The EPL5700L printer driver code

Hin-Tak Leung

1st February,2003

Contents

1 Introduction 1
1.1 How can one tell if one's printer may be compatible? . . . 1
1.2 How did I get these details? . . . 2

2 The structure of a EPL-5700L print job 2
2.1 Job Header . . . 2
2.2 Page Header . . . 4
2.2.1 Custom Paper Size . . . 4
2.3 Page Footer . . . 4
2.4 Job Footer . . . 5
2.5 Stripe Header . . . 5

3 The Stripe compression algorithm 6
3.1 An example . . . 6
3.2 Other snipplets of information . . . 7
3.3 The Run Length Encoding . . . 8

4 Patents and Intellectual Property Right Issues 8
4.1 The MS Windows driver . . . 10

5 Misc 10
5.1 On-line Resources . . . 10
5.2 The USB interface . . . 11

1 Introduction

This document details everything I learn of how to drive the Epson printer EPL 5700L in the last two years since I bought one. The information may apply to some of the lower-end Epson laser printers; in particular, the EPL 5800L, 5900L sold in Europe, There is also a new model 6100L sold in Europe.

It has been confirmed that the 5700L 5800L, and 5900L shares the same core band/stripe compression algorithm, but have rather different job header, page header stripe header and page footers.

There is a lot of hex codes and a lot of maths in this document. If you don't like it, turn away now…

This document was last updated on February 1, 2003.

No guarantee to the correctness to any part of this document. Use at your own risk.

TODO: Many parts are getting terribly out-dated, wrong, etc. Re-write needed.

1.1 How can one tell if one's printer may be compatible?

I have a EPL-5700L from UK, It has been confirmed that the 5800L and 5900L are similar enough. Some newer models, such as 6100L, might be in this case, or some other Epson printers.

TODO: details of difference between 5800L, 5900L

Note that some “EPL-5x00”, “EPL-5x00N”, “EPL-5x00-PS” understands ESC/P (Epson’s printer language), PCL (HP’s printer language), or Postscript (Adobe’s printer language), and you are better off using ECS/P, PCL, or Postscript to drive these printers.

The compression algorithm shared by the 5700L, 5800L, 5900L generates these 26 bytes for an entirely white area (e.g. top of a blank page), which can be regarded as a signature for which this driver applies:

```
a0 1d 74 03 0e 80 01 d0
40 3a e8 07 1d 00 03 a0
80 74 d0 0e 3a 01 07 40
00 e8
```

It is almost a hundred byte into the spool file, for the 5900L (which has the longest header).

Note that if you have Win2k instead of Win98se, you need to disable “enhanced printer support” or something like that in the printer driver control panel to see the actual spool file. Otherwise, Win2k seems to keep the pages as WMF (windows metafiles) until the last minute before conversion to something that the printer understands. (“Enhanced - my ass”). Oh, if one pause printing, the spool files are kept as “sp00001.spl”, etc (the number is reset to 1 every time a windows box reboots) in the system spool directory under win98 or win2k. Just pause the printer (or disconnect the parallel cable), print and search for “*.spl”.

1.2 How did I get these details?

Ghostsript on MS Windows can use the GDI sub system of the host and print to any printer that the OS itself knows about.

Win32 Ghostscript also has a print script which is drivable through batch scripts to print-spool thousands of postscript files.

So what I can do, is to draw some simple lines and shapes with `xfig`, export as postscript, and print and collect the spool files and examine them. For example, I have a little perl program which generates postscript files with one single horizontal line of increasing length at steps of 600th of an inch. So I have a few thousands of spool files to analyse.

It is a lot of guess work. The major vector-based driving languages are Postscript (Adobe), PCL (HP), ESC/Page (Epson), LIPS (Canon). Tried all of them and none of them works. The specification for Postcript, PCL and ESC/Page are very well-documented and publicly available - and I have them somewhere.

I don’t think Epson would choose to invent one more vector language (instead of re-using their own ESC/Page, or just use postscript or PCL), so the EPL-5700L has to take some sort of compressed bitmap, if it isn’t fitted with much intelligence. Apparently it can be fitted with a PCL chip and/or a postscript chip to make it understand PCL and/or postscript, and the EPL-5700 (without L) does understand PCL according to the spec, I think. So I tried printing simple pages, like a blank page with only a page number at the bottom, with one single horizontal line, etc and start from there. Over time, by looking at a lot of spool files of simple documents that I come up with, I gained a certain understanding of how it works.

2 The structure of a EPL-5700L print job

The structure of a typical print job is as shown in Figure 1. Please have a look at the figure quickly to get a mental picture.

The details of individual parts are described below.

2.1 Job Header

A total of 8 bytes (I include the off-set because I get confused sometimes myself as my tests are all written in C/Perl for which the first byte is byte 0). See table 2 for details.

There is no header field corresponding to “skip blank pages”, as expected.

