AmigaMail

COLLABORATORS

	TITLE : AmigaMail	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

AmigaMail

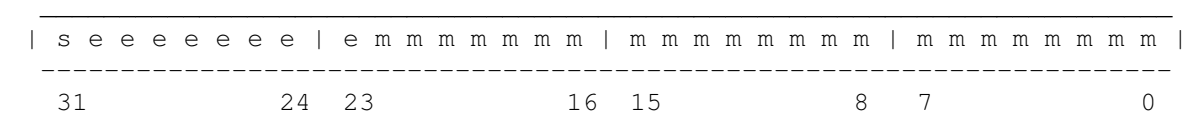
1.1 VII-1: The DR2D FORM

by Ross Cunniff and John Orr

A standard IFF FORM to describe 2D drawings has been sorely needed for a long time. Several commercial drawing packages have been available for some time but none has established its file format as the Amiga standard. The absence of a 2D drawing standard hinders the development of applications that use 2D drawings as it forces each application to understand several private standards instead of a single one. Without a standard, data exchange for both the developer and the user is difficult, if not impossible.

The DR2D FORM fills this void. This FORM was developed by Taliesin, Inc. for use as the native file format for their two-dimensional structured drawing package, ProVector. Saxon Industries and Soft Logik Publishing Corporation are planning to support this new FORM in the near future.

Many of the values stored in the DR2D FORM are stored as IEEE single precision floating point numbers. These numbers consist of 32 bits, arranged as follows:



where:

- s is the sign of the number where 1 is negative and 0 is positive.
- e is the 8 bit exponent in excess 127 form. This number is the power of two to which the mantissa is raised (Excess 127 form means that 127 is added to the exponent before packing it into the IEEE number.)
- m is the 23 bit mantissa. It ranges from 1.0000000 to 1.999999..., where the leading base-ten one is

```
          assumed.

An IEEE single precision with the value of 0.0000000 has all its bits
cleared.
```


1.2 The DR2D Chunks

```
 FORM (0x464F524D)       /* All drawings are a FORM */
 DR2D (0x44523244)       /* ID of 2D drawing */
```

1.3 FORM (0x464F524D) /* All drawings are a FORM */

```
    struct FORMstruct {
        ULONG       ID;             /* DR2D */
        ULONG       Size;
    };
```

1.4 DR2D (0x44523244) /* ID of 2D drawing */

```
The DR2D chunks are broken up into three groups: the global drawing
attribute chunks, the object attribute chunks, and the object chunks.
The global drawing attribute chunks describe elements of a 2D drawing
that are common to many objects in the drawing.  Document preferences,
palette information, and custom fill patterns are typical
document-wide settings defined in global drawing attribute chunks.
The object attribute chunks are used to set certain properties of the
object chunk(s) that follows the object attribute chunk.  The current
fill pattern, dash pattern, and line color are all set using an object
attribute chunk.  Object chunks describe the actual DR2D drawing.
Polygons, text, and bitmaps are found in these chunks.
```

1.5 The Global Drawing Attribute Chunks

```
The following chunks describe global attributes of a DR2D document.
```

```
 DRHD (0x44524844)       /* Drawing header */
 PPRF (0x50505249)       /* Page preferences */
 CMAP (0x434D4150)       /* Color map (Same as ILBM CMAP) */
 FONS (0x464F4E53)       /* Font chunk (Same as FTXT FONS chunk) */
 DASH (0x44415348)       /* Line dash pattern for edges */
 AROW (0x41524F57)       /* An arrow-head pattern */
 FILL (0x46494C4C)       /* Object-oriented fill pattern */
 LAYR (0x4C415952)       /* Define a layer */
```

1.6 DRHD (0x44524844) /* Drawing header */

The DRHD chunk contains the upper left and lower right extremes of the document in (X, Y) coordinates. This chunk is required and should only appear once in a document in the outermost layer of the DR2D file (DR2Ds can be nested).

```
    struct DRHDstruct {
        ULONG       ID;
        ULONG       Size;                   /* Always 16 */
        IEEE        XLeft, YTop,
                    XRight, YBot;
    };
```

The point (XLeft,YTop) is the upper left corner of the project and the point (XRight,YBot) is its lower right corner. These coordinates not only supply the size and position of the document in a coordinate system, they also supply the project's orientation. If XLeft < XRight, the X-axis increases toward the right. If YTop < YBot, the Y-axis increases toward the bottom. Other combinations are possible; for example in Cartesian coordinates, XLeft would be less than XRight but YTop would be greater than YBot.

