

Track 2

```
1  #include "windows.h"
2  #include "track.h"
3  #define abs(x) max(x, -(x))
4
5  /* Drawing Objects */
6  static HBRUSH hbrBlack;
7  static HBRUSH hbrHollow;
8  static HBRUSH hbrWhite;
9  static HBRUSH hbrGray;
10
11 static HPEN hpnBlack;
12 static HPEN hpnWhite;
13 static HPEN hpnNull;
14
15 static HANDLE hObj[MAX_OBJ+1];
16
17
18 /* Cursors */
19 static HCURSOR hcurCross;
20 static HCURSOR hcurDot;
21 static HCURSOR hcurNone;
22
23
24 /* Brush Shapes */
25 static HBITMAP hmapDot;
26
27 static int dxBrushMap, dyBrushMap;
28 static HDC hDCBrushMap;
29
30
31 /* Current State */
32 static HCURSOR hcurNow;
33 static char cCurrentShape = TRECT;
34 static char cCurrentPen = PN_BLACK;
35 static char cCurrentBrush = BR_WHITE;
36
37 static RECT rcClient; /* Client rectangle of window */
38 static POINT ptOrigin; /* Screen-relative origin of window */
39
40
41 /* Instance state */
42 static FARPROC lpSketch;
43 static FARPROC lpToolboxDialog;
44
45 static HANDLE myInstance;
46
47
48 /* Dragging info */
49 static RECT rcPaint;
50 static POINT ptStart, ptOld;
51
52
53 /* Backing Bitmap */
54 static HBITMAP hmapO;
55 static HDC hDCO;
56 static char OMapArray[28000];
```

```

57
58
59
60 /* ----- */
61 /*           Painting Routines           */
62 /* ----- */
63
64 void pascal DrawRect (hDC)
65     HDC     hDC;
66 {
67     Rectangle(hDC, rcPaint.left, rcPaint.top, rcPaint.right, rcPaint.bottom);
68 }
69
70 void pascal DrawEllipse (hDC)
71     HDC     hDC;
72 {
73     Ellipse(hDC, rcPaint.left, rcPaint.top, rcPaint.right, rcPaint.bottom);
74 }
75
76 void pascal DrawTriangle (hDC)
77     HDC     hDC;
78 {
79     POINT vertices[3];
80     vertices[0].x= rcPaint.left;
81     vertices[0].y= rcPaint.bottom;
82     vertices[1].x= rcPaint.right;
83     vertices[1].y= rcPaint.bottom;
84     vertices[2].x= (rcPaint.right-rcPaint.left)/2 + rcPaint.left;
85     vertices[2].y= rcPaint.top;
86     Polygon(hDC, (LPPPOINT)vertices, 3);
87 }
88
89 /* Flood fill code for a triangle -----
90 {
91     POINT center;
92     center.x= (rcPaint.right-rcPaint.left)/2 + rcPaint.left;
93     center.y= (rcPaint.bottom-rcPaint.top)/2 + rcPaint.top;
94     MoveTo (hDC, rcPaint.left, rcPaint.bottom);
95     LineTo (hDC, center.x, rcPaint.top);
96     LineTo (hDC, rcPaint.right, rcPaint.bottom);
97     LineTo (hDC, rcPaint.left, rcPaint.bottom);
98     FloodFill(hDC, center.x, center.y,
99         GetPixel(hDC, rcPaint.left, rcPaint.bottom));
100 }
101 ----- */
102
103 void pascal PaintPicture(hDC)
104     HDC hDC;
105 {
106     BitBlt(hDC, 0, 0, rcClient.right, rcClient.bottom, hDC, 0, 0, SRCCOPY);
107 }
108
109
110 /* ----- */
111 /*           Paint Box Routines           */
112 /* ----- */

```



```

113
114 BOOL pascal ToolboxButton(cmd, btn, hWnd)
115     int      cmd, btn;
116     HWND     hWnd;
117 /* Message from a USERBUTTON; i.e. a paintbox tool */
118 {
119     HANDLE    htemp;
120     HDC       hDC;
121     RECT      brect;
122
123     GetClientRect(hWnd, (LPRECT)&brect);
124     switch (cmd) {
125         case BN_CLICKED:
126             switch (btn) {
127                 case 1:
128                     cCurrentBrush= BR_WHITE;
129                     break;
130                 case 2:
131                     cCurrentBrush= BR_GRAY;
132                     break;
133                 case 3:
134                     cCurrentBrush= BR_BLACK;
135                     break;
136                 case 4:
137                     cCurrentShape= SKETCH;
138                     hcurNow= hcurDot;
139                     break;
140                 case 5:
141                     cCurrentShape= TRECT;
142                     hcurNow= hcurCross;
143                     break;
144                 case 6:
145                     cCurrentShape= TRIANGLE;
146                     hcurNow= hcurCross;
147                     break;
148                 case 7:
149                     cCurrentShape= ELLIPSE;
150                     hcurNow= hcurCross;
151                     break;
152             }
153         case BN_PAINT:
154             hDC= GetDC(hWnd);
155             /* draw basic button */
156             FrameRect(hDC, (LPRECT)&brect, hbrBlack);
157             InflateRect((LPRECT)&brect, -5, -3);
158             /* set up for figure drawing */
159             CopyRect((LPRECT)&rcPaint, (LPRECT)&brect);
160             InflateRect((LPRECT)&rcPaint, -4, -2);
161             SelectObject(hDC, hbrWhite);
162             switch (btn) {
163                 case 1:
164                     FillRect(hDC, (LPRECT)&brect, hObj[BR_WHITE]);
165                     break;
166                 case 2:
167                     FillRect(hDC, (LPRECT)&brect, hObj[BR_GRAY]);
168                     break;

```

```

169         case 3:
170             FillRect(hDC, (LPRECT)&brect, hObj[BR_BLACK]);
171             break;
172         case 4:
173             SelectObject(hDC, hbrBlack);
174             InflateRect((LPRECT)&rcPaint, -11, -3);
175             DrawEllipse(hDC);
176             break;
177         case 5:
178             DrawRect(hDC);
179             break;
180         case 6:
181             DrawTriangle(hDC);
182             break;
183         case 7:
184             DrawEllipse(hDC);
185             break;
186     }
187     ReleaseDC(hWnd, hDC);
188     break;
189 case BN_HILITE:
190     hDC= GetDC(hWnd);
191     InvertRect(hDC, (LPRECT)&brect);
192     ReleaseDC(hWnd, hDC);
193     break;
194 case BN_UNHILITE:
195     hDC= GetDC(hWnd);
196     InvertRect(hDC, (LPRECT)&brect);
197     ReleaseDC(hWnd, hDC);
198     break;
199 case BN_DISABLE:
200 default:
201     return FALSE;
202 }
203 return TRUE;
204 }
205
206             /*      Paint Box WndProc      */
207             /* ----- */
208
209 BOOL FAR PASCAL ToolboxDialog(hDlg, msg, wParam, lParam, lpResult)
210     HWND          hDlg;
211     unsigned      msg;
212     WORD          wParam;
213     LONG          lParam;
214     LONG far *    lpResult;
215 /* WndProc for the Toolbox dialog box */
216 {
217     BOOL processed;
218
219     processed= TRUE; /* Assume we can handle it */
220     if (msg==WM_INITDIALOG) {
221         /* ? */
222     } else if (msg==WM_COMMAND) {
223         if (wParam==PUTAWAY) {
224             EndDialog(hDlg, TRUE);

```



```

225         } else {
226             processed= ToolboxButton(HIWORD(lParam), wParam, LOWORD(lParam)
227         }
228     } else {
229         processed= FALSE;
230     }
231     return processed;
232 } /* ToolboxDialog */
233
234
235 /* ----- */
236 /*             Menu Commands             */
237 /* ----- */
238
239 void pascal NewBrushShape(hmap)
240     HBITMAP hmap;
241 /* Make hmap the current shape of the paint brush */
242 {
243     BITMAP      bmInfo;
244
245     GetObject(hmap, sizeof(bmInfo), (LPSTR)&bmInfo);
246     dxBrushMap= bmInfo.bmWidth;
247     dyBrushMap= bmInfo.bmHeight;
248     SelectObject(hDCBrushMap, hmap);
249 }
250
251 void pascal ClearPicture(hWnd)
252     HWND hWnd;
253 {
254     HDC      hDC;
255     FillRect(hDC, (LPRECT)&rcClient, hbrWhite);
256     hDC= GetDC(hWnd);
257     PaintPicture(hDC);
258     ReleaseDC(hWnd, hDC);
259 }
260
261
262 void pascal MenuCommand(hWnd, id)
263     HWND hWnd;
264     int id;
265 {
266     HMENU      hMenu;
267
268     hMenu = GetMenu (hWnd);
269     if (id>BR_HOLLOW) {
270         if (id==PAINTBOX) {
271             CreateDialog(myInstance, (LPSTR)"paintbox", hWnd, lpToolboxDia
272         } else if (id==CLEAR) {
273             ClearPicture(hWnd);
274         } else {
275             CheckMenuItem(hMenu, cCurrentShape, FALSE);
276             cCurrentShape = id;
277             CheckMenuItem(hMenu, cCurrentShape, TRUE);
278             if (id==SKETCH)
279                 hcurNow= hcurDot;
280             else

```

```

281             hcurNow= hcurCross;
282         }
283     } else if (id<BR_WHITE) {
284         CheckMenuItem(hMenu, cCurrentPen, FALSE);
285         cCurrentPen= id;
286         CheckMenuItem(hMenu, cCurrentPen, TRUE);
287     } else {
288         CheckMenuItem(hMenu, cCurrentBrush, FALSE);
289         cCurrentBrush= id;
290         CheckMenuItem(hMenu, cCurrentBrush, TRUE);
291     }
292 }
293
294
295 /* ----- */
296 /*             Mouse Tracking Routines             */
297 /* ----- */
298
299 void pascal DrawFigure(hDC)
300     HDC hDC;
301 /* Draw the selected figure with the selected tools */
302 {
303     SelectObject(hDCO, hObj[cCurrentPen]);
304     SelectObject(hDCO, hObj[cCurrentBrush]);
305
306     switch (cCurrentShape) {
307         case TRECT:
308             DrawRect(hDCO);
309             break;
310         case ELLIPSE:
311             DrawEllipse(hDCO);
312             break;
313         case TRIANGLE:
314             DrawTriangle(hDCO);
315             break;
316     }
317     PaintPicture(hDC);
318 } /* DrawFigure */
319
320
321 BOOL far pascal Sketch(x, y, lp)
322     int x, y;
323     LPINT lp;
324 {
325     BitBlt(hDCO, x, y, dxBrushMap, dyBrushMap,
326           hDCBrushMap, 0, 0, 0x00B8074A);
327     BitBlt((HANDLE)*lp, x, y, dxBrushMap, dyBrushMap,
328           hDCO, x, y, SRCCOPY);
329 }
330
331 int pascal BrushStroke(x1, y1, x2, y2, hDC)
332     int x1, y1, x2, y2;
333     HDC hDC;
334 {
335     int dx, dy;
336     int x, y;

