
  DictStore (SysDict, NameFrom (key), value);
}

/*
 * BZS - add some register decls, make global for macrification (remove static)
 */
Object DictFind (hash, key, size)
     struct dict_entry *hash;
     Object key;
     int size;
{
  register int i, h;

  ++hash_attempts;

  if (size == 0) return Absent;

  if (TypeOf (key) == String) key = Cvn (key);

  h = (key.Length + BodyInteger (key) + (int) TypeOf (key)) % size;

  for (i = h;;) {
    if (TypeOf (hash[i].entry_key) == Null)  return Absent;
    if (DictEqual (key, hash[i].entry_key)) {
      ++hash_tries;
      return hash[i].entry_value;
    }
    if (++i == size) i = 0;
    else if (i == h) return Absent;
    ++hash_collisions;
  }
}

/* BZS - macro-ified */
#ifndef DictLoad
Object DictLoad (dict, key)
     Object dict, key;
{
  return DictFind (Body (dict)->dict_body, key, Body (dict)->dict_size);
}
#endif

Object Lookup (dict, key)
     Type dict;
     Object key;
{
  return DictFind (dict->dict_body, key, dict->dict_size);
}

Object Load (key)
     Object key;
{
  Type dict;
  struct dict_entry *hash;
  int level,i,h,size,hashval;

  /* Initialize information based solely on the key */
  if (TypeOf (key) == String) key = Cvn (key);
  hashval = key.Length + BodyInteger(key) + (int) TypeOf(key);

  /* Search through the levels of the dict stack from the top down */
  for (level = Height (DictStack) - 1; level >= 0; level--) {
    dict = Body(DictStack->stack_body[level]);
    if ((size=dict->dict_size) != 0) {
      hash = dict->dict_body;
      ++hash_attempts;
      i = h = hashval % size;

      for(;;) {
        if (TypeOf (hash[i].entry_key) == Null) break;
        if (DictEqual (key, hash[i].entry_key)) {
          ++hash_tries;
          return hash[i].entry_value;
        }
        if (++i == size) i = 0;
        if (i == h) break;
        ++hash_collisions;
      }
    }
  }
  return Absent;
}

static int ForDict (dict, proc)
     Object dict, proc;
{
  VOID Push (ExecStack, dict);
  VOID Push (ExecStack, MakeInteger (0));
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpForDict);

  return TRUE;
}

static int forDict ()
{
  Object dict, index, proc, t; /* VAX Compiler Broken. SUN optimiser broken */
  int i;

  proc =        Pop (ExecStack);
  index =       Pop (ExecStack);
  dict =        Pop (ExecStack);

  for (i = BodyInteger (index); ; i++) {
    if (i >= maxDict (dict))
      return TRUE;
    t = Body (dict)->dict_body[i].entry_key;
    if (TypeOf (t) != Null)
      break;
  }
  VOID Push (ExecStack, dict);
  VOID Push (ExecStack, MakeInteger (i + 1));
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpForDict);
```

```
  VOID Push (ExecStack, proc);

  VOID Push (OpStack, t = Body (dict)->dict_body[i].entry_key);
  VOID Push (OpStack, t = Body (dict)->dict_body[i].entry_value);
  return TRUE;
}

static int GetDict (object, key)
     Object object, key;
{
  Object res;

  res = DictLoad (object, key);
  if (TypeOf (res) != Condition)
    return Push (OpStack, res);
  else
    return Error (PUndefined);
}

static int PutDict (object, key, value)
     Object object, key, value;
{
  if (lengthDict (object) == maxDict (object)) {
    Object t;   /* SUN optimiser broken */

    t = DictLoad (object, key);
    if (TypeOf (t) == Condition)
      return Error (PDictFull);
  }
  DictStore (object, key, value);
  return TRUE;
}

static int CopyDict (object1, object2)
     Object object1, object2;
{
  if (lengthDict (object1) <= maxDict (object2) && lengthDict (object2) == 0) {
    copyDict (object1, object2);
    VOID Push (OpStack, object2);
    return TRUE;
  }
  return Error (PRangeCheck);
}

copyDict (from, to)
     Object from, to;
{
  int i, l = maxDict (from);
  struct dict_entry *body = Body (from)->dict_body;

  for (i = 0; i < l; i++)
    if (TypeOf (body[i].entry_key) != Null)
      DictStore (to, body[i].entry_key, body[i].entry_value);
}

static int PDict (size)
     Object size;
{
  if (BodyInteger (size) < 0)
    return Error (PRangeCheck);
  else
    return Push (OpStack, MakeDict (BodyInteger (size)));
}

static int PBegin (dict)
     Object dict;
{
  if (Height (DictStack) == MaxStack (DictStack))
    return Error (PDictOverflow);
  else {
    VOID Push (DictStack, dict);
    return TRUE;
  }
}

static int PEnd ()
{
  if (Height (DictStack) == 2)
    return Error (PDictUnderflow);
  else
    VOID Pop (DictStack);
  return TRUE;
}

static int PDef (key, value)
     Object key, value;
{
  Object dict,t ;       /* SUN optimiser broken */

  dict = Top (DictStack);
  if (!wCheck (dict))
    return Error (PInvAccess);
  else if (maxDict (dict) == lengthDict (dict) && (t = DictLoad (dict, key), TypeOf (t))
    return Error (PDictFull);
  else {
    DictStore (dict, key, value);
    return TRUE;
  }
}

static int PStore (key, value)
     Object key, value;
{
  Object dict, t;       /* SUN optimiser broken */

  dict = Where (key);   /* SUN optimiser broken */
  if (TypeOf (dict) == Condition)
    dict = Top (DictStack);
  if (TypeOf (key) == Null)
    return Error (PTypeCheck);
  else if (!wCheck (dict))
    return Error (PInvAccess);
  else if (maxDict (dict) == lengthDict (dict) && (t = DictLoad (dict, key), TypeOf (t))
    return Error (PDictFull);
  else {
    DictStore (dict, key, value);
    return TRUE;
  }
}

static int PKnown (dict, key)
     Object dict, key;
{
  Object t;

  t = DictLoad (dict, key);
  return Push (OpStack, MakeBool (TypeOf (t) != Condition));
}
```



```
static int PLoad (key)
     Object key;
{
  Object res;

  res = Load (key);
  if (TypeOf (res) == Condition)
    return Error (PUndefined);
  else if (TypeOf (key) == Null)
    return Error (PTypeCheck);
  else
    return Push (OpStack, res);
}

static int PWhere (key)
     Object key;        /* key --- dict true */
     /* key --- false */
{
  Object where;

  if (TypeOf (key) == Null)
    return Error (PTypeCheck);
  where = Where (key);
  if (TypeOf (where) == Condition)
    return Push (OpStack, False);
  else if (!rCheck (where))
    return Error (PInvAccess);
  else
    return Push (OpStack, where), Push (OpStack, True);
}

static int PMaxLength (dict)
     Object dict;
{
  return Push (OpStack, MakeInteger (maxDict (dict)));
}

static int LengthDict (dict)
     Object dict;
{
  return Push (OpStack, MakeInteger (lengthDict (dict)));
}

static int PCurrentDict ()
{
  return Push (OpStack, Top (DictStack));
}

static int PReadOnly (item)
     Object item;
{
  return Push (OpStack, ReadOnly (item));
}

static int PrCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (rCheck (v)));
}

static int PwCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (wCheck (v)));
}
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

#ifndef AUX
#ifndef MACINTOSH
#include <sgtty.h>
#endif
#endif
#ifdef AUX
#include <sys/termio.h>
#include <sys/ttold.h>
#endif

#define MAXFILENAME     1000

/*
 * This file implements the input/output calls in PostScript.
 *
 * A PostScript file corresponds reasonably closely to a Unix file, so
 * this is not too hard.
 *
 * The file operator itself can generate three major file types:
 *
 *      Regular filing system files
 *      Standard I/O channels (called %stdin, %stdout and %stderr)
 *      Peculiar temporary files called %lineedit and %statementedit
 *
 * %lineedit and %statementedit are files generated by the file
 * operator, rather than merely opened for reading or writing.
 *
 * %lineedit generates a temporary file containing the text that the
 * user types in on the %stdin stream for one line only.
 * The %stdin is required to be edited while this is going on, which
 * means that it must be in cooked mode (-cbreak). %stdin is normally
 * in cbreak mode.
 *
 * %statementedit is similar to %lineedit (in that it generates a
 * temporary file), but the input process continues until PostScript
 * sees a complete expression, with "(" matching ")" and "{" matching
 * "}". "[" is not required to match "]", as these are not actually
 * parsed differently from operators (except for their delimiting
 * properties).
 *
 * There are essentially two ways of implementing these temporary files:
 *
 *      o       We could use real files, and copy user input into
 *              them, allowing the PostScript interpreter to just read
 *              from them.
 *      o       We could provide some kind of cache using a buffer in
 *              main memory.
 *
 * The second alternative is likely to reduce certain overheads and,
 * since the code for parsing from strings is needed anyway (token is
 * defined to parse from strings), there is little extra effort in
 * doing things this way.
 *
 * For this reason, files under this implementation are typed. We have:
 *
 *      StreamFiles     which are conventional stdio FILE *s.
 *      StringFiles     which are pointers in strings.
 *
 * StringFiles are used internally by token to parse from strings, to
 * avoid duplicating the entire parser. These file types are defined
 * with an enumerated type.
 *
 * It is defined that closing the %stdin file have no effect other
 * than resetting an end-of-file flag. It is not defined whether the
 * output files are required to be similarly permanent. We assume they
 * are, and introduce a feature of files which is that they may be
 * PERMANENT. This is a flag. It is also not clear if Readable and
 * Writeable access to files is an Access Restriction, as such. We
 * assume so.
 *
 */

extern int errno;
extern void HardUpdate ();

static Object StdIn, Terminal, StdOut, StdErr, LineEdit;
Object StatementEdit, Fstdin, Fterminal, Fstdout, Fstderr;

FILE *popen ();

static int Token (), Eq ();

static int PFile (), PCloseFile (), PRead (), PReadHexString (), PReadLine ();
static int PReadString (), PBytesAvailable (), PWrite (), PWriteHexString ();
static int PWriteString (), PFlush ();
static int PFlushFile (), PStatus (), PCurFile (), PPrint (), PEcho ();

static Echo ();
static int PseudoFile (), EditLine (), EditStatement ();

int ExecFile ();

InitFile ()
{
  TypeInstallOp (File, "token", Token,  1, 2, 0, 0, File);
  TypeInstallOp (File, "exec",  ExecFile,       1, 0, 0, 0, File);
  TypeInstallOp (File, "eq",    Eq,     2, 1, 0, 0, File, File);

  Fstdin  = ReadOnly (FileFrom (stdin));   Fstdin.flags |= PERMANENT;
  Fterminal  = FileFrom (fopen ("/dev/tty", "r"));   Fterminal.flags |= PERMANENT;
  Fstdout = WriteOnly (FileFrom (stdout)); Fstdout.flags |= PERMANENT;
  Fstderr = WriteOnly (FileFrom (stderr)); Fstderr.flags |= PERMANENT;

  StdIn                 = StringFrom ("%stdin");
  StdOut                = StringFrom ("%stdout");
  StdErr                = StringFrom ("%stderr");
  Terminal              = StringFrom ("%terminal");
  StatementEdit         = StringFrom ("%statementedit");
  LineEdit              = StringFrom ("%lineedit");

  InstallOp ("file",            PFile,          2, 1, 0, 0, String, String);
  InstallOp ("closefile",       PCloseFile,     1, 0, 0, 0, File);
  InstallOp ("read",            PRead,          1, 1, 0, 0, File);
```



```c
  InstallOp ("readhexstring",   PReadHexString, 2, 1, 0, 0, File, String);
  InstallOp ("readline",        PReadLine,      2, 1, 0, 0, File, String);
  InstallOp ("readstring",      PReadString,    2, 1, 0, 0, File, String);
  InstallOp ("bytesavailable",  PBytesAvailable,1, 1, 0, 0, File);
  InstallOp ("write",           PWrite,         2, 0, 0, 0, File, Integer);
  InstallOp ("writehexstring",  PWriteHexString,2, 0, 0, 0, File, String);
  InstallOp ("writestring",     PWriteString,   2, 0, 0, 0, File, String);
  InstallOp ("flush",           PFlush,         0, 0, 0, 0);
  InstallOp ("flushfile",       PFlushFile,     1, 0, 0, 0, File);
  InstallOp ("status",          PStatus,        1, 1, 0, 0, File);
  InstallOp ("currentfile",     PCurFile,       0, 1, 0, 0);
  InstallOp ("print",           PPrint,         1, 0, 0, 0, String);
  InstallOp ("echo",            PEcho,          1, 0, 0, 0, Bool);

  Cbreak (TRUE);
}

static int Token (f)
     Object f;
{
  Object res;
  res = Parse (f);
  if (TypeOf (res) == Condition)
    return Error (PSyntaxError);
  else if (TypeOf (res) == Bool)
    VOID Push (OpStack, False);
  else if (TypeOf (res) != Null) {
    VOID Push (OpStack, res);
    VOID Push (OpStack, True);
  }
  else
    return Error (PUnResult);
  return TRUE;
}

#define Body(I) ((I).u.File)

#ifndef Body
static struct file_struct *Body (item)
     Object item;
{
  return item.u.File;
}
#endif

int getchbuf;

#ifdef notdef
struct file_struct *BodyFile (item)
     Object item;
{
  return item.u.File;
}

static int Getc (fp)
     FILE *fp;
{
  int c;

  for (;;) {
    errno = 0;
    if ((c = getc (fp)) != EOF)
      return c;
    /*          if (errno == 0 || Interrupted ())*/
      return EOF;
  }
}
#endif

#define Getc(a) getc(a)

/* DCJ: This function not needed with #define right below.
static int Getchar ()
 {
        return Getc (stdin);
 }
*/

#define Getchar getchar

/* DCJ: This function not needed with #define right below.
static int Status (file) Object file;
 {
        return Body (file)->available;
 }
*/

#define Status(f) (Body(f)->available)

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) == Body (b)));
}

/*
 *      FileFrom generates a file object from an existing file pointer
 */
Object FileFrom (fp)
     FILE *fp;
{
  Object res;
  struct file_struct *f;

  MakeObject (res, File);
  f = res.u.File = (struct file_struct *) Malloc (sizeof (struct file_struct));
  f->file_type = StreamFile;
  f->f.f_ptr = fp;
  f->available = 1;

  return res;
}

/*
 *      FileString is used to generate a file-like object which actually
 *      reads from a string. This is used by "file" and "token" when parsing
 *      strings.
 */
Object FileString (o)
     Object o;
{
  Object res;
  struct file_struct *f;

  MakeObject (res, File);
  f = res.u.File = (struct file_struct *) Malloc (sizeof (struct file_struct));
  f->file_type  = StringFile;
  f->f.c_ptr    = BodyString (o);
```

```
  f->available  = lengthString (o);

  return res;
}

int GeneralGetch (file)
     Object file;
{
  int c;

  if (!Status (file))
    return EOF;
  switch (Body (file)->file_type) {
  default:
    PanicIf (TRUE, "unknown file type in Getch\n");
    return 0; /* shuts lint up */

  case StreamFile:
    c = Getc (Body (file)->f.f_ptr);
    if (c == EOF) {
      Body (file)->available = 0;
      Close (file);
    }
    return c;

  case StringFile:
    if (Body (file)->available == 0)
      return EOF;
    --Body (file)->available;
    c = *Body (file)->f.c_ptr++;
    return c;
  }
}

Ungetch (o, c)
     Object o;
     unsigned char c;
{
  struct file_struct *f = Body (o);

  ++f->available;
  switch (f->file_type) {
  case StringFile: f->f.c_ptr--; break;
  case StreamFile: VOID ungetc (c, f->f.f_ptr); break;

  default:
    Panic ("Don't know how to Ungetch to this file\n");
  }
}

int ExecFile (item)
     Object item;
{
  Object res;
  res = Parse (item);
  if (TypeOf (res) == Bool) {
    if (BodyBool (res))
      return Error (PSyntaxError);
  }
  else if (TypeOf (res) == Condition)
    return Error (PSyntaxError);
  else if (TypeOf (res) == Null)
    return Error (PUnResult);
  else if (TypeOf (res) == Name && xCheck (res)) {
    /*Object binding;
     *
     * binding = Load (res);
     *
     * if (TypeOf (binding) == Operator)
     * res = binding;*/
    VOID Push (ExecStack, item);
    VOID Push (ExecStack, res);
  }
  else {
    VOID Push (ExecStack, item);
    VOID Push (OpStack, res);
  }
  return TRUE;
}

Cleanup ()
{
  Echo (TRUE);
  Cbreak (FALSE);
}

/*static cbreak = FALSE;*/
static  echoing = TRUE;

/*ARGSUSED*/
Cbreak (cond)
     int cond;
{
/*      struct sgttyb s;

        if (cond == cbreak)
                return;
        ioctl (0, TIOCGETP, &s);
        cbreak = cond;
        if (cbreak)
                s.sg_flags |= CBREAK;
        else
                s.sg_flags &= ~CBREAK;
        ioctl (0, TIOCSETP, &s);
*/ }

static Echo (cond)
     int cond;
{
#ifndef MACINTOSH
  struct sgttyb s;

  if (cond == echoing)
    return;
  ioctl (0, TIOCGETP, &s);
  echoing = cond;
  if (echoing)
    s.sg_flags |= ECHO;
  else
    s.sg_flags &= ~ECHO;
  ioctl (0, TIOCSETP, &s);
#endif
}

static FILE *open_pipes [32];/* ICK! */
static int n_open_pipes = 0;

FILE *Fopen (name, mode)
```



```
     unsigned char *name, *mode;
{
#ifndef MACINTOSH
  if (name[0] == '|') {
    FILE *fp = popen (name + 1, mode);
    int i;

    for (i = 0; i < n_open_pipes && open_pipes [i] != NULL; i++)
      ;
    open_pipes [i] = fp;
    if (i == n_open_pipes)
      ++n_open_pipes;
    return fp;
  }
  else
#endif
    return fopen (name, mode);
}

void Fclose (fp)
     FILE *fp;
{
  int i;

#ifndef MACINTOSH
  for (i = 0; i < n_open_pipes; i++)
    if (open_pipes [i] == fp) {
      pclose (fp);
      open_pipes [i] = NULL;
    }
#endif
  fclose (fp);
}

static int PFile (string1, string2)
     Object string1, string2;
{
  int mode;

  if (lengthString (string1) == 0
      || lengthString (string2) != 1
      || (mode = getString (string2, 0)) != 'r' && mode != 'w')
    return Error (PInvFileAccess);
  else if (getString (string1, 0) == '%')
    return PseudoFile (string1, mode);
  else if (lengthString (string1) > MAXFILENAME)
    return Error (PUnFilename);
  else {
    unsigned char name[MAXFILENAME+1], modes[2];
    FILE *fp;

    modes [0] = mode; modes [1] = '\0';
    VOID Bcopy (name, BodyString (string1), lengthString (string1));
    name [lengthString (string1)] = '\0';
    if (fp = Fopen (name, modes))
      if (mode == 'r')
        return Push (OpStack, ReadOnly (FileFrom (fp)));
      else
        return Push (OpStack, WriteOnly (FileFrom (fp)));

    return Error (PUnFilename);
  }
}

static int PseudoFile (string1, mode)
     Object string1;
     int mode;
{
  if (EqString (string1, StdIn))
    return mode == 'r' ? Push (OpStack, Fstdin):Error (PInvFileAccess);
  else if (EqString (string1, Terminal))
    return Push (OpStack, Fterminal);
  else if (EqString (string1, StdOut))
    return mode == 'w' ? Push (OpStack, Fstdout):Error (PInvFileAccess);
  else if (EqString (string1, StdErr))
    return mode == 'w' ? Push (OpStack, Fstderr):Error (PInvFileAccess);
  else if (EqString (string1, StatementEdit))
    return mode == 'r' ? EditStatement () : Error (PInvFileAccess);
  else if (EqString (string1, LineEdit))
    return mode == 'r' ? EditLine () : Error (PInvFileAccess);
  else
    return Error (PUnFilename);
}

static int EditLine ()
{
  unsigned char buf [BUFSIZE],*t;
  int c, length = 0;

  Cbreak (FALSE);
  do {
    if (length >= BUFSIZE) {
      Cbreak (TRUE);
      return Error (PLimitCheck);
    }
    if ((c = Getchar ()) == EOF)
      break;
    buf [length++] = c;
    if (Interrupted ()) {
      Cbreak (TRUE);
      return FALSE;
    }
  } while (buf [length - 1] != '\n');
  Cbreak (TRUE);
  Bcopy(t = (unsigned char *) Malloc ((unsigned) length),buf,length);
  return Push (OpStack,ReadOnly (FileString (MakeString (t,length))));
}

static int EditStatement ()
{
  unsigned char buf[BUFSIZE],*t;
  int c, length = 0, slevel = 0, blevel = 0, escaped = FALSE;

  Cbreak (FALSE);
  do {
    if (length >= BUFSIZE) {
      Cbreak (TRUE);
      return Error (PLimitCheck);
    }
    if ((c = Getchar ()) == EOF)
      break;
    buf [length++] = c;
    if (Interrupted ()) {
      Cbreak (TRUE);
      return FALSE;
    }
    if (slevel != 0)
      if (escaped)
```

```
        escaped = FALSE;
      else if (c == '(')
        ++slevel;
      else if (c == ')')
        --slevel;
      else if (c == '\\')
        escaped = TRUE;
      else
        ;
    else {
      if (blevel == 0 && c == '}' || c == ')') {
        Cbreak (TRUE);
        return Error (PSyntaxError);
      }
      else if (c == '{')
        ++blevel;
      else if (c == '}')
        --blevel;
      else if (c == '(')
        ++slevel;
    }
  } while (!(slevel == 0 && blevel == 0 && buf [length - 1] == '\n'));
  Cbreak (TRUE);
  Bcopy(t = (unsigned char *) Malloc ((unsigned) length),buf,length);
  return Push (OpStack,ReadOnly (FileString (MakeString (t,length))));
}

static int PCloseFile (file)
     Object file;
{
  if (Status (file))
    Close (file);
  return TRUE;
}

int Close (file)
     Object file;
{
  if (file.flags & PERMANENT)
    return TRUE;
  switch (Body (file)->file_type) {
  default:
    Panic ("Unknown file type in closefile\n");

  case StringFile:
    Body (file)->available = 0;
    Body (file)->f.c_ptr = NULL;
    return TRUE;

  case StreamFile:
    Body (file)->available = 0;
    if (Body (file)->f.f_ptr)
      Fclose (Body (file)->f.f_ptr),
      Body (file)->f.f_ptr = NULL;
    return TRUE;
  }
}

static int PRead (file)
     Object file;
{
  int c = Getch (file);

  if (c != EOF)
    return Push (OpStack, MakeInteger (c)), Push (OpStack, True);
  else
    return Push (OpStack, False);
}

static int PReadHexString (file, string)
     Object file, string;
{
  int cc = 0, i, odd = FALSE, l = lengthString (string);

  for (i = 0; i < l; ) {
    int c = Getch (file);

    if (c == EOF)
      break;
    else if (c >= '0' && c <= '9')
      c -= '0';
    else if (c >= 'a' && c <= 'f')
      c -= 'a' - 10;
    else if (c >= 'A' && c <= 'F')
      c -= 'A' - 10;
    else
      continue;
    if (odd)
      putString (string, i++, (cc << 4) | c);
    else
      cc =  c;
    odd = !odd;
  }
  VOID Push (OpStack, getIString (string, 0, i));
  VOID Push (OpStack, MakeBool (Status (file)));

  return TRUE;
}

static int PReadLine (file, string)
     Object file, string;
{
  int i, c = 0, l = lengthString (string);

  for (i = 0; (c = Getch (file)) != EOF && i < l && c != '\n'; i++)
    putString (string, i, c);
  if (c != '\n' && c != EOF)
    return Error (PRangeCheck);
  VOID Push (OpStack, getIString (string, 0, i));
  VOID Push (OpStack, MakeBool (c != EOF));

  return TRUE;
}

static int PReadString (file, string)
     Object file, string;
{
  int i, c, l = lengthString (string);

  for (i = 0; i < l && (c = Getch (file)) != EOF; i++)
    putString (string, i, c);
  VOID Push (OpStack, getIString (string, 0, i));
  VOID Push (OpStack, MakeBool (c != EOF));

  return TRUE;
}

/*ARGSUSED*/
```



```
static int PBytesAvailable (file)
     Object file;
{
  return Push (OpStack, MakeInteger (-1));
}

static int PWrite (file, integer)
     Object file, integer;
{
  if (!Status (file))
    return Error (PIOError);
  switch (Body (file)->file_type) {
  default:
    Panic ("unknown file type in write\n");

  case StringFile:
    return Error (PInvFileAccess);

  case StreamFile:
    if (Body (file)->f.f_ptr) {
      VOID fputc (BodyInteger (integer), Body (file)->f.f_ptr);
      return TRUE;
    }
    return Error (PIOError);
  }
}

static int PWriteHexString (file, string)
     Object file, string;
{
  FILE *fp;
  unsigned char *p = BodyString (string);
  int l = lengthString (string);

  if (!Status (file))
    return Error (PIOError);
  switch (Body (file)->file_type) {
  default:
    Panic ("unknown file type in writestring\n");

  case StringFile:
    return Error (PInvFileAccess);

  case StreamFile:
    fp = Body (file)->f.f_ptr;
    while (l--) {
      int c = *p++;
      static unsigned char hex[] = "0123456789abcdef";

      fprintf (fp, "%c%c", hex[c >> 4], hex[c & 0xF]);
    }
    return TRUE;
  }
}

static int PWriteString (file, string)
     Object file, string;
{
  if (!Status (file) || !wCheck (file))
    return Error (PIOError);
  switch (Body (file)->file_type) {
  default:
    Panic ("unknown file type in writestring\n");

  case StringFile:
    return Error (PInvFileAccess);

  case StreamFile:
    fprintf (Body (file)->f.f_ptr,
             "%.*s",
             lengthString (string),
             BodyString (string));
    return TRUE;
  }
}

static int PFlush ()            /* --- */
{
  VOID fflush (stdout);
  HardUpdate ();        /* yes, well ... you didn't see this... */
  return TRUE;
}

static int PFlushFile (file)
     Object file;
{
  if (!Status (file))
    return Error (PIOError);
  else if (wCheck (file))
    switch (Body (file)->file_type) {
    default:
      Panic ("unknown file type in flushfile\n");
      return FALSE; /* shuts lint up */

    case StringFile:    break;

    case StreamFile:
      if (Body (file)->f.f_ptr)
        VOID fflush (Body (file)->f.f_ptr);
      break;
    }
  else if (rCheck (file))
    while (Getch (file) != EOF)
      ;
  return TRUE;
}

static int PStatus (file)
     Object file;
{
  return Push (OpStack, MakeBool (Status (file)));
}

static int PCurFile ()          /* --- file */
{
  int c = CountTo (File, ExecStack);
  Object f;

  PanicIf (c < 0, "currentfile finds no run context");
  f = ExecStack->stack_body[Height (ExecStack) - c - 1];
  return Push (OpStack, Cvlit (f));
}

static int PPrint (string)
     Object string;
{
  int i, l = lengthString (string);
  unsigned char *s = BodyString (string);
```

```
  for (i = 0; i < l; i++)
    putchar (*s++);
  /* PRINTF BUG: printf ("%.*s", lengthString (string), BodyString (string));*/
  return TRUE;
}

static int PEcho (res)
     Object res;
{
  Echo (BodyBool (res));
  return TRUE;
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

/*
 * Trapezoid decomposition algorithm:
 *
 *      Edges are characterised by a line description, with topY,
 *      bottomY, topX, bottomX, startingY
 *      Build edge lists from path.
 *      Sort edges on top Y value.
 *      find topmost interestingY value, which will be the minimum
 *      topY. This becomes interestingY.
 *      loop
 *              examine edge list from here to end. Move edges with
 *              topY value less than interestingY into interesting
 *              edge set, computing intersections for each as you go.
 *              This should establish next interestingY value, nextY.
 *              form the pairings for all edges in the interesting
 *              set, according to the fill rule at interestingY.
 *                      to do this, while inside, keep a pointer to
 *                      the last up-transition edge, and pair it with
 *                      the next down-transition edge.
 *                      for any edges which have changed pairing, emit
 *                      a trapezoid from their startingY to
 *                      interestingY, (if they were paired) and set
 *                      their startingY to interestingY
 *              break if (ninteresting == 0 && nedges_left == 0)
 *              interestingY := nextY
 *      pool
 */

struct edge
{
  int topX, topY, bottomX, bottomY; short dir;
  struct edge *pair;
  int startingY, where, up, clip;
  int name;
};


#define EoRule 1
#define NwRule (~0)

int names = 0;

extern int in_stroke;
extern void SetClipHardware ();

int InitClip ();
static int Clip ();
static int EOClip ();
int Fill ();
static int EOFill ();
static int PClipPath ();
static int NextLowest ();

static struct edge *edge;

InitFill ()
{
  InstallOp ("initclip",        InitClip,       0, 0, 0, 0);
  InstallOp ("clip",            Clip,           0, 0, 0, 0);
  InstallOp ("eoclip",          EOClip,         0, 0, 0, 0);
  InstallOp ("fill",            Fill,           0, 0, 0, 0);
  InstallOp ("eofill",          EOFill,         0, 0, 0, 0);
  InstallOp ("clippath",        PClipPath,      0, 0, 0, 0);
  PanicIf (((edge = (struct edge *) malloc(MAXPATHELEMENTS * sizeof(struct edge))) == NUL
}

static struct hardware *output_device;
static Colour output_colour;

static int FillIt ();

/*
 * BZS - Open Code
 */
#define NotThisBit(EDGE,WHERE,EMITFN) { \
  if((EDGE)->pair != NULL) { \
     (*EMITFN)((EDGE),(EDGE)->pair,(EDGE)->startingY,WHERE); \
     (EDGE)->pair->startingY = WHERE; \
     (EDGE)->pair->where = WHERE; \
     (EDGE)->pair->pair = NULL; \
     (EDGE)->pair = NULL; \
   } \
  (EDGE)->startingY = WHERE; \
  (EDGE)->where = WHERE; \
}

#define ThisBit(LEFT,RIGHT,WHERE,EMITFN) { \
   if((LEFT)->pair != (RIGHT) || (RIGHT)->up) { \
     if((LEFT)->pair != NULL) { \
        (*EMITFN)(LEFT,(LEFT)->pair,(LEFT)->startingY,(LEFT)->where); \
        (LEFT)->pair->startingY = (LEFT)->pair->where; \
        (LEFT)->pair->pair->startingY = (LEFT)->pair->pair->where; \
        (LEFT)->pair->pair = NULL; \
     } \
     if ((RIGHT)->pair != NULL) { \
        (*EMITFN) (RIGHT, (RIGHT)->pair, (RIGHT)->startingY, (RIGHT)->where); \
        (RIGHT)->pair->startingY = (RIGHT)->pair->where; \
        (RIGHT)->pair->pair->startingY = (RIGHT)->pair->pair->where; \
        (RIGHT)->pair->pair = NULL; \
     }  \
     (LEFT)->pair = RIGHT; \
     (RIGHT)->pair = LEFT; \
     (LEFT)->startingY = (RIGHT)->startingY = WHERE; \
   } \
   (LEFT)->where = (RIGHT)->where = WHERE; \
   (LEFT)->up = TRUE; \
   (RIGHT)->up = FALSE; \
}

#define UpEdge(COUNT_A,COUNT_B,INC_A,INC_B,RULE_A,RULE_B) \
  (((COUNT_B+INC_B)&RULE_A) && !((COUNT_A)&RULE_A) && \
   ((COUNT_A+INC_A)&RULE_A)||((COUNT_A+INC_A)&RULE_A) && \
```

```
   !((COUNT_B)&RULE_B) && ((COUNT_B+INC_B)&RULE_B))

#define DownEdge(COUNT_A,COUNT_B,INC_A,INC_B,RULE_A,RULE_B) \
  (((COUNT_B+INC_B)&RULE_B) && ((COUNT_A)&RULE_A) && \
   !((COUNT_A+INC_A)&RULE_A)||((COUNT_A+INC_A)&RULE_A) && \
   ((COUNT_B)&RULE_B) && !((COUNT_B+INC_B)&RULE_B))

#define Xvalue(AX,AY,BX,BY,CY) \
  ((BX)+((CY)-(BY))*((AX)-(BX))/(float)((AY)-(BY)))

static void EmitTrapezoid (left, right, top, bottom)
     struct edge *left, *right; int top, bottom;
{
  struct edge *temp;

  if (left->topX > right->topX || left->bottomX > right->bottomX)
    temp = left, left = right, right = temp;
  PaintTrapezoid (output_device,
                  NewDevicePoint (left->topX, left->topY),
                  NewDevicePoint (left->bottomX, left->bottomY),
                  NewDevicePoint (right->topX, right->topY),
                  NewDevicePoint (right->bottomX, right->bottomY),
                  top, bottom,
                  output_colour);
}

int Fill ()
{
  int res;

  if (gstate->device->dev == NULL)
    return PNewPath ();

  res = FillPath(gstate->device->dev, gstate->path, 1);

  VOID PNewPath ();

  return res;
}

static int EOFill ()
{
  int res;

  if (gstate->device->dev == NULL)
    return PNewPath ();

  output_device = gstate->device->dev;
  output_colour = gstate->colour;
  res = FillIt (gstate->path, EoRule, gstate->clip, NwRule, EmitTrapezoid);
  /* XXX needs optimising */