1.7 PPRF (0x50505249) /* Page preferences */

The PPRF chunk contains preference settings for ProVector. Although this chunk is not required, its use is encouraged because it contains some important environment information.

```
    struct PPRFstruct {
        ULONG       ID;
        ULONG       Size;
        char        Prefs[Size];
    };
```

DR2D stores preferences as a concatenation of several null-terminated strings, in the Prefs[] array. The strings can appear in any order. The currently supported strings are:

```
            Units=<unit-type>
            Portrait=<boolean>
            PageType=<page-type>
            GridSize=<number>
```

where:

```
            <unit-type>     is either Inch, Cm, or Pica
            <boolean>       is either True or False
            <page-type>     is either Standard, Legal, B4, B5, A3,
                              A4, A5, or Custom
            <number>        is a floating-point number
```

The DR2D FORM does not require this chunk to explicitly state all the possible preferences. In the absence of any particular preference string, a DR2D reader should fall back on the default value. The

defaults are:

```
            Units=Inch
            Portrait=True
            PageType=Standard
            GridSize=1.0
```

1.8 CMAP (0x434D4150) /* Color map (Same as ILBM CMAP) */

This chunk is identical to the ILBM CMAP chunk as described in the IFF ILBM documentation.

```
    struct CMAPstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       ColorMap[Size];
    };
```

ColorMap is an array of 24-bit RGB color values. The 24-bit value is spread across three bytes, the first of which contains the red intensity, the next contains the green intensity, and the third contains the blue intensity. Because DR2D stores its colors with 24-bit accuracy, DR2D readers must not make the mistake that some ILBM readers do in assuming the CMAP chunk colors correspond directly to Amiga color registers.

1.9 FONS (0x464F4E53) /* Font chunk (Same as FTXT FONS chunk) */

The FONS chunk contains information about a font used in the DR2D FORM. ProVector does not include support for Amiga fonts. Instead, ProVector uses fonts defined in the OFNT FORM which is documented later in this article.

```
    struct FONSstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       FontID;         /* ID the font is referenced by */
        UBYTE       Pad1;           /* Always 0 */
        UBYTE       Proportional;   /* Is it proportional? */
        UBYTE       Serif;          /* does it have serifs? */
        CHAR        Name[Size-4];   /* The name of the font */
    };
```

The UBYTE FontID field is the number DR2D assigns to this font. References to this font by other DR2D chunks are made using this number.

The Proportional and Serif fields indicate properties of this font. Specifically, Proportional indicates if this font is proportional, and Serif indicates if this font has serifs. These two options were created to allow for font substitution in case the specified font is not available. They are set according to these values:

```
    0       The DR2D writer didn't know if this font is
              proportional/has serifs.
    1       No, this font is not proportional/does not have
              serifs.
    2       Yes, this font is proportional/does have serifs.
```

The last field, Name[], is a NULL terminated string containing the name of the font.

1.10 DASH (0x44415348) /* Line dash pattern for edges */

This chunk describes the on-off dash pattern associated with a line.

```
    struct DASHstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      DashID;                 /* ID of the dash pattern */
        USHORT      NumDashes;              /* Should always be even */
        IEEE        Dashes[NumDashes];      /* On-off pattern */
    };
```

DashID is the number assigned to this specific dash pattern. References to this dash pattern by other DR2D chunks are made using this number.

The Dashes[] array contains the actual dash pattern. The first number in the array (element 0) is the length of the "on" portion of the pattern. The second number (element 1) specifies the "off" portion of the pattern. If there are more entries in the Dashes array, the pattern will continue. Even-index elements specify the length of an "on" span, while odd-index elements specify the length of an "off" span. There must be an even number of entries. These lengths are not in the same units as specified in the PPRF chunk, but are multiples of the line width, so a line of width 2.5 and a dash pattern of 1.0, 2.0 would have an "on" span of length 1.0 x 2.5 = 2.5 followed by an "off" span of length 2.0 x 2.5 = 5. The following figure shows several dash pattern examples. Notice that for lines longer than the dash pattern, the pattern repeats.

figure 1 - dash patterns

By convention, DashID 0 is reserved to mean 'No line pattern at all', i.e. the edges are invisible. This DASH pattern should not be defined by a DR2D DASH chunk. Again by convention, a NumDashes of 0 means that the line is solid.