```



```

337     int m;
338     int signDx, signDy;
339     int cnt;
340
341     dx= x2-x1; dy= y2-y1;
342     signDx= dx>=0?1:-1;
343     signDy= dy>=0?1:-1;
344     cnt= 0;
345     if (abs(dx) > abs(dy)) {
346         y= y1;
347         if (dy!=0) m= dx/dy;
348         for(x= x1; x!=x2; x= x+signDx) {
349             BitBlt(hDCO, x, y, dxBrushMap, dyBrushMap,
350                 hDCBrushMap, 0, 0, 0x00B8074A);
351             BitBlt(hDC, x, y, dxBrushMap, dyBrushMap,
352                 hDCO, x, y, SRCCOPY);
353             if (dy!=0 && (cnt++ % m)==0)
354                 y= y+signDy;
355         }
356     } else {
357         x= x1;
358         if (dx!=0) m= dy/dx;
359         for(y= y1; y!=y2; y= y+signDy) {
360             BitBlt(hDCO, x, y, dxBrushMap, dyBrushMap,
361                 hDCBrushMap, 0, 0, 0x00B8074A);
362             BitBlt(hDC, x, y, dxBrushMap, dyBrushMap,
363                 hDCO, x, y, SRCCOPY);
364             if (dx!=0 && (cnt++ % m)==0)
365                 x= x+signDx;
366         }
367     }
368 } /* BrushStroke */
369
370 void pascal TrackDraw(hWnd, pt)
371     HWND hWnd;
372     POINT pt;
373 /* Track mouse and paint with the brush */
374 {
375     HDC hDCS;
376     MSG msg;
377     POINT ptAbs;
378
379     hDCS= GetDC(hWnd);
380     SetCapture(hWnd, FALSE);
381     SelectObject(hDCO, hObj[cCurrentBrush]);
382     SetCursor(hcurNone);
383     (*lpSketch)(pt.x, pt.y, (LPINT)&hDCS);
384     ptOld.x= pt.x;
385     ptOld.y= pt.y;
386
387     while (TRUE) {
388         ReplyMessage(OL);
389         GetMessage((LPMSG)&msg, NULL, 0x200, 0x210);
390         ptAbs= msg.pt;
391         ScreenToClient(hWnd, (LPPOINT)&msg.pt);
392         if (msg.message==WM_MOUSEMOVE) {

```

```

393         LineDDA(ptOld.x, ptOld.y, msg.pt.x, msg.pt.y,
394                 lpSketch, (LPINT)&hDCS);
395         ptOld.x = msg.pt.x;
396         ptOld.y = msg.pt.y;
397         SetCursorPos(ptAbs.x, ptAbs.y);
398     } else if (msg.message == WM_LBUTTONDOWN) {
399         ReleaseDC(hWnd, hDCS);
400         ReleaseCapture();
401         SetCursor(hcurNow);
402         break;
403     }
404 }
405 } /* TrackDraw */
406
407
408 void pascal TrackFigure(hWnd, pt)
409     HWND hWnd;
410     POINT pt;
411 /* Track mouse and draw figure */
412 {
413     static HDC hDCS;
414     MSG msg;
415
416     hDCS = GetDC(hWnd);
417     SetCapture(hWnd, FALSE);
418     SelectObject(hDCS, hpnWhite);
419     SelectObject(hDCS, hbrHollow);
420     SetROP2(hDCS, R2_XORPEN);
421     ptOld.x = ptStart.x = pt.x;
422     ptOld.y = ptStart.y = pt.y;
423     Rectangle (hDCS,
424               ptStart.x,
425               ptStart.y,
426               pt.x,
427               pt.y
428             );
429
430     while (TRUE) {
431         ReplyMessage(OL);
432         GetMessage((LPMSG)&msg, NULL, 0x200, 0x210);
433         ScreenToClient(hWnd, (LPPOINT)&msg.pt);
434         if (msg.message == WM_MOUSEMOVE) {
435             Rectangle (hDCS,
436                       ptStart.x,
437                       ptStart.y,
438                       ptOld.x,
439                       ptOld.y
440                     );
441             Rectangle (hDCS,
442                       ptStart.x,
443                       ptStart.y,
444                       msg.pt.x,
445                       msg.pt.y
446                     );
447             ptOld.x = msg.pt.x;
448             ptOld.y = msg.pt.y;

```



```

449     } else if (msg.message==WM_LBUTTONDOWN) {
450         rcPaint.left = min(ptStart.x, msg.pt.x);
451         rcPaint.top = min(ptStart.y, msg.pt.y);
452         rcPaint.right = max(msg.pt.x, ptStart.x);
453         rcPaint.bottom = max(msg.pt.y, ptStart.y);
454         Rectangle (hDCS,
455                 ptStart.x,
456                 ptStart.y,
457                 ptOld.x,
458                 ptOld.y
459                 );
460         SetROP2(hDCS, R2_COPYPEN);
461         ReleaseCapture();
462         DrawFigure(hDCS);
463         ReleaseDC(hWnd, hDCS);
464         break;
465     }
466 }
467 } /* TrackFigure */
468
469
470 void pascal CreatePicture(hWnd)
471     HWND hWnd;
472 /* Set up new picture window */
473 {
474     HDC      hDC;
475     RECT     rcO;
476
477     /* Create DC's */
478     hDC= GetDC(hWnd);
479     hDCO= CreateCompatibleDC(hDC);
480     hDCBrushMap= CreateCompatibleDC(hDC);
481     ReleaseDC(hWnd, hDC);
482
483     /* Set up backing bitmap */
484     hmapO= CreateBitmap(720, 300, 1, 1, (LPSTR)OMapArray);
485     SelectObject(hDCO, hmapO);
486
487     /* Set up default paint brush */
488     NewBrushShape(hmapDot);
489
490     /* Clear backing bitmap (hWnd has null client rect so far) */
491     SetRect((LPRECT)&rcO, 0, 0, 720, 300);
492     FillRect(hDCO, (LPRECT)&rcO, hbrWhite);
493 } /* CreatePicture */
494
495
496 long far pascal TrackWndProc(hWnd, message, wParam, lParam)
497     HWND      hWnd;
498     unsigned   message;
499     WORD       wParam;
500     LONG       lParam;
501 {
502     PAINTSTRUCT ps;
503     HDC          hDC;
504     POINT        pt;

```

```

505
506     switch (message) {
507
508         case WM_MOUSEMOVE:
509         case WM_LBUTTONDOWN:
510             SetCursor(hcurNow);
511             break;
512
513         case WM_LBUTTONDOWN:
514             if (cCurrentShape==SKETCH)
515                 TrackDraw(hWnd, MAKEPOINT(lParam));
516             else
517                 TrackFigure(hWnd, MAKEPOINT(lParam));
518             break;
519
520         case WM_CREATE:
521             CreatePicture(hWnd);
522             break;
523
524         case WM_DESTROY:
525             PostQuitMessage(0);
526             break;
527
528         case WM_SIZE:
529             GetClientRect(hWnd, (LPRECT)&rcClient);
530             break;
531
532         case WM_PAINT:
533             BeginPaint(hWnd, (LPPAINTSTRUCT)&ps);
534             PaintPicture(ps.hdc);
535             EndPaint(hWnd, (LPPAINTSTRUCT)&ps);
536             break;
537
538         case WM_COMMAND:
539             MenuCommand(hWnd, wParam);
540             break;
541
542         default:
543             return(DefWindowProc(hWnd, message, wParam, lParam));
544             break;
545     }
546
547     return(0L);
548 } /* WndProc */
549
550
551 int pascal InitClass( hInstance )
552     HANDLE hInstance;
553 {
554     WNDCLASS    wClass;
555
556     /* set up some brushes */
557     hbrBlack = GetStockObject(BLACK_BRUSH);
558     hbrHollow = GetStockObject(HOLLOW_BRUSH);
559     hbrWhite = GetStockObject(WHITE_BRUSH);
560     hbrGray = GetStockObject(GRAY_BRUSH);

```



```

561 hpnBlack = GetStockObject(BLACK_PEN);
562 hpnWhite = GetStockObject(WHITE_PEN);
563 hpnNull = GetStockObject(NULL_PEN);
564
565 /* Cursors */
566 hcurDot= LoadCursor(hInstance, (LPSTR)"dot");
567 hcurCross= LoadCursor(hInstance, (LPSTR)"cross");
568 hcurNone= LoadCursor(hInstance, (LPSTR)"none");
569
570 /* Brush Shapes */
571 hmapDot= LoadBitmap(hInstance, (LPSTR)"dot");
572
573 /* Fill in class description */
574 wClass.hCursor = (HANDLE)NULL;
575 wClass.hIcon = LoadIcon(hInstance, (LPSTR)"Track" );
576 wClass.lpszMenuName = (LPSTR)"Track";
577 wClass.lpszClassName = (LPSTR)"Track";
578 wClass.hbrBackground = hbrWhite;
579 wClass.hInstance = hInstance;
580 wClass.style = CS_VREDRAW | CS_HREDRAW;
581 wClass.lpfNWndProc = TrackWndProc;
582
583 wClass.cbClsExtra= 0;
584 wClass.cbWndExtra= 0;
585
586 return RegisterClass((LPWNDCLASS)&wClass);
587 } /* InitClass */
588
589
590 int pascal WinMain(hInstance, hPrevInstance, lpszCmdLine, cmdShow)
591 HANDLE hInstance, hPrevInstance;
592 LPSTR lpszCmdLine;
593 int cmdShow;
594 {
595 MSG msg;
596 HWND hWnd;
597
598 if (!hPrevInstance) {
599 if (!InitClass( hInstance ))
600 return FALSE;
601 } else {
602 GetInstanceData (hPrevInstance, (PSTR) &hbrBlack, sizeof (hbrBlack)
603 GetInstanceData (hPrevInstance, (PSTR) &hbrHollow, sizeof (hbrHoll
604 GetInstanceData (hPrevInstance, (PSTR) &hbrWhite, sizeof (hbrWhite)
605 GetInstanceData (hPrevInstance, (PSTR) &hbrGray, sizeof (hbrGray))
606 GetInstanceData (hPrevInstance, (PSTR) &hpnWhite, sizeof (hpnWhite)
607 GetInstanceData (hPrevInstance, (PSTR) &hpnBlack, sizeof (hpnBlack)
608 GetInstanceData (hPrevInstance, (PSTR) &hpnNull, sizeof (hpnNull))
609 GetInstanceData (hPrevInstance, (PSTR) &hcurDot, sizeof (hcurDot))
610 GetInstanceData (hPrevInstance, (PSTR) &hcurCross, sizeof (hcurCro
611 GetInstanceData (hPrevInstance, (PSTR) &hmapDot, sizeof (hmapDot))
612 }
613 hObj[PN_WHITE]= hpnWhite;
614 hObj[PN_BLACK]= hpnBlack;
615 hObj[PN_NULL]= hpnNull;
616 hObj[BR_WHITE]= hbrWhite;