  VOID PNewPath ();

  return res;
}

static Path output_path;

static void PathTrapezoid (left, right, top, bottom)
     struct edge *left, *right; int top, bottom;
{
  int ll, ul, lr, ur;
  struct edge *temp;

  if (left->topX > right->topX || left->bottomX > right->bottomX)
    temp = left, left = right, right = temp;

  ll = Xvalue (left->topX, left->topY, left->bottomX, left->bottomY,
               bottom);
  ul = Xvalue (left->topX, left->topY, left->bottomX, left->bottomY, top);
  lr = Xvalue (right->topX, right->topY, right->bottomX, right->bottomY,
               bottom);
  ur = Xvalue (right->topX, right->topY, right->bottomX, right->bottomY,
               top);

  MoveTo (output_path, NewHardPoint ((float) ll, (float) bottom));
  LineTo (output_path, NewHardPoint ((float) lr, (float) bottom));
  LineTo (output_path, NewHardPoint ((float) ur, (float) top));
  LineTo (output_path, NewHardPoint ((float) ul, (float) top));
  ClosePath (output_path);
}

int InitClip ()
{
  PathFree (gstate->clip);
  gstate->clip = PathCopy (gstate->device->default_clip);

  UnlinkDevice (gstate->clipdevice);
  gstate->clipdevice = NULL;

  SetClipHardware (gstate->device->dev, (struct hardware *) NULL);

  return TRUE;
}

static int Clip ()
{
  int res;

  output_path = NewPath ();
  res = FillIt (gstate->path, NwRule, gstate->clip, NwRule, PathTrapezoid);
  /* XXX - needs optimising */

  PathFree (gstate->clip);
  gstate->clip = output_path;

  if (res) {
    gstate->clipdevice = UniqueDevice (gstate->clipdevice);
    SetClipHardware (gstate->device->dev, gstate->clipdevice->dev);

    /*     output_colour = gstate->colour;
     * gstate->colour = NewColour (0.0, 0.0, 0.0);
     * res = FillPath(gstate->clipdevice->dev, gstate->clip, 1);
     */
    output_device = gstate->clipdevice->dev;
    output_colour = NewColour (0.0, 0.0, 0.0);
    res = FillIt (gstate->clip, NwRule, gstate->device->default_clip,
                  NwRule, EmitTrapezoid);
  }
  return res;
}

static int EOClip ()
{
  int res;

  output_path = NewPath ();
```



```
  res = FillIt (gstate->path, EoRule, gstate->clip, NwRule, PathTrapezoid);
  /* XXX - needs optimising */

  PathFree (gstate->clip);
  gstate->clip = output_path;

  if (res) {
    gstate->clipdevice = UniqueDevice (gstate->clipdevice);
    SetClipHardware (gstate->device->dev, gstate->clipdevice->dev);

    output_device = gstate->clipdevice->dev;
    output_colour = NewColour (0.0, 0.0, 0.0);
    res = FillIt (gstate->clip, NwRule, gstate->device->default_clip,
                  NwRule, EmitTrapezoid);
  }
  return res;
}

static int PClipPath ()
{
  PathFree (gstate->path);
  gstate->path = PathCopy (gstate->clip);
  gstate->cp_defined = TRUE;
  gstate->cp = gstate->path->last->pe.point;

  return TRUE;
}

/****************************/

static struct edge *interesting [MAXPATHELEMENTS];

static int nedges, here, ninteresting;

static int interestingY, nextY;

static int infinity;

static int FillIt (path_a, rule_a, path_b, rule_b, emitfn)
     Path path_a, path_b;
     int rule_a, rule_b;
     void (*emitfn)();
{
  static int edgecmp ();
  static void Trapezoids (), BuildEdgeList ();

  if (EmptyPath (path_a) || EmptyPath (path_b))
    return TRUE;

  path_a = FlattenPath (path_a);
  path_b = FlattenPath (path_b);

  if (!CloseAll (path_a) || !CloseAll (path_b)) {
    PathFree (path_a);
    PathFree (path_b);

    return FALSE;
  }

  nedges = 0;
  BuildEdgeList (path_a, TRUE);
  BuildEdgeList (path_b, FALSE);
  qsort ((char *) edge, (unsigned) nedges, sizeof (struct edge), edgecmp);
  PathFree (path_a);
  PathFree (path_b);
  Trapezoids (rule_a, rule_b, emitfn);

  return TRUE;
}

static void AddLowest ();

static void Trapezoids (rule_a, rule_b, emitfn)
     int rule_a, rule_b;
     void (*emitfn)();
{
  static void FindInfinity (), AddInteresting ();

  nextY = edge[0].topY;
  AddLowest (nextY);
  FindInfinity ();

  here = 0;
  ninteresting = 0;
  names = 0;

  while (here != nedges || ninteresting) {
    static void RemoveEdges ();

    interestingY = nextY;
    ProcessEdges (rule_a, rule_b, emitfn);

    AddInteresting ();

    RemoveEdges (interestingY, emitfn);

    ProcessEdges (rule_a, rule_b, emitfn);
  }
}

ProcessEdges (rule_a, rule_b, emitfn)
     int rule_a, rule_b;
     void (*emitfn)();
{
  register struct edge **intp;
  register struct edge *up_edge;
  register int count_a = 0, count_b = 0;
  register int i;

  i = ninteresting;
  intp = interesting;
  while(i-- > 0) {
    register int d_a = 0, d_b = 0;

    if ((*intp)->clip)
      d_a = (*intp)->dir;
    else
      d_b = (*intp)->dir;

    if (UpEdge (count_a, count_b, d_a, d_b, rule_a, rule_b))
      up_edge = *intp;
    else if(DownEdge (count_a, count_b, d_a, d_b, rule_a, rule_b)){
      ThisBit (up_edge,(*intp), interestingY, emitfn);
    }
    else {
      NotThisBit ((*intp),interestingY,emitfn);
    }
    count_a += d_a;
```

```
    count_b += d_b;

    intp++;
  }

  if (count_a || count_b) {
    fprintf (stderr, "count_a = %dcount_b = %d\n", count_a, count_b);
    Panic("something wrong in area fill");
  }
}

/* BZS - Open code */
#ifndef ThisBit
ThisBit (left, right, where, emitfn)
     struct edge *left, *right;
     int where;
     void (*emitfn)();
{
  if (left->pair != right || right->up) {
    if (left->pair != NULL) {
      (*emitfn) (left, left->pair, left->startingY, left->where);
      left->pair->startingY = left->pair->where;
      left->pair->pair->startingY = left->pair->pair->where;
      left->pair->pair = NULL;
    }
    if (right->pair != NULL) {
      (*emitfn) (right, right->pair, right->startingY, right->where);
      right->pair->startingY = right->pair->where;
      right->pair->pair->startingY = right->pair->pair->where;
      right->pair->pair = NULL;
    }

    left->pair = right;
    right->pair = left;
    left->startingY = right->startingY = where;
  }
  left->where = right->where = where;
  left->up = TRUE; right->up = FALSE;
}
#endif

/*
 * BZS - Open Code this, it can be called hundreds of thousands of times
 * in even simple pictures. Mooreforms spend 50% of its time in this.
 */
#ifndef NotThisBit
NotThisBit (edge, where, emitfn)
     struct edge *edge;
     int where;
     void (*emitfn)();
{
  if (edge->pair != NULL) {
    (*emitfn) (edge, edge->pair, edge->startingY, where);
    edge->pair->startingY = where;
    edge->pair->where = where;
    edge->pair->pair = NULL;
    edge->pair = NULL;
  }
  edge->startingY = where;
  edge->where = where;
}
#endif

static void RemoveEdges (interestingY, emitfn)
     int interestingY;
     void (*emitfn)();
{
  register int i, j = 0;

  for (i = 0; i < ninteresting; i++)
    if (interesting [i]->bottomY > interestingY)
      interesting [j++] = interesting [i];
    else
      NotThisBit (interesting[i], interestingY, emitfn);
  ninteresting = j;
}

static float Yintersect (a, b)
     struct edge *a, *b;
{
  float
    num = (a->bottomX * a->topY - a->bottomY * a->topX) * (b->topY - b->bottomY) -
      (b->bottomX * b->topY - b->bottomY * b->topX) * (a->topY - a->bottomY),
      denom = (a->bottomX - a->topX) * (b->topY - b->bottomY) -
        (b->bottomX - b->topX) * (a->topY - a->bottomY);

  if (denom == 0) {
    return infinity;
  }
  return num / denom;
}

#ifndef Xvalue
static int Xvalue (ax, ay, bx, by, cy)
     int ax, ay, bx, by, cy;
{
  return bx + (cy - by) * (ax - bx) / (float) (ay - by);
}
#endif

static int intercmp (aa, bb)
     char *aa, *bb;
{
  register struct edge *a = *(struct edge **) aa, *b = *(struct edge **) bb;
  int sign;

  sign = Xvalue (a->topX, a->topY, a->bottomX, a->bottomY, interestingY + 1) -
    Xvalue (b->topX, b->topY, b->bottomX, b->bottomY, interestingY + 1);
  if (sign == 0)
    return a->bottomX - b->bottomX;
  return sign;
}

static void AddInteresting ()
{
  register int i;
  register struct edge **intp;

  nextY = infinity;
  for (; here < nedges && edge[here].topY <= interestingY; here++) {
    /* look at each new interesting edge */
    register int i, n;

    i = ninteresting;
    intp = interesting;
    while(i-- > 0) { /* look at all possible intersections */
      int inter = Yintersect (&edge[here], (*intp));
```



```
      if (inter >= interestingY &&
          inter <= edge[here].bottomY &&
          inter <= (*intp)->bottomY)
        AddLowest (inter);
      intp++;
    }
    n = ninteresting++;
    intp = &interesting[n];
    *intp = &edge[here];
    (*intp)->pair = NULL;
    (*intp)->up = FALSE;
    (*intp)->startingY = (*intp)->where = edge[here].topY;
    (*intp)->name = names++;
  }
  i = NextLowest (interestingY);
  if (i)
    nextY = i;
  if (here != nedges && edge[here].topY < nextY)
    nextY = edge[here].topY;
  i = ninteresting;
  intp = interesting;
  while(i-- > 0) {
    if ((*intp)->topY > interestingY && (*intp)->topY < nextY)
      nextY = (*intp)->topY;
    if ((*intp)->bottomY > interestingY && (*intp)->bottomY < nextY)
      nextY = (*intp)->bottomY;
    intp++;
  }
  qsort ((char *) interesting, (unsigned) ninteresting,
         sizeof (struct edge *), intercmp);
}

static void FindInfinity ()
{
  register int i;

  infinity = edge[0].topY;
  for (i = 0; i < nedges; i++)
    if (edge[i].bottomY > infinity)
      infinity = edge[i].bottomY;
}

/****************************/

static int edgecmp (a, b)
     char *a, *b;
{
  struct edge *aa = (struct edge *) a, *bb = (struct edge *) b;

  return aa->topY - bb->topY;
}

static int AddEdge (e, from, to, clip)
     struct edge *e;
     HardPoint from, to;
     int clip;
{
  int dir = 1;
  HardPoint temp;

  if ((int) (from.hy) == (int) (to.hy))
    return 0;
  if ((int) (from.hy) > (int) (to.hy)) {
    temp = from; from = to; to = temp;
    dir = -dir;
  }
  e->topX = from.hx;
  e->topY = from.hy;
  e->bottomX = to.hx;
  e->bottomY = to.hy;
  e->dir = dir;
  e->clip = clip;

  return 1;
}

static void BuildEdgeList (path, clip)
     Path path;
     int clip;
{
  register Path p;
  HardPoint move, here;

  for (p = path->next; p != path; p = p->next)
    switch (p->ptype) {
    case EMove: move = here = p->pe.point; break;
    case ELine:
      nedges += AddEdge (&edge[nedges], here, p->pe.point, clip);
      here = p->pe.point;
      break;

    case EClose:
      nedges += AddEdge (&edge[nedges], here, move, clip);
      here = move;
      break;
    }
}

/* keep list of interesting entries to come */

struct lowest
{
  struct lowest *higher;
  int e;
} *lowest_free = NULL, *lowest = NULL;

static int NextLowest (y)
     int y;
{
  int res;
  register struct lowest *p;

  for (p = lowest; p && p->e <= y; p = lowest) {
    /* delete any which are now irrelevent */
    lowest = p->higher;
    p->higher = lowest_free;
    lowest_free = p;
  }

  if (lowest == NULL)
    return 0;
  res = lowest->e;

  return res;
}

static void AddLowest (e)
     int e;
```

```
{
  register struct lowest *res, *p, *q;

  for (p = lowest; p && p->e < e; q = p, p = p->higher)
    ;

  if (p && p->e == e)
    return;
  if (lowest_free == NULL)
    res = (struct lowest *) Malloc (sizeof (struct lowest));
  else
    res = lowest_free, lowest_free = lowest_free->higher;

  res->e = e;
  res->higher = p;
  if (p != lowest)
    q->higher = res;
  else
    lowest = res;
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

#define MOVETO  0x80
#define BYTE    0xFF

#define FONT_USER       3
#define MakeVector(V, X, Y, T) V->vx = X; V->vy = Y; V->vt = T;

/* Also change #define in X11.c */
#define UpdateControl(h, on) {}

static struct show_context show_stack [MAXGSAVES];

int MoveTo (), LineTo ();

static int PCurrentFont ();
static int PDefineFont ();
static int PFindFont ();
static int PScaleFont ();
static int PMakeFont ();
static int PSetFont ();

static int BuildHershey ();
static int PShow ();
static int PWidthShow ();
static int PAShow ();
static int PAWidthShow ();
static int PStringWidth ();
static int PAdjust ();

static Vector *GetMetrics ();
extern Vector NewVector ();

static int ShowString ();
static int ShowStop ();
static int ShowStopped ();

Object Fid, FD, FontMatrix, FontType, Encoding, FontBBox;
Object OpShowStop, OpShowString, OpStopped;
Object UserShow, CharStrings, Metrics, BuildChar, FontName;

static last_fid = 0;

static int Update ();

static HardPoint zeroHardPoint;

extern Matrix HardPointTranslate ();

InitFont ()
{
  Fid           = NameFrom ("FID");
  FontMatrix    = NameFrom ("FontMatrix");
  FontType      = NameFrom ("FontType");
  Encoding      = NameFrom ("Encoding");
  FontBBox      = NameFrom ("FontBBox");
  UserShow      = NameFrom (".show");
  CharStrings   = NameFrom ("CharStrings");
  Metrics       = NameFrom ("Metrics");
  BuildChar     = NameFrom ("BuildChar");
  FontName      = NameFrom ("FontName");

  OpShowStop = MakeOp ("(showstop)",            ShowStop,       1, 0, 0, 0,
                       Bool);
  OpShowString = MakeOp ("(showstring)",        ShowString,     1, 0, 0, 0,
                         String);
  OpStopped = MakeOp ("(showstopped)",          ShowStopped,0, 1, 1, 0);

  InstallOp ("currentfont",     PCurrentFont,   0, 1, 0, 0);
  InstallOp ("definefont",      PDefineFont,    2, 1, 0, 0,
             Poly, Dictionary);
  InstallOp ("findfont",        PFindFont,      1, 1, 0, 0, Poly);
  InstallOp ("scalefont",       PScaleFont,     2, 1, 0, 0,
             Dictionary, Float);
  InstallOp ("makefont",        PMakeFont,      2, 1, 0, 0,
             Dictionary, Array);
  InstallOp ("setfont",         PSetFont,       1, 0, 0, 0,
             Dictionary);

  InstallOp ("show",            PShow,          1, 0, 0, 0, String);
  InstallOp ("widthshow",       PWidthShow,     4, 0, 0, 0,
             Float, Float, Integer, String);
  InstallOp ("ashow",           PAShow,         3, 0, 0, 0,
             Float, Float, String);
  InstallOp ("awidthshow",      PAWidthShow,    6, 0, 0, 0,
             Float, Float, Integer, Float, Float, String);
  InstallOp ("stringwidth",     PStringWidth,   1, 2, 0, 0, String);
  InstallOp (".adjust",         PAdjust,        1, 0, 0, 0, Integer);
  InstallOp ("buildhershey",    BuildHershey,   2, 0, 0, 0,
             Dictionary, Integer);
  InstallOp ("update",          Update,         1, 0, 0, 0, Bool);

  Install ("FontDirectory",     FD = MakeDict (20));

  zeroHardPoint = NewHardPoint (0.0, 0.0);
}

HardPoint DExtToInt (p)
     UserPoint p;
{
  Matrix m;
  HardPoint res;

  m = gstate->CTM;
  res.hx = p.x * m.A + p.y * m.C;
  res.hy = p.x * m.B + p.y * m.D;

  return res;
}

static int update_state = TRUE;

static int Update (bool)
     Object bool;
```

```
{
  int new = BodyBool (bool);

  if (new != update_state)
    UpdateControl (gstate->device->dev, update_state = new);
  return TRUE;
}

InitShowContext ()
{
  gstate->show = show_stack;
}

HardPoint Adjust (basic, code)
     HardPoint basic;
     char code;
{
  if (code == gstate->show->space)
    return MoveHardPoint (MoveHardPoint (gstate->show->shim,
                                         gstate->show->space_shim),
                          basic);
  else
    return MoveHardPoint (gstate->show->shim, basic);
}

static int PAdjust (code)
     Object code;
{
  gstate->CTM = HardPointTranslate (gstate->CTM,
                                    Adjust (ExtToInt (NewUserPoint (0.0,
                                                                    0.0)),
                                            BodyInteger (code)));
  return TRUE;
}

static int PCurrentFont ()
{
  return Push (OpStack, gstate->font);
}

static int PFindFont (key)
     Object key;
{
  Object res;

  res = DictLoad (FD, key);
  if (TypeOf (res) == Condition)
    return Error (PInvFont);
  return Push (OpStack, res);
}

static Object *encodingCache = NULL;
static Matrix fontmatrixCache;
static int fonttypeCache;
int fidCache;
float fontbboxCache[4];

static int PSetFont (font)
     Object font;
{
  Object tmp;

  if (!CheckFont (font))
    return Error (PInvFont);
  gstate->font = font;
  ExtractMatrix (&fontmatrixCache, DictLoad (font, FontMatrix));
  tmp = DictLoad (font, Encoding);
  encodingCache = BodyArray (tmp);
  tmp = DictLoad (font, FontType);
  fonttypeCache = BodyInteger (tmp);
  fidCache = BodyFontID (DictLoad (font, Fid));
  ExtractBBox (fontbboxCache, DictLoad (font, FontBBox));
  return TRUE;
}

int CheckFont (font)
     Object font;
{
  Object mat, bbox, enc, fonttype;
  Matrix dummy;
  float bboxdummy[4];

  mat = DictLoad (font, FontMatrix);
  if (TypeOf (mat) == Condition || !ExtractMatrix (&dummy, mat))
    return FALSE;
  fonttype = DictLoad (font, FontType);
  if (TypeOf (fonttype) != Integer)
    return FALSE;
  bbox = DictLoad (font, FontBBox);
  if (TypeOf (bbox) != Array || lengthArray (bbox) != 4 || !ExtractBBox (bboxdummy, bbox)
    return FALSE;
  enc = DictLoad (font, Encoding);
  if (TypeOf (enc) != Array)
    return FALSE;
  /*
   * DCJ: Since LaserWriter allows smaller arrays, we should
   * too...
   */
  if ( lengthArray (enc) != 256 )
    Message("Warning: Encoding array does not contain 256 elements.");

  switch (BodyInteger (fonttype)) {
  case FONT_USER: return TRUE;

  default: return TRUE;
  }
}

int CheckHershey (font, encoding)
     Object font, *encoding;
{
  Object cs, metrics;
  int i;
  cs = DictLoad (font, CharStrings);
  if (TypeOf (cs) != Dictionary)
    return FALSE;
  metrics = DictLoad (font, Metrics);
  if (TypeOf (metrics) != Dictionary)
    return FALSE;

  for (i = 0; i < 256; i++) {
    Object met;

    met = DictLoad (metrics, encoding[i]);
    if (TypeOf (met) == Array) {
      Object *m = BodyArray (met);
      int j, l = lengthArray (met);
```



```
      if (l != 1 && l != 2 && l != 4)
        return FALSE;

      for (j = 0; j < l; j++)
        if (TypeOf (m[j]) != Integer && TypeOf (m[j]) != Real)
          return FALSE;
    }
    else if (TypeOf (metrics) != Integer && TypeOf (metrics) != Real)
      return FALSE;
  }
  return TRUE;
}

int ExtractBBox (bbox, array)
     float *bbox;
     Object array;
{
  Object num;
  int i;

  if (lengthArray (array) != 4)
    return FALSE;
  for (i = 0; i < 4; i++) {
    num = getArray (array, i);

    if (TypeOf (num) == Integer)
      bbox [i] = BodyInteger (num);
    else if (TypeOf (num) == Real)
      bbox [i] = BodyReal (num);
    else
      return FALSE;
  }
  return TRUE;
}

static Object MakeFontID ()
{
  Object res;

  MakeObject (res, FontID);
  res.u.Font = last_fid++; /* just for uniqueness */

  return res;
}

int BodyFontID (fid)
     Object fid;
{
  return fid.u.Font;
}

static int PDefineFont (key, dict)
     Object key, dict;
{
  if (maxDict (dict) == lengthDict (dict) || maxDict (FD) == lengthDict (FD))
    return Error (PDictFull);

  if (!CheckFont (dict))
    return Error (PInvFont);

  DictStore (FD, key, ReadOnly (dict));
  DictStore (dict, Fid, MakeFontID ());
  return Push (OpStack, ReadOnly (dict));
}

struct font_entry
{
  Object font;
  Matrix mat;
  int fid;
  struct font_entry *font_next;
 } *font_list = NULL;

#define EqFont(m,n) (m.A == n.A && m.B == n.B && m.C == n.C && m.D == n.D)

Object LookupFont (font, matrix)
     Object font;
     Matrix matrix;
{
  Object res, mat, new;
  Matrix m, n;
  struct font_entry *p, *fe;
  int fid = BodyFontID (DictLoad (font, Fid));

  mat = DictLoad (font, FontMatrix);
  ExtractMatrix (&m, mat);
  n = MatMult (matrix, m);

  for (p = font_list; p != NULL; p = p->font_next)
    if (p->fid == fid && EqFont (p->mat, n)) {
      Message ("font matched");
      return p->font;
    }
  Message ("building a font dict");

  copyDict (font, res = MakeDict (maxDict (font)));
  new = MakeArray ((Object *) Malloc ((unsigned) 6 * sizeof (Object)), 6);
  VOID AssignMatrix (new, n);
  DictStore (res, FontMatrix, new);

  res = ReadOnly (res);

  fe = (struct font_entry *) Malloc (sizeof (struct font_entry));
  fe->mat = n;
  fe->font = res;
  fe->font_next = font_list;
  fe->fid = fid;
  font_list = fe;

  return res;
}

static int PScaleFont (font, scale)
     Object font, scale;
{
  Object res;
  float s = BodyReal (scale);
  Matrix m;

  if (!CheckFont (font))
    return Error (PInvFont);
  NewMatrix(m, s, 0.0, 0.0, s, 0.0, 0.0);
  res = LookupFont (font, m);

  return Push (OpStack, res);
}

static int PMakeFont (font, matrix)
```

```
	Object font, matrix;
{
  Object res;
  Matrix m;

  if (!CheckFont (font))
    return Error (PInvFont);
  ExtractMatrix (&m, matrix);
  res = LookupFont (font, m);

  return Push (OpStack, res);
}

static Vector *GetMetrics (v, ob)
	Vector *v;
	Object ob;
{
  if (TypeOf (ob) == Integer || TypeOf (ob) == Real) {
    MakeVector (v, BodyFloat (ob), 0.0, 0.0);
  } else if (TypeOf (ob) != Array) {
    MakeVector (v, 0.0, 0.0, 1.0);
  } else if (lengthArray (ob) == 2) {
    MakeVector (v, BodyFloat (getArray (ob, 1)), 0.0, 0.0);
  } else if (lengthArray (ob) == 4) {
    MakeVector (v, BodyFloat (getArray (ob, 2)),
		BodyFloat (getArray (ob, 3)), 0.0);
  } else {
    MakeVector (v, 0.0, 0.0, 1.0);
  }

  return v;
}
/*
 * BZS - some small changes to allow macrification of DictLoad()
 */
static int BuildHershey (font, code)
	Object font, code;
{
  Vector vtmp, *met;
  Object *bbox, string, nm, tmp;
  unsigned char *s;
  Path p;
  Matrix *mp;
  int (*func)();
  register float a,b,c,d,tx,ty;
  register int x,y, i, l;

  tmp = DictLoad (font, FontBBox);
  bbox  = BodyArray (tmp);
  tmp = DictLoad (font, Encoding);
  nm    = BodyArray (tmp) [BodyInteger (code)];
  /*
   * met        = GetMetrics (DictLoad (DictLoad (font, Metrics), nm));
   */
  tmp = DictLoad(font,Metrics);
  met = GetMetrics(&vtmp, DictLoad(tmp,nm));
  met->vx -= 2; /* hershey bodge - they look better closer */
  /*
   * string     = DictLoad (DictLoad (font, CharStrings), nm);
   */
  tmp = DictLoad(font,CharStrings);
  string = DictLoad(tmp,nm);

  SetCacheDevice (nm, NewUserPoint (met->vx, met->vy),
```

```
		  BodyReal (bbox[0]), BodyReal (bbox[1]),
		  BodyReal (bbox[2]), BodyReal (bbox[3]));

  if (TypeOf (string) != String) return TRUE;
  gstate->line_width = 1.5;
  gstate->line_join = 2;
  gstate->line_cap = 0;
  gstate->dash_length = 0;

  s = BodyString (string);
  l = lengthString (string) / 2;
  p = gstate->path;

  VOID PNewPath ();
  mp = &gstate->CTM;
  a = mp->A; b = mp->B; c = mp->C; d = mp->D; tx = mp->tx; ty = mp->ty;
  for (i = 0; i < l; i++) {
    HardPoint po;
    func = (*s & MOVETO) ? MoveTo : LineTo;
    x = (*s++ & ~MOVETO) - 64;
    y = (*s++ & ~MOVETO) - 73;
    po.hx = x*a + y*c + tx;
    po.hy = x*b + y*d + ty;
    (*func)(p,po);
  }
  return Push (ExecStack, Cvx (NameFrom ("stroke")));
}

/*
	Persistent state important across a BuildChar call-back
		InShow		- the fact that we're in a call back
		gstate		- graphics state
		ccache		- the current cache device
		gstate->device	- whether a cache device has been set or not.
				  The old device if a cache device is
				  installed.
				  The old CTM.
		Width		- current width information - used for
				   moving cp
		awx, awy	- shim adjustment parameters
		scode,
		axcode, aycode	- spacing adjustment parameters

	XXXXXXX		XXXXXX		XXXXXXXXXX	XXXXXXXXX
	XXXXXXX		XXXXXX		XXXXXXXXXX	XXXXXXXXX
	XXXXXXX		XXXXXX		XXXXXXXXXX	XXXXXXXXX
	XXXXXXX		XXXXXX		XXXXXXXXXX	XXXXXXXXX
	XXXXXXX		XXXXXX		XXXXXXXXXX	XXXXXXXXX
	       |_______|      |_______|          |_______|
		   |
		ashow/widthshow width vector

	ShowUser shows the first character (to avoid a leading gap).
			It may have to call buildchar if the character
			is not cached.
	ShowString finishes off a BuildChar (if necessary)
			then shows gaps and characters until it encounters
			one which is not cached.
		It then stacks:
			a stopped context to reset things if an error occurs.
			another ShowString to continue output.
			A buildchar to build a character
*/
```



```
static int Show (string, space, shim, space_shim)
	Object string;
	char space;
	HardPoint shim, space_shim;
{
  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);

  switch (fonttypeCache) {
  case FONT_USER:
    return ShowUser (string, fontmatrixCache, space, shim,
		     space_shim);

  default:
    return Error (PInvFont);
  }
}

static int ShowStopped ()
{
  VOID Pop (ExecStack);
  VOID Push (OpStack, False);
  return TRUE;
}

int ShowUser (string, m, space, shim, space_shim)
	Object string;
	Matrix m;
	char space;
	HardPoint shim, space_shim;
{
  unsigned char *s = BodyString (string);
  int l = lengthString (string);

  if (l == 0)
    return TRUE;

  /* GSave (); */

  ++gstate->show;
  gstate->show->InShow = TRUE;
  gstate->show->shim = shim;
  gstate->show->space_shim = space_shim;
  gstate->show->space = space;
  gstate->show->mat = gstate->CTM;

  gstate->CTM.tx = gstate->cp.hx;
  gstate->CTM.ty = gstate->cp.hy;

  gstate->CTM = MatMult (m, gstate->CTM);
  FindCache ();

  UpdateControl (gstate->device->dev, FALSE);

  gstate->cp.hx = gstate->CTM.tx;
  gstate->cp.hy = gstate->CTM.ty;
  if (CacheShow (encodingCache [*s], gstate->cp))
    return ShowStringAux (string, s, l, gstate->show->Width);

  UpdateControl (gstate->device->dev, TRUE);

  GSave ();
  VOID Push (ExecStack, OpShowStop);
  PStopped (OpShowString);
```

```
  VOID Push (OpStack, string);
  VOID Push (OpStack, gstate->font);
  VOID Push (OpStack, MakeInteger (*s));
  VOID Push (ExecStack, DictLoad (gstate->font, BuildChar));
  return TRUE;
}

static int ShowString (string)
	Object string;
{
  int l = lengthString (string);
  unsigned char *st = BodyString (string);
  UserPoint last_width;

  Pop (ExecStack);
  Pop (ExecStack);
  Pop (ExecStack);
  GRestore ();

  UpdateControl (gstate->device->dev, FALSE);

  last_width = gstate->show->Width;
  CacheShow (encodingCache [*st], gstate->cp);

  return ShowStringAux (string, st, l, last_width);
}

ShowStringAux (string, st, l, last_width)
	Object string;
	unsigned char *st;
	int l;
	UserPoint last_width;
{
  unsigned char *s = st;
  HardPoint loc;

  loc.hx = gstate->CTM.tx; loc.hy = gstate->CTM.ty;
  for (;;) {
    int code = *++s;
    HardPoint offset;

    offset = Adjust (DExtToInt (last_width), code);
    loc.hx += offset.hx;
    loc.hy += offset.hy;
    if (--l == 0)
      break;
    if (!CacheShow (encodingCache [code], loc)) {
      gstate->cp = loc;
      gstate->CTM.tx = loc.hx; gstate->CTM.ty = loc.hy;

      VOID Push (ExecStack, OpShowStop);
      PStopped (OpShowString);

      VOID Push (OpStack, getIString (string, s - st, l));

      VOID Push (OpStack, gstate->font);
      VOID Push (OpStack, MakeInteger (*s));
      VOID Push (ExecStack, DictLoad (gstate->font,
				      BuildChar));
      UpdateControl (gstate->device->dev, TRUE);

      GSave ();

      return TRUE;
```

```
    }
    last_width = gstate->show->Width;
  }
  gstate->CTM.tx = loc.hx; gstate->CTM.ty = loc.hy;
  UpdateControl (gstate->device->dev, TRUE);

  return ShowStopAux ();
}

static int ShowStop (st)
     Object st;
{
  GRestore ();
  ShowStopAux ();
  if (BodyBool (st))
    return PStop ();
  return TRUE;
}

ShowStopAux ()
{
  HardPoint cp;

  cp.hx = gstate->CTM.tx;
  cp.hy = gstate->CTM.ty;
  /* GRestore (); */
  gstate->CTM = gstate->show->mat;
  --gstate->show;
  gstate->cp = cp;

  return TRUE;
}

static int PShow (string)
     Object string;
{
  return Show (string, 0, zeroHardPoint, zeroHardPoint);
}

static int PWidthShow (x, y, code, string)
     Object x, y, code, string;
{
  return Show (string, BodyInteger (code), zeroHardPoint,
               DExtToInt (NewUserPoint (BodyReal (x), BodyReal (y))));
}

static int PAShow (ax, ay, string)
     Object ax, ay, string;
{
  return Show (string, 0, DExtToInt (NewUserPoint (BodyReal (ax),
                                                   BodyReal (ay))),
               zeroHardPoint);
}

static int PAWidthShow (x, y, code, ax, ay, string)
     Object x, y, code, ax, ay, string;
{
  return Show (string,
               BodyInteger (code),
               DExtToInt (NewUserPoint (BodyReal (x), BodyReal (y))),
               DExtToInt (NewUserPoint (BodyReal (ax), BodyReal (ay))));
}

static int PStringWidth (string)
     Object string;
{
  Matrix m;
  Vector *v, vtmp;
  Object *enc, met, tmp;
  int i, l = lengthString (string);
  unsigned char *s = BodyString (string);
  float x = 0, y = 0;

  if (!CheckFont (gstate->font))
    return Error (PInvFont);
  tmp = DictLoad (gstate->font, FontType);
  if (BodyInteger (tmp) == 3) {
    VOID Push (OpStack, string);
    VOID Push (ExecStack, Cvx (NameFrom (".stringwidth")));
    return TRUE;
  }
  tmp = DictLoad (gstate->font, Encoding);
  enc = BodyArray (tmp);
  met = DictLoad (gstate->font, Metrics);
  for (i = 0; i < l; i++) {
    v = GetMetrics (&vtmp, DictLoad (met, enc[s[i]]));
    x += v->vx;
    y += v->vy;
  }
  ExtractMatrix (&m, DictLoad (gstate->font, FontMatrix));
  vtmp = Transform (NewVector (x, y, 0.0), m);
  return Push (OpStack, MakeReal (vtmp.vx)), Push (OpStack,
                                                   MakeReal (vtmp.vy));
}
```