1.11 AROW (0x41524F57) /* An arrow-head pattern */

The AROW chunk describes an arrowhead pattern. DR2D open polygons (OPLY) can have arrowheads attached to their endpoints. See the description of the OPLY chunk later in this article for more

information on the OPLY chunk.

```
    #define ARROW_FIRST  0x01 /* Draw an arrow on the OPLY's first point */
    #define ARROW_LAST   0x02 /* Draw an arrow on the OPLY's last point */

    struct AROWstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       Flags;          /* Flags, from ARROW_*, above */
        UBYTE       Pad0;           /* Should always 0 */
        USHORT      ArrowID;        /* Name of the arrow head */
        USHORT      NumPoints;
        IEEE        ArrowPoints[NumPoints*2];
    };
```

The Flags field specifies which end(s) of an OPLY to place an arrowhead based on the #defines above. ArrowID is the number by which an OPLY will reference this arrowhead pattern.

The coordinates in the array ArrowPoints[] define the arrowhead's shape. These points form a closed polygon. See the section on the OPLY/CPLY object chunks for a descriptionof how DR2D defines shapes. The arrowhead is drawn in the same coordinate system relative to the endpoint of the OPLY the arrowhead is attached to. The arrowhead's origin (0,0) coincides with the OPLY's endpoint. DR2D assumes that the arrowhead represented in the AROW chunk is pointing to the right so the proper rotation can be applied to the arrowhead. The arrow is filled according to the current fill pattern set in the ATTR object attribute chunk.

1.12 FILL (0x46494C4C) /* Object-oriented fill pattern */

The FILL chunk defines a fill pattern. This chunk is only valid inside nested DR2D FORMs. The GRUP object chunk section of this article contans an example of the FILL chunk.

```
    struct FILLstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      FillID;                 /* ID of the fill */
    };
```

FillID is the number by which the ATTR object attribute chunk references fill patterns. The FILL chunk must be the first chunk inside a nested DR2D FORM. A FILL is followed by one DR2D object plus any of the object attribute chunks (ATTR, BBOX) associated with the object.

Figure 2 - fill patterns

DR2D makes a "tile" out of the fill pattern, giving it a virtual bounding box based on the extreme X and Y values of the FILL's object (Fig. A). The bounding box shown in Fig. A surrounding the pattern (the two ellipses) is invisible to the user. In concept, this rectangle is pasted on the page left to right, top to bottom like

floor tiles (Fig. B). Again, the bounding boxes are not visible. The only portion of this tiled pattern that is visible is the part that overlaps the object (Fig. C) being filled. The object's path is called a clipping path, as it "clips" its shape from the tiled pattern (Fig. D). Note that the fill is only masked on top of underlying objects, so any "holes" in the pattern will act as a window, leaving visible underlying objects.

1.13 LAYR (0x4C415952) /* Define a layer */

A DR2D project is broken up into one or more layers. Each DR2D object is in one of these layers. Layers provide several useful features. Any particular layer can be "turned off", so that the objects in the layer are not displayed. This eliminates the unnecessary display of objects not currently needed on the screen. Also, the user can lock a layer to protect the layer's objects from accidental changes.

```
    struct LAYRstruct {
        ULONG    ID;
        ULONG    Size;
        USHORT   LayerID;              /* ID of the layer */
        char     LayerName[16];        /* Null terminated and padded */
        UBYTE    Flags;                /* Flags, from LF_*, below */
        UBYTE    Pad0;                 /* Always 0 */
    };
```

LayerID is the number assigned to this layer. As the field's name indicates, LayerName[] is the NULL terminated name of the layer. Flags is a bit field who's bits are set according to the #defines below:

```
    #define LF_ACTIVE       0x01    /* Active for editing */
    #define LF_DISPLAYED    0x02    /* Displayed on the screen */
```

If the LF_ACTIVE bit is set, this layer is unlocked. A set LF_DISPLAYED bit indicates that this layer is currently visible on the screen. A cleared LF_DISPLAYED bit implies that LF_ACTIVE is not set. The reason for this is to keep the user from accidentally editing layers that are invisible.

