Barcode Writer in Pure Postscript

Terry Burton <tez@terryburton.co.uk>

September 9, 2005

Abstract

This document describes the implementation of the Barcode Writer in Pure Postscript project, explains by example how to use this to generate your own barcodes, and provides a simple reference to using the symbologies that it supports.

Contents

1 Introduction

Often there is a need to implement routines in several different languages that output Adobe Postscript for the purpose of printing barcodes. This project implements the printing of barcodes entirely within level 2 PostScript. This means that the entire process of converting the input string into the printed output is performed by the printer or print system, thus avoiding the need to later reimplement the barcode generation process when your development language changes.

To make it as easy as possible to incorporate this project into your own systems, whether they be freely available or proprietary, it is licensed under the permissive MIT/X-Consortium License given in section 5.

The project homepage is at http://www.terryburton.co.uk/barcodewriter.

This is the main resource for the project providing the latest downloads of code and documentation, as well as access to the support and development mailing list.

Acknowledgements

The author wishes to take this opportunity to thank the small, growing community of developers that have helped to develop, test and document this project with their suggestions and code. Most especially Michael Landers and Ross McFarland for freely giving their encoder implementations, and Lawrence Horwitz for his suggestions and testing.

2 Code Commentary

This commentary assumes familiarity with the PostScript language[1].

The code is split cleanly into two types of procedure:

The encoders Each of these represents a barcode symbology[2], e.g. EAN-13 or Code-128. It takes a string containing the barcode data and a string containing a list of options that modify the output of the encoder. It generates a structured representation of the barcode and its text for the symbology, including the calculation of check digits where necessary.

The renderer This takes the output of an encoder and generates a visual representation of the barcode.

This means that all barcodes can be generated simply in a similar manner:

```
(78858101497) (includetext height=0.6) upca barcode
(0123456789) (includecheck) interleaved2of5 barcode
```

2.1 The Barcode Data Structure

The following table describes the structured representation of a barcode that is passed by an encoder to the renderer as a dictionary when the PostScript is executed.

Element	Key	Value
Space bar succession	`sbs`	String containing the integer widths, in points, of each bar and space, starting with the leftmost bar.
Bar height succession	`bhs`	Array containing the height of each bar in inches, starting with the leftmost bar.
Bar base succession	`bbs`	Array containing the offset of the base of each bar in inches, starting with the leftmost bar.
Human readable text	`txt`	Array of arrays that contain the character, position, height, font and scale factor (font size), in points, for each of the visible text characters.
Renderer options	`opt`	String containing the user-defined renderer options.

[1] The PostScript Language Tutorial and Cookbook (a.k.a. the Blue Book), which is freely available online, serves as both a useful tutorial and reference manual to the language.

[2] By symbology we mean an accepted standard for representation of data as a barcode

2.2 An Encoder

The procedure labelled code2of5 is a simple example of an encoder, which we will now consider. Its purpose is to accept as input a string containing the barcode contents and a string containing a list of options, and to process these in a way that is specific to this encoder, and finally to output an instance of the dictionary-based data structure described in section 2.1 that represents the barcode contents in the Code 2 of 5 symbology.

As with all of the encoders, the input string is assumed to be valid for the corresponding symbology, otherwise the behaviour is undefined.

The variables that we use in this procedure are confined to local scope by declaring the procedure as follows:

```
/code2of5 {

     0 begin

...

     end

} bind def
/code2of5 load 0 1 dict put
```

We start by immediately reading the contents strings that are passed as arguments to this procedure by the user. We duplicate the options string because it is later passed unamended to the renderer.

```
     /options exch def
     /renderopts options def
     /barcode exch def
```

We initialise a few default variables. Those variables corresponding to options that can be enabled with the options argument are initially set to false.

```
     /includetext false def
     /textfont /Courier def
     /textsize 10 def
     /textpos -7 def
     /height 1 def
```

The options string is tokenised with each successive token defining either a name value pair which we instantiate or a lone variable that we define as true, allowing us to override the given default variables given above.

```
options {
    token false eq {exit} if dup length string cvs (=) search
    true eq {cvlit exch pop exch def} {cvlit true def} ifelse
} loop
```

Since any user given options create variables that are strings we need to convert them back to their intended types.

```
/textfont textfont cvlit def
/textsize textsize cvr def
/textpos textpos cvr def
/height height cvr def
```

We then create an array of string encodings for each of the available characters which we then declare in another string. This information can be derived from careful reading of the relevant specification, although this is often surprisingly difficult to obtain.

```
/encs
[ (1111313111) (3111111131) (1131111131) (3131111111)
  (1111311131) (3111311111) (1131311111) (1111113131)
  (3111113111) (1131113111) (313111) (311131)
] def

/barchars (0123456789) def
```

We now store the length of the content string and calculate the total number of bars and spaces in the resulting barcode. We initialise a string of size dependant on this length into which we will build the space bar succession. Similarly, we create an array into which we will add the human readable text information.

```
/barlen barcode length def
/sbs barlen 10 mul 12 add string def
/txt barlen array def
```

We now begin to populate the space bar succession by adding the encoding of the start character to the beginning.

```
sbs 0 encs 10 get putinterval
```

We now enter the main loop which iterates over the content string from start to finish, looking up the encoding for each character, adding this to the space bar succession.