```

```
617     hObj[IBR_GRAY]= hbrGray;
618     hObj[IBR_BLACK]= hbrBlack;
619     hObj[IBR_HOLLOW]= hbrHollow;
620
621     hcurNow= hcurCross;
622
623     hWnd = CreateWindow((LPSTR) "Track",
624                         (LPSTR) "Doodler",
625                         WS_TILEDWINDOW,
626                         0,
627                         0,
628                         0,
629                         100,
630                         (HWND) NULL,
631                         (HMENU) NULL,
632                         (HANDLE) hInstance,
633                         (LPSTR) NULL);
634
635     ShowWindow(hWnd, cmdShow);
636     UpdateWindow(hWnd);
637
638     lpSketch= MakeProcInstance((FARPROC) Sketch, hInstance);
639     lpToolboxDialog= MakeProcInstance((FARPROC) ToolboxDialog, hInstance);
640
641     myInstance= hInstance;
642
643     while (GetMessage((LPMSG)&msg, NULL, 0, 0)) {
644         TranslateMessage((LPMSG)&msg);
645         DispatchMessage((LPMSG)&msg);
646     }
647
648     return 0;
649 } /* WinMain */
```


Procedure/Function names for FontList.p

FontList	[Main]	FontList.p
DefFontList	[PresNew]	
FnFonts	[S1Draw]	
FBuildMenu		
FCheckFont		
FGetProps		
FGetDefProps		
FSetDefProps		
SetFList	[FontInstall]	
InsFList		
DelFList		
ListH		
DrawBoxItem		
ClickFilter		
LClikLoopProc		
FInstallFonts		
Unsel		
Sel		
SetCell		
GetSelCell		
TickStyle		
ComplStyle		
AddNewSlot		
BuildFaceBox		
BuildSizeBox		
SetMenuBoxCell		
BuildMenuBox		
InitLists		
SizeToEdit		
ShowFontInfo		
MenuToOthers		
DoInstall		
DoRemove		
GrayButtons		

*** End ProcNames: 34 Procedures and Functions

Procedure/Function names for ShareDefs.p

ShareDefs	[Main]	ShareDefs.p
VNewEnv	[PresNew]	
VDisposEnv		
VCloneEnv		
VReadEnv	[Files]	
VWriteEnv		
VConvertEnv	[PresConvert]	

*** End ProcNames: 7 Procedures and Functions

Procedure/Function names for Ruler.p

Ruler	
RNewPool	[Main] Ruler.p
RDestroyPool	[PresNew]
RReadPool	
RWritePool	[Files]
RConvertPool	
RFindPRuler	[PresConvert]
RPoolFindPRuler	[SlDraw]
RCopyPRuler	
RDeletePRuler	
RAddPRuler	
RSetPRuler	
RChangePRuler	
RUsePool	
ScaleMeasures	
Setvrect	
RActivate	
DrawMarker	[Ruler]
ClipDown	
DrawTabXY	
DrawTab	
DrawTabWells	
RDraw	
RErase	
RClick	
DragMarker	
UpdateIndents	
DragTab	
TouchTack	

*** End ProcNames: 29 Procedures and Functions

Units of PowerPoint

Dennis Austin
December 18, 1985

Tom Rudkin
October 30, 1987

This note describes the basic internal structure of PowerPoint in terms of MPW Pascal units. It describes the structure of the version 1.0 program, and does not include changes made for version 1.1. All units are kept in files by the same name (with '.p' appended), although some units are split into multiple files (these cases are discussed in the sections on the relevant units).

Prog

Prog is the program unit for PowerPoint and contains the entry point of program execution and the main event loop. It serves as the window controller and top level event dispatcher.

As window controller, it manages each *document window* (i.e., a window containing a PowerPoint document, or presentation), a debugging window (not compiled into the released product, but sometimes compiled in for debugging purposes), and desk accessories. It handles the Apple menu (About PowerPoint, Help, and desk accessories) and the Window menu (list of all windows on the desktop, constructed by Prog whenever the user pulls down a menu). It handles the basics of New, Open, Close, and Quit commands in order to implement document windows. It also handles the Debug menu commands to show/hide the window and toggle debug mode. It implements dragging (moving), resizing, and zooming for all document windows.

As event dispatcher, Prog does the following:

- Desk accessories receive all events that are targeted for them.
- Events that manipulate document windows themselves (resizing, etc.) are handled by Prog (with help from the DocWin unit).
- The debug window receives events targeted for it; see below.
- Events for document windows go to the DocWin unit; this includes all clicks inside the contents region of a window, but not those in the rest of a window.
- Menu commands that are not handled directly by Prog (see above) are passed on to the Cmd unit.
- Prog handles three "application-defined" events.

Events are not passed to a central parser in DocWin, but rather to specific procedures for each type of event. This unit is somewhat like TextEdit in this respect; it has procedures for update, activate, click, etc. (Note: we use the word "click" to mean a mouse-down event; technically, the word should mean a mouse-down-and-up episode without moving very far, so our click handlers actually handle both clicks and drags.)

For each document window, Prog remembers a document handle that it gets from DocWin when the window is created, storing it in the "reference constant" field of the window record. It is a handle to a "document context" record, defined in unit DocData, but Prog does not know any of its details. Prog supplies the document handle and the window pointer to DocWin whenever the call does not refer to the currently active document window. In this way, DocWin doesn't need to use or save any information in the window record. Prog remembers the current document handle and current window pointer in global variables private to the unit (these are both NIL whenever no PowerPoint window is active). Similarly, other units that deal with documents and/or windows remember the current document handle and/or window pointer in their own private global variables. These variables are always set to the argument document and/or window by the routine in each unit that handle activate events, and set to NIL by deactivate event handlers. In this way, a document or window argument need only be passed to an operation that does not always operate on the current document window (e.g. update events).

If a certain period of time has elapsed since handling of the last non-null event was completed (10 secs., changeable in the program resources), Prog's event dispatcher offers each document window a chance to do any "background" processing it has to do; in the present version, this consists of computing the miniature images of slides for the slide sorter. See the section on the Background unit for more details.

The debug window is implemented by three procedures within Prog, but they are in fact only an interface to the Apple unit WriteInWindow, which does most of the work. When these procedures are compiled in, there is also a Debug menu containing a command to hide and show the debug window (initially hidden) and any other commands desired, e.g. to output specific debugging information. To output to the debug window, call routines exported by WriteInWindow such as WWAddText and WWNewLine.

The initial code in program Prog, after initializing the various parts of the Macintosh toolbox and all the units of the PowerPoint program, processes the Finder "startup" information. This information tells whether the user selected one or more presentation documents from the Finder desktop or selected the PowerPoint application itself, and whether he chose to open or to print them. Prog creates the necessary document windows for new or existing presentations, or prints the selected presentations and then exits if Print was chosen.

As a general rule, all the code in PowerPoint that knows about the list of windows on the desktop, the position of any window on the screen, the parts of a window outside of the contents region, etc., is isolated to Prog. On the other hand, very little of Prog is really PowerPoint-specific, but could just as well be managing windows for other types of documents instead.

Cmd

Cmd implements command menus and dialogs directly related to commands. Its initialization code gets all the menus from the program resources and installs them in the menu bar. The unit enables or disables some menus in the menu bar based on whether a document window is active or not. It "reviews" menus, enabling or disabling commands and placing check marks or diamonds on commands as appropriate, whenever the user pulls down any menu (or enters any command-key accelerator). It executes all menu commands except those listed above under program Prog; it also exports routines called by Prog to run dialogs and perform other parts of the New, Open, Close, and Help commands and Print from the Finder. Finally, it manages PowerPoint's use of the Macintosh desk scrap (i.e. system clipboard).

PowerPoint does not normally use the desk scrap to store the current clipboard contents. Instead, it uses internal data structures to store information that cannot even be represented in the system-supported clipboard types: a list of slides, a list of objects from within a slide, PowerPoint-specific text, etc. (see the section on the ShareDefs unit for more details). However, when the user quits PowerPoint, or when he changes the active window from a document window to a desk accessory, Cmd causes a representation of the internal clipboard to be posted to the desk scrap in one or more system-supported types, so that the user may paste (a PowerPoint-independent representation of) whatever he cut or copied from PowerPoint into a desk accessory or other program; and before certain operations, such as Paste, Cmd checks the desk scrap to see whether something new has been placed there since it last checked, so that the user may paste into PowerPoint whatever he cut or copied from a desk accessory or other program. Cmd is also responsible for managing the internal clipboard and the "undo clipboard" (a copy of the internal clipboard made before the Cut and Copy commands are executed, so the commands can be undone), and for "extracting" data on the clipboard to a new environment when appropriate (see ShareDefs). Cmd does not itself understand the representation of data on the internal clipboard, but calls routines in the Pres unit to do the work.

Cmd knows nothing about windows or graphics ports (i.e. grafPorts), but it does keep track of the currently active document handle. None of the menu-handling routines take a document handle as an argument—they operate on the current document. Cmd does access some fields of the document context record.

The Cmd unit is split into two files, because of the size of the unit. The primary file is Cmd.p, which contains the unit's interface and some of its implementation; this is the source file given to the compiler. It text-includes (via the `$I` compiler directive) the file FileCmds.p, which contains all the routines related to commands on the File menu.

OutputPr

OutputPr implements PowerPoint printing. It is PowerPoint's sole interface to the Macintosh Print Manager. It handles the Page Setup and Print commands, including their dialog boxes. It also creates the default page setup information for a brand new presentation, which is a landscape page (if possible) for the currently chosen printer, if one is chosen, else a set of LaserWriter-like defaults. Finally, it creates a QuickDraw picture

containing the "handout frames," which are dotted-outline rectangles showing where handouts would go on the master handout page for two and for six slides per page (this operation shares much code with printing handouts, so it is in OutputPr). See the note "Presenter Printing" (November 6, 1986) for a discussion of printing.

