# Contents

*Advanced Path Construction Methods Using Level 2 Features*  5

1. Introduction  5

2. Binary Formats  6
   - Binary Token Encoding  6
   - Binary Object Sequence  8
   - Portability  9

3. User Paths  10
   - When To Use User Paths  10
   - When to Cache  11
   - Modifying Cache Limits  12

4. Types of Paths  13
   - Performance Tests  16

5. Sample Code  20
   - Calculating the User Path Bounding Box  20
   - Encoding Numbers and Operators in Binary Token Encoding  22
   - Outputting Data  24

6. Conclusions  25

*Appendix: Changes Since Earlier Versions*  27

*Index*  29
1 Introduction

This technical paper focuses on using binary formats and the user path mechanism to construct and render paths under PostScript™ Level 2. We assume you are familiar with section 3.12, “Binary Encoding Details,” and section 4.6, “User Paths,” in the *PostScript Language Reference Manual, Second Edition.*

The PostScript language Level 2 and the Display PostScript™ system offer a variety of features that can aid in constructing and rendering paths. Data and operators can be coded in ASCII or binary formats. Paths can be constructed regularly or they can make use of the user path mechanism. A combination of these different methods can yield a smaller number of bytes needed to represent a path and superior performance during the rendering of that path.

This paper is divided into four sections.

- The first section deals with the various binary formats available, including binary token encoding and binary object sequence.
- The second section discusses the user path mechanism and when it should be used.
- The third section discusses the different variations of path construction that can be utilized (using the different binary formats and user paths), and their relative performance and compactness. As part of the research into binary formats and user paths, a path generator was created to output random paths in a variety of formats.
- The last section describes some of the sample C language routines used by the path generator. These routines might benefit driver writers.

For information on the use of user paths in a Display PostScript system, see *Programming the Display PostScript System with NeXTstep: Chapter 5, “Path Construction and Rendering,”* and Chapter 9, “Drawing Issues and Scrolling.”
2 Binary Formats

Binary formats are a standard feature of PostScript Level 2 and are already implemented in Display PostScript systems. Binary token encoding (BTE) and binary object sequence (BOS) offer different trade-offs: the size of the resultant PostScript language file and the speed of execution of the PostScript language code.

Note A file does not need to be strictly binary or ASCII. Binary formats and ASCII can be freely inter-mixed within a PostScript language program.

2.1 Binary Token Encoding

With binary token encoding (BTE), operators and numbers are encoded using a series of tokens. The identifying header byte declares the data format and is followed by a fixed number of data bytes. (The number of data bytes depend on the data format.)

Three binary token encoding number formats are available.

- IEEE floating point. This format uses 4 bytes to represent a floating-point number.
- Fixed point. This format uses either 3 or 5 bytes to represent a fixed-point number.
- Integer. This format uses either 1, 2, or 4 bytes to represent an integer.

The number formats also support high-order byte first (big endian) and low-order byte first (little endian) byte orders.
The format to use depends on the type of application generating the path. For example, an illustration application might use 16-bit fixed point numbers in its graphics model. This type of representation gives it a speed advantage when calculating complex transformations on screen. Instead of converting the graphics data to ASCII when printing, the application can choose a 16-bit BTE format for its data.

Not only can data be represented in binary format, other binary objects include homogeneous number arrays (HNAs), binary strings, and even executable operators. More information about these formats can be found in section 3.12.1, “Binary Tokens,” of the *PostScript Language Reference Manual, Second Edition.*
Figure 2 Binary token encodings for homogeneous number arrays, operators, and binary strings

As can be seen in Figure 2, BTE can offer significant reductions in the number of bytes necessary to represent a number. BTE is especially helpful in environments in which disk storage is at a premium. In a PostScript language file where a considerable amount of graphics data is present, reductions in file sizes are typically 30–50%. This savings also applies to I/O bound environments limited to slower transmission channels (like serial) in which every byte adds to the print time. Note, however, that the BTE format does not execute any faster inside the PostScript interpreter.

2.2 Binary Object Sequence

A binary object sequence (BOS) is an executable array of objects whose format is similar to the internal format used by the interpreter after the scanner has processed a group of related tokens. It can be thought of as ‘pre-scanned’ PostScript language code. A BOS can be a simple object, a compound object, or an array nested to an arbitrary depth. It is very fast and efficient for the PostScript interpreter to process a BOS, but the format is less than compact.

Each object is represented by at least eight bytes. This makes it an excellent choice for high-bandwidth communications channels (such as a display system where internal transfers of large blocks of data are very fast) but of little use for printer devices. PostScript language files that use BOS are typically 150–200% larger than their ASCII counterparts.