It is important to understand how the encoding for a given character is derived. Firstly, given a character, we find its position in the string of all available characters. We then use this position to index the array of character encodings to obtain the encoding for the given character, which is added to the space/bar succession. Likewise, the character is added to the array of human readable text along with positioning and font information.

```
0 1 barlen 1 sub {
    /i exch def
    barcode i 1 getinterval barchars exch search
    pop
    length /indx exch def
    pop pop
    /enc encs indx get def
    sbs i 10 mul 6 add enc putinterval
    txt i [barcode i 1 getinterval i 14 mul 10 add -7
           textfont textsize] put
} for
```

The encoding for the end character is obtained and added to the end of the space bar succession.

```
sbs barlen 10 mul 6 add encs 11 get putinterval
```

Finally we prepare to push a dictionary containing the space bar succession (and any additional information defined in section 2.1) that will be passed to the renderer.

```
/retval 1 dict def
retval (sbs) sbs put
retval (bhs) [sbs length 1 add 2 idiv {height} repeat] put
retval (bbs) [sbs length 1 add 2 idiv {0} repeat] put
includetext {
    retval (txt) txt put
} if
retval (opt) renderopts put
retval
```

2.3 The Renderer

The procedure labelled barcode is known as the renderer, which we now consider. Its purpose is to accept as input an instance of the dictionary-based data structure described in section 2.1 that represents a barcode in some arbitrary symbology and produce a visual rendering of this at the current point.

The variables that we use in this procedure are confined to local scope by declaring the procedure as follows:

```
/barcode {

     0 begin

...

     end

} bind def
/barcode load 0 1 dict put
```

We then immediately read the dictionary-based data structure which is passed as a single argument to this procedure by an encoder, from which we extract the space bar succession, bar height succession and bar base succession.

```
    /args exch def
    /sbs args (sbs) get def
    /bhs args (bhs) get def
    /bbs args (bbs) get def
    /renderopts args (opt) get def
```

We attempt to extract from the dictionary the array containing the information about human readable text. However, this may not exist in the dictionary in which case we create a default empty array.

```
    args (txt) known {
        /txt args (txt) get def
    } {
        /txt [] def
    } ifelse
```

Just as with the encoders, we read and tokenise the supplied options allowing specific rendering options to be overridden.

```
    /inkspread 0.15 def
    renderopts {
        token false eq {exit} if dup length string cvs (=) search
        true eq {cvlit exch pop exch def} {cvlit true def} ifelse
    } loop
    /inkspread inkspread cvr def
```

We have extracted or derived all of the necessary information from the input, and now use the space bar succession, bar height succession and bar base succession in calculations that create a single array containing elements that give coordinates for each of the bars in the barcode.

We start by creating a bars array that is half the length of the space bar succession. We build this by repeatedly adding array elements that contain the height, x-coordinate, y-coordinate and width of single bars. The height and y-coordinates are read from the bar height succession and the bar base succession, respectively, whilst the x-coordinate and the width are made from a calculation of the total indent, based on the space bar succession and a compensating factor that accounts for ink spread.

```
/bars sbs length 1 add 2 idiv array def
/x 0.00 def
0 1 sbs length 1 sub {
    /i exch def
    /d sbs i get 48 sub def
    i 2 mod 0 eq {
        /h bhs i 2 idiv get 72 mul def
        /c d 2 div x add def
        /y bbs i 2 idiv get 72 mul def
        /w d inkspread sub def
        bars i 2 idiv [h c y w] put
    } if
    /x x d add def
} for
```

Finally, we perform the actual rendering in two phases. Firstly we use the contents of the bars array that we just built to render each of the bars, and secondly we use the contents of the text array extracted from the input argument to render the text. We make an efficiency saving here by not performing loading and rescaling of a font if the scale factor for the font size is 0. The graphics state is preserved across calls to this procedure to prevent unexpected interference with the users environment.

```
gsave

bars {
    {} forall
    setlinewidth moveto 0 exch rlineto stroke
} forall

txt {
    {} forall
    dup 0 ne {exch findfont exch scalefont setfont}
    {pop pop}
    ifelse
    moveto show
} forall

grestore
```

2.4 Notes Regarding Coding Style

PostScript programming veterans are encouraged to remember that the majority of people who read the code are likely to have little, if any, prior knowledge of the language.