The OutputPr unit was written originally by Bear River Associates, but has been modified by Forethought (mainly in the interest of segmenting code into overlays, plus fixing many bugs). The unit was originally named "Output"—which makes more sense—but that conflicts with a name in the Macintosh toolbox. The unit is split into two files, because of the size of the unit. The primary file is `OutputPr.p`, which contains the unit's interface and most of its implementation; this is the source file given to the compiler. It text-
includes (via the `$I` compiler directive) the file `DialogPr.p`, which contains all the routines related to the Page Setup and Print dialogs.

DocWin

DocWin is the central unit controlling the document window contents region. To do this, the unit defines several concepts relating to document windows.

- The *view rectangle* of a window is the area in which its document is viewed; it is the entire contents region of the window minus the scroll bars and the tools (in sorter views, the column of tools buttons and slide changer is omitted, so the view rectangle is wider than in non-sorter views).
- The *virtual page* of a document is its complete display, independent of the current window size. For example, in a slide view the virtual page is the entire display of the slide at the current scale. The virtual page of a title sorter is "infinitely wide," but for scrolling purposes we pretend it's very narrow (so that horizontal scrolling does not occur). The virtual page of a slide sorter is variable width, depending on the actual width of the view rectangle.
- A window's graphics port is said to be in *window coordinates* when the port's origin is at the upper left corner of the window's contents region and its clipping is the entire coordinate space. Thus, when the port is in window coordinates, arguments passed to QuickDraw routines are relative to the upper left corner of the contents of the window.
- The port is said to be in *virtual page coordinates* when its origin is set such that arguments passed to QuickDraw routines are expressed relative to the origin of the virtual page, as it is currently scrolled within the view rectangle, and its clipping rectangle is set to the portion of the virtual page actually visible in the view rectangle.

One of DocWin's basic roles is the implementation of scrolling. It handles the scroll bar events and keeps track of the current scroll position. It exports routines to set the current port into virtual page coordinates and to restore it to window coordinates. It controls the view rectangle's contents by invoking the Pres unit to draw the graphics and to process other events for the document. Pres and lower-level units ignore scrolling com-

pletely and function within the coordinate system and clipping region already set up by DocWin.

The view rectangle is expressed in window coordinates (so the top is always zero, and the left is zero in sorter views, since there is no tools column); DocWin keeps track of the view rectangle of the current window in a private global variable. As far as DocWin is concerned, the virtual page is expressed in arbitrary coordinates; DocWin requests the virtual page rectangle from Pres, which in turn requests it from Sorter or from Slide, and uses it along with the view rectangle to compute the bounds of the two scroll bars. As long as Sorter and Slide deal with their virtual pages in consistent fashions—i.e. the four edges of the virtual page represent the bounds of scrolling in each direction—DocWin can manage the scroll bars and the conversion of the port to virtual page coordinates. In particular, DocWin does not care if the virtual page has (0,0) at its upper left corner or not (in fact, Slide places (0,0) at the *center* of its virtual page, and therefore deals with negative coordinates). The bounds of the scroll bars are computed such that, when scrolled to its minimum value, the top (or left) edge of the virtual page is displayed exactly at the top (or left) edge of the view rectangle, and similarly for the bottom (or right) edges when scrolled to its maximum value. If the virtual page is smaller than the view rectangle in either dimension, then the maximum value is set equal to the minimum value and no scrolling is possible (the scroll bar is made inactive). (Note that, unlike some applications, this means PowerPoint never allows the user to scroll beyond the point where the bottom or right extreme of the display has just come into view.) See the section on the Slide units for more details on Slide's virtual page.

DocWin exports many routines called by Prog and Cmd to perform various operations on document windows; in most cases, any operation on a presentation is handled by a DocWin routine, which in turn may call routines in Pres and/or ToolsWin, although in some cases Cmd calls routines in Pres or Slide directly. For example, when Prog's event dispatcher receives a click event in the contents region of the active window, it calls the DocWin routine DClick; this in turn tests which part of the contents region the mouse was in and handles it directly if it is over a scroll bar, passes it to ToolsWin (TClick) if it is in any of the various tools, or passes it to Pres (PClick), after setting the port into virtual page coordinates, if it was in the view rectangle. In addition to handling events such as activate, deactivate, update, and click, DocWin handles resizing and zooming a window, changing the size of the virtual page (e.g. when changing viewing scale or adding slides to a sorter), changing the view (slides, notes, sorters), creating a brand new document or a clone of an open one, opening or saving a document, closing a document, and other operations.

DocWin keeps track of the currently active document handle and window pointer in private global variables. Many DocWin routines operate on the current document window, but many do not. In particular, Prog or Cmd frequently call DocWin to deactivate the current window in order to unhighlight the window and its selection before a dialog box is displayed, even though it remains the active document window as far as Prog and Cmd are concerned; then the document handle must be passed to the DocWin operation. DocWin routines that take document handle and/or window pointer arguments typically pass one or both arguments on to the corresponding ToolsWin and/or Pres routine, since those units also operate on documents (presentations). DocWin directly accesses many fields of the document context record.

ToolsWin

ToolsWin implements the tools portion of a PowerPoint document window: the buttons for changing to sorter views, the name of the current view (and slide or notes page number if applicable), the slide changer (known internally as the "slider"), and the drawing tools. Like DocWin, it exports routines for handling events for the tools, namely activate, deactivate, update, grow (resize), and click; these routines are called only from the equivalent DocWin routines.

ToolsWin also provides operations to get and set the current slide number, get and set the current tool, change the current view, and change the number of slides in the presentation. These routines may be called by the Pres and Slide units. ToolsWin needs to know the current view, not only so it can display its name in the lower-left corner of the window, but also so it knows whether or not to display the column of tools buttons and slide changer. It needs to know the number of slides so it can set the maximum value on the custom scroll bar.

ToolsWin displays a picture resource for the standard tools; one of the tools is shown inverted to indicate the current choice. It displays another picture resource for the sorter buttons. It uses a custom scroll bar (see unit BarCntl) for the slide changer, but the program interface to the custom bar is the same as to the standard scroll bars. TextEdit is used to display and edit the current slide or notes page number. For more details on the implementation of ToolsWin, see the paper, "Implementation of Presenter Tools," last revised April 17, 1987.

Like DocWin, ToolsWin keeps track of the currently active document handle and window pointer in private global variables. Almost all ToolsWin routines operate on the current document window. ToolsWin directly accesses some of the fields of the document context record, which contain tools-related information (such as the current tool and handles to the slider control and to the TextEdit record). ToolsWin's routines for handling activate and deactivate events set the current document and window, and also activates or deactivates, respectively, the tools; when activated, the tools portions of the window are all drawn in black and the current drawing tool is inverted, whereas when deactivated the tools portions are drawn in gray.

Pres

Pres implements the concept of a *presentation*. It creates and destroys presentations, reads and writes presentations on disk, and clones presentations. It manipulates the list of slides of the presentation, performing operations such as Cut and Paste from a sorter, New Slide, changing to a different view or different slide, etc. Basically, Pres is responsible for all global aspects of the presentation, but for none of the issues of viewing presentations in windows (that's DocWin's job).

Pres also handles events and other operations that apply to a presentation regardless of which view it is in, even if the exact behavior is very different for different views. It does this by passing the operation on to the Slide units if the view is a slide or notes page, and passes it on to the Sorter unit and/or handles it in Pres if the view is a sorter. For example, when Pres's click handler (PClick) receives a mouse click over the view rec-

tangle—converted into virtual page coordinates by DClick; it checks the current view, and either passes the click on to Slide (SClick) if the view is any slide or notes page, or passes it to Sorter (ZClick) if the view is a sorter; in the latter case, PClick tests the result of ZClick and then rearranges the order of slides if the user dragged slides in the sorter, or changes view to a slide or notes page if the user double-clicked in the sorter.

An open presentation (whether opened from disk or created and not yet saved) is represented by a document context record. This is the same data structure known to Cmd, DocWin, and ToolsWin, but documents are called "presentations" inside Pres to emphasize the specific type of document it deals with. Pres directly accesses most of the fields of the document context record. For example, it keeps track of the number of slides in the presentation, the current slide number, a handle to the list of slides, and whether or not the presentation has been modified since last saved. When a new presentation is created, Pres creates and initializes a new document context record. When a presentation is saved, Pres writes a copy of its document context record to disk, as well as other information; this record is read back into memory when the presentation is opened. Pres keeps track of the currently active presentation in a private global variable.

The representation of an open presentation contains the "packed" representation of all slides, notes pages, and the master handout page, plus the "unpacked" representation of the slide master, the notes master, and the slide, notes page or master handout page currently on view, if any. (See the description of the Slide units below for an explanation of these two representations.) In other words, Pres opens the slide master and notes master when it opens the presentation, stores pointers to their slide contexts in the document context, and leaves them open until the presentation is closed. In addition, whenever a slide, notes page, or handout master is on view, Pres opens it and leaves it open until the view changes (if it's not one of the two masters already open), and stores its slide context pointer in the document context. The packed representations of each slide and its notes page are stored within a slide-list entry, which are all linked together with the entry for the slide master and notes master at its head, and a handle to the head of the list stored in the document context record; see unit SlideList. The packed representation of the handout master is stored as the notes page in a separate slide-list entry that has no slide, and a handle to it is also stored in the document context. Only the packed representations and slide-list entries are saved on disk; the unpacked slide contexts are not. See the paper "Presenter Notes Pages" (November 25, 1986) for a fuller discussion of this subject.

The Pres unit is split into four files, because of the size of the unit. The primary file is `Pres.p`, which contains the unit's interface and some of its implementation; this is the source file given to the compiler. It text-includes (via the `$I` compiler directive) three files: `PFiles.p` contains all of Pres's routines relating to the non-printing commands on the File menu; `PPrint.p` contains the routines relating to Print and Page Setup commands; and `PEdit.p` contains the routines relating to modifying the slide list, except for creating new slides.

Sorter

The Sorter unit implements the slide and title sorters. It only handles the appearance of the sorters on the screen and mouse events for sorters; it does not manipulate the slide

list (Cut, Copy, Paste, Clear, New Slide, reordering slides), which is done by Pres. It exports routines for handling events on sorter views, such as activate, draw (update), and click. It also exports routines for changing the scale for viewing miniatures in the slide sorter, reporting or changing the sorter selection (an insertion point at a specific point, or a set of selected slides), for changing the document window's view to or from a sorter view, etc.

Sorter is a "submodule" of Pres, a "private" unit used only by Pres (although, of course, Pascal does not enforce this restriction).