Because PostScript language files represented using BOS are so large, they are not discussed further in this paper. More information on the use of BOS in a Display PostScript system can be found in Programming the Display PostScript System with NeXTstep, Chapter 13, “Binary Encoding Formats.”
2.3 Portability

When using binary data portability issues exist. Some binary bytes cannot be transmitted on 7-bit channels (for example, serial) and others can interfere with the communications protocol. Either a transparent communications channel such as AppleTalk® or a binary protocol such as Adobe™ Binary Communications Protocol must be used.

Applications that send binary image data in the print stream can avoid this restrictive situation by encoding the binary image data in a binary string and using either a hexadecimal (expansion factor of 1:2) or ASCII base-85 (expansion factor of 4:5) encoding. The resulting data will be portable at the cost of a larger file size. Note that although ASCII base-85 encoding is the more compact of the two encodings, it is only available on Level 2 printers.

\[ \text{e.g. } <\text{8A423948B}> \text{ vs. } <\text{-MF8\RMZ}> \]

When using binary data inside a PostScript language file, drivers should write out the proper document structuring conventions (DSC) comments, including \%LanguageLevel:, \%DocumentData:, \%BeginData:, and \%EndData. These comments increase the chances of portability of the PostScript language file if an intelligent spooler is available.

Spoolers can recognize these comments and react accordingly. For example, a spooler can re-route a raw binary file (indicated by the header-level comment \%DocumentData: Binary) to use a transparent communications channel. A spooler can also re-route the file to a PostScript Level 2 printer if the \%LanguageLevel: of the document indicates that Level 2 operators are used. More information on the DSC can be found in Appendix G of the PostScript Language Reference Manual, Second Edition.
3 User Paths

In PostScript Level 2 and the Display PostScript system there is a new method for specifying a path as a completely self-contained description called the user path. New operators have been defined in the PostScript language to use user paths: `setbbox`, `uappend`, `ucache`, `ueofill`, `ufill`, `upath`, `ustroke`, and `ustrokepath`. Using user paths can have excellent performance gains in certain situations, especially if the path is held in the user path cache for subsequent use. A version of the user path, the *encoded* user path, offers a compact and convenient representation of a path.

3.1 When To Use User Paths

User paths offer performance benefits to applications that make frequent use of identical paths. Examples of these types of programs include CAD and architectural layout applications, object-oriented drawing and illustration programs, and other applications that typically present a palette of objects that can be manipulated by the user. The definition of an identical path is best illustrated through the example in Figure 3, which contains nine identical paths.

**Figure 3** Identical paths

![Identical paths](image)

In example 3a, the basic path is shown. It consists of two rectangles representing the back and seat of a chair, and two irregular shapes (drawn using lines and arcs) to represent the arms. In example 3b, the same path is used twice. It is filled with black and then stroked with white. Examples 3c and 3d show that the coordinate system can be translated, scaled, and/or rotated still using the same path in each instance. Example 3e uses the basic path three times. It is filled with gray to represent the shadow of the chair, filled with a cyan-colored pattern, and finally stroked with black.

In each of the examples the sequence of commands sent to the PostScript interpreter to draw the chair are the same; it is only the graphics state and coordinate system that changes. Identical paths are candidates for uncached user paths.
An identical translated path is a member of a group of identical paths whose coordinate systems are the same except for the translation component; rotation and scaling remains constant. Examples 3a, 3b, and 3e together have six identical translated paths. Example 3d has two identical translated paths (the stroke and the fill). Identical translated paths are candidates for cached user paths.

Note that user paths can only deal with a subset of the regular PostScript language drawing operators: moveto, rmoveto, lineto, rlineto, curveto, rcurveto, arc, arcn, arct, and closepath. Text operators are not available. If it is likely that objects in the application will use text, examine Technical Note #5113, “Emulation of the execform Operator,” and consider defining these objects as forms.

### 3.2 When to Cache

Figure 4 illustrates the relative performance values for regular, uncached, and cached user paths. User paths should be used when there are one or more identical paths in the illustration. If there are two or more identical translated paths in an illustration, the user path should be cached. Note that user path caching involves a non-trivial amount of overhead, typically 15–20% of the normal rendering time for the first invocation of the user path.