```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */


extern Object Fid, FD, FontMatrix, FontType, Encoding, FontBBox, FontName;
extern Object UserShow, CharStrings, Metrics;
```

```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#include "device.h"
#include "point.h"
#include "font.h"

#ifndef PI
#define PI (float) 3.14159265358979
#endif

#define MAXDASH 11

#define MAXGSAVES       20

#define BBOX_LEFT       0
#define BBOX_TOP        1
#define BBOX_RIGHT      2
#define BBOX_BOTTOM     3

struct show_context
{
  char InShow;
  char space;
  UserPoint Width;
  HardPoint shim, space_shim;
  Object CharName;
  struct cache *ccache;
  Matrix mat;
};

struct state
{
  Matrix CTM;
  Colour colour;
  HardPoint cp; int cp_defined;
  Path path;
  Path clip;
  Object font;
  float line_width;
  int line_cap;
  int line_join;
  struct {
    float frequency, rotation, *thresh;
    int count;
    Object spot_function;
  } screen;
  struct {
    Object transfn;
    float *tran;
    int tcount;
  } transfer;
  int flatness;
  float miter_limit;
  float dash_offset, dash_array [MAXDASH];
  int dash_length;
  struct device *device;
  struct show_context *show;
  struct device *clipdevice;
};

extern struct state *gstate;
extern int stroke_method, fill_method;
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

static struct state gstate_stack [MAXGSAVES];

struct state *gstate = gstate_stack;

static int gstate_height = 0;

int GSave (), GRestore (), GRestoreAll ();

InitGSave ()
{
  InstallOp ("gsave",           GSave,          0, 0, 0, 0);
  InstallOp ("grestore",        GRestore,       0, 0, 0, 0);
  InstallOp ("grestoreall",     GRestoreAll,    0, 0, 0, 0);

  gstate->screen.count = 0;     /* stops thresh getting freed accidentally */
  gstate->transfer.tcount = 0;  /* stops tran getting freed accidentally */
}

int GSave ()
{
  if (gstate_height == MAXGSAVES - 1)
    return Error (PLimitCheck);
  gstate_stack [gstate_height + 1] = gstate_stack [gstate_height];
  ++gstate_height; ++gstate;
  gstate->path = PathCopy (gstate->path);
  gstate->clip = PathCopy (gstate->clip);
  LinkDevice (gstate->device);
  LinkDevice (gstate->clipdevice);
  ++gstate->screen.count;
  ++gstate->transfer.tcount;

  return TRUE;
}

int GRestore ()
{
  if (gstate_height == 0)
    VOID InitGraphics ();
  else {
    int sflag = FALSE, tflag = FALSE;

    UnlinkDevice (gstate->device);
    UnlinkDevice (gstate->clipdevice);
    PathFree (gstate->path);
    PathFree (gstate->clip);

    if (gstate->screen.count == 1) {
      Free ((char *) gstate->screen.thresh);
      sflag = TRUE;
    }
    if (gstate->transfer.tcount == 1) {
      Free ((char *) gstate->transfer.tran);
      tflag = TRUE;
    }
    --gstate_height;
    --gstate;
    if (sflag)
      SetScreen (gstate->screen.frequency, gstate->screen.rotation, gstate->screen.thresh
    if (tflag)
      SetTransfer (gstate->transfer.tran);
    (void) SetClipHardware (gstate->device->dev,
                            (gstate->clipdevice ? gstate->clipdevice->dev : (struct hardw
  }

  return TRUE;
}

int GRestoreAll ()
{
  while (gstate != gstate_stack)
    VOID GRestore ();
  InitGraphics ();

  return TRUE;
}
```

```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#include "mat.h"
#include "rop.h"
#include "colour.h"
#include "path.h"

typedef struct device_point { int dx, dy; } DevicePoint;

struct hardware
{
  union hard {
    char *addr;
    int handle;
  } hard;
  int flags;
  DevicePoint extent;
  HardPoint realsize;
  struct hardware *aux, *clip;
  short touched;
  short changed;
};

extern struct hardware *HardwareFromString (), *NewBitmapHardware ();
extern struct hardware *NewWindowHardware (), *InitHardware ();
extern DevicePoint HardwareExtent ();
extern unsigned char *StringFromHardware ();

extern void BitBlt (), BitBltBlob (), BitBltLine (), DestroyHardware ();
extern void HardUpdate (), UpdateControl ();
extern Matrix DeviceMatrix ();

extern void RasterTile (), BitBltTrapezoid ();

extern int IsWindowHardware (), TransferSize ();
extern void SetTransfer ();
extern void Paint (), PaintLine (), PaintTrapezoid ();
extern int ScreenSize ();
extern void BuildScreen (), SetScreen ();
extern DevicePoint NewDevicePoint ();
```



```
#define icon_width 40
#define icon_height 40
static char icon_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x00, 0x44, 0x00, 0x00, 0x00,
   0x00, 0x44, 0x80, 0x18, 0x00, 0x20, 0x44, 0x80, 0x24, 0x40, 0x20, 0x3c,
   0xc0, 0x05, 0x00, 0x70, 0x84, 0xb1, 0xc4, 0x59, 0x27, 0x44, 0x8a, 0x48,
   0x55, 0x29, 0x44, 0xb2, 0x50, 0x44, 0x29, 0x44, 0xa2, 0x50, 0x44, 0x29,
   0x84, 0x99, 0x92, 0x45, 0x27, 0x0e, 0x00, 0x0c, 0x00, 0x01, 0x00, 0x00,
   0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
   0x01, 0x00, 0x00, 0x00, 0x00, 0x01, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x22,
   0x00, 0x00, 0x00, 0x00, 0x22, 0x00, 0x00, 0x00, 0x00, 0x22, 0x00, 0x04,
   0x00, 0x00, 0x1e, 0x00, 0x00, 0x00, 0x00, 0x62, 0x17, 0x75, 0x49, 0x67,
   0x52, 0xa5, 0x54, 0x55, 0x55, 0x12, 0xa3, 0x34, 0x55, 0x13, 0x12, 0x41,
   0x14, 0x22, 0x11, 0x12, 0x46, 0x64, 0x22, 0x16, 0x07, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x18, 0x00, 0x00, 0x12, 0x9d,
   0x13, 0x00, 0x00, 0xa2, 0xb4, 0x10, 0x00, 0x00, 0x42, 0xa4, 0x13, 0x00,
   0x00, 0xa2, 0x34, 0x12, 0x00, 0x00, 0x16, 0x9d, 0x1b, 0x00, 0x00, 0x00,
   0x04, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00};
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

static int RealImage ();
static int RealImageMask ();

struct hardware *ScaleBitmap ();

InitImage ()
{
  InstallOp ("realimage",       RealImage,      5, 0, 0, 0,
             Integer, Integer, Integer, Array, String);
  InstallOp ("realimagemask",   RealImageMask,  5, 0, 0, 0,
             Integer, Integer, Bool, Array, String);
}

static int RealImageMask (width, height, invert, mat, string)
     Object width, height, invert, mat, string;
{
  Matrix m, m2;
  Vector origin, right, bottom;
  int w = BodyInteger (width), h = BodyInteger (height);
  int inv = BodyBool (invert), l = lengthString (string);
  int dwidth, dheight;
  unsigned char *s = BodyString (string);

  VOID GSave ();
  if (!ExtractMatrix (&m, mat))
    return Error (PTypeCheck);
  if (l != (w + 7) / 8 * h || w < 0 || h < 0)
    return Error (PRangeCheck);
  if (w == 0 || h == 0)
    return TRUE;
  m = MatInvert (m);
  gstate->CTM = m2 = MatMult (m, gstate->CTM);
  origin = Transform (NewVector (0.0, 0.0, 1.0), m2);
  right  = Transform (NewVector ((float) w, 0.0, 1.0), m2);
  bottom = Transform (NewVector (0.0, (float) h, 1.0), m2);

  /*    fprintf (stderr, "origin = (%g, %g), right = (%g, %g), bottom = (%g, %g)\n", orig
   * fprintf (stderr, "matrix == [%g %g %g %g %g %g]\n", m2.A, m2.B, m2.C, m2.D, m2.tx, m

  dwidth = (int) (right.vx - origin.vx);
  dheight = (int) (bottom.vy - origin.vy);

  if (origin.vx == bottom.vx && origin.vy == right.vy &&
      dwidth > 0 && dheight > 0/* && dwidth >= w && dheight >= h*/) {
    /* simple orientation */
    struct hardware *from, *to;

    from = HardwareFromString (s, w, h);
```

```
    if (!inv)
      BitBlt (from, from,
              NewDevicePoint (0, 0), NewDevicePoint (0, 0),
              HardwareExtent (from), ROP_NOTSOURCE);
    if (dwidth != w || dheight != h) {
      to = ScaleBitmap (from, (float) dwidth / w,
                        (float) dheight / h);
      DestroyHardware (from);
    }
    else
      to = from;

    Paint (to, gstate->device->dev,
           NewDevicePoint (0, 0),
           NewDevicePoint ((int) origin.vx, (int) origin.vy),
           NewDevicePoint (dwidth, dheight),
           gstate->colour);
    DestroyHardware (to);
  }
  else
    DrawBitmap (s, w, h, inv);
  VOID GRestore ();

  return TRUE;
}

DrawBitmap (s, width, height, inv)
     unsigned char *s;
     int width, height, inv;
{
  int w, h;

  VOID PNewPath ();

  for (h = 0; h < height; h++)
    for (w = 0; w < width; w++)
      if (((s [h * (width >> 3) + (w >> 3)] & (1 << (7 - (w & 0x7)))) != 0) == inv)
        FillUnitBox (w, h);
}

void Tile (t, h)
     struct hardware *t, *h;
{
  DevicePoint xt, xh;
  int i, sw, sh;

  xt = HardwareExtent (t);
  xh = HardwareExtent (h);

  BitBlt (t, h, NewDevicePoint (0, 0), NewDevicePoint (0, 0), xt,
          ROP_SOURCE);

  sw = xt.dx;
  for (i = sw;;) {
    BitBlt (h, h, NewDevicePoint (0, 0), NewDevicePoint (i, 0),
            NewDevicePoint (sw, xt.dy), ROP_SOURCE);
    if (i >= xh.dx)
      break;
    i += sw;
    sw <<= 1;
  }

  sh = xt.dy;
  for (i = sh;;) {
```



```
    BitBlt (h, h, NewDevicePoint (0, 0), NewDevicePoint (0, i),
            NewDevicePoint (xh.dx, sh), ROP_SOURCE);
    if (i >= xh.dy)
      break;
    i += sh;
    sh <<= 1;
  }
}

void FillMask (h, mask, width)
     struct hardware *h;
     char mask;
     int width;
{
  struct hardware *m;

  mask <<= 8 - width;

  m = HardwareFromString (&mask, width, 1);

  Tile (m, h);

  DestroyHardware (m);
}

struct hardware *UnpaddedHardwareFromString (s, w, h)
     unsigned char *s;
     int w, h;
{
  struct hardware *thin;
  struct hardware *res;
  int i;

  if (w & 7 == 0)
    return HardwareFromString (s, w, h);

  thin = HardwareFromString (s, w * h, 1);
  res = NewBitmapHardware (w, h);

  for (i = 0; i < h; i++)
    BitBlt (thin, res, NewDevicePoint (i * w, 0),
            NewDevicePoint (0, i), NewDevicePoint (w, 1),
            ROP_SOURCE);

  DestroyHardware (thin);
  return res;
}

static void VerticalReverseImage (h)
     struct hardware *h;
{
  DevicePoint ex;
  int i, n;

  ex = HardwareExtent (h);

  n = ex.dy;

  for (i = 0; i < n / 2; i++)
    BitBlt (h, h, NewDevicePoint (0, i),
            NewDevicePoint (0, n - i - 1),
            NewDevicePoint (ex.dx, 1), ROP_XOR),
    BitBlt (h, h, NewDevicePoint (0, n - i - 1),
            NewDevicePoint (0, i),
```

```
            NewDevicePoint (ex.dx, 1), ROP_XOR),
    BitBlt (h, h, NewDevicePoint (0, i),
            NewDevicePoint (0, n - i - 1),
            NewDevicePoint (ex.dx, 1), ROP_XOR);
}

static int RealImage (width, height, depth, mat, string)
     Object width, height, depth, mat, string;
{
  Matrix m, m2;
  Vector origin, right, bottom;
  int w = BodyInteger (width), h = BodyInteger (height);
  int vrev = 0, dep = BodyInteger (depth), l = lengthString (string);
  int dwidth, dheight;
  unsigned char *s = BodyString (string);

  VOID GSave ();
  if (!ExtractMatrix (&m, mat))
    return Error (PTypeCheck);
  if (l != (w * dep + 7) / 8 * h || w < 0 || h < 0)
    return Error (PRangeCheck);
  if (w == 0 || h == 0)
    return TRUE;
  m = MatInvert (m);
  gstate->CTM = m2 = MatMult (m, gstate->CTM);
  origin = Transform (NewVector (0.0, 0.0, 1.0), m2);
  right  = Transform (NewVector ((float) w, 0.0, 1.0), m2);
  bottom = Transform (NewVector (0.0, (float) h, 1.0), m2);

  dwidth = (int) (right.vx - origin.vx);
  dheight = (int) (bottom.vy - origin.vy);

  if (dheight < 0)
    origin.vy += (float) dheight, right.vy += (float)dheight,
    ++vrev, dheight = -dheight;

  /* fprintf (stderr, "origin.vx = %g, origin.vy = %g, bottom.vx = %g, right.vy = %g, dwi
   * origin.vx, origin.vy, bottom.vx, right.vy, dwidth, dheight, w, h); */

  if (dep != 16 && (int) origin.vx == (int) bottom.vx && (int) origin.vy == (int) right.v
      dwidth > 0 && dheight > 0/* && dwidth >= w && dheight >= h */) {
    /* simple orientation */
    struct hardware *from , *to;

    if (dep == 1) {
      from = HardwareFromString (s, w, h);
      if (vrev)
        VerticalReverseImage (from);
      if (dwidth != w || dheight != h) {
        to = ScaleBitmap (from, (float) dwidth / w,
                          (float) dheight / h);
        DestroyHardware (from);
      }
      else
        to = from;

      Paint (to, gstate->device->dev,
             NewDevicePoint (0, 0),
             NewDevicePoint ((int) origin.vx,
                             (int) origin.vy),
             NewDevicePoint (dwidth, dheight),
             NewGray (1.0));
      BitBlt (to, to,
              NewDevicePoint (0, 0), NewDevicePoint (0, 0),
```

```
              NewDevicePoint (dwidth, dheight),
              ROP_NOTSOURCE);
      Paint (to, gstate->device->dev,
             NewDevicePoint (0, 0),
             NewDevicePoint ((int) origin.vx,
                             (int) origin.vy),
             NewDevicePoint (dwidth, dheight),
             NewGray (0.0));
      DestroyHardware (to);
    } else {
      struct hardware *mask, *new, *temp;
      int i, j;

      from = HardwareFromString (s, w * dep, h);
      if (vrev)
        VerticalReverseImage (from);
      new = HardwareFromString (s, w, h);
      mask = NewBitmapHardware (w * dep, h);
      temp = NewBitmapHardware (w * dep, h);

      for (i = 0; i < (1<<dep); i++) {
        FillMask (mask, i, dep);

        BitBlt (from, mask, NewDevicePoint (0, 0),
                NewDevicePoint (0, 0),
                HardwareExtent (from), ROP_NXOR);
        BitBlt (NULL, temp, NewDevicePoint (0, 0),
                NewDevicePoint (0, 0),
                HardwareExtent (temp), ROP_TRUE);
        for (j = 0; j < dep; j++)
          BitBlt (mask, temp,
                  NewDevicePoint (j, 0),
                  NewDevicePoint (0, 0),
                  HardwareExtent (mask),
                  ROP_AND);
        for (j = 0; j < w; j++)
          BitBlt (temp, new,
                  NewDevicePoint (j * dep, 0),
                  NewDevicePoint (j, 0),
                  NewDevicePoint (1, h),
                  ROP_SOURCE);
        to = ScaleBitmap (new, (float) dwidth / w,
                          (float) dheight / h);
        Paint (to, gstate->device->dev,
               NewDevicePoint (0, 0),
               NewDevicePoint ((int) origin.vx,
                               (int) origin.vy),
               NewDevicePoint (dwidth, dheight),
               NewGray ((float) i / ((1 << dep)- 1)));
        DestroyHardware (to);
      }
      DestroyHardware (from);
      DestroyHardware (mask);
      DestroyHardware (temp);
      DestroyHardware (new);
    }
  } else {
    if (vrev)
      origin.vy += (float) dheight,
      right.vy += (float) dheight, dheight = -dheight;
    DrawColourBitmap (s, w, h, dep);
  }
  VOID GRestore ();
```

```
  return TRUE;
}

DrawColourBitmap (s, width, height, depth)
     unsigned char *s;
     int width, height, depth;
{
  int w, h, c, i;

  VOID PNewPath ();

  switch(depth) {
    case 16:
      i = 0;
      for (h = 0; h < height; h++)
        for (w = 0; w < width; w++) {
          gstate->colour = NewColour (0.0, 0.0,
              (float) ((s[i] << 8) | s[i+1]) / (1 << 16));
          FillUnitBox(w, h);
          i += 2;
        }
      break;
    case 8:
      for (c = 0; c < 256; c++) {
        gstate->colour = NewColour (0.0, 0.0,(float) c/255);
        i = 0;
        for (h = 0; h < height; h++)
          for (w = 0; w < width; w++)
            if (c == s[i++]) FillUnitBox(w, h);
      }
      break;
    case 4:
      for (c = 0; c < 16; c++) {
        gstate->colour = NewColour (0.0, 0.0,(float) c/15);
        i = 0;
        for (h = 0; h < height; h++) {
          for (w = 0; w < width; w++)
            if(c == ((s[i+(w>>1)] >> ((1-(w&1)<<2))) & 15)) FillUnitBox(w,h);
          i += width>>1;
        }
      }
      break;
    case 2:
      for (c = 0; c < 4; c++) {
        gstate->colour = NewColour (0.0, 0.0,(float) c/3);
        i = 0;
        for (h = 0; h < height; h++) {
          for (w = 0; w < width; w++)
            if (c == ((s[i+(w>>2)] >> ((3-(w&3))<<1)) & 3)) FillUnitBox(w,h);
          i += width>>2;
        }
      }
      break;
    case 1:
      gstate->colour = NewColour (0.0, 0.0, 0.0);
      i = 0;
      for (h = 0; h < height; h++) {
        for (w = 0; w < width; w++)
          if (0 == ((s[i+(w>>3)] >> (7-(w&7))) & 1)) FillUnitBox(w,h);
        i += width>>3;
      }
      break;
  }
}
```



```
FillUnitBox (w, h)
     int w, h;
{
  Matrix *m;
  float a,b,c,d,x,y;
  HardPoint P;

  x = (float) w;  y = (float) h;
  m = &gstate->CTM;
  a = m->A;  b = m->B;  c = m->C;  d = m->D;
  P.hx = x*a + y*c + m->tx;
  P.hy = x*b + y*d + m->ty;  VOID MoveTo (gstate->path, P);
  P.hx += a;  P.hy += b;     VOID LineTo (gstate->path, P);
  P.hx += c;  P.hy += d;     VOID LineTo (gstate->path, P);
  P.hx -= a;  P.hy -= b;     VOID LineTo (gstate->path, P);
  VOID ClosePath (gstate->path);
  VOID Fill ();
}

struct hardware *ScaleBitmap (from, xscale, yscale)
     struct hardware *from;
     float xscale, yscale;
{
  struct hardware *middle, *middle2, *high, *high2;
  int i, w, h, dwidth, dheight, ixscale, iyscale;
  DevicePoint extent;

  extent = HardwareExtent (from);
  w = extent.dx; h = extent.dy;
  dwidth = (int) ((float) w * xscale + 0.5);
  dheight = (int) ((float) h * yscale + 0.5);

  if (dwidth > w) {
    middle = NewBitmapHardware (dwidth, h);
    for (i = 0; i < w; i++)
      BitBlt (from,     middle,
              NewDevicePoint (i, 0),
              NewDevicePoint ((int) (i*xscale), 0),
              NewDevicePoint (1, h),
              ROP_OR);
    middle2 = NewBitmapHardware (dwidth, h);
    ixscale = (int) ((float) dwidth / w + 0.5);
    for (i = 0; i < ixscale + 1; i++)
      BitBlt (middle, middle2,
              NewDevicePoint (0, 0), NewDevicePoint (i, 0),
              NewDevicePoint (dwidth - ixscale + 1, h),
              ROP_OR);
    DestroyHardware (middle);
  } else {
    middle2 = NewBitmapHardware (dwidth, h);
    for (i = 0; i < dwidth; i++)
      BitBlt (from,     middle2,
              NewDevicePoint ((int) (i/xscale), 0),
              NewDevicePoint (i, 0),
              NewDevicePoint (1, h),
              ROP_OR);
  }

  if (dheight > h) {
    high = NewBitmapHardware (dwidth, dheight);
    for (i = 0; i < h; i++)
      BitBlt (middle2, high,
              NewDevicePoint (0, i),
```

```
              NewDevicePoint (0, (int) (i*yscale)),
              NewDevicePoint (dwidth, 1),
              ROP_OR);
    DestroyHardware (middle2);

    high2 = NewBitmapHardware (dwidth, dheight);
    iyscale = (int) ((float) dheight / h + 0.5);
    for (i = 0; i < iyscale + 1; i++)
      BitBlt (high,     high2,
              NewDevicePoint (0, 0), NewDevicePoint (0, i),
              NewDevicePoint (dwidth, dheight - iyscale + 1),
              ROP_OR);
    DestroyHardware (high);
  } else {
    high2 = NewBitmapHardware (dwidth, dheight);
    for (i = 0; i < dheight; i++)
      BitBlt (middle2, high2,
              NewDevicePoint (0, (int) (i/yscale)),
              NewDevicePoint (0, i),
              NewDevicePoint (dwidth, 1),
              ROP_OR);
  }
  return high2;
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include <signal.h>
#include "main.h"

static Object OpFor;

static int For (), PFor (), Eq (), Lt (), Le (), Gt (), Ge (), Not ();
static int And (), Or (), Xor (), BitShift (), Abs (), Add (), Sub ();
static int Mul (), Div (), Mod (), Neg (), Exp (), Sqrt (), Sin ();
static int Cos (), Atan (), Ln (), Log (), Identity (), Cvr (), Cvs ();
static int EqEq ();

static int WordSize, Word2, LowMask;

#define Body(O) (O.u.Integer)

InitInteger ()
{
  unsigned word;

  OpFor = MakeOp ("(forinteger)", For, 0, 1, 5, 6);

  TypeInstallOp (Integer, "cvi",        Identity,       1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "cvr",        Cvr,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "cvs",        Cvs,            2, 1, 0, 0, Integer,
                 String);
  TypeInstallOp (Integer, "==",         EqEq,           1, 0, 0, 0, Integer);
  TypeInstallOp (Integer, "for",        PFor,           4, 0, 0, 6, Integer,
                 Integer, Integer, Array);
  TypeInstallOp (Integer, "eq",         Eq,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "lt",         Lt,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "le",         Le,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "gt",         Gt,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "ge",         Ge,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "not",        Not,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "and",        And,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "or",         Or,             2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "xor",        Xor,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "bitshift",   BitShift,       2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "abs",        Abs,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "add",        Add,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "sub",        Sub,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "mul",        Mul,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "div",        Div,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "mod",        Mod,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "neg",        Neg,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "exp",        Exp,            2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "sqrt",       Sqrt,           1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "sin",        Sin,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "cos",        Cos,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "atan",       Atan,           2, 1, 0, 0, Integer,
                 Integer);
  TypeInstallOp (Integer, "ln",         Ln,             1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "log",        Log,            1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "ceiling",    Identity,       1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "floor",      Identity,       1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "round",      Identity,       1, 1, 0, 0, Integer);
  TypeInstallOp (Integer, "truncate",   Identity,       1, 1, 0, 0, Integer);

  word = -1;
  PanicIf (word != ~0, "Need 2's complement machine! Sorry.");

  for (WordSize = 0; word != 0; WordSize++)
    word >>= 1;
  Word2 = WordSize / 2;
  LowMask = (1<<Word2) - 1;
}

int StrictMul (a, b)
     int a, b;
{
  int atop, abot, btop, bbot, sum, signedp;

  signedp = (a < 0) != (b < 0);
  a = a < 0 ? -a : a;
  b = b < 0 ? -b : b;
  abot = a & LowMask;
  bbot = b & LowMask;
  atop = a >> Word2;
  btop = b >> Word2;
  sum = abot * btop + atop * bbot;
  sum += ((unsigned) (abot * bbot) >> Word2);
  sum = ((unsigned) sum >> Word2) + atop * btop;
  if (sum != 0 || a * b < 0)
    kill (getpid (), SIGFPE);
  return signedp ? -a * b : a * b;
}

int StrictAdd (a, b)
     int a, b;
{
  if ((a < 0) == (b < 0) && (a < 0) != (a + b < 0))
    kill (getpid (), SIGFPE);
  return a + b;
}

Object MakeInteger (i)
     int i;
{
  Object res;
```



```
  MakeObject (res, Integer);
  res.u.Integer = i;

  return res;
}

static Object Make (i)
     int i;
{
  Object res;

  MakeObject (res, Integer);
  res.u.Integer = i;

  return res;
}

static int Cvr (item)
     Object item;
{
  return Push (OpStack, RealInteger (item));
}

static int Cvs (v, string)
     Object v, string;
{
  int length;
  char buf [BUFSIZE];

  VOID sprintf (buf, "%d", BodyInteger (v));
  if ((length = strlen (buf)) > lengthString (string))
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), buf, length);
  return Push (OpStack, getIString (string, 0, length));
}

static int EqEq (v)
     Object v;
{
  printf ("%d", BodyInteger (v));
  return TRUE;
}

Object IntReal (o)
     Object o;
{
  return Make ((int) BodyReal (o));
}

static int PFor (initial, increment, limit, proc)
       Object initial, increment, limit, proc;
{
  VOID Push (ExecStack, Nil);
  VOID Push (ExecStack, increment);
  VOID Push (ExecStack, limit);
  VOID Push (ExecStack, initial);
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpFor);

  return TRUE;
}

static int For ()
{
  Object current, limit, increment, proc;

  proc          = Pop (ExecStack);
  current       = Pop (ExecStack);
  limit         = Pop (ExecStack);
  increment     = Pop (ExecStack);

  if (Body (increment) > 0
      ? Body (current) > Body (limit)
      : Body (current) < Body (limit))
    VOID Pop (ExecStack);
  else {
    VOID Push (ExecStack, increment);
    VOID Push (ExecStack, limit);
    VOID Push (ExecStack, Make (StrictAdd (Body (current), Body (increment))));
    VOID Push (ExecStack, proc);
    VOID Push (ExecStack, OpFor);
    VOID Push (ExecStack, proc);
    VOID Push (OpStack, current);
  }
  return TRUE;
}

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) == Body (b)));
}

static int Lt (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) < Body (b)));
}

static int Le (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) <= Body (b)));
}

static int Gt (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) > Body (b)));
}

static int Ge (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) >= Body (b)));
}

static int Not (integer)
     Object integer;
{
  return Push (OpStack, Make (~ Body (integer)));
}

static int And (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) & Body (b)));
}
```

```c
static int Or (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) | Body (b)));
}

static int Xor (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) ^ Body (b)));
}

static int BitShift (a, b)
     Object a, b;
{
  if (Body (b) > 0)
    return Push (OpStack, Make ((int) ((unsigned) Body (a) << Body (b))));
  else
    return Push (OpStack, Make ((int) ((unsigned) Body (a) >> (-Body (b)))));
}

static int Abs (v)
     Object v;
{
  return Push (OpStack, Make (Body (v) >= 0 ? Body (v) : -Body (v)));
}

static int Add (a, b)
     Object a, b;
{
  return Push (OpStack, Make (StrictAdd (Body (a), Body (b))));
}

static int Sub (a, b)
     Object a, b;
{
  return Push (OpStack, Make (StrictAdd (Body (a), -Body (b))));
}

static int Mul (a, b)
     Object a, b;
{
  return Push (OpStack, Make (StrictMul (Body (a), Body (b))));
}

static int Div (a, b)
     Object a, b;
{
  if (Body (b) == 0)
    return Error (PUnResult);
  return Push (OpStack, MakeReal ((float) Body (a) / (float) Body (b)));
}

static int Mod (a, b)
     Object a, b;
{
  if (Body (b) == 0)
    return Error (PUnResult);
  return Push (OpStack, Make (Body (a) % Body (b)));
}

static int Neg (a)
     Object a;
{
  return Push (OpStack, Make (-Body (a)));
}

static int Sqrt (v)
     Object v;
{
  if (Body (v) < 0)
    return Error (PUnResult);
  return Push (OpStack, MakeReal ((float) sqrt ((double) Body (v))));
}

static int Exp (a, b)
     Object a, b;
{
  return Push (OpStack, MakeReal ((float) pow ((double) Body (a), (double) Body (b))));
}

static int Identity (v)
     Object v;
{
  return Push (OpStack, v);
}

static int Sin (v)
     Object v;
{
  return Push (OpStack, MakeReal ((float) sin (Rad ((double) Body (v)))));
}

static int Cos (v)
     Object v;
{
  return Push (OpStack, MakeReal ((float) cos (Rad ((double) Body (v)))));
}

static int Atan (a, b)
     Object a, b;
{
  return Push (OpStack, MakeReal ((float) Deg (atan2 ((double) Body (a), (double) Body (b
}

static int Ln (v)
     Object v;
{
  return Push (OpStack, MakeReal ((float) log ((double) Body (v))));
}

static int Log (v)
     Object v;
{
  return Push (OpStack, MakeReal ((float) log10 ((double) Body (v))));
}
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include <signal.h>
#include "main.h"

#define ERRORDICTSIZE    25

Object PDictFull;
Object PDictOverflow,           PInvFont,               PSyntaxError;
Object PDictUnderflow,          PInvRestore,            PTypeCheck;
Object PExecOverflow,           PIOError,               PUndefined;
Object PExecUnderflow,          PLimitCheck,            PUnFilename;
Object PInterrupt,              PNoCurrentPoint,        PUnResult;
Object PInvAccess,              PRangeCheck,            PUnMatched;
Object PInvExit,                POpOverflow,            PUnregistered;
Object PInvFileAccess,          POpUnderflow,           PVMError;

Object Lbracket, Rbracket;
Object OpInterp, ErrorDict;

/* External variables for hashing statistics */
extern int hash_tries, hash_collisions, hash_attempts;

static int ErrorFn ();
static Object InsErr ();
static void InitErrors (), InitArguments (), Interpreter ();
static Object OpError, error_name, execName;

static catchint ();
static int EnableInterrupts ();

char *library = NULL;

Object InitialObject = {READABLE|WRITEABLE};
Object InitialExecObject = {READABLE|WRITEABLE|EXECUTABLE};

static int files_present = 0, interrupted = FALSE, protected = 0;
int interactive = FALSE, verbose = FALSE, RUNSYNC = FALSE, TESTING = FALSE;
FILE *vfp = NULL;

main (argc, argv)
     int argc;
     char **argv;
{
  char profile[BUFSIZE], rcfile [BUFSIZE];
#ifndef MACINTOSH
  char *getenv ();
#endif
  FILE *fp;
  int here;

#ifndef MACINTOSH
  library = getenv ("POSTSCRIPTLIB");
#endif
  if (library == NULL) library = default_library;

  VOID sprintf (rcfile, "%s/psrc", library);

#ifdef MACINTOSH
  vfp = stderr;
#else
  if (close (dup (9)) == 0) {
    verbose = TRUE;
    vfp = fdopen (9, "w");
  }
#endif

  Message ("POSTSCRIPT");

  Init ();
  Message ("Init Errors");
  InitErrors ();
  Message ("Init Arguments");

  VOID Push (ExecStack, Cvx (NameFrom ("executive")));

  here = Height (ExecStack);

  Install ("Library", StringFrom (library));

  InitArguments (argc, argv);
#ifdef MACINTOSH
  interactive = interactive || !files_present;
#else
  interactive = interactive || isatty (0) && !files_present;
#endif
  ReverseStack (ExecStack, Height (ExecStack) - here);

#ifndef MACINTOSH
  VOID strcpy (profile, getenv ("HOME") ? getenv ("HOME") : "/etc");
#endif
  VOID strcat (profile, "/.postscript");

  if (fp = fopen (profile, "r"))
    VOID Push (ExecStack, Cvx (FileFrom (fp)));

  if (!interactive)
    VOID Push (ExecStack,
              Cvx (StringFrom (files_present ?
                                "/executive null def" :
                                "/executive (%stdin) (r) file cvx def")));

  if (TESTING)
    VOID Push (ExecStack,
      Cvx (StringFrom ("/showpage { copypage initgraphics erasepage } def")));

  if ((fp = fopen (rcfile, "r")) == NULL)
    fprintf (stderr, "%s: cannot open %s\n", argv[0], rcfile),
    exit (1);
  else
    VOID Push (ExecStack, Cvx (FileFrom (fp)));
  InstallOp ("enableinterrupts",        EnableInterrupts, 0, 0, 0, 0);
/*      SysDict = ReadOnly (SysDict);   */
  Message ("Entering Interpreter");
  Interpreter ();
}
```