Like Pres and DocWin, Sorter operates on documents, identified by document handles and represented by document context records. Inside the unit, documents are sometimes called "sorters," to emphasize the fact that the unit deals with the sorter aspects of a presentation. Sorter is the sole unit to access some fields of the document context record, such as the current slide sorter scale and information about how the slide sorter is currently laid out in the window—how many columns of miniatures, how much space between each column, etc. In addition, Sorter accesses some other fields, such as the slide list.

Sorter keeps track of the currently active sorter in a private global variable. This is always either the same handle as is active in Pres or is NIL. Some Sorter routines operate on the current sorter and some do not. Sorter's routines for handling activate and deactivate events set the current sorter, and also highlight or unhighlight, respectively, its selection. Pres calls these routines in response to activate and deactivate events on the presentation, and also when changing to or from the title or slide sorter view.

The Sorter unit is split into two files. The primary file is `Sorter.p`, which contains the unit's interface and some of its implementation; this is the source file given to the compiler. It text-includes (via the `$I` compiler directive) the file `ZClick.p`, which contains the routine for handling click events in the sorter, plus all the routines called only from it.

SlideList

SlideList implements the concept of the list of slides belong to a single presentation. It exports the record `SListEntry`, which is one entry in a slide list, and the type `SListHandle`, which is a handle to an `SListEntry`. `SListEntry` contains the packed representations of a slide and its notes page, a handle to the miniature image of the slide (if it has been computed), a handle to the text of the slide's title (if computed), a Boolean telling whether the slide is selected (used only in sorter views), and handles to the next and previous slide-list entries. Thus the entries are doubly linked in the list.

SlideList exports routines which manipulate slide lists without knowing anything about presentations. This includes routines to allocate a slide-list entry, to dispose slide lists, to copy slide lists, to allocate or reallocate (in case it has been purged) storage for a slide's miniature image, to dispose images of the slides of a slide list, to link one list of slides into another list at a given point, and others.

SlideList is a "submodule" of Pres, a "private" unit imported only by Pres and its other submodule (although, of course, Pascal does not enforce this restriction).

DocData

DocData exports the record type DocRec that is the basic repository for information about a presentation, also known as a document context record, and the type DocHandle, which is a handle to a DocRec and is also known as a document handle. The record's fields are divided informally into groups that are each accessed primarily by one of the units using the record: Cmd, DocWin, ToolsWin, Pres, and Sorter, plus information primarily used by Slide and passed to it by Pres when a slide is opened. However, this division is not enforced by the compiler—since all those units, and even Prog, use DocData—nor is it strictly followed even by programming convention. The most that can be said is that each field is only supposed to be *written* by the unit in whose group it is, but even that rule isn't always followed.

DocData also exports the record type OutputRec, containing information relating to printing. A DocRec contains a field of type OutputRec, containing the presentation's *shape* (overhead, 35-mm, or custom), starting slide number, slide and paper size, and a handle to the print record used by the Macintosh Print Manager. This information is in a separate record so that it can more easily be manipulated by the OutputPr unit, and so OutputPr can create a default output record for a new presentation even before the document context is created.

In addition, DocData exports several constants used by the units that manipulate presentations and document windows, and the routine PLock, which locks a document handle and returns a pointer to the document context record.

The Slide Units

Slide actually comprises four distinct units, which together implement the concept of a *slide* in PowerPoint. We use the term "slide," in this context, to mean not only the slides of the presentation (i.e. the entities that are typically shown to an audience as overhead transparencies, 35-mm slides, or video images on screen), but also the notes pages and the slide, notes, and handout masters. In most respects, the Slide units do not distinguish between slides, notes pages, and handout master; e.g. these are all edited in the same way (except their "titles," which for a notes page means the picture of its corresponding slide, and for the handout master means the picture of the handout frames).

Slide exports the record SlideRec, which contains the basic representation of a slide. SlideRec has a handle for an object array and a handle for a text string that hold the bulk of the actual slide data. A single object on a slide is represented by type ObjRec, which is exported by one of the units (OService) but not used outside the Slide units. The contents of a SlideRec is called the "packed" representation of a slide, because all of the slide's objects and text are stored in just two heap objects. This is the way slides are stored with a presentation on disk, and most of the time this is also how they are stored when in memory.

However, to display or otherwise access the contents of a slide it must be *open*. This produces the "unpacked" representation of the slide, in which each slide object is represented by a separate, unrellocatable heap object, and information about the slide itself is stored in an unrellocatable heap object called a "slide context" record. This representation is optimized for speed of access, rather than for memory considerations. The slide context includes information only needed while editing the slide, such as its current selection, the scale at which it is being viewed, a pointer to the slide context for the slide or notes master (as appropriate, unless this slide *is* the master) so that master items can be drawn, and the "drawing environment" (see unit ShareDefs for a description of the latter). When the slide is on view, changes are made to its unpacked representation only, so before it is closed (and at various other times) its changes must be "flushed"; this consists of writing the current unpacked representation from a slide context back into a packed form in its SlideRec, if it has been modified since last flushed.

Opening or creating a slide takes a SlideRec argument and returns a slide context pointer that can be used in subsequent calls on Slide operations to identify the slide. Many Slide operations, however, implicitly refer to the currently active slide. This is the slide (or notes page, etc.), if any, currently on view in the active presentation, if any. Slide stores a pointer to the (open) slide context for the current slide in a global variable (exported by OService but not used outside the Slide units). Slide's routines for handling activate and deactivate events set the current slide, and also highlight or unhighlight, respectively, its selection. Pres calls these routines in response to activate and deactivate events on the presentation, and also when changing which slide (if any) is on view within the current presentation.

The coordinate system for slides places (0,0) at the center of the slide; the positions of objects on the slide, the positions of the guides, the bounds of the slide, etc., are all measured (in pixels) from the center. Thus negative coordinates are quite common. The main reason for this is so that when the size of the slide is changed by the user in the Page Setup command, only the slide's bounds need to be changed; the coordinates of all objects are left unchanged, so their positions relative to the center of the slide remain the same, although some objects may now be partially or entirely off the slide. Another advantage is that when the window is sized larger than is necessary to view the entire slide (at its current scale), it is easy to center the slide within the view rectangle. A striped pattern is drawn in the part of the view rectangle outside the slide bounds to make it clear where the slide edges are in this case, and in fact at least two pixels of this pattern are always drawn around the edges of the slide so that it can easily be ascertained when the slide has been scrolled to an edge.

However, this coordinate system is not known outside the Slide units; as mentioned above, DocWin does not need to know any details about the virtual page coordinates (for slides and sorters), as long as they are handled consistently. Slide exports an operation that computes the rectangle to use for the slide bounds given the height and width of the slide; an operation that returns the virtual page rectangle for a given open slide at its current viewing scale given the window's view rectangle, which takes into account the two pixels of border around the edges of the slide, the existence of any objects off the slide, and the case where the virtual page would be smaller than the view rectangle in either dimension; an operation that returns the "standard" virtual page rectangle for a given open slide, ignoring the size of the view rectangle and any objects that may be off the slide, used when zooming the window; and an operation that computes the scale to

use so that a slide of a given (full-scale) size can be shown in its entirety within a given view rectangle (representing the largest view rectangle possible on the screen). These operations isolate all the details of the coordinate system within the Slide units.

The four Slide units are SActive, Slide, SMouse, and OService. SActive exports the routines that handle activate and deactivate events for slides and routines that modify or report information about the current slide; it primarily handles the direct results of commands, but not of mouse clicks. Slide creates, destroys, opens, closes, copies, and draws slides, and performs other operations that do not always operate on the current slide. SMouse handles mouse clicks and idle-time in the active slide. These three units together export the interface to slides and are imported by many higher-level units; unfortunately, importing units must know which (one or more) of the Slide units to *use*, as there is no single Slide interface to import. OService contains the fundamental types, global variables, and service routines that support the other slide units; it is a "private" unit imported only by the other three Slide units (although, of course, Pascal does not enforce this restriction).

EText

EText is the interface to CoreEdit, the assembly-language unit that displays and edits "rich" text (see below). Essentially, this unit maps CoreEdit, which was written for MacWrite and knows nothing about multiple text boxes at arbitrary locations on the drawing surface at various drawing scales, to the needs of PowerPoint. It is used by Slide and Cmd units. See the papers "Text" (updated July 10, 1986) and "Text Scaling" (updated November 17, 1986) for discussions of PowerPoint's implementation of text.

EText exports routines to handle various events on text, such as update, click, and key, to handle all editing commands (undo, cut, paste, etc.) for text, and to get and set text attributes from menu commands. It also exports routines to open and close an edit session; Slide opens an edit session for the text of a label or text box whenever that box is text-selected, setting up all the information CoreEdit needs to edit the text.

The EText unit is split into two files. The primary file is EText.p, which contains the unit's interface and some of its implementation; this is the source file given to the compiler. It text-includes (via the \$I compiler directive) the file ETEdit.p, which implements the editing operations on text and Undo on text.

CoreEdit

CoreEdit is the core text-editing routines written originally by the authors of MacWrite and used in that product, and licensed to Forethought by Apple. It handles "rich" text (i.e. character-by-character formatting of font, size, and style), tabs, and indents. Forethought had to modify it extensively, however, to adapt it from use in a word processor to PowerPoint's particular requirements. See the Apple document "CoreEdit: A Programmer's Guide" (August 15, 1983) for the CoreEdit specification, and the paper "Presenter's Use of CoreEdit" (November 23, 1986) for a discussion of the adaptation.

CoreEdit consists of several files. All of them reside in subfolder `CoreEdit`. `CoreEdit.p` contains the Pascal interface to CoreEdit (type and procedure definitions), suitable for being imported by EText, and implementations for the procedures consisting simply of the *external* directive. `CE.a` is the primary assembly-language file, the source file given to the assembler. It text-includes the other assembly-language source files: `CEqus.a`, `ReDsp.a`, `FmtStuf.a`, `Procs.a`, `Misc.a`, and `FillStuf.a`.

ShareDefs

ShareDefs defines several types used by many PowerPoint units. The principal ones are those representing the drawing environment, the internal clipboard, and the different views that a presentation may be in. The views are represented by an enumeration type naming the seven window views: current slide, slide master, current notes page, notes master, handout master, slide sorter, and title sorter.

The drawing environment is represented by a record, `DrawEnv`, containing six handles, which reference the environment's picture pool, ruler pool, installed font list, custom tool table (not used), color table (not used in 1.0), and an object containing the presentation's "command state": the various drawing defaults (defaults for all the attributes for new objects), whether the grid is enabled, whether the guides are displayed and if so where, whether object edges are shown, etc. A field of type `DrawEnv` is contained within the document context record, the scrap record, and the slide context record. Thus the environment of an open presentation is shared by each open slide of the presentation, and by the clipboard if it contains data cut or copied from the presentation.