**Figure 4** Relative times between different types of paths

Caching is not performed automatically, it must be explicitly invoked using the `ucache` operator in the definition of the user path. The application must decide whether user paths should be cached. Section 4.6.3, “User Path Cache,” of the PostScript Language Reference Manual, Second Edition discusses the conditions in which the user path cache will be effective.
In general, multiple instances of a user path painted at different places on the page take advantage of the user path cache when the current transformation matrix has been altered only by translate (identical translated paths). If paths are scaled or rotated from the original path, then caching is not effective (identical paths).

### 3.3 Modifying Cache Limits

Even though caching is specifically requested by the ucache operator, a path might not be cached depending on the size of the reduced path. The user path cache has two control parameters that determine path caching behavior. The MaxUPathItem parameter denotes the maximum number of bytes that can be occupied by a single cached user path—reduced paths that are larger than this limit will not be cached. The MaxUPathCache parameter limits the total number of bytes occupied by the user path cache.

Different devices will have different default values for MaxUPathItem and MaxUPathCache. These limits might not be sufficient depending on the size of the paths being drawn. It might be useful to an application developer to know the current cache parameters. This can be done using the ucachestatus operator. Its format is as follows:

```
– ucachestatus mark bsize bmax rsize rmax blimit
```

- **blimit** is the value of the MaxUPathItem parameter. It represents the maximum number of bytes that each user path can occupy.

- **rmax** is the value of the MaxUPathCache parameter. It represents the total size of the cache.

- **rsize** is the current amount of VM used in the cache.

- **bmax** is the maximum number of user paths that can be cached.

- **bsize** is the current number of cached user paths.

The purpose of the mark is to delimit the values returned by ucachestatus. This enable a program to determine how many values were returned (using counttomark) and to discard any unused ones (using cleartomark).

A user application can change the MaxUPathItem parameter. The per user path limit can be changed to \( m \) bytes by issuing the following code:

```
<< /MaxUPathItem m >> setuserparams
```
The application should never alter the total user path cache size; MaxUPathCache is a system-level parameter. A change is normally made by the systems administrator on a device-by-device basis using the setsystemparams operator.

4 Types of Paths

A combination of the binary token encoding format and user paths can provide better methods for constructing paths under Level 2. Performance tests were done on a variety of paths using different combinations of the binary/user path alternatives. A total of 7 different path types were tested: ASCII regular paths, binary paths, ASCII user paths, binary user paths, ASCII encoded user paths, binary encoded user paths, and HNA encoded user paths.

For the purposes of illustration, all binary examples are represented as hexadecimal numbers with spaces in between values. All example paths that are use binary token encoding use the IEEE number format.

ASCII Regular Path

An ASCII regular path is a path drawn with PostScript language marking operators and does not use the user path mechanism. This is by far the most common method of path construction; most existing PostScript language drivers output this type of path. Note that in the following example the path construction operators have been redefined to one or two letter equivalents for compactness.

```
256.94 87.87 m
267.27 473.81 85.87 351.12 113.22 247.39 c
-71.27 -52.56 rm
184.04 491.72 267.70 382.84 323.24 383.57 c
411.48 75.97 225.17 267.60 124.88 an
cp
...rest of the path...
stroke
```
Binary Regular Path

A binary regular path is similar to an ASCII regular path in that it is drawn with PostScript language marking operators and does not use the user path mechanism. Its format, however, is in binary token encoding; numbers and operators are expressed as binary tokens. There is a variety of number formats available including integer, fixed point, and IEEE formats. The sample code included with this paper has C language routines that can output a number in any of these formats. See section 5.2, “Encoding Numbers and Operators in Binary Token Encoding,” for more information.

```
8A43807852 8A42AFBD71 926B % moveto
8A4385A28F 8A43ECE7AE 8A42ABBD71 8A43AF8F5C 8A42E270A4
8A437763D7 922B % curveto
8A428EB8A3D 8AC2523D71 9286 % rmoveto
8A43380A3D 8A43F5DC29 8A4385D99A 8A43BF6B85 8A43A19EB8
8A43BFC8F6 922B % curveto
8A43CDDB71 8A4297F0A4 8A43612B85 8A4385CCCD 8A42F9C28F
9206 % arcn
9216 % closepath
...rest of the path...
92A7 % stroke
```

ASCII User Path

An ASCII user path has an ASCII data representation and uses the user path mechanism. Two items in the definition differentiate it from a regular path: the `setbbox` and `ucache` operators. The `setbbox` operator defines the bounds of the user path. This is where user paths obtain part of their enhanced performance. By knowing the bounds of a graphic, the user path mechanism can render it more quickly. The `ucache` operator specifies that the user path should be cached. A low-level description of the path is kept by the user path cache mechanism for later use. Translated identical paths can take advantage of the cached user path.

```
/aupath {
  ucache
  -253.64 -150.21 756.46 779.93 setbbox
  256.94 87.87 moveto
  267.27 473.81 85.87 351.12 113.22 247.39 curveto
  -71.27 -52.56 rmoveto
  184.04 491.72 267.70 382.84 323.24 383.57 curveto
  411.48 75.97 225.17 267.60 124.88 arcn
  closepath
  ...rest of the path...
} cvlit def
aupath ustroke
```
Binary User Path