```
Protect ()
{
  ++protected;
}

Unprotect ()
{
  --protected;
}

static int EnableInterrupts ()
{
  if (interactive)
    VOID signal (SIGINT, catchint);
  return TRUE;
}

static void InitArguments (argc, argv)
     int argc;
     char **argv;
{
  int i, command = FALSE;

  for (i = 1; i < argc; i++) {
    FILE *fp;

    if (command) {
      command = FALSE;
      VOID Push (ExecStack, Cvx (StringFrom (argv[i])));
    }
    else if (*argv[i] == '-')
      switch (argv[i][1]) {
      case 'n':
        RUNSYNC = TRUE;
        break;

      case 't':
        TESTING = TRUE;
        break;

      case 'i':
        interactive = TRUE;
        break;

      case 'c':
        ++files_present;
        command = TRUE;
        break;

      case 's': case '\0':
        ++files_present;
        VOID Push (ExecStack, Cvx (Fstdin));
        break;

      default:
        fprintf (stderr, "%s: unknown option '%c'\n", argv[0],
                 argv[i][1]);
        exit (1);
      }
    else {
      ++files_present;
      if (fp = Fopen (argv[i], "r"))
        VOID Push (ExecStack, Cvx (FileFrom (fp)));
      else
        fprintf (stderr, "%s: cannot open %s\n", argv[0],
                 argv[i]);
    }
  }
}

static void Interpreter ()
{
  register Stack ExecSt,OpSt,DictSt;

  Self = execName = NameFrom ("exec");
  ExecSt = ExecStack;
  OpSt = OpStack;
  DictSt = DictStack;

  while (Height (ExecSt) != 0) {
    int res;
    Object item, exop;
    Type itype;

    item = Pop (ExecSt);
    if (!xCheck (item))
      res = Push (OpSt, item);
    else if ((itype = TypeOf (item)) == Operator)
      /* Marginally slower when inlined... */
      res = ExecOperator(item);
    else if (itype == Array) {
      /* Arrays get special treatment for speed */
      int l = lengthArray (item);
      register Object *rp;
      Type t;

      if (l != 0) {
        if(ExecSt->stack_fill != ExecSt->stack_size) {
          rp = &ExecSt->stack_body[ExecSt->stack_fill++];
          rp->type = Array;
          rp->u.Integer = 0;
          rp->u.Array = item.u.Array + 1;
          rp->Length = l - 1;
          rp->flags = item.flags;
        }

        rp = BodyArray(item);
        t = TypeOf(*rp);
        if(t == Name || t == Operator) Push (ExecSt, *rp);
        else                           Push (OpSt, *rp);
      }
      res = TRUE;
    }
    else if (itype == Name) {
      /* Names get special treatment for speed */
      Type dict;
      struct dict_entry *hash;
      int level,i,h,size,hashval;

      /* Try to load the object from the dictionary */
      /* Initialize information based solely on the item */
      hashval = item.Length + BodyInteger(item) + (int) itype;

      /* Search through the levels of the dict stack from the top down */
      for (level = Height (DictSt) - 1; level >= 0; level--) {
        dict = BodyDict(DictSt->stack_body[level]);
        if ((size=dict->dict_size) != 0) {
          hash = dict->dict_body;
```



```
          ++hash_attempts;
          i = h = hashval % size;

          for(;;) {
            if (TypeOf (hash[i].entry_key) == Null) break;
            if (DictEqual (item, hash[i].entry_key)) {
              ++hash_tries;
              item = hash[i].entry_value;
              level = 0;
              break;
            }
            if (++i == size) i = 0;
            if (i == h) break;
            ++hash_collisions;
          }
        }
      }

      if (!xCheck (item))  res = Push (OpSt, item);
      else if ((itype = TypeOf (item)) == Operator) res = ExecOperator (item);
      else if (itype == Array)  res = ExecArray (item);
      else if (itype == File)   res = ExecFile (item);
      else {
        res = Push (OpSt, item);
        exop = Lookup (TypeOf (item), execName);
        if (TypeOf (exop) != Condition)
          VOID Push (ExecSt, exop);
      }
    }
    else if (itype == File)
      res = ExecFile (item);
    else {
      res = Push (OpSt, item);
      exop = Lookup (TypeOf (item), execName);
      if (TypeOf (exop) != Condition)
        VOID Push (ExecSt, exop);
    }

    if (interrupted && !protected) {
      interrupted = FALSE;
      error_name = PInterrupt;
      res = FALSE;
    }

    if (!res) {
      Object error_op;

      VOID Push (OpSt, Self);
      error_op = DictLoad (ErrorDict, error_name);
      if (TypeOf (error_op) == Condition)
        VOID Push (ExecSt, OpError);
      else
        VOID Push (ExecSt, error_op);
      Cbreak (TRUE);
    }
  }
}

static void InitErrors ()
{
  ErrorDict = MakeDict (ERRORDICTSIZE);
  PDictFull = InsErr (ErrorDict, "dictfull",                    ErrorFn);
  PDictOverflow = InsErr (ErrorDict, "dictstackoverflow",       ErrorFn);
  PDictUnderflow = InsErr (ErrorDict, "dictstackunderflow",     ErrorFn);
  PExecOverflow = InsErr (ErrorDict, "execstackoverflow",       ErrorFn);
  PExecUnderflow = InsErr (ErrorDict, "execstackunderflow",     ErrorFn);
  PInterrupt = InsErr (ErrorDict, "interrupt",                  ErrorFn);
  PInvAccess = InsErr (ErrorDict, "invalidaccess",              ErrorFn);
  PInvExit = InsErr (ErrorDict, "invalidexit",                  ErrorFn);
  PInvFileAccess = InsErr (ErrorDict, "invalidfileaccess",      ErrorFn);
  PInvFont = InsErr (ErrorDict, "invalidfont",                  ErrorFn);
  PInvRestore = InsErr (ErrorDict, "invalidrestore",            ErrorFn);
  PIOError = InsErr (ErrorDict, "ioerror",                      ErrorFn);
  PLimitCheck = InsErr (ErrorDict, "limitcheck",                ErrorFn);
  PNoCurrentPoint = InsErr (ErrorDict, "nocurrentpoint",        ErrorFn);
  PRangeCheck = InsErr (ErrorDict, "rangecheck",                ErrorFn);
  POpOverflow = InsErr (ErrorDict, "stackoverflow",             ErrorFn);
  POpUnderflow = InsErr (ErrorDict, "stackunderflow",           ErrorFn);
  PSyntaxError = InsErr (ErrorDict, "syntaxerror",              ErrorFn);
  PTypeCheck = InsErr (ErrorDict, "typecheck",                  ErrorFn);
  PUndefined = InsErr (ErrorDict, "undefined",                  ErrorFn);
  PUnFilename = InsErr (ErrorDict, "undefinedfilename",         ErrorFn);
  PUnResult     = InsErr (ErrorDict, "undefinedresult",         ErrorFn);
  PUnMatched = InsErr (ErrorDict, "unmatchedmark",              ErrorFn);
  PUnregistered = InsErr (ErrorDict, "unregistered",            ErrorFn);
  PVMError = InsErr (ErrorDict, "VMerror",                      ErrorFn);
  Install ("errordict", ErrorDict);
  OpError = MakeOp ("(errorfn)", ErrorFn, 1, 0, 0, 1, Name);
}

/*
 * This is the code to deal with user interrupts.
 * In order to do this cleanly, we effectively poll for interrupts in
 * the main interpreter loop (above). Possibly any routine which could
 * take an inordinate amount of time out of this loop should also poll
 * the 'interrupted' variable and generate an error if one is found.
 * Currently none do - "run"ning a file is done by the interpreter
 * above.
 *
 */

static catchint ()
{
  VOID signal (SIGINT, catchint);
  interrupted = TRUE;
}

int Interrupted ()
{
  return interrupted && !protected;
}

/*
 *      An intrinsic operator generates an error condition by
 *      returning FALSE.  The usual way to do this is to call Error
 *      with the PostScript name of the error condition.
 *
 */

static Object InsErr (dict, name, fn)
     Object dict;
     unsigned char *name;
     int (*fn)();
{
  Object op;

  DictStore (dict, op = NameFrom (name), MakeOp (name, fn, 1, 0, 0, 1, Name));
  return op;
}
```

```
}

static ErrorFn (name)
     Object name;
{
  PrintName (Self);
  printf (" in operator ");
  PrintName (name);
  putchar ('\n');
  return Push (ExecStack, Cvx (NameFrom ("stop")));
}

int Error (error)
     Object error;
{
  error_name = error;
  return FALSE;
}

int OpCheck (args, res)
     int args, res;
{
  if (OpStack->stack_fill < args)
    return Error (POpUnderflow);
  else if (OpStack->stack_fill - args + res > OpStack->stack_size)
    return Error (POpOverflow);
  else
    return TRUE;
}

Panic (s)
     char *s;
{
  fprintf (stderr, "PostScript panic: %s\n", s);
  fprintf (stderr, "Please report this fault to the support person for this program\n");
  Cleanup ();
  exit (1);
}

Object Cvx (o)
     Object o;
{
  o.flags |= EXECUTABLE;
  return o;
}

Object Cvlit (o)
     Object o;
{
  o.flags &= ~EXECUTABLE;
  return o;
}

Object ExecOnly (o)
     Object o;
{
  if (o.type == Dictionary)
    o.u.Dictionary->dict_flags &= ~(READABLE | WRITEABLE);
  else
    o.flags &= ~(READABLE | WRITEABLE);
  return ReadOnly (o);
}

Object ReadOnly (o)
     Object o;
{
  if (o.type == Dictionary)
    o.u.Dictionary->dict_flags &= ~WRITEABLE;
  else
    o.flags &= ~WRITEABLE;
  return o;
}

Object WriteOnly (o)
     Object o;
{
  if (o.type == Dictionary)
    o.u.Dictionary->dict_flags &= ~READABLE;
  else
    o.flags &= ~READABLE;
  return o;
}

Object SameFlags (a, b)
     Object a, b;
{
  b.flags = a.flags;
  return b;
}
```



```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <setjmp.h>
#include <strings.h>

#define NONE            (-2)
#define EOS             '\0'
#define TRUE            1
#define FALSE           0

#ifdef BUFSIZE
#undef BUFSIZE
#endif
#define BUFSIZE      1000
#define STRSIZE      4000

#ifndef unix
#define MACINTOSH
#endif

#define VOID    (void)

#define READABLE         01
#define WRITEABLE        02
#define EXECUTABLE       04
#define PERMANENT       010
#define PIPED           020

typedef struct dict_struct *Type;

typedef struct object {
  int flags;
  Type type;
  int Length;
  union {
    int Integer, Bool, Font;
    float Real;
    Type Dictionary;
    unsigned char *String;
    struct name_struct *Name;
    struct op_struct *Operator;
    struct file_struct *File;
    struct object *Array;
  } u;
} Object;

struct name_struct
{
  struct name_struct *next;
  int string_length;
  unsigned char *string_body;
};

enum file_type { StringFile, StreamFile };

struct file_struct {
  enum file_type file_type;
  int available;
  union {
    unsigned char *c_ptr;
    FILE          *f_ptr;
  } f;
};

struct dict_entry { Object entry_key, entry_value; };

struct dict_struct {
  int dict_flags, dict_size, dict_fill;
  struct dict_entry *dict_body;
};

typedef struct stack {
  int stack_fill, stack_size;
  Object overflow, underflow, *stack_body;
} *Stack, StackOb;

struct op_struct {
  Object name;
  int (*fn)(), arguments, results, execpop, execpush, execuse;
  Type *argtypes;
};

Object SameFlags (), Cvx (), Cvlit ();
Object ReadOnly (), WriteOnly (), ExecOnly ();
int OpCheck ();
Object MakeArray (), ParseArray (), getArray (), getIArray (), *BodyArray ();
Object MakeBool ();

Object MakeDict (), DictLoad (), Lookup (), DictFrom (), Load ();
Type MakeType (), TypeOf (), BodyDict ();

extern int EqTrue (), Equal ();

Object FileFrom (), FileString ();

int Getch ();
struct file_struct *BodyFile ();
Object MakeInteger (), IntReal ();

Object ParseNumber ();
float Deg (), Rad ();
Object ParseId (), NameFrom (), MakeName (), Cvn (), StringName ();
unsigned char *BodyName ();
Object MakeOp (), NameOperator ();
Object Parse ();

int PolyFirst (), PolySecond (), PolyThird (), PolyPair ();

Object MakeReal (), RealInteger ();

float BodyReal (), BodyFloat ();

extern char *malloc();
extern int free();
#define Malloc(I) malloc(I)
#define Free(S) free(S)

Object Where (), DictLookup ();

extern Object MakeString (), StringFrom (), getIString ();
extern Object ParseString (), ParseHexString ();

extern Object PDictFull;
```

```
extern Object PDictOverflow,            PInvFont,               PSyntaxError;
extern Object PDictUnderflow,           PInvRestore,            PTypeCheck;
extern Object PExecOverflow,            PIOError,               PUndefined;
extern Object PExecUnderflow,           PLimitCheck,            PUnFilename;
extern Object PInterrupt,               PNoCurrentPoint,        PUnResult;
extern Object PInvAccess,               PRangeCheck,            PUnMatched;
extern Object PInvExit,                 POpOverflow,            PUnregistered;
extern Object PInvFileAccess,           POpUnderflow,           PVMError;

extern Type Bool, Mark, String, Real, Poly, Operator;
extern Type Name, File, Dictionary, Condition, Null, Integer;
extern Type Array, Mark, Condition, Null, Float, FontID;

extern Object SysDict, Absent, Nil;
extern Object True, False, Marker, Self;
extern Object OpInterp, Lbracket, Rbracket;
extern Object StatementEdit, Fstdin, Fstdout, Fstderr;

extern Stack OpStack, ExecStack, DictStack;
extern jmp_buf env;
extern int interactive, verbose;
extern char default_library[], *library;
extern FILE *vfp, *Fopen ();
extern void Fclose ();

/* Stack macros */
#define Top(S)        ((S)->stack_body[(S)->stack_fill - 1])
#define Height(S)     ((S)->stack_fill)
#define MaxStack(S)   ((S)->stack_size)
#define Space(S)      ((S)->stack_size - (S)->stack_fill)
#define Pop(S)        ((S)->stack_body[--(S)->stack_fill])
#define FastPush(S,O) ((S)->stack_body[(S)->stack_fill]=(O),(S)->stack_fill++)
#define Push(S, O)    (((S)->stack_fill != (S)->stack_size) ? \
                      ((S)->stack_body[(S)->stack_fill] = (O), \
                      (S)->stack_fill++, TRUE) : FALSE)

/* File macros */
extern int getchbuf;
#define BodyFile(file)   ((file).u.File)
#define StatusFile(file) (BodyFile(file)->available != 0)
#define Getch(file) \
  ((StatusFile(file) && BodyFile(file)->file_type == StreamFile) \
  ? ((getchbuf = getc (BodyFile(file)->f.f_ptr)), \
    ((getchbuf != EOF) ? getchbuf : ((BodyFile(file)->available = 0), \
                                    Close (file), EOF))) \
  : GeneralGetch (file))

/* Dictionary macros */
Object DictFind();
#define DictLoad(DICT,KEY) DictFind((DICT.u.Dictionary)->dict_body,KEY,\
                                   (DICT.u.Dictionary)->dict_size)
#define DictEqual(A,B) ((TypeOf(A) == TypeOf(B)) && \
                        (A.Length == B.Length) && \
                        ((A.u.Integer == B.u.Integer)|| \
                         (TypeOf(A) == Array) && (A.Length == 0)))
#define maxDict(D)     (BodyDict(D)->dict_size)
#define lengthDict(D)  (BodyDict(D)->dict_fill)

/* Flag macros */
#define rCheck(O) ((O.type == Dictionary) \
                   ? (O.u.Dictionary->dict_flags & READABLE) \
                   : (O.flags & READABLE))
#define wCheck(O) ((O.type == Dictionary) \
                   ? (O.u.Dictionary->dict_flags & WRITEABLE) \
                   : (O.flags & WRITEABLE))
#define xCheck(O) (O.flags & EXECUTABLE)

/* Body macros */
#define BodyInteger(O)  ((O).u.Integer)
#define BodyReal(O)     ((O).u.Real)
#define BodyOperator(O) ((O).u.Operator)
#define BodyString(O)   ((O).u.String)
#define BodyArray(O)    ((O).u.Array)
#define BodyDict(O)     ((O).u.Dictionary)
#define BodyName(O)     ((O).u.Name->string_body)
#define lengthName(O)   ((O).u.Name->string_length)
#define lengthString(O) ((O).Length)

/* Path macros */
extern struct path_element *free_pelem;
#define NewElem(T, P) { P = free_pelem; if (P) free_pelem = P->next;\
     else P = (Path) Malloc (sizeof (struct path_element));\
     P->ptype = T; }
#define CurEType(P) ((P)->last->ptype)
#define EmptyPath(P) ((!(P))?TRUE:((P)->next == (P)))
#define ElemFree(P)  { PanicIf ((P) == NULL, "ElemFree given NULL");\
                       (P)->next = free_pelem; free_pelem = (P); }
#define PathInsert(P, I) { PanicIf ((I) == NULL, "PathInsert given NULL");\
                                (I)->next = (P); (I)->last = (P)->last;\
                                (P)->last->next = (I); (P)->last = (I); }
#define PathFree(P) { PanicIf((P) == NULL, "PathFree given NULL");\
                           (P)->last->next = free_pelem; \
                                (P)->last = (P); \
                           free_pelem = (P); }
#define PathDelete(P) { Path R;\
                        PanicIf ((P)->next == (P), "delete from empty path");\
                        R=(P)->last; R->last->next = (P); (P)->last = R->last;\
                        R->next = free_pelem; free_pelem = R; }


/* Object macros */
extern Object InitialObject,InitialExecObject;
#define MakeObject(O, T) { O = InitialObject; O.type = T;}
#define MakeExecObject(O, T) { O = InitialExecObject; O.type = T;}

/* Matrix macros */
#define NewMatrix(m, pA, pB, pC, pD, ptx, pty) { \
  (m).A = pA; (m).B = pB; (m).C = pC; (m).D = pD; (m).tx = ptx; (m).ty = pty; }
#define RotatePosQ(m,t) { t=m.A; m.A=m.C; m.C = -t; t=m.B; m.B=m.D; m.D = -t; }
#define RotateNegQ(m,t) { t=m.A; m.A = -m.C; m.C=t; t=m.B; m.B = -m.D; m.D=t; }

/* Miscellaneous macros */
#define lengthArray(a)   ((a).Length)
#define NameOperator(op) ((op).u.Operator->name)
#define TypeOf(a)        ((a).type)
#define Message(S) { if (vfp != NULL) fprintf (vfp, "%s\n", S), fflush (vfp); }
#define PanicIf(COND,STR) if(COND) Panic(STR)
#define Max(a, b) ((a > b) ? a : b)
#define Min(a, b) ((a < b) ? a : b)

#ifdef AUX
#define Bcopy(D,S,N) memcpy(D,S,N)
#else
#ifdef MACINTOSH
#define Bcopy(D,S,N) bcopy(S,D,N)
#else
#define Bcopy(D,S,N) bcopy(S,D,N)
#endif
```



```
#endif
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

Matrix identity =   { 1.0, 0.0, 0.0, 1.0, 0.0, 0.0 };

Vector NewVector (A, B, vt)
     float A, B, vt;
{
  Vector v;

  v.vx = A; v.vy = B; v.vt = vt;

  return v;
}

Matrix Translate (m, x, y)
     Matrix m;
     float x, y;
{
  m.tx += x * m.A + y * m.C;
  m.ty += x * m.B + y * m.D;
  return m;
}

HereTranslate (pm, p)
     Matrix *pm;
     UserPoint p;
{
  pm->tx += p.x * pm->A + p.y * pm->C;
  pm->ty += p.x * pm->B + p.y * pm->D;
}

Matrix Scale (m, x, y)
     Matrix m;
     float x, y;
{
  m.A *= x;  m.B *= x;  m.C *= y;  m.D *= y;
  return m;
}

Matrix Rotate (m, a)
     Matrix m;
     float a;
{
  float ca = cos(a), sa = sin(a);
  Matrix r;
  NewMatrix(r, m.A * ca + m.C * sa, m.B * ca + m.D * sa,
            m.C * ca - m.A * sa, m.D * ca - m.B * sa,
            m.tx,                m.ty);
  return r;
}

Matrix MatMult (a, b)
     Matrix a, b;
{
  Matrix r;
  NewMatrix (r,    a.A  * b.A + a.B * b.C, a.A * b.B + a.B * b.D,
             a.C  * b.A + a.D * b.C, a.C * b.B + a.D * b.D,
             a.tx * b.A + a.ty * b.C + b.tx,
             a.tx * b.B + a.ty * b.D + b.ty);
  return r;
}

Matrix MatInvert (m)
     Matrix m;
     /* know any good matrix inversion algorithms ? */
{                                         /* this one will be simplistic */
  Matrix r;
  float detinv = 1 / (m.A * m.D - m.B * m.C);

  NewMatrix(r,m.D * detinv,
            -m.B * detinv,
            -m.C * detinv,
            m.A * detinv,
            (m.C * m.ty - m.D * m.tx) * detinv,
            -(m.A * m.ty - m.B * m.tx) * detinv);
  return r;
}

Vector Transform (v, m)
     Vector v;
     Matrix m;
{
  return NewVector (v.vx * m.A + v.vy * m.C + v.vt * m.tx,
                    v.vx * m.B + v.vy * m.D + v.vt * m.ty,
                    v.vt);
}

Vector DTransform (v, m)
     Vector v;
     Matrix m;
{
  return NewVector (v.vx * m.A + v.vy * m.C,
                    v.vx * m.B + v.vy * m.D,
                    v.vt);
}

Vector ITransform (v, mi)
     Vector v;
     Matrix mi;
{
  Matrix m;

  m = MatInvert (mi);
  return NewVector (v.vx * m.A + v.vy * m.C + v.vt * m.tx,
                    v.vx * m.B + v.vy * m.D + v.vt * m.ty,
                    v.vt);
}

Vector IDTransform (v, mi)
     Vector v;
     Matrix mi;
{
  Matrix m;
```



```
  m = MatInvert (mi);
  return NewVector (v.vx * m.A + v.vy * m.C,
                    v.vx * m.B + v.vy * m.D,
                    v.vt);
}

Vector DiffVector (a, o)
     Vector a, o;
{
  return NewVector (a.vx - o.vx, a.vy - o.vy, 1.0);
}
```

```c
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

typedef struct matrix { float A, B, C, D, tx, ty; } Matrix;

typedef struct vector { float vx, vy, vt; } Vector;

extern Matrix MatMult (), MatInvert (), Translate (), Rotate (), Scale ();
extern Vector NewVector (), DiffVector ();
extern Vector Transform (), ITransform (), DTransform (), IDTransform ();
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include <signal.h>

#include "main.h"

#ifndef PI
#define PI 3.14159265358979
#endif

static int PRand (), PSrand (), PRrand (), Idiv ();
static int ParseInteger ();

static catchmath ();
extern int errno;
jmp_buf penv;

InitMath ()
{
  InstallOp ("abs",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("add",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("div",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("mod",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("mul",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("neg",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("sub",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("sqrt",    PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("exp",     PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("ceiling", PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("floor",   PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("round",   PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("truncate",        PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("atan",    PolyPair,       2, 1, 0, 0, Poly, Poly);
  InstallOp ("cos",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("sin",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("ln",      PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("log",     PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("rand",    PRand,          0, 1, 0, 0);
  InstallOp ("srand",   PSrand,         1, 0, 0, 0, Integer);
  InstallOp ("rrand",   PRrand,         0, 1, 0, 0);
  InstallOp ("idiv",    Idiv,           2, 1, 0, 0, Float, Float);

  /*    PanicIf (setjmp (penv), "Unexpected floating point error");
        VOID signal (SIGFPE, catchmath);
*/
}

static catchmath ()
{
  VOID signal (SIGFPE, catchmath);
  longjmp (penv, TRUE);
}

float Deg (r)
     float r;
{
  float res = 360 * r / (2 * PI);
  return res < 0 ? res + 360 : res;
}

float Rad (d)
     float d;
{
  return 2 * PI * d / 360;
}

float BodyFloat (ob)
     Object ob;
{
  if (TypeOf (ob) == Integer)
    return (float) BodyInteger (ob);
  else if (TypeOf (ob) == Real)
    return BodyReal (ob);
  else
    Panic ("BodyFloat passed arg of bad type");
  return 0.0;
}

static float pow10[]={1,1e1,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9,1e10,1e11};

#define RetInt(V) {res.type=Integer; res.u.Integer=sign?-(V):(V); return res;}
#define RetReal(V) {res.type=Real; res.u.Real = sign?-(V):(V); return res;}

Object ParseNumber (s, length) unsigned char *s; int length;
{
  Object res;
  int c, sign, ival, dval, olength;
  float fval;

  if (length == 0) return Nil;

#ifdef MACINTOSH
  setmem(&res, sizeof(Object), ’\0’);
#else
  bzero(&res, sizeof(Object));
#endif

  res.flags =  READABLE|WRITEABLE;
  if ((c = *s) == ’+’) { sign = 0;  c = *++s; --length; }
  else if (c == ’-’) { sign = 1; c = *++s; --length; }
  else sign = 0;

  if (c == ’.’) {
    if ((++s, --length) == 0 || (c = *s) < ’0’ || c > ’9’) return Nil;
    olength = length;  dval = c - ’0’;
    while ((++s, --length) > 0 && (c = *s) >= ’0’ && c <= ’9’)
      dval = dval*10 + c-’0’;
    fval = (float) dval / pow10[olength-length];
    if (length == 0) RetReal(fval);
  }
  else {
    if (length == 0 || c < ’0’ || c > ’9’) return Nil;
    ival = c - ’0’;
    while ((++s, --length) > 0 && (c = *s) >= ’0’ && c <= ’9’)
      ival = ival*10 + c-’0’;
    if (length == 0) RetInt(ival);
```

```
    if ((c = *s) == '.') {
      if ((s++, --length) == 0 || (c = *s) < '0' || c > '9') return Nil;
      olength = length;  dval = c - '0';
      while ((++s, --length) > 0 && (c = *s) >= '0' && c <= '9')
        dval = dval*10 + c-'0';
      fval = ival + (float) dval / pow10[olength-length];
      if (length == 0) RetReal(fval);
    }
    else if (c == '#') {
      if (ival < 2 || ival > 36) return Nil;
      s++; --length;
      ival = ParseInteger (&s, &length, ival);
      if (length >= 0) RetInt(ival);
      return Nil;
    }
    else fval = (float) ival;
  }

  if ((c = *s) == 'e' || c == 'E') {
    if ((s++, --length) == 0) return Nil;

    if (sign) fval = -fval;
    if ((c = *s) == '+') { sign = 0;  c = *++s; --length; }
    else if(c == '-') { sign = 1; c = *++s; --length; }
    else sign = 0;

    if (length == 0 || c < '0' || c > '9') return Nil;
    ival = c - '0';
    while ((++s, --length) > 0 && (c = *s) >= '0' && c <= '9')
      ival = ival*10 + c-'0';
    if (sign) ival = -ival;
    res.type = Real; res.u.Real = fval * pow(10.0,(double) ival);
    return res;
  }
  return Nil;
}

static int ParseInteger (p, length, base)
     unsigned char **p;
     int *length, base;
{
  int present = FALSE, ival = 0, digit;
  if (*length == 0) {
    *length = -1;
    return 0;
  }
  for (;;) {
    int c = *(*p)++;

    if (c >= '0' && c <= '9')      digit = c - '0';
    else if (c >= 'a' && c <= 'z') digit = c - 'a' + 10;
    else if (c >= 'A' && c <= 'Z') digit = c - 'A' + 10;
    else break;

    if (digit < base) ival = ival * base + digit;
    else break;
    ++present;
    if (--*length == 0) return ival;
  }
  --*p;
  if (!present) *length = -1;
  return ival;
}
```

```
static int seed = 1;

static int random ()
{
  return seed = ((seed * 1103515245 + 12345) & 0x7fffffff);
}

static int PRand ()             /* --- integer */
{
  VOID random ();
  return Push (OpStack, MakeInteger (seed));
}

static int PSrand (see)
     Object see;
{
  srand ((unsigned) (seed = BodyInteger (see)));
  return TRUE;
}

static int PRrand ()
{
  return Push (OpStack, MakeInteger (seed));
}

static int Idiv (a, b)
     Object a, b;
{
  if (BodyReal (b) == 0)
    return Error (PUnResult);
  return Push (OpStack, MakeInteger ((int) BodyReal (a) / (int) BodyReal (b)));
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

Object AssignMatrix ();
extern Matrix identity;
static int PMatrix ();
static int PIdentMatrix ();
static int PDefaultMatrix ();
static int PCurrentMatrix ();
static int PSetMatrix ();
static int PTranslate ();
static int PScale ();
static int PRotate ();
static int PConcat ();
static int PConcatMatrix ();
static int PTransform ();
static int PDTransform ();
static int PITransform ();
static int PIDTransform ();
static int PInvertMatrix ();

InitMatrix ()
{
  InstallOp ("matrix",          PMatrix,        0, 1, 0, 0);
  InstallOp ("initmatrix",      PInitMatrix,    0, 0, 0, 0);
  InstallOp ("identmatrix",     PIdentMatrix,   1, 1, 0, 0, Array);
  InstallOp ("defaultmatrix",   PDefaultMatrix, 1, 1, 0, 0, Array);
  InstallOp ("currentmatrix",   PCurrentMatrix, 1, 1, 0, 0, Array);
  InstallOp ("setmatrix",       PSetMatrix,     1, 0, 0, 0, Array);
  InstallOp ("translate",       PTranslate,     0, 0, 0, 0);
  InstallOp ("scale",           PScale,         0, 0, 0, 0);
  InstallOp ("rotate",          PRotate,        0, 0, 0, 0);
  InstallOp ("concat",          PConcat,        1, 0, 0, 0, Array);
  InstallOp ("concatmatrix",    PConcatMatrix,  3, 1, 0, 0, Array, Array,
             Array);
  InstallOp ("transform",       PTransform,     0, 0, 0, 0);
  InstallOp ("dtransform",      PDTransform,    0, 0, 0, 0);
  InstallOp ("itransform",      PITransform,    0, 0, 0, 0);
  InstallOp ("idtransform",     PIDTransform,   0, 0, 0, 0);
  InstallOp ("invertmatrix",    PInvertMatrix,  2, 1, 0, 0, Array, Array);
}

Object MatrixFrom (m)
     Matrix m;
{
  return AssignMatrix (MakeArray ((Object *) Malloc (6 * sizeof (Object)), 6), m);
}

int ExtractMatrix (pm, o)
     Matrix *pm;
     Object o;
```

```
{
  int i;
  float temp[6];

  for (i = 0; i < 6; i++) {
    Object item;

    item = getArray (o, i);
    if (TypeOf (item) == Integer)
      temp[i] = (float) BodyInteger (item);
    else if (TypeOf (item) == Real)
      temp[i] = BodyReal (item);
    else
      return FALSE;
  }

  NewMatrix(*pm, temp[0], temp[1], temp[2], temp[3], temp[4], temp[5]);
  return TRUE;
}

static int PMatrix ()
{
  return Push (OpStack, MatrixFrom (identity));
}

int PInitMatrix ()
{
  gstate->CTM = gstate->device->default_matrix;

  return TRUE;
}

static int PIdentMatrix (array)
     Object array;
{
  if (lengthArray (array) != 6)
    return Error (PRangeCheck);
  VOID AssignMatrix (array, identity);
  return Push (OpStack, array);
}

static int PDefaultMatrix (array)
     Object array;
{
  if (lengthArray (array) != 6)
    return Error (PRangeCheck);
  VOID AssignMatrix (array, gstate->device->default_matrix);
  return Push (OpStack, array);
}

static int PCurrentMatrix (array)
     Object array;
{
  if (lengthArray (array) != 6)
    return Error (PRangeCheck);
  VOID AssignMatrix (array, gstate->CTM);
  return Push (OpStack, array);
}

static int PSetMatrix (array)
     Object array;
{
  if (lengthArray (array) != 6)
    return Error (PRangeCheck);
```

```
  else if (!ExtractMatrix (&gstate->CTM, array))
    return Error (PTypeCheck);
  else
    return TRUE;
}

static int PTranslate ()
{
  Object tx, ty, mat;
  float x, y;
  Matrix m;

  if (!OpCheck (2, 1))
    return FALSE;
  mat = Pop (OpStack);
  if (TypeOf (mat) == Array)
    if (!wCheck (mat))
      return Push (OpStack, mat), Error (PInvAccess);
    else if (lengthArray (mat) != 6)
      return Push (OpStack, mat), Error (PRangeCheck);
    else {
      ty = Pop (OpStack);
      if (TypeOf (ty) == Integer)
        y = (float) BodyInteger (ty);
      else if (TypeOf (ty) == Real)
        y = BodyReal (ty);
      else
        return Push (OpStack, ty), Push (OpStack, mat), Error (PTypeCheck);
      tx = Pop (OpStack);
      if (TypeOf (tx) == Integer)
        x = (float) BodyInteger (tx);
      else if (TypeOf (tx) == Real)
        x = BodyReal (tx);
      else
        return Push (OpStack, tx), Push (OpStack, ty), Push (OpStack, mat), Error (PTypeC