The picture pool contains handles to all the pictures used in the presentation; an object in the presentation that contains a picture stores a reference into the pool in its `ObjRec`, rather than the picture handle itself (see unit `Collect`). In this way, multiple occurrences of a picture can appear in a presentation without duplicating the picture itself, which is often quite large. A special case of multiple occurrences of a picture within a presentation is when a slide or object containing a picture has been copied to the clipboard; then both the original slide or object in the presentation and the copy on the clipboard contain the picture. In such cases, all occurrences of the picture are represented within the slide data structures by references to the same entity in the presentation's picture pool. Similarly, the ruler pool contains records describing all the text rulers used in the presentation; a text box in the presentation stores a ruler index into the pool in its `ObjRec`, rather than the ruler itself (see unit `Ruler`). This permits the same ruler to be shared by many text boxes at very little cost; this is quite common since, for example, all text boxes created with the (current) default ruler share the same ruler until the user actually modifies the ruler on one of the boxes.

The internal clipboard, or scrap, is represented by a record called `ScrapRec` and an enumeration type called `ScrapType`. `ScrapType` enumerates the kinds of data the clipboard can contain, including a slide list, a single slide (or notes page), an object list, text, or a picture; it can also indicate that the system desk scrap is to be used. `ScrapRec` contains a type, a 32-bit field whose interpretation is dependent on the type, some auxiliary fields required for some types, and some information about the presentation from which the scrap data was cut or copied. The 32-bit field generally contains a handle or pointer to the data, but only the unit(s) which are responsible for managing each scrap

type—Pres, Slide, or EText—know its meaning. The presentation information includes the drawing environment, the size of a slide, and the document handle itself; these are needed to give the necessary context to the scrap data. For example, all pictures and rulers in the objects of the scrap data are denoted by references into the picture pool or ruler pool of the scrap's environment. And, if a picture of a slide on the clipboard is drawn—e.g. to save to a scrapbook file—then the slide size is used to compute the frame of the picture. The document handle is included in the scrap record so that we can identify which presentation the scrap data came from. See the paper "Cut & Paste" (September 18, 1986) for a more complete discussion on the clipboard.

One disadvantage of using picture and ruler pools is that it complicates pasting slides or objects into a different presentation than the one from which they were cut or copied, because all references to pictures and rulers in the data on the clipboard must be replaced with references to pictures and rulers in the destination presentation, and those pictures and rulers must be added to the destination picture and ruler pools. It also complicates closing or saving a presentation when slides or objects on the clipboard were cut or copied from that presentation, because any references to pictures or rulers in the data on the clipboard are references in a pool that is about to be destroyed (if the presentation is closing) or written to disk—and we don't want the copy of the pool stored on disk to contain reference counts for references that happened to be on the clipboard at the time it was saved but are not in the presentation itself. To solve these problems, we have to *extract* all picture and ruler references of the data on the clipboard from their original pools—those of the presentation from which they were cut or copied—and replace them with references to newly-added pictures and rulers in newly-created pools. The new pools exist in a new environment, created for the scrap, not associated with any presentation. The document handle in the scrap record is used to recognize whether references in the scrap data are to the environment of the presentation being closed or saved and hence whether they need to be extracted; this handle is NIL when the scrap has already been extracted from its original presentation.

Unit ShareDefs also exports routines that operate on drawing environments: to create a new, default environment, to destroy an environment and its parts, to clone an environment (as part of cloning a presentation), to read and write an environment to disk, and to convert a previous-version environment read from disk into the current version.

Ruler

The Ruler unit manages a pool of rulers for each presentation. It also draws text box rulers and handles mouse clicks for rulers, thus handling modifications to a ruler. ...

FontList

The FontList unit maintains the list of fonts that is installed in the font menu for a each presentation. It also includes the code that implements the Other Fonts command, including its dialog box, for adding, deleting, replacing, or reordering fonts installed for a presentation. ...

Collect

Collect manages collections (pools) of handles to arbitrary entities, allowing an entity to be used multiple times within a presentation without multiple copies of it. It uses reference counts to know when a handle is no longer referenced within the collection, at which time it disposes the handle. An entity within the collection is referenced via an EntityReference, a type exported by Collect. The unit exports operations to create and destroy a collection, to read and write a collection on disk, to add a handle to a collection, to return the handle corresponding to a given reference, to make a copy of a reference, and to delete a reference.

Collections are currently used within PowerPoint to store QuickDraw pictures that have been pasted onto slides. See the description of unit ShareDefs above for more information on these collections, called picture pools.

BFile

BFile implements the concept of a *block file*, or b-file, used by PowerPoint to store presentations on disk. The caller opens and closes the actual Macintosh file. It begins access to the file as a b-file by passing the file's reference number to either of two BFile operations depending on whether the file is to be read or written; both of these operations return a handle that must be passed to subsequent BFile operations.

The representation of a presentation on disk contains a set of data blocks, each of which is referenced by a block number and each corresponding to one heap object of the presentation's in-memory representation. The contents of a block in the file is identical to the contents of the corresponding heap object in memory, except that handles to other heap objects are replaced by block numbers. See the paper "BFile" (August 1, 1986) for a complete description of the subject.

Pictures

The Pictures unit contains support for creating a QuickDraw picture and drawing into it. PowerPoint uses this mechanism to create a picture of a slide for the slide sorter, for display on a notes page, and for saving to a scrapbook, and to create a picture of a slide or some objects from a slide to paste as picture into a slide and to post to the system desk scrap. The original motivation for creating a separate unit to do this was that we had to implement an elaborate work-around to a QuickDraw bug in scaling text within pictures; that bug has since been fixed by Apple.

The unit exports a routine CreatePicture, which opens a new graphics port, sets up its visible and clipping regions to match the size of picture being drawn (which is typically larger than the screen), opens a QuickDraw picture, then calls a procedure parameter to do the actual drawing, closes the picture and the port, and returns the picture handle. The unit also exports a routine to check whether there is enough memory to draw a given picture when the drawing destination is a picture—i.e. to draw a picture into a picture—a situation that can require arbitrarily-large amounts of memory.

Background

The Background unit implements a mechanism for allowing some computations to be done "in background"—i.e. when the main event loop has no events to handle—and yet be able to abort the computation if user input occurs and to handle normal idle-loop processing (blink the insertion point and change the shape of the pointer as the mouse moves).

This mechanism is used as follows: Whenever a computation is about to start in background that is interruptible by user input, call `BgStartBackground`; call `BgStopBackground` when it is complete. At periodic intervals during the computation, call `BgIsCheck`, which returns `TRUE` if it's time to do idle-time processing and to check for user events. `BgIsCheck` can be called by routines that run both in background and normally, since it never returns `TRUE` if background processing isn't in progress (as indicated by `BgStartBackground`). If background *is* in progress, `BgIsCheck` calls `SystemTask` if it's been at least one clock tick (1/60th of a second) since it last did so, to give desk accessories a chance to run periodically, and it returns `TRUE` if it has been at least 1/10th of a second since user events were last checked. If `BgIsCheck` returns `TRUE`, then call `BgEventCheck`. This routine does the idle-loop processing and then checks if a user-input event is available; if so it does *not* return to its caller but calls `Signal`, which raises an exception that must be handled by any routine that needs to clean up before being aborted (see unit `ErrSignal`). If there is no user input, then after `BgEventCheck` returns, continue the computation. If the computation modifies global variables or other global state in such a way that ordinary idle-loop processing would not work (as slide drawing and text drawing do), then it must restore the global state before calling `BgEventCheck` and set it up for the computation in progress again after it returns.

PowerPoint currently checks for user-input events in two loops: before drawing each object on a slide, and before drawing each paragraph in a text box. This is not so often as to cause too much time to be spent checking, but it is usually often enough to give the user the appearance of instant response to the mouse or keyboard and to keep the insertion point blinking at the desired rate even when sorter images are being computed in background. Ideally, the user never notices that any processing is happening in background; however, whenever a paragraph is very long or has complicated formatting in it, and whenever an object contains a picture that takes a long time to draw, there is a noticeable interval during which user input is ignored and blinking stops.

MoreOrLess

`MoreOrLess` implements the conversion of Living Video Text's `More` and `ThinkTank` files into PowerPoint slides. It exports one routine, which is called from the `Cmd` unit when the `Paste From` command is executed for a presentation in a sorter view. It in turn calls routines in `Pres` and `Slide` to create new slides and set their titles and text boxes from the outline in the file, after first running the standard get-file dialog to ask the user what file to paste. The unit was written by Bear River Associates, with slight modification by Forethought. See the memo from Dennis Austin, "Reading More files into Presenter" (December 22, 1986) for more information on this subject.

ReadPict

ReadPict implements the pasting of MacPaint and PICT files onto PowerPoint slides. It exports one routine, which is called from the Cmd unit when the Paste From command is executed for a presentation in a slide view. The routine runs the standard get-file dialog to ask the user what file to paste, then reads the file and returns a picture handle. For PICT files, the file contains simply a picture plus a 512-byte header; the header is skipped and the rest of the file is turned into a picture object. For MacPaint files, the unit has to unpack the bitmap in the file, crop away all the white space around the four sides of the bitmap (because MacPaint files are always assumed to be the size of the Macintosh screen, but typically a much smaller area contains the image of interest), and then write the bitmap into a QuickDraw picture in "bands" of at most 3K bytes each.

The unit was written by Bear River Associates, with slight modification by Forethought.

SIMakeResFile

SIMakeResFile contains the routines to create a scrapbook file of a user-specified name, write pictures into the file, and close it. A "scrapbook" file is one of the same type and format as the Macintosh system file used by the Scrapbook desk accessory, except that it may have any name and reside in any folder and volume. The pictures written to the scrapbook each become a separate "page" in the scrapbook.

PowerPoint uses this unit for saving pictures of the slides of a presentation into a scrapbook, which the user does via a radio button on the "Save as" dialog box. The unit was written by Solutions, Inc. (hence the "SI" in the name), who also wrote the SmartScrap desk accessory for viewing arbitrary scrapbook files.

Memory

The Memory unit manages PowerPoint's use of global memory. Its initialization code sets up the application heap zone, allocates several master pointer blocks, and sets up the mechanism for ensuring we don't run out of memory without warning the user about it first (a "reserve" object and a "grow-zone" procedure). The unit also exports operations to find out whether enough memory exists for doing something, in order to "pre-flight" check operations like loading a presentation from disk and computing miniature images for the slide sorter. Finally, it provides a mechanism for posting an application event that issues an alert to the user when memory is low, but does not issue the alert until control has returned to the main event loop. See the paper "Memory Management in PowerPoint" (February 27, 1987) for a complete, though slightly out-of-date, discussion of the subject.