A binary user path is the binary counterpart to the ASCII user path. The path data is in a binary token representation such as a binary regular path and uses the user path mechanism.

```
/bupath { 92B1  % ucache
  8AC37DA3D7 8AC31635C3 8A443D1D71 8A4442FB85 928F % setbbox
  8A43807B52 8A42AFBD71 926B % moveto
  8A4385A28F 8A43ECE7AE 8A42ABBD71 8A43AF8F5C 8A42E270A4
  8A437763D7 922B % curveto
  8A4328E8A3D 8AC2523D71 9286 % rmoveto
  8A43380A3D 8A43F5DC29 8A4385D99A 8A43BF6B85 8A43A19EB8
  8A43BCF8F6 922B % curveto
  8A43CDBD71 8A4297F0A4 8A43612B85 8A4385CCCD 8A42F9C28F
  9206 % arc
  9216 % closepath
  ...rest of the path...
} 922D 9233 % cvlit def
bupath 92B7 % ustroke
```

ASCII Encoded User Path

An ASCII encoded user path is a very compact representation of a user path. It consists of a data array and an operator string. The data array contains all the operands used by the operators. The encoding scheme for operators is specialized to encoded user paths and is found in section 4.6.2, “Encoded User Paths,” in the PostScript Language Reference Manual, Second Edition. The operator string should be represented in either hex or ASCII base-85 (preferred).

```
/aeupath {  
  { -253.64 -150.21 756.46 779.93 256.94 87.87 267.27 473.81 85.87
    351.12 113.22 247.39 -71.27 -52.56 184.04 491.72 267.70 382.84
    323.24 383.57 411.48 75.97 225.17 267.60 124.88
    ...rest of the operands... }  
  <0B001050205080A ...rest of the operators...>
} cvlit def
aeupath ustroke
```
**Binary Encoded User Path**

A binary encoded user path is similar to an ASCII encoded user path, but binary token encoding is used to represent the operands, and the operator string is represented as a binary string. A binary encoded user path is more compact than an ASCII encoded user path.

```
/beupath {
    8AC37DA3D7 8AC31635C3 8A443D1D71 8A4442FB85 8A43807852 8A42AFBD71 8A4385A28F 8A43ECE7AE
    8A42ABBD71 8A43AF8F5C 8A42E270A4 8A437763D7 8A28E8A3D 8AC2523D71 8A4380A3D 8A43F5DC29
    8A438599A 8A43BF6B85 8A43A19EB8 8A43BFC8F6 8A43CDBD71 8A4297F0A4 8A43612B85 8A4385CCCD
    8A42F9C28F ...more operands...
    8E 64 0B 00 01 05 02 05 08 0A ...more operators...
} 922D 9233 % cvlit def
beupath 92B7 % ustroke
```

**HNA Encoded User Path**

An HNA encoded user path is similar to an ASCII encoded user path, but the data array is represented by a homogenous number array and the operators array as a binary string. This is the most compact representation of an encoded user path. In this example, both the data array and the operator string are encoded in ASCII85 format to ensure portability across 7-bit communications channels.

```
/heupath {
   <-PqW5'_gS:_\P{t6p2QP6pgd[6\"Ba6Ee]B6\V6b6gXmj6E8bJ6a1Nn6K21K6["# ?_N"F6_GW7"6T>A)6h7426\x3<6bdE56_u)666g_{6d<:)6C7,\86X_M66\Wb16XR6\4a6V8E ...rest of the operands... ->
   <-\NL22!X/i: ...rest of the operators... ->
} cvlit def
heupath ustroke
```

### 4.1 Performance Tests

The tests whose results appear in Table 1, were run on a pre-release Level 2 printer. The communications channel was a 9600 baud serial line, and the printer was configured to use the Adobe Binary Communications Protocol (BCP). Test files that were not BCP clean (that is, they contained binary data that conflicted with the communications protocol) were run through a BCP filter.