      NewMatrix(m, 1.0, 0.0, 0.0, 1.0, x, y);
      VOID AssignMatrix (mat, m);

      return Push (OpStack, mat);
    }
  else if (TypeOf (mat) == Integer)
    y = (float) BodyInteger (mat);
  else if (TypeOf (mat) == Real)
    y = BodyReal (mat);
  else
    return Push (OpStack, mat), Error (PTypeCheck);

  tx = Pop (OpStack);
  if (TypeOf (tx) == Integer)
    x = (float) BodyInteger (tx);
  else if (TypeOf (tx) == Real)
    x = BodyReal (tx);
  else
    return Push (OpStack, tx), Push (OpStack, mat), Error (PTypeCheck);
  gstate->CTM = Translate (gstate->CTM, x, y);
  return TRUE;
}

static int PScale ()
{
  Object tx, ty, mat;
  float x, y;
  Matrix m;

  if (!OpCheck (2, 1))
    return FALSE;
  mat = Pop (OpStack);
  if (TypeOf (mat) == Array)
    if (!wCheck (mat))
      return Push (OpStack, mat), Error (PInvAccess);
    else if (lengthArray (mat) != 6)
      return Push (OpStack, mat), Error (PRangeCheck);
    else {
      ty = Pop (OpStack);
      if (TypeOf (ty) == Integer)
        y = (float) BodyInteger (ty);
      else if (TypeOf (ty) == Real)
        y = BodyReal (ty);
      else
        return Push (OpStack, ty), Push (OpStack, mat), Error (PTypeCheck);
      tx = Pop (OpStack);
      if (TypeOf (tx) == Integer)
        x = (float) BodyInteger (tx);
      else if (TypeOf (tx) == Real)
        x = BodyReal (tx);
      else
        return Push (OpStack, tx), Push (OpStack, ty), Push (OpStack, mat), Error (PTypeC

      NewMatrix (m, x, 0.0, 0.0, y, 0.0, 0.0);
      VOID AssignMatrix (mat, m);

      return Push (OpStack, mat);
    }
  else if (TypeOf (mat) == Integer)
    y = (float) BodyInteger (mat);
  else if (TypeOf (mat) == Real)
    y = BodyReal (mat);
  else
    return Push (OpStack, mat), Error (PTypeCheck);

  tx = Pop (OpStack);
  if (TypeOf (tx) == Integer)
    x = (float) BodyInteger (tx);
  else if (TypeOf (tx) == Real)
    x = BodyReal (tx);
  else
    return Push (OpStack, tx), Push (OpStack, mat), Error (PTypeCheck);
  gstate->CTM = Scale (gstate->CTM, x, y);
  return TRUE;
}

static int PRotate ()
{
  Object ang, mat;
  float a,ca,sa;
  Matrix m;

  if (!OpCheck (1, 1))
    return FALSE;
  mat = Pop (OpStack);
  if (TypeOf (mat) == Array)
    if (!wCheck (mat))
      return Push (OpStack, mat), Error (PInvAccess);
    else if (lengthArray (mat) != 6)
      return Push (OpStack, mat), Error (PRangeCheck);
    else {
      ang = Pop (OpStack);
```



```
      if (TypeOf (ang) == Integer)
        a = (float) BodyInteger (ang);
      else if (TypeOf (ang) == Real)
        a = BodyReal (ang);
      else
        return Push (OpStack, ang), Push (OpStack, mat), Error (PTypeCheck);

      ca = cos(a);  sa = sin(a);
      NewMatrix (m, ca, sa, -sa, -ca, 0.0, 0.0);
      VOID AssignMatrix (mat, m);

      return Push (OpStack, mat);
    }
  else if (TypeOf (mat) == Integer)
    a = (float) BodyInteger (mat);
  else if (TypeOf (mat) == Real)
    a = BodyReal (mat);
  else
    return Push (OpStack, mat), Error (PTypeCheck);

  gstate->CTM = Rotate (gstate->CTM, Rad (a));
  return TRUE;
}

static int PConcat (mat)
     Object mat;
{
  Matrix m;

  if (!ExtractMatrix (&m, mat))
    return Error (PTypeCheck);
  gstate->CTM = MatMult (m, gstate->CTM);
  return TRUE;
}

static int PConcatMatrix (mat1, mat2, mat3)
     Object mat1, mat2, mat3;
{
  Matrix m1, m2;

  if (!ExtractMatrix (&m1, mat1) || !ExtractMatrix (&m2, mat2) ||
      lengthArray (mat3) != 6)
    return Error (PTypeCheck);
  VOID AssignMatrix (mat3, MatMult (m1, m2));
  VOID Push (OpStack, mat3);
  return TRUE;
}

static int Transfn (fn)
     Vector (*fn)();
{
  Matrix m;
  Object tx, ty, mat;
  Vector v;
  float x, y;
  int mflag = FALSE;

  m = gstate->CTM;
  if (!OpCheck (2, 1))
    return FALSE;

  mat = Pop (OpStack);
  if (TypeOf (mat) == Array) {
    if (lengthArray (mat) != 6 || !ExtractMatrix (&m, mat))
      return Push (OpStack, mat), Error (PTypeCheck);
    mflag = TRUE;
  }
  else
    VOID Push (OpStack, mat);

  if (!OpCheck (2, 1))
    return Push (OpStack, mat), FALSE;
  ty = Pop (OpStack);
  if (TypeOf (ty) == Integer)
    y = (float) BodyInteger (ty);
  else if (TypeOf (ty) == Real)
    y = BodyReal (ty);
  else {
    VOID Push (OpStack, ty);
    if (mflag)
      VOID Push (OpStack, mat);
    return Error (PTypeCheck);
  }
  tx = Pop (OpStack);
  if (TypeOf (tx) == Integer)
    x = (float) BodyInteger (tx);
  else if (TypeOf (tx) == Real)
    x = BodyReal (tx);
  else {
    VOID Push (OpStack, tx);
    VOID Push (OpStack, ty);
    if (mflag)
      VOID Push (OpStack, mat);
    return Error (PTypeCheck);
  }

  v = (*fn) (NewVector (x, y, 1.0), m);

  return Push (OpStack, MakeReal (v.vx)), Push (OpStack, MakeReal (v.vy));
}

static int PTransform ()
{
  return Transfn (Transform);
}

static int PDTransform ()
{
  return Transfn (DTransform);
}

static int PITransform ()
{
  return Transfn (ITransform);
}

static int PIDTransform ()
{
  return Transfn (IDTransform);
}

static int PInvertMatrix (mat1, mat2)
     Object mat1, mat2;
{
  Matrix m;

  if (!ExtractMatrix (&m, mat1) || lengthArray (mat2) != 6)
    return Error (PTypeCheck);
```

```
  VOID AssignMatrix (mat2, MatInvert (m));
  VOID Push (OpStack, mat2);
  return TRUE;
}

UserPoint IntToExt (P)
     HardPoint P;
{
  register Matrix *mp;
  UserPoint p;
  float x,y,det;

  /* TMD - Try really open coding this...
  **    v = Transform (NewVector ((float) p.hx, (float) p.hy, 1.0),
  **                   MatInvert (gstate->CTM));
  **    return NewPoint (v.vx, v.vy);
  */
  mp = &gstate->CTM;
  det = mp->A * mp->D - mp->B * mp->C;
  x = P.hx - mp->tx;  y = P.hy - mp->ty;
  p.x = (x * mp->D - y * mp->C) / det;
  p.y = (y * mp->A - x * mp->B) / det;
  return p;
}

HardPoint ExtToInt (p)
     UserPoint p;
{
  HardPoint P;
  register Matrix *mp;
/*
 * BZS - try open coding this
 *
        v = Transform (NewVector (p.x, p.y, 1.0), gstate->CTM);
 *
 * TMD - try really open coding this...
 */
  mp = &gstate->CTM;
  P.hx = p.x * mp->A + p.y * mp->C + mp->tx;
  P.hy = p.x * mp->B + p.y * mp->D + mp->ty;
  return P;
}

Matrix PointTranslate (m, p)
     Matrix m;
     UserPoint p;
{
  return Translate (m, p.x, p.y);
}

Matrix HardPointTranslate (m, p)
     Matrix m;
     HardPoint p;
{
  m.tx += p.hx;
  m.ty += p.hy;

  return m;
}

Object AssignMatrix (o, m)
     Object o;
     Matrix m;
{
  Object *vec = BodyArray (o);

  vec[0] = MakeReal (m.A);
  vec[1] = MakeReal (m.B);
  vec[2] = MakeReal (m.C);
  vec[3] = MakeReal (m.D);
  vec[4] = MakeReal (m.tx);
  vec[5] = MakeReal (m.ty);

  return o;
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include <sys/param.h>
#ifdef AUX
#include <signal.h>
#include <sys/types.h>
#endif /* AUX */
#include <sys/times.h>
#ifndef HZ
#       define HZ 60
#endif

static int PUserTime ();

InitMisc ()
{
  InstallOp ("usertime",        PUserTime,      0, 1, 0, 0);
  InstallOp ("==",              PolyFirst,      1, 1, 0, 0, Poly);
}

static int PUserTime ()
{
  struct tms tbuf;
  long times ();

  times (&tbuf);
  return Push (OpStack, MakeInteger ((int) (tbuf.tms_utime * 1000 / HZ)));
}
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static int Cvs (), EqEq (), Eq (), ExecName ();
static Object OpExecName;

Object Self;

InitName ()
{
  OpExecName = MakeOp ("exec", ExecName,        1, 1, 0, 4, Name);

  TypeInstall (Name, "exec",    OpExecName);

  TypeInstallOp (Name, "==",    EqEq,   1, 0, 0, 0, Name);
  TypeInstallOp (Name, "cvs",   Cvs,    2, 1, 0, 0, Name, String);
  TypeInstallOp (Name, "eq",    Eq,     2, 1, 0, 0, Name, Name);
}

static int Cvs (v, string)
     Object v, string;
{
  int length;

  if (lengthString (string) < (length = lengthName (v)))
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), BodyName (v), length);
  return Push (OpStack, getIString (string, 0, length));
}

static int EqEq (v)
     Object v;
{
  if (!xCheck (v))
    putchar ('/');
  PrintName (v);
  return TRUE;
}

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (BodyName (a) == BodyName (b) && lengthName (a) == lengt
}

static int ExecName (item)
     Object item;
{
  Object v;

  v = DictLookup (item);
  if (TypeOf (v) == Condition)
    return Error (PUndefined);
  else
    return Push (ExecStack, v);
}

#if 0
static struct name_struct *FindTreeName (s, length, root)
     unsigned char *s;
     int length;
     struct name_struct **root;
{
  struct name_struct *r;
  int rootlen,cmp;

  while (*root) {
    rootlen = (*root)->string_length;
    if(rootlen == length) {
      cmp = strncmp(s, (*root)->string_body, length);
      if (cmp == 0) return *root;
    } else
      cmp = length - rootlen;

    root = (cmp < 0) ? &(*root)->prev_name : &(*root)->next_name;
  }

  r = *root = (struct name_struct *) Malloc (sizeof (struct name_struct));

  r->next_name = r->prev_name = NULL;
  Bcopy(r->string_body = (unsigned char *)Malloc((unsigned)length),s,length);
  r->string_length = length;

  return r;
}

#define HASH_NAME_SIZE  1024
int name_tries = 0, name_hits = 0;
static struct name_struct *hash_name [HASH_NAME_SIZE];
static struct name_struct *name_tree = NULL;

static struct name_struct *FindName (s, length)
     unsigned char *s;
     int length;
{
  struct name_struct *p;
  int hash = 0,tmplen = length;

  while(tmplen--) hash += *s++;
  hash &= (HASH_NAME_SIZE - 1);

  p = hash_name [hash];
  ++name_tries;
  ++name_hits;

  if (p == NULL || p->string_length != length ||
      strncmp (s, p->string_body, length)) {
    p = FindTreeName (s, length, &name_tree);

    if (p != NULL) {
      hash_name [hash] = p;
      --name_hits;
    }
  }
  return p;
}
```



```c
#else

#define HASH_NAME_SIZE  1009
int name_tries = 0, name_hits = 0;
static struct name_struct *hash_name [HASH_NAME_SIZE];

static struct name_struct *FindName (s, length)
     unsigned char *s;
     int length;
{
  struct name_struct *p,**slot;
  unsigned char *t1,*t2;
  int hash,shift,tmplen;

  /* Convert each character to a 5 bit number between 3 and 28.
   * Xor these into a 32 bit integer in the following pattern:
   * ---------------------------------
   *        44444333332222211111100000   <-- first 5 bits go in 00000
   *      9999988888777776666655555
   *    44444333332222211111100000
   *       9999988888777776666655555
   *     44444333332222211111100000
   */
  t1 = s; hash = 0; shift = 0; tmplen = length;
  while (tmplen--) {
    hash ^= *t1++ << shift;
    if((shift += 5) >= 22) shift -= 22;
  }
  hash %= HASH_NAME_SIZE;
  slot = &hash_name[hash];
  p = *slot;
  ++name_tries;
  ++name_hits;

  while(p != NULL) {
    if(p->string_length == length) {
      tmplen = length;
      t1 = p->string_body; t2 = s;
      while(*t1++ == *t2++ && --tmplen > 0);
      if(tmplen == 0) return p;
    }
    slot = &p->next;
    p = *slot;
    name_tries++;
  }

  --name_hits;
  *slot = p = (struct name_struct *) Malloc (sizeof (struct name_struct));
  p->next = NULL;
  Bcopy(p->string_body = (unsigned char *)Malloc((unsigned)length),s,length);
  p->string_length = length;
  return p;
}
#endif

Object NameFrom (s)
     unsigned char *s;
{
  Object res;

  MakeObject (res, Name);
  res.u.Name = FindName (s, strlen (s));
  return res;
}

Object Cvn (o)
     Object o;
{
  Object res;

  MakeObject (res, Name);
  res.u.Name = FindName (BodyString (o), lengthString (o));

  return res;
}

Object StringName (o)
     Object o;
{
  return MakeString (BodyName (o), lengthName (o));
}

PrintName (n)
     Object n;
{
  printf ("%.*s", lengthName (n), BodyName (n));
}

Object ParseId (o)
     Object o;
{
  unsigned char buf[BUFSIZE], *p = buf;
  int c, length;
  Object res;

  if(!StatusFile(o)) return Absent;
  if(BodyFile(o)->file_type == StreamFile) {
    FILE *fp = BodyFile(o)->f.f_ptr;
    for (;;) {
      switch (c = getc(fp)) {
        case EOF: case ' ': case '\t': case '\n': break;

        case '/': case '<': case '>': case '(': case ')':
        case '%': case '{': case '}': case '[': case ']':
/*        BodyFile(o)->available++; */
          ungetc(c,fp);
          break;
        default: *p++ = c; continue;
      }
      break;
    }
  }
  else {
    for (;;) {
      switch (c = GeneralGetch(o)) {
        case EOF: case ' ': case '\t': case '\n': break;

        case '/': case '<': case '>': case '(': case ')':
        case '%': case '{': case '}': case '[': case ']':
          Ungetch (o, c);
          break;
        default: *p++ = c; continue;
      }
      break;
    }
  }

  length = (int) (p - buf);
```

```
  if (length == 0) return Absent;

  /* Only attempt to parse as a number if it could possibly be one */
  c = buf[0];
  if((c >= '0' && c <= '9') || c == '.' || c == '+' || c == '-') {
    res = ParseNumber (buf, length);
    if (TypeOf (res) != Null) return res;
  }

  /* Not a number, so place it in the name space */
  MakeObject(res, Name);
  res.u.Name = FindName(buf,length);
  return res;
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

int ExecOperator ();
static int EqEq (), Cvs (), Eq (), Signature ();

InitOperator ()
{
  TypeInstallOp (Operator, "==",  EqEq, 1, 0, 0, 0, Operator);
  TypeInstallOp (Operator, "cvs", Cvs,  2, 1, 0, 0, Operator, String);
  TypeInstallOp (Operator, "exec",ExecOperator, 1, 0, 0, 0, Operator);
  TypeInstallOp (Operator, "eq",  Eq,          2, 1, 0, 0, Operator, Operator);
  TypeInstallOp (Operator, "signature", Signature,1, 0, 0, 0, Operator);
}

/*VARARGS6*/
/*ARGSUSED*/
Object MakeOp (name, fn, arguments, results, execpop, execpush,
               arg1, arg2, arg3, arg4, arg5, arg6, arg7)
     char *name;
     int (*fn)(), arguments, results, execpop, execpush;
     Type arg1, arg2, arg3, arg4, arg5, arg6, arg7;
{
  Object res;
  struct op_struct *op;
  int i;

  MakeExecObject (res, Operator);
  res.u.Operator = op = (struct op_struct *) Malloc (sizeof (struct op_struct));

  op->name      = NameFrom (name);
  op->fn        = fn;
  op->arguments = arguments;
  op->results   = results;
  op->execpop   = execpop;
  op->execpush  = execpush;
  op->execuse   = (execpop + execpush) > 0;
  op->argtypes  = (Type *) Malloc ((unsigned) sizeof (Type) * arguments);

  for (i = 0; i < arguments; i++)
    op->argtypes[i] = (&arg1)[i];

  return res;
}

/*VARARGS6*/
/*ARGSUSED*/
InstallOp (name, fn, arguments, results, execpop, execpush,
           arg1, arg2, arg3, arg4, arg5, arg6, arg7)
     char *name;
     int (*fn)(), arguments, results, execpop, execpush;
     Type arg1, arg2, arg3, arg4, arg5, arg6, arg7;
{
  Object res;
  struct op_struct *op;
  /*    int i;
   */
  MakeExecObject (res, Operator);
  res.u.Operator = op = (struct op_struct *) Malloc (sizeof (struct op_struct));

  op->name      = NameFrom (name);
  op->fn        = fn;
  op->arguments = arguments;
  op->results   = results;
  op->execpop   = execpop;
  op->execpush  = execpush;
  op->execuse   = (execpop + execpush) > 0;

  op->argtypes  = (Type *) Malloc ((unsigned) sizeof (Type) * arguments);

  /*    for (i = 0; i < arguments; i++)
   * op->argtypes[i] = (&arg1)[i];
   */
  switch (arguments) {
  default:
    Panic ("too many arguments in InstallOp");
    break;
  case 7: op->argtypes[6] = arg7;
  case 6: op->argtypes[5] = arg6;
  case 5: op->argtypes[4] = arg5;
  case 4: op->argtypes[3] = arg4;
  case 3: op->argtypes[2] = arg3;
  case 2: op->argtypes[1] = arg2;
  case 1: op->argtypes[0] = arg1;
  case 0: break;
  }
  DictStore (SysDict, NameOperator (res), res);
}

/*VARARGS7*/
/*ARGSUSED*/
TypeInstallOp (type, name, fn, arguments, results, execpop, execpush,
               arg1, arg2, arg3, arg4, arg5, arg6, arg7)
     char *name;
     int (*fn)(), arguments, results, execpop, execpush;
     Type type, arg1, arg2, arg3, arg4, arg5, arg6, arg7;
{
  Object res;
  struct op_struct *op;
  int i;

  MakeExecObject (res, Operator);
  res.u.Operator = op = (struct op_struct *) Malloc (sizeof (struct op_struct));

  op->name      = NameFrom (name);
  op->fn        = fn;
  op->arguments = arguments;
  op->results   = results;
  op->execpop   = execpop;
  op->execpush  = execpush;
  op->execuse   = (execpop + execpush) > 0;

  op->argtypes  = (Type *) Malloc ((unsigned) sizeof (Type) * arguments);

  for (i = 0; i < arguments; i++)
    op->argtypes[i] = (&arg1)[i];
```

```
  TypeInstall (type, name, res);
}

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (BodyOperator(a)==BodyOperator(b)));
}

static int Cvs (v, string)
     Object v, string;
{
  Object vv;
  int length;

  vv = NameOperator (v);

  if (lengthString (string) < (length = lengthName (vv)))
    return Error (PRangeCheck);
  VOID strncpy (BodyString (string), BodyName (vv), length);
  return Push (OpStack, getIString (string, 0, length));
}

static int EqEq (v)
     Object v;
{
  VOID Push (OpStack, NameOperator (v));
  VOID Push (ExecStack, Cvx (NameFrom ("==")));
  return TRUE;
}

/*
 * Operator types.
 *
 * PostScript has 200 or so operators which can be divided into groups by the
 * arguments required.
 *
 * Type Real      causes an Integer argument to get floated
 * Type Unchecked causes an argument to be passed through whatever type it has
 * Type Number    causes all Number arguments to be Integer or Real
 *
 * An operator object comprises several pieces of information:
 *
 * 1    Number of arguments required
 * 2    Number of results which may be returned
 * 3    Number of items required on the ExecStack
 * 4    Number of items added to the ExecStack
 * 5    A List of argument types
 *
 * Functions requiring strange arguments can declare themselves to take few
 * arguments and then use some directly from the OpStack.
 *
 * The first four pieces of information are only required for checking Stack
 * over and underflow, thus operators are not required to hold to their
 * requests.
 *
 * After resolving any peculiar type rules, Exec will lookup the operator
 * name in the type dictionary for the argument type which is specified as
 * being the controlling argument.  The operator will receive the correct
 * number of arguments and will be expected to stack its own results. There
 * will be a fixed maximum limit of the number of arguments an operator
 * can declare for itself. This is likely to be the maximum that PostScript
 * uses, i.e. 6 or 7.
 */

/*
 * BZS - try to speed up a little
 * TMD - speed it up more by reordering the typechecking if statements
 *       and by adding an execuse flag which indicates if the exec stack
 *       is used or not.
 */
int ExecOperator (item)
     Object item;
{
  register struct op_struct *op = BodyOperator(item);
  register int i, opargs, res;
  Object arg[7];

  Self = NameOperator (item);
  opargs = op->arguments;
  if(op->results > Space(OpStack)) return Error (POpOverflow);
  if(opargs > Height(OpStack))     return Error (POpUnderflow);
  if(op->execuse) {
    if(op->execpush > Space(ExecStack)) return Error (PExecOverflow);
    if(op->execpop > Height(ExecStack)) return Error (PExecUnderflow);
  }

  /* Pop all arguments off of Opstack */
  for (i = opargs - 1; i >= 0; i--)  arg[i] = Pop (OpStack);

  /* Do type checking of the arguments */
  for (i = opargs - 1; i >= 0; i--) {
    register Type formal = op->argtypes[i], actual = TypeOf (arg[i]);

    if ((formal==Float && actual==Real) || formal==actual || formal==Poly)
      continue;
    if (formal == Float && actual == Integer) {
      /* Convert operand and continue */
      arg[i].type = Real;
      arg[i].u.Real = (float) arg[i].u.Integer;
      continue;
    }

    /* Incorrect type, push all args back on and report an error */
    for (i = 0; i < opargs; i++) FastPush (OpStack, arg[i]);
    return Error (PTypeCheck);
  }

  /* Perform the operation */
  switch (opargs) {
    case 0: res=(*(op->fn))(); break;
    case 1: res=(*(op->fn))(arg[0]); break;
    case 2: res=(*(op->fn))(arg[0],arg[1]); break;
    case 3: res=(*(op->fn))(arg[0],arg[1],arg[2]); break;
    case 4: res=(*(op->fn))(arg[0],arg[1],arg[2],arg[3]); break;
    case 5: res=(*(op->fn))(arg[0],arg[1],arg[2],arg[3],arg[4]); break;
    case 6: res=(*(op->fn))(arg[0],arg[1],arg[2],arg[3],arg[4],arg[5]); break;
    case 7: res=(*(op->fn))(arg[0],arg[1],arg[2],arg[3],arg[4],arg[5],arg[6]);
            break;
    default: Panic ("primitive with too many arguments");
  }

  /* If the result was false, push back all the operands */
  if (!res)
    for (i = 0; i < opargs; i++) FastPush (OpStack, arg[i]);

  return res;
```



```
}

static int Signature (op)
     Object op;
{
  Object tn;
  struct op_struct *ops = BodyOperator(op);
  int i;

  tn = NameFrom ("type");
  if (!OpCheck (0, ops->arguments + 1))
    return FALSE;
  for (i = 0; i < ops->arguments; i++)
    VOID Push (OpStack, Lookup (ops->argtypes[i], tn));
  VOID Push (OpStack, MakeInteger (ops->arguments));
  VOID Push (OpStack, MakeInteger (ops->results));

  return TRUE;
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

#include "graphics.h"

Path NewPath ();
UserPoint NewUserPoint (), IntToExt ();

HardPoint NewHardPoint (), ExtToInt ();

struct path_element *free_pelem = NULL;

#ifdef __STDC__
int Bezier(float,float,float,float,float,float,float,float);
#endif

#ifndef NewElem
void NewElem (type, p)
     enum pelem_type type;
     Path *p;
{
  p = free_pelem;

  if (p != NULL)
    free_pelem = p->next;
  else
    p = (Path) Malloc (sizeof (struct path_element));
  p->ptype = type;
}
#endif

#ifndef ElemFree
void ElemFree (p)
     Path p;
{
  PanicIf (p == NULL, "ElemFree given NULL");
  p->next = free_pelem;
  free_pelem = p;
}
#endif

Path NewPath ()
{
  Path res;

  NewElem (EHeader, res);

  return res->next = res->last = res;
}

#ifndef PathFree
PathFree (p)
     Path p;
{
  while (!EmptyPath (p))
    PathDelete (p);
  ElemFree (p);
}
#endif

Path PathCopy (path)
     Path path;
{
  Path p, tmp, new;

  NewElem (EHeader, new);
  new->next = new->last = new;

  for (p = path->next; p != path; p = p->next) {
    NewElem (p->ptype, tmp);
    tmp->pe = p->pe;
    PathInsert (new, tmp);
  }
  return new;
}

#ifndef CurEType
enum pelem_type CurEType (p)
     Path p;
{
  return p->last->ptype;
}
#endif

#ifndef EmptyPath
int EmptyPath (p)
     Path p;
{
  return p->next == p;
}
#endif

#ifndef PathDelete
PathDelete (p)
     Path p;
{
  Path res = p->last, ret;

  PanicIf (p->next == p, "delete from empty path");
  ret = PathRemove (res);
  ElemFree (ret);

  return TRUE;
}
#endif

Path PathRemove (p)
     Path p;
{
  p->last->next = p->next;
  p->next->last = p->last;

  return p;
}

#ifndef PathInsert
```



```
PathInsert (p, i)
     Path p, i;
{
  PanicIf (i == NULL, "PathInsert given NULL");
  i->next = p;
  i->last = p->last;
  p->last->next = i;
  p->last = i;
}
#endif

int MoveTo (p, np)
     Path p;
     HardPoint np;
{
  Path res;

  NewElem (EMove, res);
  res->pe.point = np;

  if (!EmptyPath (p) && CurEType (p) == EMove)
    PathDelete (p);
  gstate->cp_defined = TRUE;
  gstate->cp = np;
  PathInsert (p, res);
  return TRUE;
}

int LineTo (p, np)
     Path p;
     HardPoint np;
{
  Path res;

  NewElem (ELine, res);
  res->pe.point = np;

  gstate->cp = np;
  PathInsert (p, res);

  return TRUE;
}

int CurveTo (p, p0, p1, p2)
     Path p;
     HardPoint p0, p1, p2;
{
  Path new;

  NewElem (ECurve, new);

  new->pe.curve.x0 = p0;
  new->pe.curve.x1 = p1;
  new->pe.curve.x2 = p2;
  gstate->cp = p2;
  PathInsert (p, new);
  return TRUE;
}

float Normalise (ang)
     float ang;
{
  while (ang < -PI || ang > PI)
    if (ang < PI)  ang += 2 * PI;
    else           ang -= 2 * PI;
  return ang;
}

int Arc (p, dir, centre, radius, ang1, ang2)
     Path p;
     int dir;
     UserPoint centre;
     float radius, ang1, ang2;
{
  Path new;
  HardPoint P1,P2,P3;
  Matrix m;
  float diff,ca,sa,x,y,t1,t2;
  int quarters;

  /* Find absolute difference between the two angles */
  diff = dir > 0 ? ang2 - ang1 : ang1 - ang2;
  while (diff < (float) 0.0) diff += (float) (2 * PI);

  /* Translate center of arc to origin */
  m = gstate->CTM;
  m.tx += centre.x * m.A + centre.y * m.C;
  m.ty += centre.x * m.B + centre.y * m.D;

  /* Don’t bother rotating if starting angle is 0 */
  if(ang1 != (float) 0.0) {
    ca = cos(ang1); sa = sin(ang1);
    t1 = m.A;  t2 = m.B;
    m.A = m.A * ca + m.C * sa; m.B = m.B * ca + m.D * sa;
    m.C = m.C * ca - t1 * sa;  m.D = m.D * ca - t2 * sa;
  }

  /* For arcs larger than a quarter circle, draw quarter circles first */
  quarters = (int) (diff / (float) (PI/2 - .00001));
  if(quarters > 0) {
    diff -= quarters * (float) (PI/2);
    t2 = ((float) .5522847) * radius;  /* (4/3)*(sqrt(2) - 1) */
    if(dir > 0)
      do {
        /* Add a quarter circle, rotate m by 90 degrees */
        P1.hx = m.tx + radius*m.A + t2*m.C; P1.hy = m.ty + radius*m.B + t2*m.D;
        P3.hx = m.tx + radius*m.C;          P3.hy = m.ty + radius*m.D;
        P2.hx = P3.hx + t2*m.A;             P2.hy = P3.hy + t2*m.B;

        NewElem (ECurve, new);
        new->pe.curve.x0 = P1;  new->pe.curve.x1 = P2;  new->pe.curve.x2 = P3;
        gstate->cp = P3; PathInsert (p, new);

        RotatePosQ(m,t1);
      } while(--quarters > 0);
    else
      do {
        /* Add a quarter circle, rotate m by -90 degrees */
        P1.hx = m.tx + radius*m.A - t2*m.C; P1.hy = m.ty + radius*m.B - t2*m.D;
        P3.hx = m.tx - radius*m.C;          P3.hy = m.ty - radius*m.D;
        P2.hx = P3.hx + t2*m.A;             P2.hy = P3.hy + t2*m.B;

        NewElem (ECurve, new);
        new->pe.curve.x0 = P1;  new->pe.curve.x1 = P2;  new->pe.curve.x2 = P3;
        gstate->cp = P3; PathInsert (p, new);

        RotateNegQ(m,t1);
      } while(--quarters > 0);
```

```
  }

  /* Draw the rest of the arc, if there is any left */
  if(diff <= (float) 0.0) return TRUE;

  /* Compute cosine and sine of half the angle just once */
  ca = cos(diff/2);  sa = sin(diff/2);
  if (dir < 0) {
    /* Rotate m by -diff/2 and invert x axis */
    t1 = m.A; t2 = m.B;
    m.A = m.A * ca - m.C * sa;  m.B = m.B * ca - m.D * sa;
    m.C = -m.C * ca - t1 * sa;  m.D = -m.D * ca - t2 * sa;
    x = radius * ca; y = -radius * sa;
    P3.hx = x * m.A - y * m.C + m.tx;  P3.hy = x * m.B - y * m.D + m.ty;
  }
  else {
    /* Rotate m by diff/2 */
    t1 = m.A; t2 = m.B;
    m.A = m.A * ca + m.C * sa;  m.B = m.B * ca + m.D * sa;
    m.C = m.C * ca - t1 * sa;   m.D = m.D * ca - t2 * sa;
    x = radius * ca; y = radius * sa;
    P3.hx = x * m.A + y * m.C + m.tx;  P3.hy = x * m.B + y * m.D + m.ty;
  }

  /* Compute P1 and P2 and P3 from x, y and m */
  x = (4 * radius - x) / 3;
  y = sa * (radius / ca - x) / ca;
  P1.hx = x * m.A - y * m.C + m.tx;  P1.hy = x * m.B - y * m.D + m.ty;
  P2.hx = x * m.A + y * m.C + m.tx;  P2.hy = x * m.B + y * m.D + m.ty;

  NewElem (ECurve, new);
  new->pe.curve.x0 = P1;  new->pe.curve.x1 = P2;  new->pe.curve.x2 = P3;
  gstate->cp = P3; PathInsert (p, new);
  return TRUE;
}

ArcTo (path, cp, x1, y1, x2, y2, r, xt1, yt1, xt2, yt2)
     Path path;
     HardPoint cp;
     float x1, y1, x2, y2, r;
     float *xt1, *yt1, *xt2, *yt2;
{
  UserPoint p0;
  float x0, y0, x3, y3, dist,
  ang, anga, angb, absang, delta;
  int dir;

  p0 = IntToExt (cp); x0 = p0.x, y0 = p0.y;
  angb = atan2 (y2 - y1, x2 - x1);
  if (angb <= 0)
    angb += 2 * PI;
  anga = atan2 (y0 - y1, x0 - x1);
  if (anga <= 0)
    anga += 2 * PI;
  ang = angb - anga;
  if (ang <= 0)
    ang += 2 * PI;
  if (ang >= 2 * PI)
    ang -= 2 * PI;
  if (ang >= PI)
    ang -= 2 * PI;
  dir = ang <= 0 ? 1 : -1;
  dist = fabs (r / sin (ang / 2));
  absang = ang / 2 + anga;
  x3 = dist * cos (absang) + x1;
  y3 = dist * sin (absang) + y1;
  absang += PI;
  delta = (ang + (ang > 0 ? - PI : PI)) / 2;
  *xt1 = x3 + r * cos (absang - delta);
  *yt1 = y3 + r * sin (absang - delta);
  *xt2 = x3 + r * cos (absang + delta);
  *yt2 = y3 + r * sin (absang + delta);
  LineTo (path, ExtToInt (NewUserPoint (*xt1, *yt1)));
  Arc (path,
       dir,
       NewUserPoint (x3, y3),
       r, absang - delta,
       absang + delta);
  return TRUE;
}

int ClosePath (p)
     Path p;
{
  Path pp, res;

  if (!gstate->cp_defined) return TRUE;