Extra

The Extra unit mediates the recursive "uses" relation, allowing units that are "low" in the "uses" chain to *use* units that are "higher" in the chain. That is, lower units should import

unit Extra in order to gain access to routines in higher units, rather than importing the higher units directly.

At present, the only routines accessible through Extra are two in DocWin: DIdle and DAutoScroll. The Background unit needs to call DIdle to do idle-time processing during background computations. The Sorter and EText units need to call DAutoScroll to scroll the virtual page within the window if the user drags the mouse out of the view rectangle during certain operations (e.g. marquee selection and moving slides in a sorter), or if the user types so as to move the insertion point out of the view rectangle. These units call XIdle and XAutoScroll, which in turn call DIdle and DAutoScroll, respectively.

Utility

Utility defines some miscellaneous constants, types, global variables, and routines used throughout the program. It exports constants for the characters generated by several keys on the Macintosh keyboard, such as the four arrow keys, return, enter, etc. It exports the constant BIGINT, which is a number guaranteed to be "large" compared to the screen size, but small enough not to overflow when it is offset, doubled, etc. It exports several variables that would be constants except that their values have to be computed at run time, such as the number of pixels per inch vertically and horizontally, a rectangle covering the entire QuickDraw coordinate space, and a null rectangle. It exports globals describing the program and its environment, such as a Boolean telling whether "new" (i.e. 128K) ROMs are on the machine and a Boolean telling whether to use metric units or English units of measurement.

Utility exports several general-purpose arithmetic and geometric routines, such as routines to scale and unscale integers, points, and rectangles by PowerPoint's fixed scales (1:1, 2:3, 1:2, and 1:3). It exports a routine to return the current date and time in the format PowerPoint uses during printing, a routine to show the watch cursor, routines to save and restore a rectangle of bits from the screen, and so forth. These are all routines which do things not obviously belonging in any other unit of the program, and usually not even particularly PowerPoint specific.

In addition to the file `Utility.p`, containing the unit's interface and all of the implementation that is written in Pascal, there is also a file `UtilAsm.a`, containing assembly language code to implement some Utility routines. For example, `EqualMem` contains a tight assembly-language loop to tell whether the objects pointed at by two pointers are byte-for-byte equal.

ErrSignal

ErrSignal is a unit supplied by Apple that implements a "throw and catch" type of mechanism for handling exceptions. It exports two principal routines: `Signal` and `CatchSignal`. A routine that wants to handle an exception first calls `CatchSignal`, then performs the operation that might cause an exception. When some other routine detects an exceptional condition, it calls `Signal`, which causes control to pass to the caller of `CatchSignal` as though `CatchSignal` is returning again. Thus the routine that called `Signal`, and all other routines in the call chain between the one that called `CatchSignal`

and the one that called Signal, are exited "prematurely." The first call to CatchSignal (before the exception occurs) returns zero, whereas after a signal has been generated CatchSignal returns the (non-zero) argument that was passed to Signal. Thus the routine that is going to handle the exception checks the result of CatchSignal: if it is zero, this is the first time here so do the normal operation; otherwise handle the exception.

PowerPoint uses signals only when aborting background computation due to user input. When the routine in the Background unit detects user input during background, it calls Signal. Certain routines involved in drawing miniatures for the slide sorter are known to be contained in call chains that eventually call the Background event-check routine. Those routines call CatchSignal first if they have any clean-up to do when drawing is aborted; after their clean-up code, they must call Signal again to reraise the signal. For example, CreatePicture has to close the picture and the port if drawing is aborted. Any routine that does not call CatchSignal will be bypassed as control passes back up the chain, ultimately to the routine in the main program unit that controls all background processing.

The ErrSignal unit consists of two files: ErrSignal.p contains the interface to the unit, written in Pascal, suitable for being imported by other units. ErrSignal.a contains the implementation of all routines in the unit, all written in assembly language. Although the unit was originally written by Apple, Forethought modified it to save and restore register values as part of CatchSignal and Signal, so that all local variables, etc., will be correct when control passes back to an exception handler, and to fix one bug.

LMenu

LMenu implements a menu definition (MDEF) for PowerPoint's Line menu. This menu shows lines in various thicknesses and with or without arrow heads, instead of command names, and so it requires a custom menu. It is not linked into the main PowerPoint program, but is linked by itself into an MDEF resource. The unit was written by Bear River Associates.

PMenu

PMenu implements an MDEF for PowerPoint's Pattern menu. This menu shows two columns of rectangles, each containing a different pattern, so it requires a custom menu. Like LMenu, it is linked by itself into an MDEF resource, and was written by Bear River Associates.

BarCntl

BarCntl implements a control definition (CDEF) for PowerPoint's slide changer. The slide changer's behavior is very similar to a scroll bar, but its appearance is different: it looks like a thin rod with an arrowhead at each end, and its "thumb" looks like a knob that slides along the rod. In almost every respect, the interface to this custom control is identical to the interface to standard scroll bars: it has a minimum and maximum value and a current value represented by the relative distance of the thumb between the top

and bottom of the control; it can be moved, sized, shown and hidden, enabled and disabled; it responds to mouse events in the same way—all these operations are invoked using the standard Toolbox routines for scroll bars in the standard ways. There is one addition to the interface: when the custom control is highlighted with the value 253, it is drawn in a gray pattern and without its thumb; this is used when the window is deactivated.

BarCntl is not linked into the main PowerPoint program, but is linked by itself into a CDEF resource. The unit was written by Bear River Associates.

PowerPoint Programming Conventions

November 24, 1987

Dennis Austin

PowerPoint is unusually tolerant of individual programming styles, and different modules use different conventions. Nevertheless, we do try to maintain a consistent style within a single module. To make this easier, I have noted some of the conventions that are observed.

WARNING: DOGMA AHEAD.

Indentation schemes

Scheme Dennis

This one takes the view that the use of a compound statement to group the statements controlled by a conditional or repetitive statement is a historical mistake. Most recent Pascal-like languages (Modula, Ada, etc.) use structured statement syntax that does not require compound statements.

This indentation scheme pretends that Pascal is such a language. Consequently, the final *end* bracketing an *if*, *while*, or *for* statement is indented at the same level as its mating *if*, *while*, or *for*. (an *else* or *end else begin* is also indented at the level of the *if*.) The *begin* is given no special consideration, being regarded just another part of the syntax; the phrases *then begin*, *do begin*, and *end else begin* are regarded as single tokens.

To carry the idea to extreme, every structured statement would use a *begin-end* pair. The standard actually followed is less strict: If the controlled statement is only a single line (not just single statement), then the *begin-end* can be omitted. In the case of an *if-then-else*, the *begin-end* can be omitted only if both branches qualify and therefore both are omitted.

Benefits of this approach are:

Whenever a line A is indented under line B, it means that A controls B. Every level of indentation represents exactly one level of control.

Indentation never changes by more than one level at a time, except in the case where the *begin-end* has been omitted in the last statement in a controlled block. (A good thing to avoid.)

The program does not take up "noise" lines that are added only because of the syntax.

The phrase "*end else begin*" is always written as a single token. Therefore extra semicolons do not cause syntax errors. You can end every line with a semicolon if you want to.

Scheme Tom

This more classic approach emphasizes the bracketing nature of *begin* and *end*. Matching brackets are always in the same "column". *Begin* and *end* always appear on a line by themselves. *Else* always appears on a line by itself, indented at the level of the *if*.

Benefits of this approach are:

Whenever a line A is indented under line B, it means that A controls B. Every level of indentation represents exactly one level of control.

begins and *ends* are easy to match visually.

ifs and *elses* are easy to match visually.

Tab settings

Dennis likes fairly large tab settings, in the range of, say, 4 to 8. Tabs are set every 5 spaces in many of the files because that was the default on the Lisa Workshop where those file originated. Newer files have tabs set every 4 spaces since that is the default under MPW.

Most of the files Tom created have tabs set every 3 spaces. He arrived at that figure by observing that 4 was too many and 2 was too few...

For new files, any setting is probably okay, but it is unwise to change the setting on existing files.

Identifiers

Procedures

Following Apple conventions, procedure names begin with a capital letter. Each "word" unit in the name also begins with a capital; all other letters are lower case. Underscores are not used. Procedures that are exported from a module are prefixed with a particular letter representing that module. We're running out of letters, so we may need to start using 2-letter prefixes.

Constants

Following C practice, but not Apple practice, constants are usually written in all upper case. In particular, NIL, TRUE, and FALSE are always all caps. Constants exported from the Apple interfaces are often written in all caps so that they meet our convention. To make them more readable, the underscore is sometimes used as a separator.

Variables

Following Apple practice, variables and field names begin with a lower case letter. Each "word" unit in the identifier also begins with a capital; all other letters are lower case. No other convention is used consistently.

Types

Following Apple practice, type names begin with a capital letter. An exception is made for standard types, which may begin with a lower case letter (except for Boolean which may be capitalized because it is a proper noun.) Each "word" unit in the identifier also begins with a capital; all other letters are lower case. No other convention is used consistently.

Procedure headings

Procedure headings always have underlining under the word *procedure* and the identifier. If the procedure is exported, the underlining is with equal signs, if not, the underlining is with hyphens. Underlining is not done in the interface portion of a unit. Following the underlining, there is always a comment briefly describing the procedure. If the procedure is exported, the comment appears in both the interface and implementation sections. Exceptions are sometimes made for "one-liners" whose function is obvious from the identifier.

The underline and comment are indented at the level of the word *procedure*. Local declarations are indented an additional tab. The procedure body *begin* and *end* are always placed at the same indentation level as the word *procedure*. The *end* has a comment with the name of the procedure; if there are many local declarations (especially local procedures), the *begin* has the same comment.

AR

Comments that are inserted to note a place where some future action is required before the code is "finished" are marked with letters AR. (action required)

Assert, range check

Range checking is on everywhere possible. In version 1.0, we even shipped with checking on. Where it must be turned off to use some of Apple's array types, it is turned off in the most limited context reasonable.

Whenever a condition must obviously be true, we use the utility procedure `Assert` to make sure it is. This can catch errors much earlier than they would otherwise show up. If you don't think that an `OSErr` could possibly have been returned, call `Assert(err=NOERR)`;

Comment symbols

We use braces as comments rather than `(* ... *)`

Reserved words

Ignoring Apple convention, we do not write Pascal reserved words in all capitals; they are written in all lower case.

Spacing

Punctuation is usually spaced as it is in English: no space before period, comma, colon, or semicolon; one space after. No space after a left parenthesis or before a right.