Each test defined a random path of 100 nodes that was then tiled 5×5 on a letter sized page. The ‘Ratio’ statistics indicate the percentage decrease in execution time or the decrease in file size for each of the tests. Increases in
execution time/file size are shown in parentheses. Files sizes (in bytes) were calculated based on the definition and rendering of the path only. Extraneous comments and timing code were not included.

There are two variations to each test: a stroked version and a filled version.

**Level 1 Tests**

Test 1.1 is an ASCII regular path. This is typical of the output you would expect from most PostScript printer drivers. All of the ratio calculations are in comparison to this base case.

Tests 1.2–1.4 test binary regular paths in which the data points are encoded using BTE. Integers are represented in the smallest number of bytes possible in the best fit integer (BFI), and best fit 16-bit fixed (BFF) cases. Note that using BTE decreases file size but does not affect execution time.

**Level 2 User Path Tests**

Tests 2.1 and 2.5 use regular user paths, cached and uncached. Cached user paths have a definite performance edge.

Tests 2.2–2.4 and 2.6–2.8 test binary user paths in which data points and operators are encoded using BTE, cached and uncached.

**Level 2 Encoded User Path Tests**

Tests 3.1 and 3.5 test regular encoded user paths. The data array is in ASCII and the operator array is a binary string in ASCII base-85 format.

Tests 3.2–3.4 and 3.6–3.8 test binary encoded user paths. The data array is in BTE format and the operator array is a binary string.

**Level 2 HNA User Path Tests**

Tests 4.1–4.6 are encoded user paths using a homogeneous number array (HNA) for the data array and a binary string as the operator array; both arrays are in ASCII base-85 format.

Table 1 shows the results of the performance test that were run on a pre-release Level 2 printer.
Table 1  Performance test results

<table>
<thead>
<tr>
<th>Format</th>
<th>Test</th>
<th>Stroke time</th>
<th>Fill time</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sec.</td>
<td>Ratio</td>
<td>Sec.</td>
</tr>
<tr>
<td>Level 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td>1.1</td>
<td>54.21</td>
<td>0%</td>
<td>61.34</td>
</tr>
<tr>
<td>Best Fit Int</td>
<td>1.2</td>
<td>54.66</td>
<td>(1%)</td>
<td>58.86</td>
</tr>
<tr>
<td>Best Fit Fix</td>
<td>1.3</td>
<td>54.48</td>
<td>(0%)</td>
<td>59.36</td>
</tr>
<tr>
<td>IEEE Float</td>
<td>1.4</td>
<td>54.48</td>
<td>(1%)</td>
<td>59.82</td>
</tr>
<tr>
<td>Level 2 user paths</td>
<td>uncached ASCII</td>
<td>2.1</td>
<td>53.32</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2 encoded user paths</td>
<td>uncached ASCII</td>
<td>3.1</td>
<td>52.58</td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2 HNA user paths</td>
<td>uncached Fix 16</td>
<td>4.1</td>
<td>52.78</td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Conclusions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Uncached user paths typically exhibit a 1–6% performance gain over regular drawn paths. Because the bounding box of the object is known, the number of calculations performed during drawing is decreased (for example, clipping calculations).
Caching user paths is only feasible if the path will be used more than once on the page since there is a performance penalty of approximately 15–20% when the object is first put into the cache. Once cached, however, performance gains of 76% (4 times faster) for filled paths and 59% (2.5 times faster) for stroked paths can be realized.

Encoded user paths, while not faster in execution time than regular user paths, can be a convenient and compact method of path representation. This is especially true in the case of HNA encoded paths, where files can be 15% smaller than their unencoded counterparts.

Using binary token encoding does not affect execution time, but can decrease the file size (and thus the transmission time). Best fit integers are the most compact of all binary representations, but the inability to represent fractional numbers might be of consequence to some applications.

The clear ‘winner’ in these tests is test 4.4: a Level 2 HNA cached user path using a 16-bit fixed point format. As well as being 2.5× faster for stroked and 4× faster for filled user paths, the file size was 65% smaller than regular ASCII paths.

It is important to note that stroked user paths can take up to four times more cache storage than filled user paths. The stroked user path had a reduced path size of 57,000 to 59,000 bytes depending on the type of user path representation. The filled user path had a reduced path size of 14,000 to 15,000 bytes. The difference in sizes is due to the increased number of reduced path primitives that are needed to represent a stroke in the cache as opposed to a fill.
5 Sample Code

The following section describes sample C language routines used by the path generator discussed in this technical note. These routines are provided in the Postscript Language Software Development Kit and might be useful to printer driver writers.