  NewElem (EClose, res);
  res->pe.point.hx = gstate->CTM.tx;
  res->pe.point.hy = gstate->CTM.ty;

  for (pp = p; pp->ptype != EMove; pp = pp->last)
    ;
  PathInsert (p, res);

  NewElem (EMove, res);
  gstate->cp = res->pe.point = pp->pe.point;
  PathInsert (p, res);
  return TRUE;
}

UserPoint NewUserPoint (x, y)
     float x, y;
{
  UserPoint res;

  res.x = x; res.y = y;
  return res;
}

HardPoint NewHardPoint (x, y)
     float x, y;
{
  HardPoint res;

  res.hx = x; res.hy = y;
  return res;
}

HardPoint MoveHardPoint (a, b)
     HardPoint a, b;
{
  return NewHardPoint (a.hx + b.hx, a.hy + b.hy);
}

UserPoint MoveUserPoint (a, b)
     UserPoint a, b;
```



```
{
  return NewUserPoint (a.x + b.x, a.y + b.y);
}

float PointX (p)
     UserPoint p;
{
  return p.x;
}

float PointY (p)
     UserPoint p;
{
  return p.y;
}

/*
 * This algorithm subdivides a Bezier curve.
 * The algorithm is taken from:
 *
 *      "The Beta2-split: A special case of the Beta-spline Curve and
 *       Surface Representation." B. A. Barsky and A. D. DeRose. IEEE, 1985.
 *
 * A vector of control points W is transformed by two matrices into two
 * Vectors which describe the sub-polygons. This is done twice, once each
 * for the X and Y coordinates. The matrices required for the 'left' and
 * 'Right' sub-polygons are as follows:
 *
 *       /  1    0    0    0  \                 / 1/8  3/8  3/8  1/8 \
 * W  = |  1/2  1/2   0    0   |       W  =    |   0   1/4  1/2  1/4  |
 *  L   |  1/4  1/2  1/4   0   |        R      |   0    0   1/2  1/2  |
 *       \ 1/8  3/8  3/8  1/8 /                 \  0    0    0    1  /
 *
 * The algorithm subdivides the curve recursively until each sub-polygon
 * is deemed to be flat to within the required tolerance. Flatness is
 * defined as the greater distance of the inner control points (P1 & P2)
 * from the line through the outer ones (P0 & P3):
 *
 *                                                    P2
 *                                    /-------------X
 *            P1          ------------             |\_
 *             _X---------/                         |  \_
 *           _/ |                                   |    \_
 *         _/   |                                   |      \_
 *       _/     |                                   |        \_
 *     _/       | D1                             D2 |          \_
 *   _/         |                                   |            \_
 *  /           |                                   |              \
 * X----------------------------------------------------------------X
 * P0
 *                                                           P3
 */

static Path path;

#ifdef __STDC__
int Bezier (float x0,float y0,float x1,float y1,
            float x2,float y2,float x3,float y3)
#else
int Bezier (x0,y0, x1, y1, x2, y2, x3, y3)
     float x0, y0, x1, y1, x2, y2, x3, y3;
#endif
{
  Path new;
  HardPoint P3;
  float xsave,ysave,y,opp,adj;
  float flatness;

  /* If bezier curve is a point, return a path to (x3,y3) */
  opp = y3-y0;    adj = x3-x0;
  if(opp == (float) 0.0 && adj == (float) 0.0) {
    P3.hx = x3;   P3.hy = y3;
    NewElem (ELine, new);
    gstate->cp = new->pe.point = P3;
    PathInsert (path, new);
    return TRUE;
  }

  /* If flatness of (x1,y1) and (x2,y2) are good, return a path to (x3,y3) */
  flatness = gstate->flatness * sqrt(opp*opp + adj*adj);
  y = - opp*(x1-x0) + adj*(y1-y0);
  if (y >= -flatness && y <= flatness) {
    y = - opp*(x2-x0) + adj*(y2-y0);
    if (y >= -flatness && y <= flatness) {
      P3.hx = x3;   P3.hy = y3;
      NewElem (ELine, new);
      gstate->cp = new->pe.point = P3;
      PathInsert (path, new);
      return TRUE;
    }
  }

  /* Subdivide bezier curve into two smaller ones and recur */
  xsave = (x0+x3)/8 + 3*(x1+x2)/8;
  ysave = (y0+y3)/8 + 3*(y1+y2)/8;
  if (!Bezier(x0,                  y0,
              (x0+x1)/2,           (y0+y1)/2,
              (x0+x2)/4 + x1/2,    (y0+y2)/4 + y1/2,
              xsave,               ysave))
    return 0;
  return Bezier(xsave,            ysave,
                (x1+x3)/4 + x2/2, (y1+y3)/4 + y2/2,
                (x2+x3)/2,        (y2+y3)/2,
                x3,               y3);
}

static int RevSeg (new, p, next)
     Path new, next;
     HardPoint p;
{
  switch (next->ptype) {
  case EClose:
  case EHeader:
  case EMove:   return MoveTo (new, p);
  case ELine:   return LineTo (new, p);
  case ECurve:  return CurveTo (new, next->pe.curve.x1, next->pe.curve.x0, p);
  }
  Panic ("RevSeg: unknown pe type");
  return TRUE;
}

Path ReversePath (path)
     Path path;
{
  Path p, new, pnew;
  int closed = FALSE;

  NewElem (EHeader, new);
```

```c
  new->next = new->last = new;
  pnew = new;

  for (p = path->last; p != path; p = p->last)
    switch (p->ptype) {
    case EClose:
      closed = TRUE;
      break;

    case EMove:
      if (!RevSeg (new, p->pe.point, p->next)) {
        PathFree (new);
        return NULL;
      }
      if (closed) {
        if (!ClosePath (new)) {
          PathFree (new);
          return NULL;
        }
      }
      new = new->next;
      closed = FALSE;
      break;

    case ELine:
      if (!RevSeg (new, p->pe.point, p->next)) {
        PathFree (new);
        return NULL;
      }

      break;

    case ECurve:
      if (!RevSeg (new, p->pe.curve.x2, p->next)) {
        PathFree (new);
        return NULL;
      }

      break;

    default:
      Panic ("ReversePath: unknown element type");
    }
  PathFree (p);

  return pnew;
}

Path FlattenPath (arg)
     Path arg;
{
  Path p, pp, new, add;
  HardPoint lastp;

  NewElem (EHeader, new);
  new->next = new->last = new;
  path = new;
  for (p = arg->next; p != arg; p = p->next)
    switch (p->ptype) {
      case EMove:
        lastp = p->pe.point;
        NewElem (EMove, add);
        add->pe.point = lastp;
        if (!EmptyPath (new) && CurEType (new) == EMove) PathDelete (new);
        gstate->cp_defined = TRUE;
        gstate->cp = lastp;
        PathInsert (new, add);
        break;
      case ELine:
        lastp = p->pe.point;
        NewElem (ELine, add);
        add->pe.point = lastp;
        gstate->cp = lastp;
        PathInsert (new, add);
        break;
      case EClose:
        if (gstate->cp_defined) {
          NewElem (EClose, add);
          add->pe.point.hx = gstate->CTM.tx;
          add->pe.point.hy = gstate->CTM.ty;

          for (pp = new; pp->ptype != EMove; pp = pp->last)
            ;
          PathInsert (new, add);

          NewElem (EMove, add);
          gstate->cp = add->pe.point = pp->pe.point;
          PathInsert (new, add);
        }
        break;
      case ECurve:
        if (!Bezier(lastp.hx,          lastp.hy,
                    p->pe.curve.x0.hx, p->pe.curve.x0.hy,
                    p->pe.curve.x1.hx, p->pe.curve.x1.hy,
                    p->pe.curve.x2.hx, p->pe.curve.x2.hy)) {
          PathFree (new);
          return NULL;
        }
        lastp = p->pe.curve.x2;
        break;
      default:
        Panic ("Flattenpath discovers unknown path element type");
        break;
    }
  return path;
}

int CloseAll (path)
     Path path;
{
  /* closes all open portions of a path *in place* */
  Path p;
  enum pelem_type last_type = EHeader;

  for (p = path->next; p != path; last_type = p->ptype, p = p->next)
    if (p->ptype == EMove && last_type != EClose && last_type != EHeader)
      if (!ClosePath (p))
        return Error (PLimitCheck);
  if (last_type == EMove) {
    PathDelete (p);
    return TRUE;
  }
  if (last_type == EHeader || last_type == EClose)
    return TRUE;
  return ClosePath (p);
}

SetPath (p, v)
```



```c
     Path *p, v;
{
  PathFree ((*p));
  *p = v;
}

Bound (left, right, top, bottom, p)
     float *left, *right, *top, *bottom;
     HardPoint p;
{
  if (p.hx < *left)     *left   = p.hx;
  if (p.hx > *right)    *right  = p.hx;
  if (p.hy < *bottom)   *bottom = p.hy;
  if (p.hy > *top)      *top    = p.hy;
}

int PathBBox (left, right, top, bottom)
     float *left, *right, *top, *bottom;
{
  Path p;

  if (!gstate->cp_defined || EmptyPath (gstate->path))
    return Error (PNoCurrentPoint);

  *left = *right = gstate->cp.hx;
  *top = *bottom = gstate->cp.hy;

  for (p = gstate->path->next; p != gstate->path; p = p->next)
    switch (p->ptype) {
    case EMove:
    case ELine:
      Bound (left, right, top, bottom, p->pe.point);
      break;
    case EClose:
      break;
    case ECurve:
      Bound (left, right, top, bottom, p->pe.curve.x0);
      Bound (left, right, top, bottom, p->pe.curve.x1);
      Bound (left, right, top, bottom, p->pe.curve.x2);
      break;
    case EHeader:
      Panic ("PathBBox, header found.");
    default:
      break;
      Panic ("PathBBox, unknown path element type.");
      break;
    }

  return TRUE;
}

UserBound (left, right, top, bottom, p)
     float *left, *right, *top, *bottom;
     UserPoint p;
{
  if (p.x < *left)      *left   = p.x;
  if (p.x > *right)     *right  = p.x;
  if (p.y < *bottom)    *bottom = p.y;
  if (p.y > *top)       *top    = p.y;
}
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

#include "graphics.h"

#define PATHFORALLDEPTH 500

static int PCurrentPoint ();
static int PMoveTo ();
static int PRMoveTo ();
static int PLineTo ();
static int PRLineTo ();
static int PArc ();
static int PArcN ();
static int PArcTo ();
static int PCurveTo ();
static int PRCurveTo ();
static int PClosePath ();
static int PPathBBox ();
static int PReversePath ();
static int PPathForAll ();

static int PathForAll ();
static int PPathProc ();

int PFlattenPath ();

Object OpPathForAll;

InitPath ()
{
  OpPathForAll = MakeOp ("(oppathforall)", PathForAll, 0, 6, 7, 7);

  InstallOp ("newpath",         PNewPath,       0, 0, 0, 0);
  InstallOp ("currentpoint",    PCurrentPoint,  0, 2, 0, 0);
  InstallOp ("moveto",          PMoveTo,        2, 0, 0, 0, Float, Float);
  InstallOp ("rmoveto",         PRMoveTo,       2, 0, 0, 0, Float, Float);
  InstallOp ("lineto",          PLineTo,        2, 0, 0, 0, Float, Float);
  InstallOp ("rlineto",         PRLineTo,       2, 0, 0, 0, Float, Float);
  InstallOp ("arc",             PArc,           5, 0, 0, 0, Float, Float,
             Float, Float, Float);
  InstallOp ("arcn",            PArcN,          5, 0, 0, 0, Float, Float,
             Float, Float, Float);
  InstallOp ("arcto",           PArcTo,         5, 4, 0, 0, Float, Float,
             Float, Float, Float);
  InstallOp ("curveto",         PCurveTo,       6, 0, 0, 0, Float, Float,
             Float, Float, Float, Float);
  InstallOp ("rcurveto",        PRCurveTo,      6, 0, 0, 0, Float,
             Float, Float, Float, Float, Float);
  InstallOp ("closepath",       PClosePath,     0, 0, 0, 0);
  InstallOp ("pathbbox",        PPathBBox,      0, 4, 0, 0);
  InstallOp ("flattenpath",     PFlattenPath,   0, 0, 0, 0);
  InstallOp ("reversepath",     PReversePath,   0, 0, 0, 0);
  InstallOp ("pathforall",      PPathForAll,    4, 0, 0, 6, Array, Array,
              Array, Array);
  InstallOp ("pathproc",        PPathProc,      0, 1, 0, 0);
  gstate->path = NewPath ();
  gstate->clip = NewPath ();
}

int PNewPath ()
{
  Path new;

  gstate->cp_defined = FALSE;
  PathFree (gstate->path);
  NewElem(EHeader,new);
  new->next = new->last = new;
  gstate->path = new;
  return TRUE;
}

static int PCurrentPoint ()
{
  UserPoint cp;

  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  cp = IntToExt (gstate->cp);
  VOID Push (OpStack, MakeReal (PointX (cp)));
  VOID Push (OpStack, MakeReal (PointY (cp)));
  return TRUE;
}

static int PMoveTo (x, y)
     Object x, y;
{
  UserPoint new;
  extern FILE *vfp;
  char foo[BUFSIZ];

  if(vfp != NULL) {
    sprintf(foo, "PMoveto XY: %f %f", BodyReal(x), BodyReal(y));
    Message(foo);
  }
  new.x = BodyReal(x);    new.y = BodyReal(y);
  return MoveTo (gstate->path, ExtToInt (new));
}

static int PRMoveTo (x, y)
     Object x, y;
{
  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  return MoveTo (gstate->path, ExtToInt (MoveUserPoint (NewUserPoint (BodyReal (x), BodyR
}

static int PLineTo (x, y)
     Object x, y;
{
  UserPoint new;

  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  new.x = BodyReal(x);    new.y = BodyReal(y);
  return LineTo (gstate->path, ExtToInt (new));
```



```c
}

static int PRLineTo (x, y)
     Object x, y;
{
  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  return LineTo (gstate->path, ExtToInt (MoveUserPoint (NewUserPoint (BodyReal (x), BodyR
}

static int PArc (x, y, rad, ang1, ang2)
     Object x, y, rad, ang1, ang2;
{
  UserPoint centre, start;
  float a1 = Rad (BodyReal (ang1)), a2 = Rad (BodyReal (ang2));
  float r = BodyReal (rad);

  start = NewUserPoint (cos (a1) * r, sin (a1) * r);
  centre = NewUserPoint (BodyReal (x), BodyReal (y));
  /* DCJ:
   * Compiler barfed on original line.  Don't blame it.
   */
  if (EmptyPath (gstate->path)) {
    if (!MoveTo (gstate->path, ExtToInt (MoveUserPoint (centre, start))))
      return FALSE;
  } else {
    if (!LineTo (gstate->path, ExtToInt (MoveUserPoint (centre, start))))
      return FALSE;
  }

  /*    if (!(*(EmptyPath (gstate->path) ? MoveTo : LineTo)) (gstate->path, ExtToInt (Mov
   * return FALSE;
   */
  return Arc (gstate->path, 1, centre, r, a1, a2);
}

static int PArcN (x, y, rad, ang1, ang2)
     Object x, y, rad, ang1, ang2;
{
  UserPoint centre, start;
  float a1 = Rad (BodyReal (ang1)), a2 = Rad (BodyReal (ang2));
  float r = BodyReal (rad);

  start = NewUserPoint (cos (a1) * r, sin (a1) * r);
  centre = NewUserPoint (BodyReal (x), BodyReal (y));
  /* DCJ:
   * Compiler barfed on original line.  Don't blame it.
   */
  if (EmptyPath (gstate->path)) {
    if (!MoveTo (gstate->path, ExtToInt (MoveUserPoint (centre, start))))
      return FALSE;
  } else {
    if (!LineTo (gstate->path, ExtToInt (MoveUserPoint (centre, start))))
      return FALSE;
  }

  /*    if (!(*(EmptyPath (gstate->path) ? MoveTo : LineTo)) (gstate->path, ExtToInt (Mov
   * return FALSE;
   */
  return Arc (gstate->path, -1, centre, r, a1, a2);
}

static int PArcTo (X1, Y1, X2, Y2, rad)
     Object X1, Y1, X2, Y2, rad;
{
  float xt1, xt2, yt1, yt2;
  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);

  return ArcTo (gstate->path, gstate->cp,
                BodyReal (X1), BodyReal (Y1), BodyReal (X2), BodyReal (Y2), BodyReal (rad
                &xt1, &yt1, &xt2,  &yt2) &&
                  Push (OpStack, MakeReal (xt1)) &&
                    Push (OpStack, MakeReal (yt1)) &&
                      Push (OpStack, MakeReal (xt2)) &&
                        Push (OpStack, MakeReal (yt2));
}

static int PCurveTo (x0, y0, x1, y1, x2, y2)
     Object x0, y0, x1, y1, x2, y2;
{
  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  return CurveTo (gstate->path,
                  ExtToInt (NewUserPoint (BodyReal (x0), BodyReal (y0))),
                  ExtToInt (NewUserPoint (BodyReal (x1), BodyReal (y1))),
                  ExtToInt (NewUserPoint (BodyReal (x2), BodyReal (y2))));
}

static int PRCurveTo (x0, y0, x1, y1, x2, y2)
     Object x0, y0, x1, y1, x2, y2;
{
  UserPoint cp;

  if (!gstate->cp_defined)
    return Error (PNoCurrentPoint);
  cp = IntToExt (gstate->cp);
  return CurveTo (gstate->path,
                  ExtToInt (MoveUserPoint (cp,
                                           NewUserPoint (BodyReal (x0),
                                                         BodyReal (y0)))),
                  ExtToInt (MoveUserPoint (cp,
                                           NewUserPoint (BodyReal (x1),
                                                         BodyReal (y1)))),
                  ExtToInt (MoveUserPoint (cp,
                                           NewUserPoint (BodyReal (x2),
                                                         BodyReal (y2)))));
}

static int PClosePath ()
{
  return ClosePath (gstate->path);
}

int PFlattenPath ()
{
  Path res;

  if ((res = FlattenPath (gstate->path)) == NULL)
    return Error (PLimitCheck);
  SetPath (&gstate->path, res);
  return TRUE;
}

static int PReversePath ()
{
  Path res;
```

```
  if ((res = ReversePath (gstate->path)) == NULL)
    return Error (PLimitCheck);
  SetPath (&gstate->path, res);
  return TRUE;
}

static Path pathforall [PATHFORALLDEPTH], *pathp = pathforall;
static int path_depth = 0;

static int PPathForAll (move, line, curve, close)
     Object move, line, curve, close;
{
  Path res;

  if ((res = PathCopy (gstate->path)) == NULL || path_depth == PATHFORALLDEPTH - 1)
    return Error (PLimitCheck);
  *++pathp = res;
  ++path_depth;
  VOID Push (ExecStack, Nil);
  VOID Push (ExecStack, move);
  VOID Push (ExecStack, line);
  VOID Push (ExecStack, curve);
  VOID Push (ExecStack, close);
  VOID Push (ExecStack, OpPathForAll);
  return TRUE;
}

static int PathForAll ()
{
  Path res;
  Object move, line, curve, close, fn;
  UserPoint p;

  close = Pop (ExecStack);
  curve = Pop (ExecStack);
  line = Pop (ExecStack);
  move = Pop (ExecStack);
  if (EmptyPath (*pathp)) {
    PathFree (*pathp);
    *pathp--;
    --path_depth;
    VOID Pop (ExecStack);
    return TRUE;
  }
  res = (*pathp)->next;
  switch (res->ptype) {
  case EMove:
    p = IntToExt (res->pe.point);
    VOID Push (OpStack, MakeReal (p.x));
    VOID Push (OpStack, MakeReal (p.y));
    fn = move;
    break;

  case ELine:
    p = IntToExt (res->pe.point);
    VOID Push (OpStack, MakeReal (p.x));
    VOID Push (OpStack, MakeReal (p.y));
    fn = line;
    break;

  case ECurve:
    p = IntToExt (res->pe.curve.x0);
    VOID Push (OpStack, MakeReal (p.x));
    VOID Push (OpStack, MakeReal (p.y));
    p = IntToExt (res->pe.curve.x1);
    VOID Push (OpStack, MakeReal (p.x));
    VOID Push (OpStack, MakeReal (p.y));
    p = IntToExt (res->pe.curve.x2);
    VOID Push (OpStack, MakeReal (p.x));
    VOID Push (OpStack, MakeReal (p.y));
    fn = curve;
    break;

  case EClose:
    fn = close;
    break;

  default:
    Panic ("OpPathForAll - encounters unknown path element");
  }
  VOID Push (ExecStack, move);
  VOID Push (ExecStack, line);
  VOID Push (ExecStack, curve);
  VOID Push (ExecStack, close);
  VOID Push (ExecStack, OpPathForAll);
  VOID Push (ExecStack, fn);

  PathDelete (res->next);
  return TRUE;
}

static int PPathProc ()
{
  Object *body, *p;
  int sum = 0;
  Path pa;
  UserPoint po;

  for (pa = gstate->path->next; pa != gstate->path; pa = pa->next)
    switch (pa->ptype) {
    case EMove: case ELine: sum += 3; break;
    case ECurve: sum += 7; break;
    case EClose: ++sum; break;
    }

  p = body = (Object *) Malloc ((unsigned) sizeof (Object) * sum);
  for (pa = gstate->path->next; pa != gstate->path; pa = pa->next)
    switch (pa->ptype) {
    case EMove:
      po = IntToExt (pa->pe.point);
      *p++ = MakeReal (po.x);
      *p++ = MakeReal (po.y);
      *p++ = Cvx (NameFrom ("moveto"));
      break;

    case ELine:
      po = IntToExt (pa->pe.point);
      *p++ = MakeReal (po.x);
      *p++ = MakeReal (po.y);
      *p++ = Cvx (NameFrom ("lineto"));
      break;

    case ECurve:
      po = IntToExt (pa->pe.curve.x0);
      *p++ = MakeReal (po.x);
      *p++ = MakeReal (po.y);
      po = IntToExt (pa->pe.curve.x1);
      *p++ = MakeReal (po.x);
```



```
      *p++ = MakeReal (po.y);
      po = IntToExt (pa->pe.curve.x2);
      *p++ = MakeReal (po.x);
      *p++ = MakeReal (po.y);
      *p++ = Cvx (NameFrom ("curve"));
      break;

    case EClose:
      *p++ = Cvx (NameFrom ("closepath"));
      break;
    }
  return Push (OpStack, Cvx (MakeArray (body, sum)));
}

static int PPathBBox ()
{
  UserPoint right_top, left_bottom, left_top, right_bottom;
  float left, right, top, bottom, uleft, uright, utop, ubottom;

  if (!PathBBox (&left, &right, &top, &bottom))
    return FALSE;

  left_bottom   = IntToExt (NewHardPoint (left, bottom));
  right_bottom  = IntToExt (NewHardPoint (right, bottom));
  right_top     = IntToExt (NewHardPoint (right, top));
  left_top      = IntToExt (NewHardPoint (left, top));

  uleft = uright = left_bottom.x; utop = ubottom = left_bottom.y;
  UserBound (&uleft, &uright, &utop, &ubottom, left_top);
  UserBound (&uleft, &uright, &utop, &ubottom, right_top);
  UserBound (&uleft, &uright, &utop, &ubottom, right_bottom);

  VOID Push (OpStack, MakeReal (uleft));
  VOID Push (OpStack, MakeReal (ubottom));
  VOID Push (OpStack, MakeReal (uright));
  VOID Push (OpStack, MakeReal (utop));

  return TRUE;
}
```

```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#define CBUTT   0
#define CROUND  1
#define CSQUARE 2

#define JMITRE  0
#define JROUND  1
#define JBEVEL  2

#define STROKE_FILL 0
#define STROKE_THIN 1

#define FILL_SINGLE             0
#define FILL_CACHE_DOUBLE       1
#define FILL_DOUBLE             2
#define FILL_DOUBLE_STROKE      3

#ifdef MACINTOSH
#define MAXPATHELEMENTS 3000
#else
#define MAXPATHELEMENTS 3000
#endif

typedef struct hard_point { float hx, hy; } HardPoint;

enum pelem_type { EHeader, EMove, ELine, EArc, ECurve, EClose };

struct path_element
{
  enum pelem_type ptype;
  union {
    HardPoint point;
    struct arc { int dir; HardPoint centre; float radius, from, to; } arc;
    struct bezier { HardPoint x0, x1, x2; } curve;
  } pe;
  struct path_element *next, *last;
};

typedef struct path_element *Path;

extern HardPoint NewHardPoint (), ExtToInt (), MoveHardPoint ();
extern Path NewPath (), PathCopy (), NewMove ();
extern int PNewPath (), PInitMatrix ();
extern Path NewClipPath (), PathRemove (), ReversePath (), FlattenPath ();
extern float Normalise (), PointX (), PointY ();
```



```
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

typedef struct point { float x, y; } UserPoint;

extern Matrix PointTranslate ();
extern UserPoint NewUserPoint (), IntToExt (), MoveUserPoint ();
extern int MoveTo (), LineTo ();
extern Object AssignMatrix ();
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not sold for
 * profit or incorporated into a product except under licence from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes to the
 * source made by persons other than the author should be marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static int Copy ();

int PolyFirst (), PolySecond (), PolyThird (), PolyPair ();

InitPoly ()
{
  Lbracket = Cvx (NameFrom("["));
  Rbracket = Cvx (NameFrom("]"));

  InstallOp ("token",       PolyFirst,  1, 1, 0, 0, Poly);
  InstallOp ("copy",        Copy,       0, 0, 0, 0);
  InstallOp ("length",      PolyFirst,  1, 1, 0, 0, Poly);
  InstallOp ("forall",      PolySecond, 2, 2, 0, 0, Poly, Poly);
  InstallOp ("get",         PolySecond, 2, 2, 0, 0, Poly, Poly);
  InstallOp ("put",         PolyThird,  3, 3, 0, 0, Poly, Poly, Poly);
  InstallOp ("getinterval", PolyThird,  3, 3, 0, 0, Poly, Poly, Poly);
  InstallOp ("putinterval", PolyThird,  3, 3, 0, 0, Poly, Poly, Poly);
  InstallOp ("signature",   PolyFirst,  1, 0, 0, 0, Poly);
}

Object Parse (f)
     Object f;
{
  Object res;
  int c;

  if(BodyFile(f)->file_type == StreamFile) {
    FILE *fp = BodyFile(f)->f.f_ptr;
    for (;;)
      switch (c = getc(fp)) {
        default:
/*        BodyFile(f)->available++; */
          ungetc(c,fp);
          res = ParseId (f);
          if (!(TypeOf (res) == Integer || TypeOf (res) == Real))
            res.flags |= EXECUTABLE;
          return res;

        case '/': return ParseId (f);
        case '{': res = ParseArray (f); res.flags |= EXECUTABLE; return res;
        case '}': return True;
        case '[': return Lbracket;
        case ']': return Rbracket;
        case '<': return ParseHexString (f);
        case '(': return ParseString (f);
        case ')': return Absent;

        case EOF: return False;

        case ' ':
        case '\t':
        case '\n': continue;

        case '%':
          while ((c = getc(fp)) != '\n' && c != EOF)
            ;
          if(c == EOF) return False;
          continue;
      }
  }
  else {
    for (;;)
      switch (c = Getch (f)) {
        default:
          Ungetch (f, c);
          res = ParseId (f);
          if (!(TypeOf (res) == Integer || TypeOf (res) == Real))
            res.flags |= EXECUTABLE;
          return res;

        case '/': return ParseId (f);
        case '{': res = ParseArray (f); res.flags |= EXECUTABLE; return res;
        case '}': return True;
        case '[': return Lbracket;
        case ']': return Rbracket;
        case '<': return ParseHexString (f);
        case '(': return ParseString (f);
        case ')': return Absent;

        case EOF: return False;

        case ' ':
        case '\t':
        case '\n': continue;

        case '%':
          while ((c = Getch (f)) != '\n' && c != EOF)
            ;
          if(c == EOF) return False;
          continue;
      }
  }
}

/*
 * The following are a few of the polymorphic generic routines which actually
 * get called by user PostScript.
 * They call type-checked routines on behalf of the types of their arguments.
 *
 */

int PolyFirst (arg1) Object arg1;       /* type dictionary choice determined by top of st
 {
        if (!Apply (TypeOf (arg1)))
                return Error (PTypeCheck);
        return Push (OpStack, arg1);
 }

int PolySecond (arg1, arg2) Object arg1, arg2;  /* type dictionary choice determined by f
 {
        if (!Apply (TypeOf (arg1)))
                return Error (PTypeCheck);
        return Push (OpStack, arg1), Push (OpStack, arg2);
 }
```



```
int PolyThird (arg1, arg2, arg3) Object arg1, arg2, arg3;       /* type dictionary choice
 {
        if (!Apply (TypeOf (arg1)))
                return Error (PTypeCheck);
        return Push (OpStack, arg1), Push (OpStack, arg2), Push (OpStack, arg3);
 }

int PolyPair (arg1, arg2) Object arg1, arg2;
 {
        if (TypeOf (arg1) == Integer && TypeOf (arg2) == Real)
                arg1 = RealInteger (arg1);
        else if (TypeOf (arg2) == Integer && TypeOf (arg1) == Real)
                arg2 = RealInteger (arg2);
        if (TypeOf (arg1) != TypeOf (arg2))
                return Error (PTypeCheck);
        if (!Apply (TypeOf (arg1)))
                return Error (PTypeCheck);
        return Push (OpStack, arg1), Push (OpStack, arg2);
 }

int Apply (type) Type type;
 {
        Object fn;

        fn = Lookup (type, Self);
        if (TypeOf (fn) == Condition)
                return Error (PTypeCheck);
        else
                return Push (ExecStack, fn);
 }

static int Copy ()      /* any1 . . . anyn N    --- any1 . . . anyn any1 . . . anyn */
                /* other1  other2       --- subother2 */
 {
        Object object1, object2;
        int h = Height (OpStack);

        object2 = Pop (OpStack);
        if (h == 0)
                return Error (POpUnderflow);
        else if (TypeOf (object2) == Integer)
         {
                int n = BodyInteger (object2);

                if (n < 0 || n >= h)
                        return Error (PRangeCheck);
                else if (h - 1 + n > MaxStack (OpStack))
                        return Error (POpOverflow);
                else
                 {
                        int i;

                        for (i = h - 1 - n; i < h; i++)
                                OpStack->stack_body[i+n] = OpStack->stack_body[i];
                        OpStack->stack_fill += n;
                        return TRUE;
                 }
         }
        object1 = Top (OpStack);
        VOID Push (OpStack, object2);
        if (TypeOf (object1) != TypeOf (object2))
                return Error (PTypeCheck);
        else if (!rCheck (object1) || !wCheck (object2))
                return Error (PInvAccess);
        else
                return Apply (TypeOf (object1));
 }
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static int PCvs (), PCvrs (), PCvn (), PType (), PCvlit(), PxCheck ();
int PCvx ();

Object type;

InitProperty ()
{
  type = NameFrom ("type");

  InstallOp ("type",            PType,          1, 1, 0, 0, Poly);
  InstallOp ("cvlit",           PCvlit,         1, 1, 0, 0, Poly);
  InstallOp ("cvn",             PCvn,           1, 1, 0, 0, String);
  InstallOp ("cvrs",            PCvrs,          3, 1, 0, 0, Integer, Integer, String);
  InstallOp ("cvs",             PCvs,           2, 1, 0, 0, Poly, String);
  InstallOp ("cvx",             PCvx,           1, 1, 0, 0, Poly);
  InstallOp ("xcheck",          PxCheck,        1, 1, 0, 0, Poly);
  InstallOp ("cvi",             PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("cvr",             PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("readonly",        PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("rcheck",          PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("wcheck",          PolyFirst,      1, 1, 0, 0, Poly);
  InstallOp ("executeonly",     PolyFirst,      1, 1, 0, 0, Poly);
}

static int PType (item)
     Object item;
{
  return Push (OpStack, Lookup (TypeOf (item), type));
}

static int PCvs (item, string)
     Object item, string;
{
  int l;
  Object t;

  t = Lookup (TypeOf (item), NameFrom ("cvs"));
  if (TypeOf (t) == Condition)
    if (lengthString (string) < (l = strlen ("--nostringval--")))
      return Error (PRangeCheck);
    else {
      putIString (string, 0, StringFrom ("--nostringval--"));
      return Push (OpStack, getIString (string, 0, l));
    } else {
      VOID Push (OpStack, item);
      VOID Push (OpStack, string);
      return Apply (TypeOf (item));
    }
}

static int PCvlit (item)
     Object item;
{
  return Push (OpStack, Cvlit (item));
}

static int PCvn (string)
     Object string;
{
  return Push (OpStack, SameFlags (string, Cvn (string)));
}

static int PCvrs (num, base, string)
     Object num, base, string;
{
  unsigned n = BodyInteger (num);
  unsigned char buf [BUFSIZE], *p = buf, *q = BodyString (string);
  int b, length;

  if (!wCheck (string))
    return Error (PInvAccess);
  else if ((b = BodyInteger (base)) < 2 || b > 36)
    return Error (PRangeCheck);
  do {
    int dig_val = n % b;

    n /= b;
    *p++ = dig_val >= 10 ? 'A' + dig_val - 10 : '0' + dig_val;
  } while (n != 0);

  if ((length = p - buf) > lengthString (string))
    return Error (PRangeCheck);

  while (--p >= buf)
    *q++ = *p;
  return Push (OpStack, getIString (string, 0, length));
}