1.14 The Object Attribute Chunks

```
 ATTR (0x41545452)       /* Object attributes */
 BBOX (0x42424F48)       /* Bounding box of next object in FORM */
 XTRN (0x5854524E)       /* Externally controlled object */
```

1.15 ATTR (0x41545452) /* Object attributes */

The ATTR chunk sets various attributes for the objects that follow it. The attributes stay in effect until the next ATTR changes the attributes, or the enclosing FORM ends, whichever comes first.

```
    /* Various fill types */
    #define FT_NONE         0    /* No fill                 */
    #define FT_COLOR        1    /* Fill with color from palette */
    #define FT_OBJECTS      2    /* Fill with tiled objects */


    struct ATTRstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       FillType;    /* One of FT_*, above      */
        UBYTE       JoinType;    /* One of JT_*, below      */
        UBYTE       DashPattern; /* ID of edge dash pattern */
        UBYTE       ArrowHead;   /* ID of arrowhead to use  */
        USHORT      FillValue;   /* Color or object with which to fill */
        USHORT      EdgeValue;   /* Edge color index        */
        USHORT      WhichLayer;  /* ID of layer it's in     */
        IEEE        EdgeThick;   /* Line width              */
    };
```

FillType specifies what kind of fill to use on this ATTR chunk's objects. A value of FT_NONE means that this ATTR chunk's objects are not filled. FT_COLOR indicates that the objects should be filled in with a color. That color's ID (from the CMAP chunk) is stored in the FillValue field. If FillType is equal to FT_OBJECTS, FillValue contains the ID of a fill pattern defined in a FILL chunk.

JoinType determines which style of line join to use when connecting the edges of line segments. The field contains one of these four values:

```
    /* Join types */
    #define JT_NONE    0            /* Don't do line joins */
    #define JT_MITER        1       /* Mitered join */
    #define JT_BEVEL        2       /* Beveled join */
    #define JT_ROUND        3       /* Round join */
```

DashPattern and ArrowHead contain the ID of the dash pattern and arrow head for this ATTR's objects. A DashPattern of zero means that there is no dash pattern so lines will be invisible. If ArrowHead is 0, OPLYs have no arrow head. EdgeValue is the color of the line segments. WhichLayer contains the ID of the layer this ATTR's objects are in. EdgeThick is the width of this ATTR's line segments.

1.16 BBOX (0x42424F48) /* Bounding box of next object in FORM */

The BBOX chunk supplies the dimensions and position of a bounding box surrounding the DR2D object that follows this chunk in the FORM. A BBOX chunk can apply to a FILL or AROW as well as a DR2D object. The BBOX chunk appears just before its DR2D object, FILL, or AROW chunk.

```
    struct BBOXstruct {
        ULONG       ID;
        ULONG       Size;
        IEEE                XMin, YMin,     /* Bounding box of obj. */
                            XMax, YMax;     /* including line width */
    };
```

In a Cartesian coordinate system, the point (XMin, YMin) is the coordinate of the lower left hand corner of the bounding box and (XMax, YMax) is the upper right. These coordinates take into consideration the width of the lines making up the bounding box.

1.17 XTRN (0x5854524E) /* Externally controlled object */

The XTRN chunk was created primarily to allow ProVector to link DR2D objects to ARexx functions.

```
    struct XTRNstruct {
        ULONG   ID;
        ULONG   Size;
        short   ApplCallBacks;              /* From #defines, below */
        short   ApplNameLength;
        char    ApplName[ApplNameLength];  /* Name of ARexx func to call */
    };
```

ApplName[] contains the name of the ARexx script ProVector calls when the user manipulates the object in some way. The ApplCallBacks field specifies the particular action that triggers calling the ARexx script according to the #defines listed below.

```
    /* Flags for ARexx script callbacks */
    #define    X_CLONE     0x0001    /* The object has been cloned */
    #define    X_MOVE      0x0002    /* The object has been moved */
    #define    X_ROTATE    0x0004    /* The object has been rotated */
    #define    X_RESIZE    0x0008    /* The object has been resized */
    #define    X_CHANGE    0x0010    /* An attribute (see ATTR) of the
                                        object has changed */
    #define    X_DELETE    0x0020    /* The object has been deleted */
    #define    X_CUT       0x0040    /* The object has been deleted, but
                                        stored in the clipboard */
    #define    X_COPY      0x0080    /* The object has been copied to the
                                        clipboard */
    #define    X_UNGROUP   0x0100    /* The object has been ungrouped */
```