5.1 Calculating the User Path Bounding Box

Part of the overhead of generating a user path is keeping track of the bounding box. The generate.c file contains a path generator that demonstrates how to keep track of the bounding box while constructing the path. The method used in this paper could be more accurate. It was designed with the idea of keeping calculations to a minimum.

Keep track of the current point in two variables (cx and cy); these will be referenced when using the relative operators. Establish four variables (llx, lly, urx, ury) to hold the current bounding box. Initially set these values as follows:

\[
\begin{align*}
\text{llx} &= \text{lly} = \text{MAX\_INT}; \\
\text{urx} &= \text{ury} = \text{MIN\_INT};
\end{align*}
\]

The values of \text{MAX\_INT} and \text{MIN\_INT} are the maximum and minimum integer values representable by the host machine. These definitions are usually found in \text{<limits.h>}.

When a \text{moveto} or a \text{lineto} is generated, update the current point variables to reflect the new position \((px, py)\):

\[
\begin{align*}
\text{cx} &= \text{px}; \\
\text{cy} &= \text{py};
\end{align*}
\]

In addition, the new coordinates should be compared to the bounding box:

\[
\text{Check\_BBox}(\text{cx, cy});
\]

Where \text{Check\_BBox} performs some simple comparisons:

\[
\begin{align*}
\text{static void Check\_BBox}(\text{float32 f, float32 g}) \{ \\
\text{if (f > urx) urx} &= \text{f; } \\
\text{if (f < llx) llx} &= \text{f; } \\
\text{if (g > ury) ury} &= \text{g; } \\
\text{if (g < lly) lly} &= \text{g}
\}
\]

When a \text{curveto} is generated there are six data points \((mx, my, nx, ny, px, py)\). The current point should be updated to reflect the end position of the curve.

\[
\begin{align*}
\text{cx} &= \text{px}; \\
\text{cy} &= \text{py};
\end{align*}
\]
The bounding box check is now more complicated, the resulting Bézier should be differentiated to find the maxima and minima of the curve. To simplify calculations, assume that the control points of the curve \((mx, my, nx, ny)\) are limits. The bounding box check can be made directly against them.

\[
\text{Check\_BBox}(mx, my);
\text{Check\_BBox}(nx, ny);
\]

The bounding box check should also be made against the endpoint of the curve.

\[
\text{Check\_BBox}(cx, cy);
\]

When an \texttt{moveto} or \texttt{rlineto} is generated, update the current point variables to reflect the change in position \((rpx, rpy)\).

\[
\text{cx} = \text{cx} + \text{rpx}; \text{cy} = \text{cy} + \text{rpy};
\]

The bounding box should be checked in the regular manner.

\[
\text{Check\_BBox}(cx, cy);
\]

When an \texttt{rcurveto} is generated, the method is similar to \texttt{curveto}, but with relative points \((mrx, mry, nrx, nry, prx, pry)\):

\[
\text{Check\_BBox}(cx + mrx, cy + mry);
\text{Check\_BBox}(cx + nrx, cy + nry);
\text{cx} = \text{cx} + prx; \text{cy} = \text{cy} + pry;
\text{Check\_BBox}(cx, cy);
\]

When an \texttt{arc} or \texttt{arcn} is generated, update the current point (where \(px\) and \(py\) are the center of the arc, \(r\) is the radius, and \(a1\) and \(a2\) are the start and ending angles of the arc).

\[
\text{cx} = px + r * \cos(a2 * \pi / 180); \text{cy} = py + r * \cos(a2 * \pi / 180);
\]

Check the bounding box. In this instance, simplifying the checking routine consists of checking all four extremes of a circle with center \((px, py)\) and radius \(r\).

\[
\text{Check\_Angles}(px, py, r);
\]

Where \texttt{Check\_Angles} is:

\[
\text{static void Check\_Angles(float32 cx, float32 cy, float32 r) \{} \\
\text{  /* We punt on the bounding box for an arc -> consider it a circle */} \\
\text{  \text{Check\_BBox}(cx+r,cy); /* 0 degrees */} \\
\text{  \text{Check\_BBox}(cx,cy+r); /* 90 degrees */} \\
\text{  \text{Check\_BBox}(cx-r,cy); /* 180 degrees */} \\
\text{  \text{Check\_BBox}(cx,cy-r); /* 270 degrees */} \\
\text{\}}
\]
This routine can be modified to make a more intelligent check.