Dennis observes the above convention even for `:=`

Spacing within expressions varies on a case-by-case basis, as the author thinks best enhances the readability.

Observations of MS-Windows Modules

Dennis Austin

The Microsoft Windows documentation has little to say about the use of modules aside from the simple case of a multiple-instance application. Since we would like to use modules in our applications, and we have been using them in previous distributions of Windows, I have explored some of their features. My understanding is grossly incomplete, but it has taken many hours of experiments even to turn up this much information.

The Windows Entry Sequence

The -Gw compiler flag causes cc to generate the windows entry (and exit) code for all procedures declared FAR. The entry sequence is

```

push    ds      ; These first three instructions, one
pop      ax      ; byte each, set AX to the current DS.
nop

inc      bp      ; The rest of the sequence enters the
push     bp      ; procedure saving DS and setting DS to
mov      bp,sp   ; the value in AX. (I don't know why
push     ds      ; the stored BP is forced to be odd.)
mov      ds,ax
sub      sp,<const>

```

The first three bytes of the sequence may be changed by the linker or loader, depending on the specifications of the module definition file (.def).

Application modules are tasks and can have multiple data segments, one per instance. For exported procedures, i.e. those named as exports in the .def file, the linker changes the first three bytes of the entry sequence to nop's. To call an exported procedure, AX must be preset with the correct data segment address for the instance that is to receive the call --even when calling from within the same module.

The Windows routine MakeProcInstance creates a code fragment that sets up AX as required, in effect binding an exported procedure to a particular instance. It yields a long pointer to code that loads AX with the data segment address of the instance and then branches to the actual procedure. The code ragment is allocated in Windows own data space.

The routine GetProcAddress serves the same purpose as MakeProcInstance, but is intended to be called from other modules. MakeProcInstance can only be called from the module defining the procedure. (In this release, GetProcAddress has been changed to require the DOS file name of the exporting module rather than its module name --a step backwards, in my opinion.)

Procedures exported with the NODATA attribute are slightly different. They are expected not to access the module data segment. Their entry sequences are thus not modified and

DS remains that of the caller. Apparently, NODATA entry points need not be compiled with the windows entry sequence (-Gw), although I'm not certain about this.

Library modules have only one data segment, or none at all. If there is no data segment, the linker makes no code modifications so the code can be compiled without the "-Gw" flag and no windows entry code will be generated. As far as entry points are concerned, declaring no data segment is apparently the same as declaring every exported procedure to be NODATA.

If the library has a data segment, the module loader changes the first three bytes of each entry sequence to

```
mov ax,<const>
```

where the constant is, of course, the address of the data segment. To call library procedures, no special arrangements like GetProcAddress or MakeProcInstance are needed.

Libraries are not started as tasks. According to John Pollock of Microsoft, they should have a procedure called "main" like a normal C program. After loading, the main procedure is called by "the system". It is not clear whether the library has its own stack for this call, but it seems to.

Problems: When linking a library module with the windows library, slibw, I get an unresolved symbol error for WinMain. When linked with slibc (still using link4x), I get no errors from the linker but the library won't load properly. _main is called as expected, but, when it returns, the library routine __astart continues by calling exit() much the same as WinStart does after WinMain returns. For a library, however, this kills windows.

In addition to the exit problem, though, none of the windows facilities are available when the library is linked with slibc. I am not sure to what extent I am facing bugs or features, but libraries certainly don't work the way I would expect them to.

I would expect a library to work like an application module except:

- It would have no stack, but run on the stack of its caller.
- It would be automatically loaded when needed and unloaded when no longer needed.
- It could be shared by applications, or have separate copies (data segments) for each application calling it, depending on the .def file.
- It's starting procedure would initialize its data segment, if any. It would be called on a system stack, but with DS set to its own data.

Miscellaneous notes:

Due to a bug in the February release, the name of a module that exports procedures must be in all upper case. If it is not, the importing module will fail to load and windows will die.

Whenever a module (or library) entry point is invoked from another application, the code will execute with the stack segment separate from the data segment. The code must be written and compiled (-Aw) so that this will work correctly.

When a single module instance (or library) is called from multiple applications (tasks), there may be synchronization problems. Microsoft does not give any guidance in handling these, but the nonpre-emptive scheduling means that most kludges should succeed.

A library module should only be loaded once, of course, but multiple loads seem to "work": they simply load another copy of the code segment, too.

Modules (and libraries) are apparently loaded on demand, although I can't guarantee this feature. Libraries could also be "unloaded" when there are no loaded modules that import from them. I don't know whether this ever happens, but it doesn't seem too.

Applications with a single data segment don't make much sense. The linker and loader don't change the entry sequences, but Windows doesn't set up DS on calls to WinProc so you don't get any data segment. That is probably a bug, but even if it were fixed the feature wouldn't be of much use.

Libraries may be declared to have multiple data segments, but it is unclear what their use might be. Perhaps a new instance could be automatically allocated for each application that uses the library. At present, they don't even load when invoked. At least symdeb gives no indication that they do. The linker does not even assign them an entry point (it gives 0000:0000).

Because the loader stores the literal data segment address all through the code of a library module, it would appear that the data segment is fixed in memory even if it is declared movable (spelled "moveable" in Bellevue). The information is available to adjust all those locations again if the segment is swapped, however, so we can't be sure. Maybe they fixup code segments anew every time they're brought into memory. Now, at second thought, that actually seems like the right choice.

Dennis Austin
19 February 1985

Observations of MS-Windows Modules

Dennis Austin

The Microsoft Windows documentation has little to say about the use of modules aside from the simple case of a multiple-instance application. Since we would like to use modules in our applications, and we have been using them in previous distributions of Windows, I have explored some of their features. My understanding is grossly incomplete, but it has taken many hours of experiments even to turn up this much information.

The Windows Entry Sequence

The -Gw compiler flag causes cc to generate the windows entry (and exit) code for all procedures declared FAR. The entry sequence is

```
push    ds      ; These first three instructions, one
pop      ax      ; byte each, set AX to the current DS.
nop

inc      bp      ; The rest of the sequence enters the
push     bp      ; procedure saving DS and setting DS to
mov      bp,sp   ; the value in AX. (I don't know why
push     ds      ; the stored BP is forced to be odd.)
mov      ds,ax
sub      sp,<const>
```

The first three bytes of the sequence may be changed by the linker or loader, depending on the specifications of the module definition file (.def).

Application modules are tasks and can have multiple data segments, one per instance. For exported procedures, i.e. those named as exports in the .def file, the linker changes the first three bytes of the entry sequence to nop's. To call an exported procedure, AX must be preset with the correct data segment address for the instance that is to receive the call --even when calling from within the same module.

The Windows routine `MakeProcInstance` creates a code fragment that sets up AX as required, in effect binding an exported procedure to a particular instance. It yields a long pointer to code that loads AX with the data segment address of the instance and then branches to the actual procedure. The code fragment is allocated in Windows own data space.

The routine `GetProcAddress` serves the same purpose as `MakeProcInstance`, but is intended to be called from other modules. `MakeProcInstance` can only be called from the module defining the procedure. (In this release, `GetProcAddress` has been changed to require the DOS file name of the

exporting module rather than its module name --a step backwards, in my opinion.)

Procedures exported with the NODATA attribute are slightly different. They are expected not to access the module data segment. Their entry sequences are thus not modified and DS remains that of the caller. Apparently, NODATA entry points need not be compiled with the windows entry sequence (-Gw), although I'm not certain about this.

Library modules have only one data segment, or none at all. If there is no data segment, the linker makes no code modifications so the code can be compiled without the "-Gw" flag and no windows entry code will be generated. As far as entry points are concerned, declaring no data segment is apparently the same as declaring every exported procedure to be NODATA.

If the library has a data segment, the module loader changes the first three bytes of each entry sequence to

```
mov     ax,<const>
```

where the constant is, of course, the address of the data segment. To call library procedures, no special arrangements like GetProcAddress or MakeProcInstance are needed.

Libraries are not started as tasks. According to John Pollock of Microsoft, they should have a procedure called "main" like a normal C program. After loading, the main procedure is called by "the system". It is not clear whether the library has its own stack for this call, but it seems to.

Problems: When linking a library module with the windows library, slibw, I get an unresolved symbol error for WinMain. When linked with slibc (still using link4x), I get no errors from the linker but the library won't load properly. `_main` is called as expected, but, when it returns, the library routine `__astart` continues by calling `exit()` much the same as WinStart does after `WinMain` returns. For a library, however, this kills windows.

In addition to the exit problem, though, none of the windows facilities are available when the library is linked with slibc. I am not sure to what extent I am facing bugs or features, but libraries certainly don't work the way I would expect them to.

I would expect a library to work like an application module except:

- It would have no stack, but run on the stack of its caller.
- It would be automatically loaded when needed and unloaded when no longer needed. It could be shared by applications, or have separate copies (data segments) for each application calling it, depending on the .def file.
- Its starting procedure would initialize its data segment, if any. It would be called on a system stack, but with DS set to its own data.

Miscellaneous notes:

Due to a bug in the February release, the name of a module that exports procedures must be in all upper case. If it is not, the importing module will fail to load and windows will die.

Whenever a module (or library) entry point is invoked from another application, the code will execute with the stack segment separate from the data segment. The code must be written and compiled (-Aw) so that this will work correctly.

When a single module instance (or library) is called from multiple applications (tasks), there may be synchronization problems. Microsoft does not give any guidance in handling these, but the nonpre-emptive scheduling means that most kludges should succeed.

A library module should only be loaded once, of course, but multiple loads seem to "work": they simply load another copy of the code segment, too.

Modules (and libraries) are apparently loaded on demand, although I can't guarantee this feature. Libraries could also be "unloaded" when there are no loaded modules that import from them. I don't know whether this ever happens, but it doesn't seem too.

Applications with a single data segment don't make much sense. The linker and loader don't change the entry sequences, but Windows doesn't set up DS on calls to WinProc so you don't get any data segment. That is probably a bug, but even if it were fixed the feature wouldn't be of much use.

Libraries may be declared to have multiple data segments, but it is unclear what their use might be. Perhaps a new instance could be automatically allocated for each application that uses the library. At present, they don't even load when invoked. At least symdeb gives no indication that they do. The linker does not even assign them an entry point (it gives 0000:0000).

Because the loader stores the literal data segment address all through the code of a library module, it would appear that the data segment is fixed in memory even if it is declared movable (spelled "moveable" in Bellevue). The information is available to adjust all those locations again if the segment is swapped, however, so we can't be sure. Maybe they fixup code segments anew every time they're brought into memory. Now, at second thought, that actually seems like the right choice.

Dennis Austin
19 February 1985