For example, given the XTRN object:

```
    FORM xxxx DR2D {
            XTRN xxxx { X_RESIZE | X_MOVE, 10, "Dimension" }
            ATTR xxxx { 0, 0, 1, 0, 0, 0, 0.0 }
            FORM xxxx DR2D {
                    GRUP xxxx { 2 }
                    STXT xxxx { 0, 0.5, 1.0, 6.0, 5.0, 0.0, 4, "3.0" }
                    OPLY xxxx { 2, { 5.5, 5.5, 8.5, 5.5 } }
            }
```

```
    }
```

ProVector would call the ARexx script named Dimension if the user resized or moved this object. What exactly ProVector sends depends upon what the user does to the object. The following list shows what string(s) ProVector sends according to which flag(s) are set. The parameters are described below.

```
    X_CLONE     "appl CLONE objID dx dy"
    X_MOVE      "appl MOVE objID dx dy"
    X_ROTATE    "appl ROTATE objID cx cy angle"
    X_RESIZE    "appl RESIZE objID cx cy sx sy"
    X_CHANGE    "appl CHANGE objID et ev ft fv ew jt fn"
    X_DELETE    "appl DELETE objID"
    X_CUT       "appl CUT objID"
    X_COPY      "appl COPY objID"
    X_UNGROUP   "appl UNGROUP objID"
```

where:

```
    appl is the name of the ARexx script
    CLONE, MOVE, ROTATE, RESIZE, etc. are literal strings
    objID is the object ID that ProVector assigns to this object
    (dx, dy) is the position offset of the CLONE or MOVE
    (cx, cy) is the point around which the object is rotated or resized
    angle is the angle (in degrees) the object is rotated
    sx and sy are the scaling factors in the horizontal and
      vertical directions, respectively.
    et is the edge type (the dash pattern index)
    ev is the edge value (the edge color index)
    ft is the fill type
    fv is the fill index
    ew is the edge weight
    jt is the join type
    fn is the font name
```

The X_CHANGE message reflects changes to the attributes found in the ATTR chunk.

If the user resized the XTRN object shown above by factor of 2, ProVector would call the ARexx script Dimension like this:

```
    Dimension RESIZE 1985427 7.0 4.75 2.0 2.0
```

1.18 The Object Chunks

The following chunks define the objects available in the DR2D FORM.

```
 VBM  (0x56424D20)       /* Virtual BitMap */
 CPLY (0x43504C59)       /* Closed polygon */
 OPLY (0x4F504C59)       /* Open polygon */
 GRUP (0x47525550)       /* Group */
 STXT (0x53545854)       /* Simple text */
 TPTH (0x54505448)       /* A text string along a path */
```

1.19 VBM (0x56424D20) /* Virtual BitMap */

The VBM chunk contains the position, dimensions, and file name of an ILBM image.

```
    struct VBMstruct {
        IEEE        XPos, YPos,     /* Virtual coords */
                    XSize, YSize,   /* Virtual size */
                    Rotation;       /* in degrees */
        USHORT      PathLen;        /* Length of dir path */
        char        Path[PathLen];  /* Null-terminated path of file */
    };
```

The coordinate (XPos, YPos) is the position of the upper left hand corner of the bitmap and the XSize and YSize fields supply the x and y dimensions to which the image should be scaled. Rotation tells how many degrees to rotate the ILBM around its upper left hand corner. ProVector does not currently support rotation of bitmaps and will ignore this value. Path contains the name of the ILBM file and may also contain a partial or full path to the file. DR2D readers should not assume the path is correct. The full path to an ILBM on one system may not match the path to the same ILBM on another system. If a DR2D reader cannot locate an ILBM file based on the full path name or the file name itself (looking in the current directory), it should ask the user where to find the image.

1.20 CPLY (0x43504C59) and OPLY (0x4F504C59)

Polygons are the basic components of almost all 2D objects in the DR2D FORM. Lines, squares, circles, and arcs are all examples of DR2D polygons. There are two types of DR2D polygons, the open polygon (OPLY) and the closed polygon (CPLY). The difference between a closed and open polygon is that the computer adds a line segment connecting the endpoints of a closed polygon so that it is a continuous path. An open polygon's endpoints do not have to meet, like the endpoints of a line segment.

```
    struct POLYstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      NumPoints;
        IEEE        PolyPoints[2*NumPoints];
    };
```

The NumPoints field contains the number of points in the polygon and the PolyPoints array contains the (X, Y) coordinates of the points of the non-curved parts of polygons. The even index elements are X coordinates and the odd index elements are Y coordinates.