/*ARGSUSED*/
int NoStringVal (v, string)
     Object v, string;
{
  char *mess = "--nostringval--";
  int length = strlen (mess);

  if (lengthString (string) < length)
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), mess, length);
  return Push (OpStack, getIString (string, 0, length));
}

int PCvx (item)
     Object item;
{
  return Push (OpStack, Cvx (item));
}

static int PxCheck (item)
     Object item;
{
  return Push (OpStack, MakeBool (xCheck (item)));
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static Object OpFor;

static int For (), PFor (), Eq (), Lt (), Le (), Gt (), Ge (), Identity ();
static int Abs (), Add (), Sub (), Mul (), Div (), Neg (), EqEq (), Cvi ();
static int Cvs (), Exp (), Sqrt (), Sin (), Cos (), Atan (), Ln (), Log ();
static int Ceiling (), Floor (), Round (), Truncate ();

InitReal ()
{
  OpFor = MakeOp ("(forreal)", For, 0, 1, 5, 6, Real, Real, Real, Array);

  TypeInstallOp (Real, "cvr",   Identity,       1, 1, 0, 0, Real);
  TypeInstallOp (Real, "cvi",   Cvi,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "cvs",   Cvs,            2, 1, 0, 0, Real, String);
  TypeInstallOp (Real, "==",    EqEq,           1, 0, 0, 0, Real);
  TypeInstallOp (Real, "for",   PFor,           4, 0, 0, 6, Real, Real, Real, Array);
  TypeInstallOp (Real, "eq",    Eq,             2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "lt",    Lt,             2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "le",    Le,             2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "gt",    Gt,             2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "ge",    Ge,             2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "abs",   Abs,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "add",   Add,            2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "sub",   Sub,            2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "mul",   Mul,            2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "div",   Div,            2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "neg",   Neg,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "exp",   Exp,            2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "sqrt",  Sqrt,           1, 1, 0, 0, Real);
  TypeInstallOp (Real, "sin",   Sin,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "cos",   Cos,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "atan",  Atan,           2, 1, 0, 0, Real, Real);
  TypeInstallOp (Real, "ln",    Ln,             1, 1, 0, 0, Real);
  TypeInstallOp (Real, "log",   Log,            1, 1, 0, 0, Real);
  TypeInstallOp (Real, "ceiling", Ceiling,      1, 1, 0, 0, Real);
  TypeInstallOp (Real, "floor", Floor,          1, 1, 0, 0, Real);
  TypeInstallOp (Real, "round", Round,          1, 1, 0, 0, Real);
  TypeInstallOp (Real, "truncate",Truncate,     1, 1, 0, 0, Real);
}

Object MakeReal (f)
     float f;
{
  Object res;

  MakeObject (res, Real);
  res.u.Real = f;

  return res;
}

static Object Make (f)
     float f;
{
  Object res;

  MakeObject (res, Real);
  res.u.Real = f;

  return res;
}

static int Identity (item)
     Object item;
{
  return Push (OpStack, item);
}

static int Cvi (item)
     Object item;
{
  return Push (OpStack, IntReal (item));
}

static int Cvs (v, string)
     Object v, string;
{
  char buf [BUFSIZ];
  int length;

  VOID sprintf (buf, "%g", BodyReal (v));
  if ((length = strlen (buf)) > lengthString (string))
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), buf, length);
  return Push (OpStack, getIString (string, 0, length));
}

static int EqEq (v)
     Object v;
{
  printf ("%g", BodyReal (v));
  return TRUE;
}

#ifndef BodyReal
float BodyReal (item)
     Object item;
{
  return item.u.Real;
}
#endif

#define Body(I) ((I).u.Real)

#ifndef Body
static float Body (item)
     Object item;
{
  return item.u.Real;
}
#endif

Object RealInteger (o)
```

```
     Object o;
{
  return Make ((float) BodyInteger (o));
}

static int PFor (initial, increment, limit, proc)
     Object initial, increment, limit, proc;
{
  VOID Push (ExecStack, Nil);
  VOID Push (ExecStack, increment);
  VOID Push (ExecStack, limit);
  VOID Push (ExecStack, initial);
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpFor);

  return TRUE;
}

static int For ()
{
  Object current, limit, increment, proc;

  proc    = Pop (ExecStack);
  current   = Pop (ExecStack);
  limit   = Pop (ExecStack);
  increment = Pop (ExecStack);

  if (Body (increment) > 0
      ? Body (current) > Body (limit)
      : Body (current) < Body (limit))
    VOID Pop (ExecStack);
  else {
    VOID Push (ExecStack, increment);
    VOID Push (ExecStack, limit);
    VOID Push (ExecStack, Make (Body (current) + Body (increment)));
    VOID Push (ExecStack, proc);
    VOID Push (ExecStack, OpFor);
    VOID Push (ExecStack, proc);
    VOID Push (OpStack, current);
  }
  return TRUE;
}

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) == Body (b)));
}

static int Lt (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) < Body (b)));
}

static int Le (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) <= Body (b)));
}

static int Gt (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) > Body (b)));
}

static int Ge (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (Body (a) >= Body (b)));
}

static int Abs (v)
     Object v;
{
  return Push (OpStack, Make (Body (v) >= 0 ? Body (v) : -Body (v)));
}

static int Add (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) + Body (b)));
}

static int Sub (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) - Body (b)));
}

static int Mul (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) * Body (b)));
}

static int Div (a, b)
     Object a, b;
{
  return Push (OpStack, Make (Body (a) / Body (b)));
}

static int Neg (a)
     Object a;
{
  return Push (OpStack, Make (-Body (a)));
}

static int Sqrt (v)
     Object v;
{
  return Push (OpStack, Make ((float) sqrt ((double) Body (v))));
}

static int Exp (a, b)
     Object a, b;
{
  return Push (OpStack, Make ((float) pow ((double) Body (a), (double) Body (b))));
}

static int Ceiling (v)
     Object v;
{
  return Push (OpStack, Make ((float) ceil ((double) Body (v))));
}

static int Floor (v)
```



```
     Object v;
{
  return Push (OpStack, Make ((float) floor ((double) Body (v))));
}

static int Round (v)
     Object v;
{
  return Push (OpStack, Make (Body (v) - floor ((double) Body (v)) >= 0.5
                              ? ceil ((double) Body (v))
                              : floor ((double) Body (v))));
}

static int Truncate (v)
     Object v;
{
  return Push (OpStack, Make (Body (v) < 0 ? ceil (Body (v)) : floor (Body (v))));
}

static int Atan (a, b)
     Object a, b;
{
  return Push (OpStack, Make ((float) Deg (atan2 (Body (a), Body (b)))));
}

static int Cos (v)
     Object v;
{
  return Push (OpStack, Make (cos (Rad (Body (v)))));
}

static int Sin (v)
     Object v;
{
  return Push (OpStack, Make (sin (Rad (Body (v)))));
}

static int Ln (v)
     Object v;
{
  return Push (OpStack, Make (log (Body (v))));
}

static int Log (v)
     Object v;
{
  return Push (OpStack, Make (log10 (Body (v))));
}
```

```c
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

#define ROP_FALSE       0       /* F            */
#define ROP_AND         1       /* S & D        */
#define ROP_ANDNOT      2       /* S & ~D       */
#define ROP_SOURCE      3       /* S            */
#define ROP_NOTAND      4       /* ~S & D       */
#define ROP_DEST        5       /* D            */
#define ROP_XOR         6       /* S ^ D        */
#define ROP_OR          7       /* S | D        */
#define ROP_NOR         8       /* ~(S | D)     */
#define ROP_NXOR        9       /* ~(S ^ D)     */
#define ROP_NOTDEST     10      /* ~D           */
#define ROP_ORNOT       11      /* S | ~D       */
#define ROP_NOTSOURCE   12      /* ~S           */
#define ROP_NOTOR       13      /* ~S | D       */
#define ROP_NAND        14      /* ~(S & D)     */
#define ROP_TRUE        15      /* T            */
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

#define   OPSTACKSIZE   500
#define EXECSTACKSIZE   250
#define DICTSTACKSIZE    20

static StackOb  DStack, EStack, OStack;

Stack   DictStack = &DStack;
Stack   ExecStack = &EStack;
Stack   OpStack   = &OStack;

static int PPop (), PDup (), PExch (), PRoll (), PIndex (), PClear ();
static int PCount (), PClearToMark ();
static int PCountToMark (), PExecStack (), PCountDictStack (), PDictStack ();
static int PReverseStack ();

static StackOb MakeStack ();

InitStack ()
{
  DStack = MakeStack (DICTSTACKSIZE, PDictOverflow,     PDictUnderflow);
  VOID Push (DictStack, SysDict);

  OStack = MakeStack (  OPSTACKSIZE, POpOverflow,       POpUnderflow);
  EStack = MakeStack (EXECSTACKSIZE, PExecOverflow,     PExecUnderflow);

  Install ("mark", Marker);

  InstallOp ("pop",             PPop,           1, 0, 0, 0, Poly);
  InstallOp ("dup",             PDup,           1, 2, 0, 0, Poly);
  InstallOp ("exch",            PExch,          2, 2, 0, 0, Poly, Poly);
  InstallOp ("roll",            PRoll,          2, 0, 0, 0, Integer, Integer);
  InstallOp ("index",           PIndex,         1, 1, 0, 0, Integer);
  InstallOp ("clear",           PClear,         0, 0, 0, 0);
  InstallOp ("count",           PCount,         0, 1, 0, 0);
  InstallOp ("cleartomark",     PClearToMark,   0, 0, 0, 0);
  InstallOp ("counttomark",     PCountToMark,   0, 1, 0, 0);
  InstallOp ("execstack",               PExecStack,     1, 1, 0, 0, Array);
  InstallOp ("countdictstack",  PCountDictStack,0, 1, 0, 0);
  InstallOp ("dictstack",               PDictStack,     1, 1, 0, 0, Array);
  InstallOp ("reversestack",    PReverseStack,  1, 0, 0, 0, Integer);
}

static StackOb MakeStack (size, over, under)
     int size; Object over, under;
{
  StackOb res;

  res.stack_body = (Object *) Malloc ((unsigned) sizeof (Object) * size);
  res.stack_fill = 0; res.stack_size = size;
  res.overflow = over; res.underflow = under;

  return res;
}

#ifdef notdef
int Push (stack, object)
     Stack stack;
     Object object;
{
  if (stack->stack_fill != stack->stack_size)
    stack->stack_body[stack->stack_fill++] = object;
  else
    return FALSE;
  return TRUE;
}

Object Pop (stack)
     Stack stack;
{
  if (stack->stack_fill)
    return stack->stack_body[--stack->stack_fill];
  printf ("%s", stack == OpStack ? "OpStack" : stack == ExecStack ? "ExecStack" : "OtherS
  Panic (": Pop empty stack");
  return Nil; /* for lint */
}

Object Top (stack)
     Stack stack;
{
  if (stack->stack_fill)
    return stack->stack_body[stack->stack_fill - 1];
  printf ("%s", stack == OpStack ? "OpStack" : stack == ExecStack ? "ExecStack" : "OtherS
  Panic (": Top empty stack");
  return Nil; /* for lint */
}

int Height (s)
     Stack s;
{
  return s->stack_fill;
}

int MaxStack (s)
     Stack s;
{
  return s->stack_size;
}
#endif

Object DictLookup (o)
     Object o;
{
  int i;

  for (i = DictStack->stack_fill - 1; i >= 0; i--) {
    Object item;

    item = DictLoad (DictStack->stack_body[i], o);
    if (TypeOf (item) != Condition)
      return item;
  }
  return Absent;
}
```

```
Object Where (key)
     Object key;
{
  int i;

  for (i = DictStack->stack_fill - 1; i >= 0; i--) {
    Object t;

    t = DictLoad (DictStack->stack_body[i], key);
    if (TypeOf (t) != Condition)
      return DictStack->stack_body[i];
  }
  return Absent;
}

static reverse (vec, n)
     Object *vec;
     int n;
{
  Object temp;
  int i, lim = n / 2;

  for (i = 0; i < lim; i++)
    temp = vec[i],
    vec[i] = vec[n-1-i],
    vec[n-1-i] = temp;
}

int CountTo (t, s)
     Type t;
     Stack s;
{
  int i;

  Object *body = s->stack_body;

  for (i = Height (s) - 1; i >= 0; i--)
    if (TypeOf (body[i]) == t)
      return Height (s) - i - 1;
  return -1;
}

int ClearTo (t, s)
     Type t;
     Stack s;
{
  int count;

  if ((count = CountTo (t, s)) >= 0)
    s->stack_fill = s->stack_fill - (count + 1);
  else
    return FALSE;
  return TRUE;
}

static int PCountDictStack ()
{
  return Push (OpStack, MakeInteger (DictStack->stack_fill));
}

static int PDictStack (array)
     Object array;
{
  Object *body = DictStack->stack_body;
  int i, l = Height (DictStack);

  if (!wCheck (array))
    return Error (PInvAccess);
  if (l > lengthArray (array))
    return Error (PRangeCheck);
  for (i = 0; i < l; i++)
    putArray (array, i, body[i]);
  return Push (OpStack, getIArray (array, 0, l));
}

/*ARGSUSED*/
static int PPop (item)
     Object item;
{
  return TRUE;
}

static int PDup (item)
     Object item;
{
  return Push (OpStack, item), Push (OpStack, item);
}

static int PExch (a, b)
     Object a, b;
{
  return Push (OpStack, b), Push (OpStack, a);
}

static int PRoll (size, count)
     Object size, count;
{
  int s = BodyInteger (size), c = BodyInteger (count);
  Object *body = OpStack->stack_body + Height (OpStack) - s;

  if (BodyInteger (size) < 0)
    return Error (PRangeCheck);
  if (BodyInteger (size) > Height (OpStack))
    return Error (POpUnderflow);
  if (BodyInteger (size) == 0)
    return TRUE;
  if (c > 0) {
    c = c % s;
    reverse (body, s - c);
    reverse (body + s - c, c);
  } else {
    c = (-c) % s;
    reverse (body, c);
    reverse (body + c, s - c);
  }
  reverse (body, s);
  return TRUE;
}

static int PIndex (index)
     Object index;
{
  if (BodyInteger (index) >= Height (OpStack) || BodyInteger (index) < 0)
    return Error (PRangeCheck);
  return Push (OpStack, OpStack->stack_body[Height (OpStack) - 1 - BodyInteger (index)]);
}
```



```
static int PClear ()
{
  OpStack->stack_fill = 0;
  return TRUE;
}

static int PCount ()
{
  return Push (OpStack, MakeInteger (Height (OpStack)));
}

static int PClearToMark ()
{
  if (ClearTo (Mark, OpStack))
    return TRUE;
  else
    return Error (PUnMatched);
}

static int PCountToMark ()
{
  int count;

  if ((count = CountTo (Mark, OpStack)) >= 0)
    return Push (OpStack, MakeInteger (count));
  else
    return Error (PUnMatched);
}

static int PExecStack (array)
     Object array;
{
  int i, l = Height (ExecStack);
  Object *body = ExecStack->stack_body;

  if(!wCheck (array))
    return Error (PInvAccess);
  if (l > lengthArray (array))
    return Error (PRangeCheck);
  for (i = 0; i < l; i++)
    putArray (array, i, body[i]);
  VOID Push (OpStack, getIArray (array, 0, l));
  return TRUE;
}

ReverseStack (stack, n)
     Stack stack; int n;
{
  Object *body = stack->stack_body;

  reverse (&body[Height (stack) - n], n);
}

static int PReverseStack (n)
     Object n;
{
  if (BodyInteger (n) > Height (OpStack))
    return Error (PRangeCheck);
  ReverseStack (OpStack, BodyInteger (n));
  return TRUE;
}
```

```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#include "graphics.h"

extern int InitGraphics ();
int ErasePage ();
static int CopyPage ();

static int SetLineWidth ();
static int GetLineWidth ();
static int SetLineCap ();
static int GetLineCap ();
static int SetLineJoin ();
static int GetLineJoin ();
static int SetDash ();
static int GetDash ();
static int SetFlat ();
static int GetFlat ();
static int SetMiterLimit ();
static int GetMiterLimit ();
static int SetHSB ();
static int GetHSB ();
static int SetRGB ();
static int GetRGB ();

static int PSetRealScreen ();
static int PBuildScreen ();
static int PScreenSize ();
static int GetScreen ();

static int PSetRealTransfer ();
static int PGetTransfer ();
static int PTransferSize ();

static int SetGray ();
static int GetGray ();

static int SetFillMethod ();
static int GetFillMethod ();
static int SetStrokeMethod ();
static int GetStrokeMethod ();

int fill_method = 1, stroke_method = 0;

InitState ()
 {
        InstallOp ("erasepage",         ErasePage,      0, 0, 0, 0);
        InstallOp ("copypage",          CopyPage,       0, 0, 0, 0);
        InstallOp ("setlinewidth",      SetLineWidth,   1, 0, 0, 0, Float);
        InstallOp ("currentlinewidth",  GetLineWidth,   0, 1, 0, 0);
        InstallOp ("setlinecap",        SetLineCap,     1, 0, 0, 0, Integer);
        InstallOp ("currentlinecap",    GetLineCap,     0, 1, 0, 0);
        InstallOp ("setlinejoin",       SetLineJoin,    1, 0, 0, 0, Integer);
        InstallOp ("currentlinejoin",   GetLineJoin,    0, 1, 0, 0);
        InstallOp ("setdash",           SetDash,        2, 0, 0, 0,
                   Array, Float);
        InstallOp ("currentdash",       GetDash,        0, 2, 0, 2);
        InstallOp ("setflat",           SetFlat,        1, 0, 0, 0, Float);
        InstallOp ("currentflat",       GetFlat,        0, 1, 0, 0);
        InstallOp ("setmiterlimit",     SetMiterLimit,  1, 0, 0, 0, Float);
        InstallOp ("currentmiterlimit", GetMiterLimit,  0, 1, 0, 0);

        InstallOp ("setgray",           SetGray,        1, 0, 0, 0, Float);
        InstallOp ("currentgray",       GetGray,        0, 1, 0, 0);
        InstallOp ("sethsbcolor",       SetHSB,         3, 0, 0, 0,
                   Float, Float, Float);
        InstallOp ("currenthsbcolor",   GetHSB,         0, 3, 0, 0);
        InstallOp ("setrgbcolor",       SetRGB,         3, 0, 0, 0,
                   Float, Float, Float);
        InstallOp ("currentrgbcolor",   GetRGB,         0, 3, 0, 0);

        InstallOp ("screensize",        PScreenSize,    2, 1, 0, 0,
                   Float, Float);
        InstallOp ("buildscreen",       PBuildScreen,   2, 2, 0, 0,
                   Float, Float);
        InstallOp ("setrealscreen",     PSetRealScreen, 4, 0, 0, 0,
                   Float, Float, Array, Array);
        InstallOp ("currentscreen",     GetScreen,      0, 3, 0, 0);

        InstallOp ("setrealtransfer",   PSetRealTransfer,       2, 0, 0, 0,
                   Array, Array);
        InstallOp ("currenttransfer",   PGetTransfer,   0, 1, 0, 0);
        InstallOp ("transfersize",      PTransferSize,  0, 1, 0, 0);

        InstallOp ("setfillmethod",     SetFillMethod,  1, 0, 0, 0, Integer);
        InstallOp ("currentfillmethod", GetFillMethod,  0, 1, 0, 0);
        InstallOp ("setstrokemethod",   SetStrokeMethod,        1, 0, 0, 0,
                   Integer);
        InstallOp ("currentstrokemethod",GetStrokeMethod,       0, 1, 0, 0);
        InstallOp ("initgraphics",      InitGraphics,   0, 0, 0, 0);

        VOID InitGraphics ();
 }

int InitGraphics ()
 {
        VOID PInitMatrix ();            /* initmatrix */
        VOID InitClip ();               /* initclip */
        VOID PNewPath ();               /* newpath */

        gstate->colour = NewColour (0.0, 0.0, 0.0);     /* 0 setgray */
        gstate->line_width = 1;         /* 1 setlinewidth */
        gstate->line_cap = CBUTT;       /* setlinecap */
        gstate->line_join = JMITRE;     /* setlinejoin */
        gstate->dash_length = 0;        /* [] 0 setdash */
        gstate->dash_offset = 0;
        gstate->miter_limit = 10;       /* 10 setmiterlimit */
        InitShowContext ();

        return TRUE;
 }

int ErasePage ()
 {
        Paint (NULL, gstate->device->dev,
               NewDevicePoint (0, 0), NewDevicePoint (0, 0),
```



```c
               HardwareExtent (gstate->device->dev),
               White);
        return TRUE;
 }

static int CopyPage ()
 {
        return TRUE;
 }

static int SetLineWidth (width) Object width;
 {
        gstate->line_width = BodyReal (width);
        return TRUE;
 }

static int GetLineWidth ()
 {
        return Push (OpStack, MakeReal (gstate->line_width));
 }

static int SetLineCap (cap) Object cap;
 {
        int v = BodyInteger (cap);

        if (v < 0 || v > 2)
                return Error (PRangeCheck);
        gstate->line_cap = v;
        return TRUE;
 }

static int GetLineCap ()
 {
        return Push (OpStack, MakeInteger (gstate->line_cap));
 }

static int SetLineJoin (join) Object join;
 {
        int v = BodyInteger (join);

        if (v < 0 || v > 2)
                return Error (PRangeCheck);
        gstate->line_join = v;
        return TRUE;
 }

static int GetLineJoin ()
 {
        return Push (OpStack, MakeInteger (gstate->line_join));
 }

static int SetDash (array, offset) Object array, offset;
 {
        Object *v = BodyArray (array);
        int i, length = lengthArray (array);
        float sum = 0;

        if (length > MAXDASH)
                return Error (PLimitCheck);
        if (length == 0)
         {
                gstate->dash_length = 0;
                return TRUE;
         }
        for (i = 0; i < length; i++)
         {
                float val;

                if (TypeOf (v[i]) == Real)
                        val = BodyReal(v[i]);
                else if (TypeOf (v[i]) == Integer)
                        val = (float) BodyInteger (v[i]);
                else
                        return Error (PLimitCheck);
                if (val < 0)
                        return Error (PRangeCheck);
                sum += val;
         }
        if (sum == 0)
                return Error (PRangeCheck);

        gstate->dash_length = length;
        gstate->dash_offset = BodyReal (offset);
        for (i = 0; i < length; i++)
         {
                if (TypeOf (v[i]) == Integer)
                        gstate->dash_array [i] = BodyInteger (v[i]);
                else
                        gstate->dash_array [i] = BodyReal (v[i]);
         }
        return TRUE;
 }

static int GetDash ()
 {
        int i;
        if (!OpCheck (0, gstate->dash_length + 1))
                return FALSE;
        VOID Push (OpStack, Marker);
        for (i = 0; i < gstate->dash_length; i++)
                VOID Push (OpStack, MakeReal (gstate->dash_array[i]));
        VOID Push (ExecStack, MakeReal (gstate->dash_offset));
        VOID Push (ExecStack, Cvx (Rbracket));
        return TRUE;
 }

static int SetFlat (flat) Object flat;
 {
        gstate->flatness = BodyReal (flat);
        return TRUE;
 }

static int GetFlat ()
 {
        return Push (OpStack, MakeInteger (gstate->flatness));
 }

static int SetMiterLimit (miter) Object miter;
 {
        float m = BodyReal (miter);

        if (m < 1)
                return Error (PRangeCheck);
        gstate->miter_limit = m;
        return TRUE;
 }

static int GetMiterLimit ()
```

```
 {
        return Push (OpStack, MakeReal (gstate->miter_limit));
 }

static int SetGray (gray) Object gray;
 {
        float g = BodyReal (gray);

        /* DCJ:
         * Postscript manual doesn't specify what happens when a value
         * to setgray is out of range.  Most printers will set the
         * value to the bound that was exceeded.  We will do so too.
         */
        if (g < 0.0)
             g = 0.0;
        if (g > 1.0)
             g = 1.0;
        gstate->colour = NewGray (g);

        return TRUE;
 }

static int GetGray ()
 {
        return Push (OpStack, MakeReal (Brightness (gstate->colour)));
 }

static int SetHSB (hue, sat, bright) Object hue, sat, bright;
 {
        float h = BodyReal (hue), s = BodyReal (sat), b = BodyReal (bright);
        if (h < 0 || h > 1 || s < 0 || s > 1 || b < 0 || b > 1)
                return Error (PRangeCheck);
        gstate->colour = NewHSBColour (h, s, b);
        return TRUE;
 }

static int SetRGB (red, green, blue) Object red, green, blue;
 {
        float R = BodyReal (red), G = BodyReal (green), B = BodyReal (blue);

        if (R < 0 || R > 1 || G < 0 || G > 1 || B < 0 || B > 1)
                return Error (PRangeCheck);

        gstate->colour = NewRGBColour (R, G, B);
        return TRUE;
 }

static int GetHSB ()
 {
        float h, s, b;

        ColourHSB (gstate->colour, &h, &s, &b);
        VOID Push (OpStack, MakeReal (h));
        VOID Push (OpStack, MakeReal (s));
        VOID Push (OpStack, MakeReal (b));

        return TRUE;
 }

static int GetRGB ()
 {
        float R, G, B;

        ColourRGB (gstate->colour, &R, &G, &B);
        VOID Push (OpStack, MakeReal (R));
        VOID Push (OpStack, MakeReal (G));
        VOID Push (OpStack, MakeReal (B));

        return TRUE;
 }

static int PScreenSize (freq, rot) Object freq, rot;
 {
        return Push (OpStack, MakeInteger (ScreenSize (BodyReal (freq),
                                                       BodyReal (rot))));
 }

static int PBuildScreen (freq, rot) Object freq, rot;
 {
        float f = BodyReal (freq), r = BodyReal (rot);
        int i, ss = ScreenSize (f, r);
        float   *x = (float *) Malloc (sizeof (float) * ss),
                *y = (float *) Malloc (sizeof (float) * ss);

        Object *px = (Object *) Malloc (sizeof (Object) * ss),
                *py = (Object *) Malloc (sizeof (Object) * ss);

        BuildScreen (f, r, x, y);

        for (i = 0; i < ss; i++)
         {
                px [i] = MakeReal (x[i]);
                py [i] = MakeReal (y[i]);
         }
        Free ((char *) x);
        Free ((char *) y);

        return Push (OpStack,
                     MakeArray (px, ss)) && Push (OpStack, MakeArray (py, ss));
 }

static int PSetRealScreen (freq, rot, spot, thresh) Object freq, rot, spot, thresh;
 {
        float f = BodyReal (freq), r = BodyReal (rot);
        int i, ss = ScreenSize (f, r);
        float *th = (float *) Malloc (sizeof (float) * ss);

        gstate->screen.frequency        = f;
        gstate->screen.rotation         = r;
        gstate->screen.spot_function    = spot;

        if (lengthArray (thresh) != ss)
         {
                free ((char *) th);
                return Error (PRangeCheck);
         }

        for (i = 0; i < ss; i++)
         {
                Object a;

                a = getArray (thresh, i);
                if (TypeOf (a) != Real)
                 {
                        Free ((char *) th);
                        return Error (PTypeCheck);
                 }
```



```
                th[i] = BodyReal (a);
         }
        if (gstate->screen.count == 1)
             /* allows 0 to mean 'not assigned yet' */
                Free ((char *) gstate->screen.thresh);
        gstate->screen.thresh = th;
        gstate->screen.count = 1;

        SetScreen (f, r, th);

        return TRUE;
 }

static int GetScreen ()
 {
        VOID Push (OpStack, MakeReal (gstate->screen.frequency));
        VOID Push (OpStack, MakeReal (gstate->screen.rotation));
        VOID Push (OpStack, gstate->screen.spot_function);
        return TRUE;
 }

static int PSetRealTransfer (transfer, values) Object transfer, values;
 {
        Object v;
        int i, size = TransferSize ();
        float *val;

        if (lengthArray (values) != size)
                return Error (PRangeCheck);

        val = (float *) Malloc (sizeof (float) * size);
        for (i = 0; i < size; i++)
         {
                v = getArray (values, i);
                if (TypeOf (v) == Real)
                        val [i] = BodyReal (v);
                else
                 {
                        Free ((char *) val);
                        return Error (PTypeCheck);
                 }
         }

        if (gstate->transfer.tcount == 1)
                Free ((char *) gstate->transfer.tran);
        gstate->transfer.transfn = transfer;
        gstate->transfer.tran = val;
        gstate->transfer.tcount = 1;

        SetTransfer (val);

        return TRUE;
 }

static int PGetTransfer ()
 {
        return Push (OpStack, gstate->transfer.transfn);
 }

static int PTransferSize ()
 {
        return Push (OpStack, MakeInteger (TransferSize ()));
 }

static int SetFillMethod (method) Object method;
 {
        fill_method = BodyInteger (method);
        return TRUE;
 }

static int GetFillMethod ()
 {
        return Push (OpStack, MakeInteger (fill_method));
 }

static int SetStrokeMethod (method) Object method;
 {
        stroke_method = BodyInteger (method);
        return TRUE;
 }

static int GetStrokeMethod ()
 {
        return Push (OpStack, MakeInteger (stroke_method));
 }
```

```c
/*
 * Copyright (C) Crispin Goswell 1987, All Rights Reserved.
 */

char *strcat (), *strncat ();
int strcmp (), strncmp (), strlen (), strspn (), strcspn ();
```



```c
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"

static Object OpForString, Make ();

int putIString ();

Object ParseString (), ParseHexString (), StringFrom ();

static int forString ();

static int Exec (), Token (), PString (), Search (), AnchorSearch ();
static int Copy (), EqEq (), Length (), ForAll (), Get (), Put ();
static int GetInterval (), PutInterval (), Eq (), Lt (), Le (), Gt ();
static int Ge (), PrCheck (), PwCheck ();
static int Cvi (), Cvr (), Cvs (), PReadOnly (), PExecOnly ();

InitString ()
{
  OpForString = MakeOp ("(forallstring)", forString, 0, 0, 3, 5);

  TypeInstallOp (String, "cvi",         Cvi,            1, 1, 0, 0, String);
  TypeInstallOp (String, "cvr",         Cvr,            1, 1, 0, 0, String);
  TypeInstallOp (String, "cvs",         Cvs,            2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "==",          EqEq,           1, 0, 0, 0, String);
  TypeInstallOp (String, "exec",        Exec,           1, 0, 0, 1, String);
  TypeInstallOp (String, "token",       Token,          1, 3, 0, 2, String);
  TypeInstallOp (String, "eq",          Eq,             2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "lt",          Lt,             2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "le",          Le,             2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "gt",          Gt,             2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "ge",          Ge,             2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "length",      Length,         1, 1, 0, 0, String);
  TypeInstallOp (String, "forall",      ForAll,         2, 0, 0, 4, String,
                Array);
  TypeInstallOp (String, "copy",        Copy,           2, 1, 0, 0, String,
                String);
  TypeInstallOp (String, "get",         Get,            2, 1, 0, 0, String,
                Integer);
  TypeInstallOp (String, "put",         Put,            3, 0, 0, 0, String,
                Integer, Integer);
  TypeInstallOp (String, "getinterval", GetInterval,    3, 1, 0, 0, String,
                Integer, Integer);
  TypeInstallOp (String, "putinterval", PutInterval,    3, 0, 0, 0, String,
                Integer, String);
  TypeInstallOp (String, "executeonly", PExecOnly,      1, 1, 0, 0, String);
  TypeInstallOp (String, "readonly",    PReadOnly,      1, 1, 0, 0, String);
  TypeInstallOp (String, "rcheck",      PrCheck,        1, 1, 0, 0, String);
  TypeInstallOp (String, "wcheck",      PwCheck,        1, 1, 0, 0, String);
  TypeInstallOp (String, "token",       Token,          1, 3, 0, 0, String);

  InstallOp ("string",          PString,        1, 1, 0, 0, Integer);
  InstallOp ("search",          Search,         2, 4, 0, 0, String, String);
  InstallOp ("anchorsearch",    AnchorSearch,   2, 3, 0, 0, String, String);
}

static int Cvi (rep)
     Object rep;
{
  Object v;

  v = ParseNumber (BodyString (rep), lengthString (rep));
  if (TypeOf (v) == Integer)
    return Push (OpStack, v);
  else if (TypeOf (v) == Real)
    return Push (OpStack, IntReal (v));
  else if (TypeOf (v) == Condition)
    return Error (PUnResult);
  else
    return Error (PSyntaxError);
}

static int Cvr (rep)
     Object rep;
{
  Object v;

  v = ParseNumber (BodyString (rep), lengthString (rep));
  if (TypeOf (v) == Real)
    return Push (OpStack, v);
  else if (TypeOf (v) == Integer)
    return Push (OpStack, RealInteger (v));
  else if (TypeOf (v) == Condition)
    return Push (OpStack, rep), Error (PUnResult);
  else
    return Push (OpStack, rep), Error (PSyntaxError);
}

static int Cvs (v, string)
     Object v, string;
{
  int length;

  if (lengthString (string) < (length = lengthString (v)))
    return Error (PRangeCheck);
  VOID Bcopy (BodyString (string), BodyString (v), length);
  return Push (OpStack, getIString (string, 0, length));
}

static int EqEq (v)
     Object v;
{
  int c, i, l = lengthString (v);
  unsigned char *s = BodyString (v);

  putchar ('(');
  for (i = 0; i < l && !Interrupted (); i++)
    switch (c = s[i]) {
    default:
      if (c < ' ' || c > 126)
```

