AN-04 (LP)
September 23, 2005

HOW TO PRINT BITMAP GRAPHICS IN LINE PRINTER MODE

This document describes how to print bitmapped graphics in Line Printer mode. Two other graphics mode are available. One is RLE compressed graphics (see AN-05) and downloaded .PCX graphics printed from Easy Print (see AN-09).

Using the ESC V n1n2 <data> to send graphics to be printed one time (and not stored) is really extremely simple, and is virtually identical to the way most graphic commands have worked on many different printers for years. It is used to print a bitmap graphic on the fly, where the image is to be printed once and then discarded rather than to be stored and printed again and again. The ESC V n1n2 precedes the actual bitmap data that forms the picture. The ESC V tells the printer to enter a graphics mode, the n1n2 tells the printer how tall the graphics image is.

A bit map graphic is an image composed of individual bits in bytes, each bit representing a pixel (or picture element) in the image, and the bytes laid end to end from left to right, then stacked, making up the image a series of very thin strips going across the image from left to right and from top to bottom.

For example, if you look at the image below, you can see a diamond with a rectangle inside (you might need a little imagination), composed of individual pixels each the size of one letter. We'll let a space represent a white pixel, and an X represent a black pixel:

```
    XXXX
  XXXXXXXX
 XX      XX
 XX  XXXX  XX
 XX  XXXX  XX
XX   XXXX   XX
 XX  XXXX  XX
 XX  XXXX  XX
  XX      XX
  XXXXXXXX
    XXXX
```

To form this image digitally, we would make the black pixels (represented by X's) a ONE bit and the white areas a ZERO bit:

```
000000000011110000000000
000000001111111100000000
000000011000000110000000
000000110011110011000000
000001100001111000110000
000001101011110001100000
000000110011110011000000
000000011000000110000000
000000001111111100000000
000000000011110000000000
```

Now we can separate these out into bytes with each byte representing 8 bits, or pixels. This would be a true binary representation of the bitmap of our image.

```
00000000 00111100 00000000
00000000 11111111 00000000
00000001 10000001 10000000
00000011 00111100 11000000
00000110 00111100 01100000
00001100 00111100 00110000
00000110 00111100 01100000
00000011 00111100 11000000
00000001 10000001 10000000
00000000 11111111 00000000
00000000 00111100 00000000
```

Since most programmers do not work in Binary, we'll convert the binary image into HEX;

```
0x00 0x3C 0x00
0x00 0xFF 0x00
0x01 0x81 0x80
0x03 0x3C 0xC0
0x06 0x3C 0x60
0x0C 0x3C 0x30
0x06 0x3C 0x60
0x01 0x81 0x80
0x00 0xFF 0x00
0x00 0x3C 0x00
```

Now we are ready to print this image, which is 24 pixels wide and 10 pixels high. We'll print it on an imaginary printer whose printhead is only 24 dots wide (3 bytes). We can use pixel and dot interchangeably here, since each pixel in our image is represented by one dot on the printhead. We would send the following to the printer which represents the ESC V n1n2 command as well as the data for the image:

0x1B 0x56 0x00 0x0A 0x00 0x3C 0x00 0x00 0xFF 0x00 0x01 0x81 0x80 0x03 0x3C 0xC0 0x06 0x3C 0x60 0x0C 0x3C 0x30 0x06 0x3C 0x60 0x01 0x81 0x80 0x00 0xFF 0x00 0x00 0x3C 0x00

The 0x1B is the ESC character, the 0x56 is the 'V', the 0x00 is the MS Byte of our 16 bit count of 10 (since the image is 10 pixels high), and the 0x0A is the LS Byte of our 16 bit count of 10 (since 10 decimal is 0A in Hex). The rest of the bytes are the data from the image, arranged left to right and top to bottom.