An arct takes five parameters, two defining points \((p1\text{ and } p2)\), and a radius \((r)\). Combined with the current point, \(p1\) and \(p2\) define two tangent lines. The center of the arc is located within the inner angle between the tangent lines. It is the only point located at distance \(r\) in a direction perpendicular to both lines. The PostScript Language Reference Manual, Second Edition has an excellent picture and description of the arct command in Chapter 8, “Operators.” Finding the resultant current point and checking the bounding box are a little complicated:

\[
\begin{align*}
\text{Check_BBox}(p1.x, p1.y); \\
v1.x &= cx - p1.x; \quad v1.y = cy - p1.y; \\
v2.x &= p2.x - p1.x; \quad v2.y = p2.y - p1.y; \\
v1len &= \sqrt{(v1.x^2 + v1.y^2)}; \\
v2len &= \sqrt{(v2.x^2 + v2.y^2)}; \\
\text{if } (v1len == 0 \text{ || } v2len == 0) \text{ msg(MSG_ERROR,"Undefined arcto\n")}; \\
v1.x &= v1len; \quad v1.y /= v1len; \\
v2.x &= v2len; \quad v2.y /= v2len; \\
cos2theta &= -(v1.x * v2.x + v1.y * v2.y); \\
\text{if } (\cos2theta == -1 \text{ || } \cos2theta == 1) \{ \\
\quad \text{cx} = p1.x; \quad \text{cy} = p1.y; /* Degenerative arc */ \\
\} \text{ else } \\
\quad \tantheta &= \sqrt{(1 - \cos2theta) / (1 + \cos2theta)}; \\
\quad \text{cx} &= p1.x + v2.x * r * \tantheta; \\
\quad \text{cy} &= p1.y + v1.y * r * \tantheta; \\
\} \text{ Check_BBox(cx, cy);}
\]

For the sake of simplicity, the point \(p1\) is checked against the current bounding box rather than finding the maximum limits of the arc generated by arct, which would provide more accurate results. In addition, the final point on the path (the end of the arc) is also checked against the bounding box.

5.2 Encoding Numbers and Operators in Binary Token Encoding

Number Formats

The following routines are available in the file bte.c and can be useful for converting IEEE numbers to binary token encoding (BTE) formats including integer, fixed point, and IEEE floating point. Also included is a routine to create a homogeneous number array from an array of floating point numbers.

Note that the format of the output your driver generates will most likely be based on the internal format that is used to store graphical data points. These routines are meant as examples and not as solutions.

\[
\text{extern void BTE_int(int32 num, bool hilo,} \\
\quad \text{uint8 *buf, uint8 *bsize);}\
\]
This best fit integer routine will examine \textit{num} and determine the minimum number of bytes needed to represent \textit{num} in an integer binary token. The token will be placed in \textit{buf} and the size of the token returned in \textit{bsize}. The order of the bytes within the token is determined by \textit{hilo}.

```c
extern void BTE_fix(float32 num, bool hilo,
           uint8 *buf, uint8 *bsize);
```

This best fit fixed-point procedure takes an IEEE floating point number and converts it to a fixed-point number of either 16 or 32 bits. Accuracy of the fractional portion of the number is ±0.031 (5 bits). The token will be placed in \textit{buf} and the size of the token returned in \textit{bsize}. The order of the bytes within the token is determined by \textit{hilo}.

```c
extern void BTE_IEEE(float32 num, bool hilo,
            uint8 *buf, uint8 *bsize);
```

This function takes an IEEE 32-bit floating point number and converts it to a IEEE floating point binary token. The token will be placed in \textit{buf} and the size of the token returned in \textit{bsize}. The order of the bytes within the token is determined by \textit{hilo}.

```c
extern void BTE_hna(float32 *nums, uint16 nlen, uint8 format,
            uint8 **buf, uint16 *blen);
```

This routine takes an array of IEEE floating point numbers and a formatting flag. Valid values for the format flag can be found in section 3.12.1, “Binary Tokens,” of the \textit{PostScript Language Reference Manual, Second Edition}.

The \textit{nums} array is converted to a homogeneous number array (HNA) and placed in the newly created buffer \textit{buf}, the size of the buffer is returned in \textit{blen}. Note that no bounds checking is done to ensure that the numbers in \textit{nums} fit in the format described by the \textit{format} flag.