To encourage development, the code has been written with these goals in mind:

- That it be easy to use and to comprehend
- That it be easy to modify and enhance

To this end the following points should be observed for all new code submissions:

- New encoders should be based on the code of a similar existing encoder
- Include comments where these clarify the operations involved, particular where something unexpected happens
- Prefer simplicity to efficency and clarity to obfuscation, except where this will be a problem

3 Resources and Examples

There are several ways of using the PostScript within your own projects.

Many example uses of the code for various languages and platforms can be downloaded from the code repository at http://www.terryburton.co.uk/barcodewriter/files/repository/

3.1 Language Specific APIs

No language specific APIs exist yet. If you have experience writing API specifications and would like to help create an API design for the project then contact the author.

3.2 Front Ends

The following is a list of the front ends available for the project.

Web based demo http://www.raise-the-bar.co.uk/demo

pst-barcode pst-barcode is a PSTricks package for LaTeX.
http://www.ctan.org/tex-archive/graphics/pstricks/contrib/pst-barcode/

3.3 Installing the Barcode Generation Capability into a Printer's Virtual Machine

Most genuine PostScript printers allow procedures to be defined such that they persist across different jobs through the use of the `exitserver` command. If your printer supports this then you will be able to print the main code containing the definitons of all the encoders and the renderer once, e.g. soon after the device is turned on, and later omit these definitons from each of the barcode documents that you print.

To install the barcode generation capabilities into the virtual machine of a PostScript printer you need to uncomment a line near the top of the code so that it reads:

```
serverdict begin 0 exitserver
```

Once this code is printed the procedural definitions for the encoders and the renderer will remain defined across all jobs until the device is reset either by power-cycling or with the following code:

```
serverdict begin 0 exitserver systemdict /quit get exec
```

3.4 Hints for Generating Precisely the Required Symbol

To create a barcode to a required width and height, without stretching the human readable text, perform the following steps.

Create a basic symbol by choosing the relevant data and text options for the corresponding encoder, and position this using `translate` such that the *bottom-left* corner of the bars is in the required location:

```
gsave
430 750 translate
(977147396801) (includetext) ean13 barcode
grestore
```

Find the uniform scale factor (same value for x and y) that makes your output of the required *width*:

```
gsave
430 750 translate
1.3 1.3 scale      %  <-- Add a line like this
(977147396801) (includetext) ean13 barcode
grestore
```

Add a height option that adjusts the bar *height* appropriately (taking the scaling into account):

```
gsave
430 750 translate
1.3 1.3 scale
% Added height=0.8 option to adjust height
(977147396801) (includetext height=0.8) ean13 barcode
grestore
```

The result should now be of the intended dimensions at the desired location with properly scaled text. You can now add any additional options to customise the symbol.

3.5 Printing in Perl

This example will print a page of EAN-13s ranging between two given values when called from a shell like this:

```
$ ./ean13s.pl 978186074271 978186074292 | lpr
```

The contents of the script `ean13s.pl` is as follows:

```
#!/usr/bin/perl -w
use strict;

die 'Requires two arguments' if (@ARGV!=2);

open(PS,'barcode.ps') || die 'File not found';
$_=join('',<PS>);
close(PS);

print "%!PS-Adobe-2.0\n";

m/
    %\ --BEGIN\ TEMPLATE--
    (.*)
    %\ --END\ TEMPLATE--
    /sx || die 'Unable to parse out the template';
print $1;

for (my $i=$ARGV[0], my $j=0; $i<$ARGV[1]; $i++, $j++) {
    my $x=100+150*(int($j/7));
    my $y=100+100*($j%7);
    print "gsave\n";
    print "$x $y translate\n";
    print "($i) (includetext) ean13 barcode\n";
    print "grestore\n";
}

print "showpage\n";
```

4 Supported Symbologies

The following section shows the symbologies that are supported by the encoders, including the available features for each. This list may not be up-to-date. If it does not contain any of the formats or features that you require then check the project source code or try the support mailing list.