A common question that arises is "What do I do if my image is not as wide as the printhead?". The answer is to pad white space around the image so that it will print properly. To use our previous example using the rectangle in the diamond, let us assume that we want to print this image in the center of another imaginary printhead that is 40 dots wide. We would pad white space in front of and behind the image – 8 pixels in front and 8 pixels behind will give use 8 pixels + 24 pixels + 8 pixels or 40 pixels total:

```
0x00 0x00 0x3C 0x00 0x00
0x00 0x00 0xFF 0x00 0x00
0x00 0x01 0x81 0x80 0x00
0x00 0x03 0x3C 0xC0 0x00
0x00 0x06 0x3C 0x60 0x00
0x00 0x0C 0x3C 0x30 0x00
0x00 0x06 0x3C 0x60 0x00
0x00 0x01 0x81 0x80 0x00
0x00 0x00 0xFF 0x00 0x00
0x00 0x00 0x3C 0x00 0x00
```

In this case, the data that would be sent to the printer which represents the ESC V n1n2 command as well as the data for the image is:

0x1B 0x56 0x00 0x0A 0x00 0x00 0x3C 0x00 0x00 0x00 0x00 0xFF 0x00 0x00 0x00 0x01 0x81 0x80 0x00 0x00 0x03 0x3C 0xC0 0x00 0x00 0x06 0x3C 0x60 0x00 0x00 0x0C 0x3C 0x30 0x00 0x00 0x06 0x3C 0x60 0x00 0x00 0x01 0x81 0x80 0x00 0x00 0x00 0xFF 0x00 0x00 0x00 0x00 0x3C 0x00 0x00

Again, the 0x1B is the ESC character, the 0x56 is the 'V'. The dotline count does not change even though the width and the number of image data bytes did change – it is still 10 as represented by the 0x00 and the 0x0A. All bytes after this are the image data again, but this time there are more bytes.

A bitmap image as we are referring to it should not be confused with a .BMP file from Microsoft, since that file has a header and is inverted. Any "standard" graphics image format file used on PCs and the Internet will need to be converted to its pure, raw, bitmap form before sending down to be printed. Information about these "standard" graphics file formats can be found on the web.

There are a few rules when printing graphics, if you follow these rules and form the data for your image according to the examples above, you should have no trouble printing graphics on the printer:

1. The graphic MUST be the same width as the printer - 384 pixels for 2" thermal, 576 pixels for the 3" thermal, 832 pixels for the 4" thermal, and 240 pixels for the 2" impact. If it is not the same width, then you must pad with white space (0x00 bytes) on each line.

2. Each pixel is represented as one dot on the printed image, therefore only black and white (one bit per pixel) is allowed
3. Bytes sent to the printer as data are printed left to right, top to bottom.
4. Bits within a byte are printed as though the byte is laying on its side, with the Most Significant Bit to the left
5. Therefore, the MSBit of the first data byte sent is in the uppermost and leftmost corner of the printed image, with additional bits in that byte printed horizontally alongside the first bit
6. The next data byte is printed in the same manner horizontally alongside the previous byte
7. This is repeated until we receive data bytes for the entire width of the printer (384 pixels -> 48 bytes, 576 pixels -> 72 bytes, 832 pixels -> 104 bytes, and 240 pixels -> 30 bytes). For example, for the 2" printer, the first 48 bytes of data will form the first row of pixels or dots in the printed image, which is approximately 2" wide but only 1/200 inch high (since the printer is approximately 200 dpi).
8. The next bytes are printed in the same manner across the row immediately below the previous row, from left to right
9. The n1n2 count is the 16 bit count of the number of ROWS of pixels that make up the height of the image, and NOT the number of data bytes sent. Again, for example, if an image is 1 inch high, it is approximately 200 pixels high, and therefore n1n2 would be 200.
10. The n1n2 count is sent so that n1 is the most significant byte of the count, and n2 is the least significant byte of the count. Since, in our example of 200 dotlines, the number 200 will fit in a single byte, n1 would be zero and n2 would be 200 (or 0xC8).
11. If the n1n2 count is wrong, the printer will act improperly. Remember that n1n2 is the count of the number of dotlines, not the number of bytes, So in our example of a 1" high image on the 2" printer, the data would be a total of 200 dotlines/inch X 48 bytes/dotline or 9,600 bytes. But n1n2 would be only 200 for the number of dotlines and NOT 9,600 which is the count of the total number of bytes..
    a. If n1n2 is too small for the data being sent, then part of the image will be printed properly, but the printer will try to interpret the rest of the image data as printable ASCII characters resulting in the spewing out of much extra paper with garbage printed occasionally.
    b. If n1n2 is too large for the data sent, the printer will appear to do nothing as it waits for more data, or "eats" printable data that should have been printed after the image printed. Or if the image is large, it may print part of the image, then nothing, again 'eating" printable data after the image.