```
        printf ("\\%o", c);
      else
        putchar (c);
      break;

    case '\n':  printf ("\\n"); break;
    case '\r':  printf ("\\r"); break;
    case '\t':  printf ("\\t"); break;
    case '\b':  printf ("\\b"); break;
    case '\f':  printf ("\\f"); break;
    }
  printf (")");
  return TRUE;
}

static int Exec (item)
     Object item;
{
  return Push (ExecStack, Cvx (FileString (item)));
}

static int Token (f)
     Object f;
{
  Object res, fs;

  res = Parse (fs = FileString (f));
  if (TypeOf (res) == Condition)
    return Error (PSyntaxError);
  else if (TypeOf (res) == Null)
    return Error (PUnResult);
  else if (TypeOf (res) != Bool) {
    VOID Push (OpStack,
               Cvx (SameFlags (f,
                                getIString (f,
                                            lengthString (f) -
                                            BodyFile (fs)->available,
                                            BodyFile (fs)->available))));
    VOID Push (OpStack, res);
    VOID Push (OpStack, True);
  } else if (BodyBool (res)) {
    VOID Push (OpStack, f);
    VOID Push (OpStack, False);
  } else
    return Error (PSyntaxError);
  return TRUE;
}

Object MakeString (s, length)
     unsigned char *s; int length;
{
  Object res;
  unsigned char *new;

  MakeObject (res, String);
  res.u.String = (new = (unsigned char *) Malloc ((unsigned) length));
  VOID Bcopy (new, s, length);
  res.Length = length;

  return res;
}

static Object Make (s, length)
     unsigned char *s; int length;
{
  Object res;
  unsigned char *new;

  MakeObject (res, String);
  res.u.String = (new = (unsigned char *) Malloc ((unsigned) length));
  VOID Bcopy (new, s, length);
  res.Length = length;

  return res;
}

Object StringFrom (s)
     unsigned char *s;
{
  Object res;

  MakeObject (res, String);
  res.u.String = s;
  res.Length = strlen ((unsigned char *) s);

  return res;
}

static int Eq (a, b)
     Object a, b;
{
  return Push (OpStack, MakeBool (EqString (a, b)));
}

int EqString (a, b)
     Object a, b;
{
  int   al = lengthString (a),
  bl = lengthString (b);

  return al == bl && 0 == strncmp ((unsigned char *) BodyString (a),
                                   (unsigned char *) BodyString (b), al);
}

static int Lt (a, b)
     Object a, b;
{
  int   al = lengthString (a),
  bl = lengthString (b),
  cmp = strncmp ((unsigned char *) BodyString (a),
                 (unsigned char *) BodyString (b), Min (al, bl));

  return Push (OpStack, MakeBool (cmp < 0 || cmp == 0 && al < bl));
}

static int Le (a, b)
     Object a, b;
{
  return Push (OpStack,
               MakeBool (strncmp ((unsigned char *) BodyString (a),
                                  (unsigned char *) BodyString (b),
                                  Min (lengthString (a),
                                       lengthString (b))) <= 0));
}

static int Gt (a, b)
     Object a, b;
{
```



```
  int   al = lengthString (a),
  bl = lengthString (b),
  cmp = strncmp ((unsigned char *) BodyString (a),
                 (unsigned char *) BodyString (b), Min (al, bl));

  return Push (OpStack, MakeBool (cmp > 0 || cmp == 0 && al > bl));
}

static int Ge (a, b)
     Object a, b;
{
  return Push (OpStack,
               MakeBool (strncmp ((unsigned char *) BodyString (a),
                                  (unsigned char *) BodyString (b),
                                  Min (lengthString (a),
                                       lengthString (b))) >= 0));
}

static int Length (object)
     Object object;
{
  return Push (OpStack, MakeInteger (lengthString (object)));
}

static int Get (object, key)
     Object object, key;
{
  int index;

  if (TypeOf (key) == Integer)
    index = BodyInteger (key);
  else if (TypeOf (key) == Real)
    index = (int) BodyReal (key);
  else
    return Error (PTypeCheck);
  if (index >= 0 && index < lengthString (object))
    return Push (OpStack, MakeInteger ((int) BodyString (object)[index]));
  else
    return Error (PRangeCheck);
}

int getString (object, index)
     Object object; int index;
{
  return BodyString (object)[index];
}

static int GetInterval (object, begin, length)
     Object object, begin, length;
{
  int b = BodyInteger (begin), l = BodyInteger (length);

  if (l < 0 || b < 0 || b + l > lengthString (object))
    return Error (PRangeCheck);
  return Push (OpStack, getIString (object, b, l));
}

Object getIString (object, begin, length)
     Object object; int begin, length;
{
  return Make (BodyString (object) + begin, length);
}

static int Put (object, key, value)
     Object object, key, value;
{
  int index;

  if (TypeOf (key) == Integer)
    index = BodyInteger (key);
  else if (TypeOf (key) == Real)
    index = (int) BodyReal (key);
  else
    return Error (PTypeCheck);
  if (index < 0 || index >= lengthString (object))
    return Error (PRangeCheck);
  else if (TypeOf (value) != Integer)
    return Error (PTypeCheck);
  BodyString (object)[index] = BodyInteger (value);
  return TRUE;
}

putString (object, index, value)
     Object object;
     int index, value;
{
  BodyString (object)[index] = value;
}

static int PutInterval (object1, begin, object2)
     Object object1, begin, object2;
{
  int b = BodyInteger (begin);

  if (lengthString (object2) + b > lengthString (object1))
    return Error (PRangeCheck);
  VOID putIString (object1, b, object2);
  return TRUE;
}

int putIString (object1, begin, object2)
     Object object1, object2;
     int begin;
{
  int l = lengthString (object2);
  unsigned char *from = BodyString (object2),
                *to = BodyString (object1) + begin;

  while (l--)
    *to++ = *from++;

  return TRUE;
}

static int Copy (object1, object2)
     Object object1, object2;
{
  if (lengthString (object1) <= lengthString (object2)) {
    VOID putIString (object2, 0, object1);
    VOID Push (OpStack, getIString (object2, 0, lengthString (object1)));
    return TRUE;
  } else
    return Error (PRangeCheck);
}

static int match (a, b, max)
     unsigned char *a, *b;
     int max;
```

```
{
  int i;

  for (i = 0; i < max && *a++ == *b++; i++)
    ;
  return i;
}

static ForAll (string, proc)
     Object string, proc;
{
  VOID Push (ExecStack, string);
  VOID Push (ExecStack, MakeInteger (0));
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpForString);

  return TRUE;
}

static int forString ()
{
  Object string, index, proc;

  proc   = Pop (ExecStack);
  index  = Pop (ExecStack);
  string = Pop (ExecStack);

  if (BodyInteger (index) >= lengthString (string))
    return TRUE;

  VOID Push (ExecStack, string);
  VOID Push (ExecStack, MakeInteger (BodyInteger (index) + 1));
  VOID Push (ExecStack, proc);
  VOID Push (ExecStack, OpForString);
  VOID Push (ExecStack, proc);

  VOID Push (OpStack, MakeInteger (getString (string, BodyInteger (index))));
  return TRUE;
}

Object ParseString (o)
     Object o;
{
  unsigned char buf[STRSIZE], *p = buf, *max;
  int c, level = 0, foo;

  max = p + STRSIZE;
  for (;;) {
    if (p == max) return Absent;
    switch (c = Getch (o)) {
      default:  *p++ = c; continue;
      case EOF: return Nil;
      case ’(’: ++level; *p++ = c; continue;
      case ’)’:
        if (--level < 0) break;
        else *p++ = c;
        continue;
      case ’\\’:
        if (BodyFile (o)->file_type != StringFile)
          switch (c = Getch (o)) {
            default:  *p++ = c; break;
            case EOF: return Nil;
            case ’\n’: break;
            case ’n’: c = ’\n’; *p++ = c;       break;
            case ’r’: c = ’\r’; *p++ = c;       break;
            case ’t’: c = ’\t’; *p++ = c;       break;
            case ’b’: c = ’\b’; *p++ = c;       break;
            case ’f’: c = ’\f’; *p++ = c;       break;
            case ’0’: case ’1’: case ’2’: case ’3’:
            case ’4’: case ’5’: case ’6’: case ’7’:
              c = c - ’0’;
              foo = Getch (o);  /* breaks PNX C compiler when inline */
              c = (c << 3) + foo - ’0’;
              foo = Getch (o);  /* breaks PNX C compiler when inline */
              c = (c << 3) + foo - ’0’;
              *p++ = c;
              break;
          }
        else
          *p++ = c;
          continue;
      }
      break;
    }
  return Make (buf, (int) (p-buf));
}

/*
 * TMD -- Perhaps the fastest way to parse a hex string into bytes.
 *        Only the stream file case is really fast, since it uses
 *        getc() directly.  To make the string file case fast,
 *        GeneralGetch() should be replaced by something close
 *        to an array reference.  But I’m not sure what exactly to use...
 */

Object ParseHexString (o)
     Object o;
{
  unsigned char buf[STRSIZE], *p = buf;
  int c, val, length = 0;

  if(!StatusFile(o)) return Absent;
  if(BodyFile(o)->file_type == StreamFile) {
    FILE *fp = BodyFile(o)->f.f_ptr;
    do {
      /* Get the first nibble */
      for(;;) {
        c = getc(fp);
        if (c >= ’0’ && c <= ’9’)      { val = (c - ’0’) << 4;      break; }
        else if (c >= ’A’ && c <= ’F’) { val = (c - ’A’ + 10) << 4; break; }
        else if (c >= ’a’ && c <= ’f’) { val = (c - ’a’ + 10) << 4; break; }
        else if (c == ’>’) return Make(buf,length);
        else if (c == EOF) return Nil;
      }

      /* Get the second nibble */
      for(;;) {
        c = getc(fp);
        if (c >= ’0’ && c <= ’9’)      { *p++ = val + c - ’0’;      break; }
        else if (c >= ’A’ && c <= ’F’) { *p++ = val + c - ’A’ + 10; break; }
        else if (c >= ’a’ && c <= ’f’) { *p++ = val + c - ’a’ + 10; break; }
        else if (c == ’>’) return Make(buf,length);
        else if (c == EOF) return Nil;
      }
    } while(++length != STRSIZE);
  }
  else {
    do {
```



```
      /* Get the first nibble */
      for(;;) {
        c = GeneralGetch(o);
        if (c >= ’0’ && c <= ’9’)      { val = (c - ’0’) << 4;      break; }
        else if (c >= ’A’ && c <= ’F’) { val = (c - ’A’ + 10) << 4; break; }
        else if (c >= ’a’ && c <= ’f’) { val = (c - ’a’ + 10) << 4; break; }
        else if (c == ’>’) return Make(buf,length);
        else if (c == EOF) return Nil;
      }

      /* Get the second nibble */
      for(;;) {
        c = GeneralGetch(o);
        if (c >= ’0’ && c <= ’9’)      { *p++ = val + c - ’0’;      break; }
        else if (c >= ’A’ && c <= ’F’) { *p++ = val + c - ’A’ + 10; break; }
        else if (c >= ’a’ && c <= ’f’) { *p++ = val + c - ’a’ + 10; break; }
        else if (c == ’>’) return Make(buf,length);
        else if (c == EOF) return Nil;
      }
    } while(++length != STRSIZE);
  }

  return Absent;
}

#ifdef notdef
Object ParseHexString (o)
     Object o;
{
  unsigned char buf [STRSIZE], *p = buf;
  int c, count = 0, val = 0, length = 0;

  while ((c = Getch (o)) != ’>’ && c != EOF && length < STRSIZE) {
    if (c >= ’0’ && c <= ’9’)
      val = val * 16 + c - ’0’;
    else if (c >= ’A’ && c <= ’F’)
      val = val * 16 + 10 + c - ’A’;
    else if (c >= ’a’ && c <= ’z’)
      val = val * 16 + 10 + c - ’a’;
    else
      --count;
    if (++count == 2) {
      *p++ = val; ++length;
      count = val = 0;
    }
  }
  if (length == STRSIZE)
    return Absent;
  return c == EOF ? Nil : Make (buf, length);
 }
#endif

static int PString (length)
     Object length;
{
  int l;
  unsigned char *body, *p;

  if ((l = BodyInteger (length)) < 0)
    return Error (PRangeCheck);
  body = (unsigned char *) Malloc ((unsigned) l);

  for (p = body; p < &body[l]; p++)
    *p = ’\0’;
  return Push (OpStack, Make (body, l));
}

static int AnchorSearch (string, seek)
     Object string, seek;
{
  unsigned char *str = BodyString (string), *see = BodyString (seek);
  int m, lstr = lengthString (string), lsee = lengthString (seek);

  if (lsee > lstr || (m = match (str, see, lsee)) != lsee)
    VOID Push (OpStack, string),
    VOID Push (OpStack, False);
  else {
    VOID Push (OpStack, Make (str + m, lstr - m));
    VOID Push (OpStack, Make (str, m));
    VOID Push (OpStack, True);
  }
  return TRUE;
}

static int Search (string, seek)
     Object string, seek;
{
  unsigned char *str = BodyString (string), *body = str,
                *see = BodyString (seek);
  int i, m, lstr = lengthString (string), lsee = lengthString (seek);

  for (i = lstr - lsee; i >= 0; i--)
    if ((m = match (str, see, lsee)) == lsee) {
      VOID Push (OpStack, Make (body + (str-body) + m, lstr - (str-body+m)));
      VOID Push (OpStack, Make (body + (str - body), m));
      VOID Push (OpStack, Make (body, str - body));
      VOID Push (OpStack, True);
      return TRUE;
    } else
      ++str;
  VOID Push (OpStack, string);
  VOID Push (OpStack, False);

  return TRUE;
}

static int PExecOnly (item)
     Object item;
{
  return Push (OpStack, ExecOnly (item));
}

static int PReadOnly (item)
     Object item;
{
  return Push (OpStack, ReadOnly (item));
}

static int PrCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (rCheck (v)));
}

static int PwCheck (v)
     Object v;
{
  return Push (OpStack, MakeBool (wCheck (v)));
```

```
}
```



```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */

#include "main.h"
#include "graphics.h"

int PStrokePath ();

int in_stroke = FALSE;

int Stroke ();

InitStroke ()
{
  InstallOp ("strokepath",      PStrokePath,    0, 0, 0, 0);
  InstallOp ("stroke",          Stroke,         0, 0, 0, 0);
}

int Stroke ()
{
  int res;

  if (!PFlattenPath ())
    return FALSE;
  if (EmptyPath (gstate->path))
    return TRUE;
  if (gstate->device->dev == NULL)
    return PNewPath ();
  if (ThinStroke ())
    return TRUE;
  in_stroke = TRUE;
  res = PStrokePath () && Fill ();
  VOID PNewPath ();
  in_stroke = FALSE;

  return res;
}

float Magnitude (v)
     Vector v;
{
  return sqrt (v.vx * v.vx + v.vy * v.vy);
}

#define Magnitude(v) sqrt (v.vx * v.vx + v.vy * v.vy)

static void Dash ()
{
  Matrix m;
  Path p, new = NewPath (), last_move, last_dash;
  HardPoint move, here, next;
  Vector v, unit;
  int marking, start_marking = TRUE, index, start_index = 0;
  float dash_left, start_dash_left = gstate->dash_offset;
  float umag, dmag;

  m = gstate->CTM;
  m.tx = 0.0;  m.ty = 0.0;

  while (start_dash_left > gstate->dash_array [start_index]) {
    start_dash_left -= gstate->dash_array [start_index];
    if (++start_index == gstate->dash_length)
      start_index = 0;
    start_marking = !start_marking;
  }
  start_dash_left = gstate->dash_array [start_index] - start_dash_left;

  for (p = gstate->path->next; p != gstate->path; p = p->next)
    switch (p->ptype) {
    case EMove:
      index = start_index;
      marking = start_marking;
      dash_left = start_dash_left;
      here = move = p->pe.point;
      VOID MoveTo (new, here);
      last_move = last_dash = new->last;
      break;

    case ELine:
    case EClose:
      if (p->ptype == ELine)
        next = p->pe.point;
      else
        next = move;
      v = NewVector (next.hx - here.hx, next.hy - here.hy, 1.0);
      dmag = Magnitude (v);
      if (dmag == 0) {
        here = next;
        break;
      }
      unit = ITransform (v, m);
      umag = Magnitude (unit);
      unit = NewVector (v.vx / umag, v.vy / umag, 1.0);
      while (umag > dash_left) {
        here.hx += unit.vx * dash_left;
        here.hy += unit.vy * dash_left;
        (*(marking ? LineTo : MoveTo)) (new, here);
        if (!marking)
          last_dash = new->last;
        marking = !marking;
        umag -= dash_left;
        if (++index == gstate->dash_length)
          index = 0;
        dash_left = gstate->dash_array [index];
      }
      if (p->ptype == ELine) {
        dash_left -= umag;
        if (marking)
          VOID LineTo (new, next);
      } else {
        if (marking) {
          if (start_marking)
            if (last_dash == last_move)
              ClosePath (new);
            else {
              /*        LineTo (new, move);     */
              MoveChunk (last_move,last_dash,new->last);
```

```
            } else
              VOID LineTo (new, move);
        }
      }
      here = next;
      break;

    default:
      Panic ("Dash: unknown Path Element type");
    }
  PathFree (gstate->path);
  gstate->path = new;
  gstate->cp = new->next->pe.point;
}

MoveChunk (dest, begin, end)
     Path dest, begin, end;
{
  dest->ptype = ELine;
  end->next->last = begin->last;
  begin->last->next = end->next;
  dest->last->next = begin;

  end->next = dest;
  begin->last = dest->last;
  dest->last = end;
}

/*
 * is 'b' to the left of 'a' ? if 'a' carries into 'b' ?
 *
 *                         /
 *                        /
 *                       b
 *                      /
 *                     /
 * --------- a -------o
 *
 */

leftof (a, b)
     float a, b;
{
  return Normalise (b - a) > 0;
}

#define leftof(a, b) Normalise (b - a) > 0

Miter (new, last_angle, angle, width)
     Path new;
     float last_angle, angle, width;
{
  Matrix old;
  float diff;

  old = gstate->CTM;
  if (leftof (last_angle, angle)) {
    diff = angle - last_angle;
    VOID MoveTo (new, ExtToInt (NewUserPoint (0.0, 0.0)));
    gstate->CTM = Rotate (gstate->CTM, -diff);
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, -width / 2)));
    gstate->CTM = Rotate (gstate->CTM, diff / 2);
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, -(width / 2) / cos (diff / 2))));
    gstate->CTM = old;
```

```
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, -width / 2)));
  } else {
    diff = last_angle - angle;
    VOID MoveTo (new, ExtToInt (NewUserPoint (0.0, 0.0)));
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, width / 2)));
    gstate->CTM = Rotate (gstate->CTM, diff / 2);
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, (width / 2) / cos (diff / 2))));
    gstate->CTM = Rotate (gstate->CTM, diff / 2);
    VOID LineTo (new, ExtToInt (NewUserPoint (0.0, width / 2)));
    gstate->CTM = old;
  }
  ClosePath (new);
}

Bevel (new, last_angle, angle, width)
     Path new;
     float last_angle, angle, width;
{
  HardPoint P1;
  Matrix *mp;
  float diff,ca,sa,x,y;

  mp = &gstate->CTM;
  width /= 2;
  diff = last_angle - angle;
  x = mp->tx;   y = mp->ty;
  P1.hx = x;    P1.hy = y;
  VOID MoveTo (new, P1);

  ca = cos(diff);   sa = sin(diff);

  if (leftof (last_angle, angle)) {
    P1.hx = -width * (mp->C*ca - mp->A*sa) + x;
    P1.hy = -width * (mp->D*ca - mp->B*sa) + y;
    VOID LineTo (new, P1);

    P1.hx = -width*mp->C + x;
    P1.hy = -width*mp->D + y;
    VOID LineTo (new, P1);
  } else {
    P1.hx = width*mp->C + x;
    P1.hy = width*mp->D + y;
    VOID LineTo (new, P1);

    P1.hx = width * (mp->C*ca - mp->A*sa) + x;
    P1.hy = width * (mp->D*ca - mp->B*sa) + y;
    VOID LineTo (new, P1);
  }
  ClosePath (new);
}

BeginCap (new, width)
     Path new;
     float width;
{
  Path add;
  HardPoint P1,P2,P3;
  Matrix *mp;
  float x,y,t2,w2,wa,wb,wc,wd;

  switch (gstate->line_cap) {
  case CBUTT:
    break;
```



```
  case CSQUARE:
    mp = &gstate->CTM;  w2 = width/2; wc = w2*mp->C; wd = w2*mp->D;
    P1.hx = -wc + mp->tx;  P1.hy = -wd + mp->ty;  VOID MoveTo (new, P1);
    wc *= 2;  wd *= 2;
    P1.hx += wc;           P1.hy += wd;            VOID LineTo (new, P1);
    P1.hx -= w2*mp->A;     P1.hy -= w2*mp->B;     VOID LineTo (new, P1);
    P1.hx += wc;           P1.hy += wd;            VOID LineTo (new, P1);
    ClosePath (new);
    break;

  case CROUND:
    mp = &gstate->CTM;  w2 = width/2;
    P1.hx = w2*mp->C + mp->tx;   P1.hy = w2*mp->D + mp->ty;
    VOID MoveTo (new, P1);

    /* Arc (new, 1, P1, w2, PI / 2, 3 * PI / 2); */
    wa = w2*gstate->CTM.C;  wb = w2*gstate->CTM.D;
    wc = -w2*gstate->CTM.A; wd = -w2*gstate->CTM.B;
    x = gstate->CTM.tx;     y = gstate->CTM.ty;
    t2 = ((float) .5522847);  /* (4/3)*(sqrt(2) - 1) */

    P1.hx = x + wa + t2*wc; P1.hy = y + wb + t2*wd;
    P3.hx = x + wc;         P3.hy = y + wd;
    P2.hx = P3.hx + t2*wa;  P2.hy = P3.hy + t2*wb;
    NewElem (ECurve, add);
    add->pe.curve.x0 = P1;      add->pe.curve.x1 = P2;  add->pe.curve.x2 = P3;
    gstate->cp = P3; PathInsert (new, add);

    P1.hx = x + wc - t2*wa; P1.hy = y + wd - t2*wb;
    P3.hx = x - wa;         P3.hy = y - wb;
    P2.hx = P3.hx + t2*wc;  P2.hy = P3.hy + t2*wd;
    NewElem (ECurve, add);
    add->pe.curve.x0 = P1;      add->pe.curve.x1 = P2;  add->pe.curve.x2 = P3;
    gstate->cp = P3; PathInsert (new, add);

    ClosePath (new);
    break;

  default:
    Panic ("BeginCap  - unknown line cap encountered");
  }
}

LineJoin (new, width, last_angle, angle)
     Path new;
     float width, last_angle, angle;
{
  float ang, sa;

  switch (gstate->line_join) {
  case JMITRE:
    ang = Normalise (leftof (last_angle, angle) ? angle - last_angle : last_angle - angle
    ang = ang < 0 ? -ang : ang;
    ang = ang > PI / 2 ? PI - ang : ang;
    sa = sin (ang / 2);
    if (sa != 0 && 1 / sa <= gstate->miter_limit &&
                   1 / sa >= -gstate->miter_limit)
      Miter (new, last_angle, angle, width);
    else
      Bevel (new, last_angle, angle, width);

    break;

  case JROUND:
```

```
    VOID MoveTo (new, ExtToInt (NewUserPoint (width / 2, 0.0)));
    Arc (new, 1, NewUserPoint (0.0, 0.0), width / 2, 0.0, 2 * PI);
    ClosePath (new);
    break;

  case JBEVEL:
    Bevel (new, last_angle, angle, width);
    break;

  default:
    Panic ("LineJoin  - unknown line join encountered");
  }
}

EndCap (new, width, length)
     Path new;
     float width, length;
{
  Path add;
  HardPoint P1,P2,P3;
  Matrix *mp;
  float x,y,t2,w2,wa,wb,wc,wd;

  switch (gstate->line_cap) {
    case CBUTT:
      break;

    case CSQUARE:
      mp = &gstate->CTM;  w2 = width/2;  wa = w2*mp->A;  wb = w2*mp->B;
      P1.hx = length*mp->A + w2*mp->C + mp->tx;
      P1.hy = length*mp->B + w2*mp->D + mp->ty;    VOID MoveTo (new, P1);
      P1.hx += wa;          P1.hy += wb;           VOID LineTo (new, P1);
      P1.hx += width*mp->C; P1.hy += width*mp->D;  VOID LineTo (new, P1);
      P1.hx -= wa;          P1.hy -= wb;           VOID LineTo (new, P1);
      ClosePath (new);
      break;

    case CROUND:
      mp = &gstate->CTM;  w2 = width/2;
      P1.hx = length*mp->A - w2*mp->C + mp->tx;
      P1.hy = length*mp->B - w2*mp->D + mp->ty;

      VOID MoveTo (new, P1);

      /* Arc (new, 1, P1, w2, -PI / 2, PI / 2); */
      wa = -w2*gstate->CTM.C; wb = -w2*gstate->CTM.D;
      wc = w2*gstate->CTM.A;  wd = w2*gstate->CTM.B;
      x = gstate->CTM.tx;     y = gstate->CTM.ty;
      t2 = ((float) .5522847);  /* (4/3)*(sqrt(2) - 1) */

      P1.hx = x + wa + t2*wc; P1.hy = y + wb + t2*wd;
      P3.hx = x + wc;         P3.hy = y + wd;
      P2.hx = P3.hx + t2*wa;  P2.hy = P3.hy + t2*wb;
      NewElem (ECurve, add);
      add->pe.curve.x0 = P1;    add->pe.curve.x1 = P2;  add->pe.curve.x2 = P3;
      gstate->cp = P3; PathInsert (new, add);

      P1.hx = x + wc - t2*wa; P1.hy = y + wd - t2*wb;
      P3.hx = x - wa;         P3.hy = y - wb;
      P2.hx = P3.hx + t2*wc;  P2.hy = P3.hy + t2*wd;
      NewElem (ECurve, add);
      add->pe.curve.x0 = P1;    add->pe.curve.x1 = P2;  add->pe.curve.x2 = P3;
      gstate->cp = P3; PathInsert (new, add);
```

```
      ClosePath (new);
      break;

    default:
      Panic ("StrokeLineEnd  - unknown line cap encountered");
  }
}

static float move_angle;
static Matrix move_matrix;

float LineSegment (p, new, ehere, enow, width, last_angle, last_type)
     Path p, new;
     UserPoint ehere, enow;
     float width, last_angle;
     enum pelem_type last_type;
{
  HardPoint P1;
  float angle,length,ca,sa;
  Matrix old,m;

  /* Compute angle, length, sin(angle) and cos(angle) of line segment */
  sa = enow.y - ehere.y;   ca = enow.x - ehere.x;

  angle = atan2 (sa,ca);   length = sqrt (sa*sa + ca*ca);
  if(length > (float) 0.0) {  sa /= length;   ca /= length; }
  else {                      sa = 0;         ca = 1; }

  /* Translate CTM by ehere and rotate by angle */
  old = gstate->CTM;
  m.A  = ca * old.A + sa * old.C;  m.B  = ca * old.B + sa * old.D;
  m.C  = ca * old.C - sa * old.A;  m.D  = ca * old.D - sa * old.B;
  m.tx = ehere.x * old.A + ehere.y * old.C + old.tx;
  m.ty = ehere.x * old.B + ehere.y * old.D + old.ty;
  gstate->CTM = m;

  /* Create a path for the outline of the line segment */
  sa = -width/2;
  P1.hx = sa*m.C + m.tx; P1.hy = sa*m.D + m.ty; VOID MoveTo (new, P1);
  P1.hx += length*m.A;   P1.hy += length*m.B;   VOID LineTo (new, P1);
  P1.hx += width*m.C;    P1.hy += width*m.D;    VOID LineTo (new, P1);
  P1.hx -= length*m.A;   P1.hy -= length*m.B;   VOID LineTo (new, P1);
  ClosePath (new);

  if (last_type == EMove) {
    move_angle = angle;
    move_matrix = gstate->CTM;
  }
  else if (last_type == ELine)
    LineJoin (new, width, last_angle, angle);

  if (p->ptype == EClose) {
    if (last_type == ELine) {
      gstate->CTM = move_matrix;
      LineJoin (new, width, angle, move_angle);
    }
  }
  else if (p->next->ptype == EMove || p->next->ptype == EHeader) {
    EndCap (new, width, length);
    gstate->CTM = move_matrix;
    BeginCap (new, width);
  }

  gstate->CTM = old;
  return angle;
}

int PStrokePath ()
{
  Path p, new;
  HardPoint prev, here, move;
  enum pelem_type last_type = EHeader;
  float angle, last_angle, width = gstate->line_width;

  NewElem(EHeader,new);
  new->next = new->last = new;

  PFlattenPath ();
  if (gstate->dash_length != 0)
    Dash ();
  for (p = gstate->path->next; p != gstate->path; last_type = p->ptype, p = p->next) {
    switch (p->ptype) {
    case EMove:
      prev = here;
      move = here = p->pe.point;
      break;

    case EClose:
      if (last_type == EMove)
        break;
      angle = LineSegment (p, new, IntToExt (here), IntToExt (move), width, last_angle, l
      prev = here;
      here = move;
      last_type = EHeader;
      break;

    case ELine:
      angle = LineSegment (p, new, IntToExt (here), IntToExt (p->pe.point), width, last_a
      prev = here;
      here = p->pe.point;
      break;

    default:
      Panic ("unknown path element type in StrokePath");
    }
    last_type = p->ptype;
    last_angle = angle;
  }
  PathFree (gstate->path);
  gstate->path = new;
  return TRUE;
}

int ThinStroke ()
{
  Path p;
  Vector v;
  HardPoint here, prev;

  if (stroke_method != STROKE_THIN)
    return FALSE; /* not used - get better results with area fill */
  v = Transform (NewVector (gstate->line_width, gstate->line_width, 0.0), gstate->CTM);
  if (fabs (v.vx) > 1.1 || fabs (v.vy) > 1.1)
    return FALSE;
  if (gstate->dash_length != 0)
    Dash ();
  for (p = gstate->path->next; p != gstate->path; p = p->next)
    switch (p->ptype) {
```



```
    case EMove:
      here = prev = p->pe.point;
      break;

    case ELine:
      DevicePaintLine (gstate->device, prev, p->pe.point, gstate->colour);
      prev = p->pe.point;
      break;

    case EClose:
      DevicePaintLine (gstate->device, prev, here, gstate->colour);
      prev = here;
      break;

    default:
      Panic ("unknown path element type in ThinStroke");
    }
  VOID PNewPath ();

  return TRUE;
}
```

```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
#include "main.h"
#ifndef MACINTOSH
#include <signal.h>

static int Chdir (), Fork (), Execv (), Wait (), Exit (), System ();
static int Signal (), Kill ();

InitUnix ()
{
  InstallOp ("chdir",   Chdir,  1, 1, 0, 0, String);
  InstallOp ("fork",    Fork,   0, 1, 0, 0);
  InstallOp ("wait",    Wait,   0, 5, 0, 0);
  InstallOp ("uexit",   Exit,   1, 0, 0, 0, Integer);
  InstallOp ("execv",   Execv,  2, 0, 0, 0, Array, String);
  InstallOp ("system",  System, 1, 1, 0, 0, String);
  InstallOp ("signal",  Signal, 2, 1, 0, 0, Integer, Integer);
  InstallOp ("kill",    Kill,   2, 1, 0, 0, Integer, Integer);
}

static int Chdir (dir)
     Object dir;
{
  char buffer [BUFSIZE];
  int l = lengthString (dir);

  VOID strncpy (buffer, BodyString (dir), l);
  buffer [l] = '\0';

  return Push (OpStack, MakeBool (!chdir (buffer)));
}

static int Fork ()
{
  return Push (OpStack, MakeInteger (fork ()));
}

static int Exit (status)
     Object status;
{
  exit (BodyInteger (status));

  return TRUE; /* shuts lint up */
}

static int Wait ()
{
  int pid, status;

  if ((pid = wait (&status)) < 0)
    return Push (OpStack, False);
  return Push (OpStack, MakeBool (status & 0200))
    && Push (OpStack, MakeInteger (status & 0177))
      && Push (OpStack, MakeInteger (status >> 8))
        && Push (OpStack, MakeInteger (pid))
          && Push (OpStack, True);
}

static int Execv (args, name)
     Object args, name;
{
  int i, nl, l = lengthArray (args);
  char **av = (char **) Malloc ((unsigned) (l+1) * sizeof (char *));
  char buffer [BUFSIZE];

  for (i = 0; i < l; i++) {
    Object elem;

    elem = getArray (args, i);
    if (TypeOf (elem) != String)
      return Error (PTypeCheck);
    else {
      int l = lengthString (elem);

      av[i] = Malloc ((unsigned) l + 1);
      VOID strncpy (av[i], BodyString (elem), l);
      av[i][l] = '\0';
    }
  }
  av[l] = NULL;

  nl = lengthString (name);
  VOID strncpy (buffer, BodyString (name), nl);
  buffer [nl] = '\0';

  if (execv (buffer, av) == -1) {
    for (i = 0; i < l; i++)
      Free (av[i]);
    Free ((char *) av);

    return Error (PInvFileAccess);
  }
  return TRUE;
}

static int System (s)
     Object s;
{
  char buffer [BUFSIZE];
  int l = lengthString (s);

  VOID strncpy (buffer, BodyString (s), l);
  buffer [l] = '\0';
  return Push (OpStack, MakeInteger (system (buffer)));
}

static int Signal (n, s)
     Object n, s;
{
  int sn = BodyInteger (n);

  if (sn < 1 || sn > NSIG)
    return Error (PRangeCheck);
  return Push (OpStack, MakeInteger (signal (sn, BodyInteger (s))));
}
```



```
static int Kill (n, s)
     Object n, s;
{
  return Push (OpStack, MakeBool (0 == kill (BodyInteger (n), BodyInteger (s))));
}
#else
InitUnix ()
{
}
#endif
```