4.1 EAN-13

Data 12 or 13 digits

Options

Option	Feature
`includetext`	Enable human readable text

Notes If just 12 digits are entered then the check digit is calculated automatically

Figure 1: `(9781860742712) (includetext guardwhitespace) ean13 barcode`

4.2 EAN-8

Data 8 digits

Options

Option	Feature
`includetext`	Enable human readable text

Figure 2: `(12345678) (includetext guardwhitespace height=0.6) ean8 barcode`

4.3 UPC-A

Data 11 or 12 digits

Options

Option	Feature
`includetext`	Enable human readable text

Notes If just 11 digits are entered then the check digit is calculated automatically

Figure 3: `(78858101497) (includetext) upca barcode`

4.4 UPC-E

Data 7 or 8 digits

Options

Option	Feature
`includetext`	Enable human readable text

Notes If just 7 digits are entered then the check digit is calculated automatically

Figure 4: `(0123456) (includetext) upce barcode`

4.5 EAN-5

Data 5 digits

Options

Option	Feature
`includetext`	Enable human readable text

Figure 5: `(90200) (includetext guardwhitespace) ean5 barcode`

4.6 EAN-2

Data 2 digits

Options

Option	Feature
`includetext`	Enable human readable text

Figure 6: `(38) (includetext guardwhitespace) ean2 barcode`

4.7 ISBN

Data 9 or 10 digits seperated appropriately with dashes

Options

Option	Feature
`includetext`	Enable human readable text

Notes If just 9 digits are entered then the human readable ISBN check digit is calculated automatically

Figure 7: `(1-58880-149) (includetext) isbn barcode`

4.8 Code-39

Data Variable number of characters, digits and any of the symbols `-. *$/+%`.

Options

Option	Feature
`includecheck`	Enable check digit
`includetext`	Enable human readable text
`includecheckintext`	Make check digit visible in text

Figure 8: `(CODE-39) (includecheck includetext) code39 barcode`

4.9 Code-128 and UCC/EAN-128

Data Variable number of ASCII characters and special funtion symbols, starting with the approriate start character for the initial character set. UCC/EAN-128s must have a manditory `FNC 1` symbol immediately following the start character.

Options

Option	Feature
`includetext`	Enable human readable text
`includecheckintext`	Make check digit visible in text

Notes Any non-printable character can be entered via its escaped ordinal value, for example `^070` for `ACK` and `^102` for `FNC 1`. Since a caret symbol serves as an escape character it must be escaped as `^062` if used in the data. The check character is always added automatically.

C o u n t   1234   !

Figure 9: `(^104^102Count^0991234^101!) (includetext) code128 barcode`

4.10 Rationalized Codabar

Data Variable number of digits and any of the symbols `-$:/.+ABCD`.

Options

Option	Feature
`includecheck`	Enable check digit
`includetext`	Enable human readable text
`includecheckintext`	Make check digit visible in text

0 1 2 3 4 5 6 7 8 9

Figure 10: `(0123456789) (includetext) rationalizedCodabar barcode`

4.11 Interleaved 2 of 5 and ITF-14

Data Variable number of digits. An ITF-14 is 14 characters and does not have a check digit.

Options

Option	Feature
`includecheck`	Enable check digit
`includetext`	Enable human readable text
`includecheckintext`	Make check digit visible in text

Notes The data may be automatically prefixed with 0 to make the data, including optional check digit, of even length.

Figure 11: `(05012345678900) (includecheck height=0.7) interleaved2of5 barcode`

4.12 Code 2 of 5

Data Variable number of digits

Options

Option	Feature
`includetext`	Enable human readable text

0 1 2 3 4 5 6 7 8 9

Figure 12: `(0123456789) (includetext textpos=75 textfont=Helvetica textsize=16) code2of5 barcode`

4.13 Postnet

Data Variable number digits

	Option	Feature
Options	`includetext`	Enable human readable text
	`includecheckintext`	Make the check digit visible in the text

Notes Check digit is always added automatically

0 1 2 3 4 5 6 7

Figure 13: `(01234567) (includetext textpos=-10 textfont=Arial textsize=10) postnet barcode`

4.14 Royal Mail

Data Variable number digits and capital letters

	Option	Feature
Options	`includetext`	Enable human readable text
	`includecheckintext`	Make the check digit visible in the text

Notes Check digit is always added automatically

L E 2 8 H S 9 Z

Figure 14: `(LE28HS9Z) (includetext) royalmail barcode`

4.15 MSI

Data Variable number digits

Options

Option	Feature
`includecheck`	Enable check digit
`includetext`	Enable human readable text
`includecheckintext`	Make check digit visible in the text

Figure 15: `(0123456789) (includecheck includetext) msi barcode`

4.16 Plessey

Data Variable number of hexadecimal characters

Options

Option	Feature
`includetext`	Enable human readable text
`includecheckintext`	Make the check digits visible in the text

Notes Check digits are always added automatically.

Figure 16: `(012345ABCDEF) (includetext) plessey barcode`

5 License