Figure 3 - Bezier curves

DR2D uses Bezier cubic sections, or cubic splines, to describe curves in polygons. A set of four coordinates (P1 through P4) defines the shape of a cubic spline. The first coordinate (P1) is the point where

the curve begins. The line from the first to the second coordinate (P1 to P2) is tangent to the curve at the first point. The line from P3 to P4 is tangent to the cubic section, where it ends at P4.

The coordinates describing the cubic section are stored in the PolyPoints[] array with the coordinates of the normal points. DR2D inserts an indicator point before a set of cubic section points to differentiate a normal point from the points that describe a curve. An indicator point has an X value of 0xFFFFFFFF. The indicator point's Y value is a bit field. If this bit field's low-order bit is set, the points that follow the indicator point make up a cubic section.

The second lowest order bit in the indicator point's bit field is the MOVETO flag. If this bit is set, the point (or set of cubic section points) starts a new polygon, or subpolygon. This subpolygon will appear to be completely separate from other polygons but there is an important connection between a polygon and its subpolygon. Subpolygons make it possible to create holes in polygons. An example of a polygon with a hole is the letter "O". The "O" is a filled circular polygon with a smaller circular polygon within it. The reason the inner polygon isn't covered up when the outer polygon is filled is that DR2D fills are done using the even-odd rule.

The even-odd rule determines if a point is "inside" a polygon by drawing a ray outward from that point and counting the number of path segments the ray crosses. If the number is even, the point is outside the object and shouldn't be filled. Conversely, an odd number of crossings means the point is inside and should be filled. DR2D only applies the even-odd rule to a polygon and its subpolygons, so no other objects are considered in the calculations.

Taliesin, Inc. supplied the following algorithm to illustrate the format of DR2D polygons. OPLYs, CPLYs, AROWs, and ProVector's outline fonts all use the same format:

```
typedef union {
    IEEE num;
    LONG bits;
} Coord;

#define INDICATOR       0xFFFFFFFF
#define IND_SPLINE      0x00000001
#define IND_MOVETO      0x00000002

/* A common pitfall in attempts to support DR2D has
        been to fail to recognize the case when an
        INDICATOR point indicates the following
        coordinate to be the first point of BOTH a
        Bezier cubic and a sub-polygon, ie. the
        value of the flag = (IND_CURVE | IND_MOVETO) */

Coord   Temp0, Temp1;
int     FirstPoint, i, Increment;

/* Initialize the path */
```

```
    NewPath();
    FirstPoint = 1;

    /* Draw the path */
    i = 0;
    while( i < NumPoints ) {
        Temp0.num = PolyPoints[2*i];    Temp1.num = PolyPoints[2*i + 1];
        if( Temp0.bits == INDICATOR ) {
            /* Increment past the indicator */
            Increment = 1;
            if( Temp1.bits & IND_MOVETO ) {
                /* Close and fill, if appropriate */
                if( ID == CPLY ) {
                    FillPath();
                }
                else {
                    StrokePath();
                }

                /* Set up the new path */
                NewPath();
                FirstPoint = 1;
            }
            if( Temp1.bits & IND_CURVE ) {
                /* The next 4 points are Bezier cubic control points */
                if( FirstPoint )
                    MoveTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
                else
                    LineTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
                CurveTo(    PolyPoints[2*i + 4], PolyPoints[2*i + 5],
                            PolyPoints[2*i + 6], PolyPoints[2*i + 7],
                            PolyPoints[2*i + 8], PolyPoints[2*i + 9] );
                FirstPoint = 0;
                /* Increment past the control points */
                Increment += 4;
            }
        }
        else {
            if( FirstPoint )
                MoveTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
            else
                LineTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
            FirstPoint = 0;

            /* Increment past the last endpoint */
            Increment = 1;
        }

        /* Add the increment */
        i += Increment;
    }

    /* Close the last path */
    if( ID == CPLY ) {
        FillPath();
    }
    else {
```

```
        StrokePath();
    }
```

1.21 GRUP (0x47525550) /* Group */

The GRUP chunk combines several DR2D objects into one. This chunk is only valid inside nested DR2D FORMs, and must be the first chunk in the FORM.

```
    struct GROUPstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      NumObjs;
    };
```