\textbf{PostScript Language Operators}

The following routine is found in \textit{mapname.c} and can help to generate binary token encodings of operators using token identifiers 145 (literal name) and 146 (executable name).

```c
extern int16 MapName (char *name);
```

The function \textbf{MapName} will take a PostScript language operator name and return the appropriate index into the system table for that name. A −1 is returned if the name is not found.

```c
putchar(146); putchar(MapName“moveto”);
```

This example will produce the executable BTE equivalent of the \textbf{moveto} operator.
5.3 Outputting Data

The *trans.c* file contains routines for outputting data in a variety of formats. Useful routines include conversions of a buffer to hex, ASCII base-85, and binary string token formats.

```c
extern void trans_byte(FILE *fout, uint8 b, bool quoted);
```

This routine will output the byte *b* to the *fout* stream. If *quoted* is set to TRUE, the byte may be quoted, as necessary, according to the Adobe binary communications protocol. For example,

```c
trans_byte(stdout, 0x04, TRUE);
```

will send the following bytes to stdout:

```
0x01 0x44
```

```c
extern void trans_hex(FILE *fout, uint8 *source, uint16 slen);
```

This function will output the buffer *source* to the *fout* stream as a hexadecimal string. For example,

```c
uint8 mybuf[5] = { 0, 3, 13, 100, 3 };
trans_hex(stdout, mybuf, 5);
```

produces the following string:

```c
<00030D6403>.
```

```c
extern void trans_ascii85(FILE *fout, uint8 *source, uint16 slen);
```

This procedure will output the buffer *source* to the *fout* stream as an ASCII base-85 string. For example,

```c
uint8 mybuf[5] = { 0, 3, 13, 100, 3 };
trans_ascii85(stdout, mybuf, 5);
```

produces the following string:

```
~<!Z@&+~>.
```
extern void trans_bst(FILE *fout, uint8 *source, uint16 slen, 
  bool hilo, bool quoted);

This function will output the buffer source to the fout stream as a binary string. It will also check to see whether a short string (token 142) or a long string (tokens 143 and 144) is necessary. For example,

```c
uint8 mybuf[5] = { 0, 3, 13, 100, 3 };
trans_bst(stdout, mybuf, 5, TRUE, TRUE);
```

will send the following bytes to stdout:

0x8E 0x05 0x00 0x03 0x13 0x64 0x03.

6 Conclusions

- Cached user paths combined with binary token encodings can offer significant performance improvements and reduced PostScript language code sizes. Graphics objects can be rendered up to 4× faster and the resultant files can be up to 65% smaller using the techniques detailed in this paper.

- Certain types of applications can benefit by using user paths and BTE:

Applications that are object-oriented and expect many identical translated paths in their output are good candidates for user path support.

Applications that already run on a Display PostScript system and use user paths can take advantage of the user path mechanism on a PostScript Level 2 device with a minimum of effort.

- User paths are not trivial to support. New data structures and routines may need to be added to the application, and the bounding box of replicated paths must be calculated. It is suggested that before an application engineer spends a great deal of effort upgrading an application to support user paths, prototype tests be constructed to ensure that the improvements in performance offset the necessary development time.
Appendix: Changes Since Earlier Versions

Changes since August 8, 1991 version

• Document was reformatted in the new document layout and minor editorial changes were made.

Changes since May 4, 1991 version

• Section 3.3, “Modifying Cache Limits,” was added. It talks about how to determine the current size of the user path cache and how to set individual item limits.

• Section 4.1, “Performance Tests,” was updated to reflect a newer version of the PostScript Level 2 interpreter. A sub-section that lists the performance conclusions was also added.

• Section 6, “Conclusions,” was added to draw together many of conclusions reached in various parts of the paper.
## Index

### A
- **arc** 21
- **arcn** 21
- **arct** 22

### B
- binary object sequence 8
- binary token encoding
  - encoding numbers and operators 22
- **blimit** 12
- **bmax** 12
- BOS. *See* binary object sequence
- **bsize** 12, 23
- BTE. *See* binary token encodings
- **bte.c** 22
- **buf** 23

### C
- **Check_Angles** 21
- **Check_BBox** 20
- **cleartomark** 12
- **counttomark** 12
- **curveto** 20

### D
- document structuring conventions
  - comments 9
- %%%DocumentData: Binary 9

### F
- files
  - **bte.c** 22
  - **generate.c** 20
  - **limits.h** 20
- **mapname.c** 23
- **trans.c** 24
- **format (flag)** 23
- **fout (stream)** 24

### G
- **generate.c** 20

### H
- **hilo** 23
- homogeneous number array 23

### L
- %%%LanguageLevel: 9
- **limits.h** 20
- literal name 23

### M
- **MapName** 23
- **mapname.c** 23
- **MAX_INT** 20
- **MaxUPathCache** 12
- **MaxUPathItem** 12
- **MIN_INT** 20

### N
- **num** 23

### P
- **paths**
  - advanced construction methods 5–25
  - ASCII encoded user path 15
  - ASCII regular path 13