The NumObjs field contains the number of objects contained in this group. Note that the layer of the GRUP FORM overrides the layer of objects within the GRUP. The following example illustrates the layout of the GRUP (and FILL) chunk.

```
   FORM { DR2D              /* Top-level drawing... */
          DRHD { ... }      /* Confirmed by presence of DRHD chunk */
          CMAP { ... }      /* Various other things... */
          FONS { ... }
          FORM { DR2D               /* A nested form... */
                 FILL { 1 }         /* Ah!  The fill-pattern table */
                 CPLY { ... }       /* with only 1 object */
          }
          FORM { DR2D               /* Yet another nested form */
                 GRUP { ..., 3 }    /* Ah! A group of 3 objects */
                 TEXT { ... }
                 CPLY { ... }
                 OPLY { ... }
          }
          FORM { DR2D               /* Still another nested form */
                 GRUP { ..., 2 }    /* A GRUP with 2 objects */
                 OPLY { ... }
                 TEXT { ... }
          }
    }
```

1.22 STXT (0x53545854) /* Simple text */

The STXT chunk contains a text string along with some information on how and where to render the text.

```
    struct STXTstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       Pad0;           /* Always 0 (for future expansion) */
        UBYTE       WhichFont;      /* Which font to use */
        IEEE        CharW, CharH,   /* W/H of an individual char */
```

```
                    BaseX, BaseY,   /* Start of baseline */
                    Rotation;       /* Angle of text (in degrees) */
        USHORT      NumChars;
        char        TextChars[NumChars];
    };
```

The text string is in the character array, TextChars[]. The ID of the font used to render the text is WhichFont. The font's ID is set in a FONS chunk. The starting point of the baseline of the text is (BaseX, BaseY). This is the point around which the text is rotated. If the Rotation field is zero (degrees), the text's baseline will originate at (BaseX, BaseY) and move to the right. CharW and CharH are used to scale the text after rotation. CharW is the average character width and CharH is the average character height. The CharW/H fields are comparable to an X and Y font size.

1.23 TPTH (0x54505448) /* A text string along a path */

This chunk defines a path (polygon) and supplies a string to render along the edge of the path.

```
    struct TPTHstruct {
        ULONG   ID;
        ULONG   Size;
        UBYTE   Justification;      /* see defines, below */
        UBYTE   WhichFont;          /* Which font to use */
        IEEE    CharW, CharH;       /* W/H of an individual char    */
        USHORT  NumChars;           /* Number of chars in the string */
        USHORT  NumPoints;          /* Number of points in the path */
        char    TextChars[NumChars];/* PAD TO EVEN #! */
        IEEE    Path[2*NumPoints];  /* The path on which the text lies */
    };
```

WhichFont contains the ID of the font used to render the text. Justification controls how the text is justified on the line. Justification can be one of the following values:

```
    #define J_LEFT          0x00    /* Left justified */
    #define J_RIGHT         0x01    /* Right justified */
    #define J_CENTER        0x02    /* Center text */
    #define J_SPREAD        0x03    /* Spread text across path */
```

CharW and CharH are the average width and height of the font characters and are akin to X and Y font sizes, respectively. A negative FontH implies that the font is upsidedown. Note that CharW must not be negative. NumChars is the number of characters in the TextChars[] string, the string containing the text to be rendered. NumPoints is the number of points in the Path[] array. Path[] is the path along which the text is rendered. The path itself is not rendered. The points of Path[] are in the same format as the points of a DR2D polygon.

1.24 A Simple DR2D Example

Here is a (symbolic) DR2D FORM:

```
    FORM { DR2D
        DRHD 16 { 0.0, 0.0, 10.0, 8.0 }
        CMAP  6 { 0,0,0, 255,255,255 }
        FONS  9 { 1, 0, 1, 0, "Roman" } 0
        DASH 12 { 1, 2, {1.0, 1.0} }
        ATTR 14 { 0, 0, 1, 0, 0, 0, 0, 0.0 }
        BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
        FORM { DR2D
            GRUP  2 { 2 }
            BBOX 16 { 3.0, 4.0, 7.0, 5.0 }
            STXT 36 { 0,1, 0.5, 1.0, 3.0, 5.0, 0.0, 12, "Hello, World" }
            BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
            OPLY 42 { 5, {2.0,2.0, 8.0,2.0, 8.0,6.0, 2.0,6.0, 2.0,2.0 }
        }
    }
```

This is what the DR2D FORM above should look like:

   Figure 4 - Simple DR2D drawing

1.25 The OFNT FORM

```
 OFNT    (0x4F464E54)    /* ID of outline font file */
 OFHD    (0x4F464844)    /* ID of OutlineFontHeaDer */
 KERN    (0x4B45524C)    /* Kerning pair */
 CHDF    (0x43484446)    /* Character definition */
```

1.26 OFNT (0x4F464E54) /* ID of outline font file */

ProVector's outline fonts are stored in an IFF FORM called OFNT. This IFF is a separate file from a DR2D. DR2D's FONS chunk refers only to fonts defined in the OFNT form.

1.27 OFHD (0x4F464844) /* ID of OutlineFontHeaDer */

This chunk contains some basic information on the font.

```
    struct OFHDstruct {
        char   FontName[32];   /* Font name, null padded */
        short  FontAttrs;      /* See FA_*, below */
        IEEE   FontTop,        /* Typical height above baseline */
               FontBot,        /* Typical descent below baseline */
               FontWidth;      /* Typical width, i.e. of the letter O */
    };
```

```
    #define FA_BOLD         0x0001
    #define FA_OBLIQUE      0x0002
    #define FA_SERIF        0x0004
```

The FontName field is a NULL terminated string containing the name of this font. FontAttrs is a bit field with flags for several font attributes. The flags, as defined above, are bold, oblique, and serif. The unused higher order bits are reserved for later use. The other fields describe the average dimensions of the characters in this font. FontTop is the average height above the baseline, FontBot is the average descent below the baseline, and FontWidth is the average character width.

1.28 KERN (0x4B45524C) /* Kerning pair */

The KERN chunk describes a kerning pair. A kerning pair sets the distance between a specific pair of characters.

```
    struct KERNstruct {
        short   Ch1, Ch2;  /* The pair to kern (allows for 16 bits...) */
        IEEE    XDisplace, /* Amount to displace -left +right */
                YDisplace; /* Amount to displace -down +up */
    };
```

The Ch1 and Ch2 fields contain the pair of characters to kern. These characters are typically stored as ASCII codes. Notice that OFNT stores the characters as a 16-bit value. Normally, characters are stored as 8-bit values. The wary programmer will be sure to cast assigns properly to avoid problems with assigning an 8-bit value to a 16-bit variable. The remaining fields, XDisplace and YDisplace, supply the baseline shift from Ch1 to Ch2.

1.29 CHDF (0x43484446) /* Character definition */

This chunk defines the shape of ProVector's outline fonts.

```
    struct CHDFstruct {
        short   Ch;         /* The character we're defining (ASCII) */
        short   NumPoints;  /* The number of points in the definition */
        IEEE    XWidth,     /* Position for next char on baseline - X */
                YWidth;     /* Position for next char on baseline - Y */
     /* IEEE    Points[2*NumPoints]*/       /* The actual points */
    };


    #define INDICATOR   0xFFFFFFFF  /* If X == INDICATOR, Y is an action */
    #define IND_SPLINE  0x00000001  /* Next 4 pts are spline control pts */
    #define IND_MOVETO  0x00000002     /* Start new subpoly */
    #define IND_STROKE  0x00000004  /* Stroke previous path */
    #define IND_FILL    0x00000008  /* Fill previous path */
```

Ch is the value (normally ASCII) of the character outline this chunk

defines. Like Ch1 and Ch2 in the KERN chunk, Ch is stored as a 16-bit value. (XWidth,YWidth) is the offset to the baseline for the following character. OFNT outlines are defined using the same method used to define DR2D's polygons (see the description of OPLY/CPLY for details).

Because the OFNT FORM does not have an ATTR chunk, it needed an alternative to make fills and strokes possible. There are two extra bits used in font indicator points not found in polygon indicator points, the IND_STROKE and IND_FILL bits (see defines above). These two defines describe how to render the current path when rendering fonts.

The current path remains invisible until the path is either filled and/or stroked. When the IND_FILL bit is set, the currently defined path is filled in with the current fill pattern (as specified in the current ATTR chunk). A set IND_STROKE bit indicates that the currently defined path itself should be rendered. The current ATTR's chunk dictates the width of the line, as well as several other attributes of the line. These two bits apply only to the OFNT FORM and should not be used in describing DR2D polygons.