The resolution-related byte 3 and 4 are used as in table 3 (oh yes, the maximum resolution of the printer is 1200dpi x 600dpi, which is called “1200dpi Class” in the printer control panel under MS Windows):

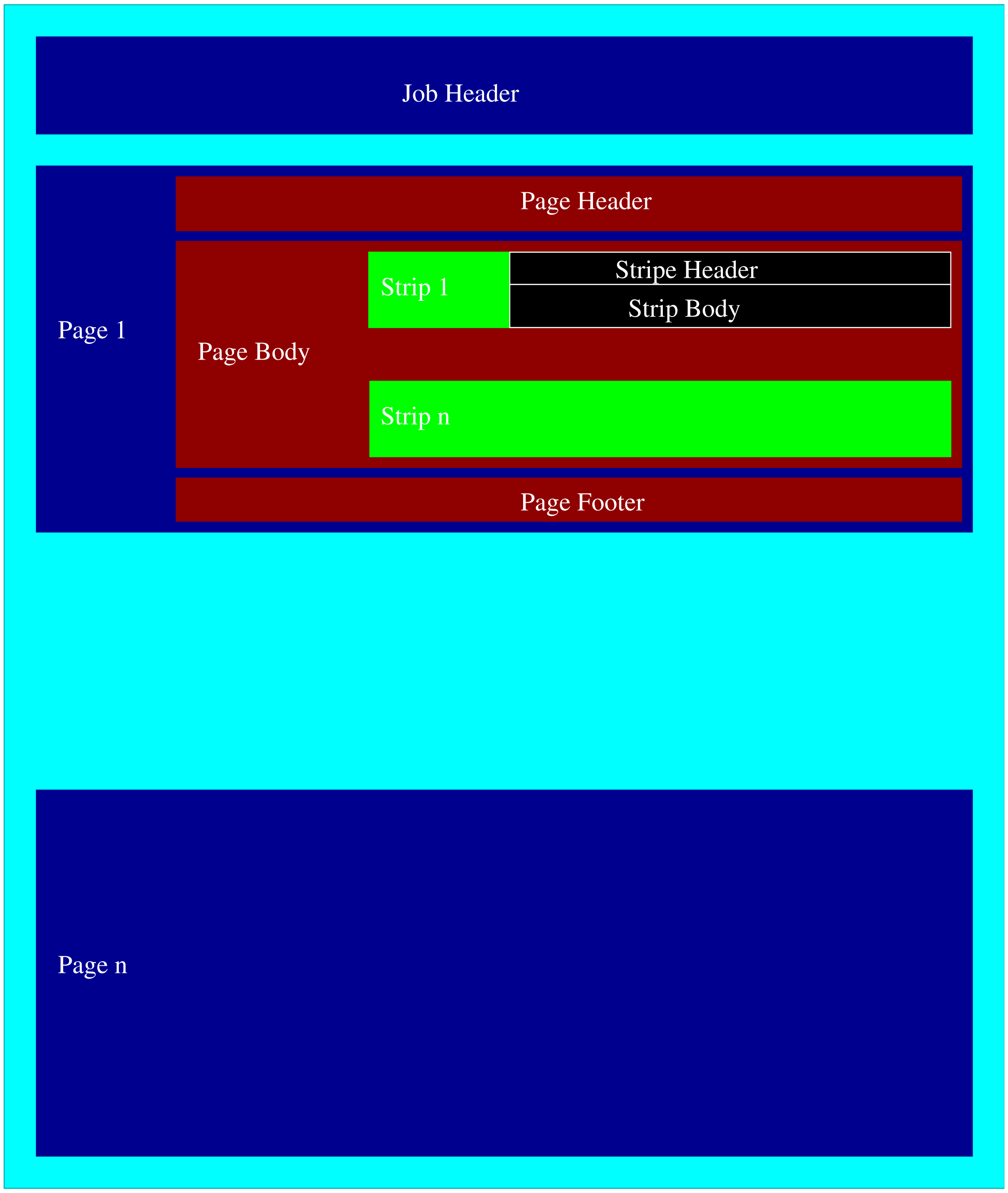

Figure 1: The structure of a typical print job

Part	size (Byte)
Job Header	8
Page Header	25
Stripe Header	7
Page Footer	2
Job Footer	2

Table 1: Sizes of the components of a print job

Byte	off set	description	values
1	0x00	unknown	0x00
2	0x01	unknown	0x00
3	0x02	resolution related	
4	0x03	resolution related	
5	0x04	RITech status: off = 0, on = 1 (default)	0,1
6	0x05	Toner save: off = 0 (default), on = 1	0,1
7	0x06	Paper type: Normal = 0 (default) Thick (N) = 2, Thick (W) = 1, transparency = 3	0,1,2,3
8	0x07	Density: density 1 = 1, density 3 = 3 (default), density 5 = 5,	1,2,3,4,5

Table 2: Job Header

2.2 Page Header

The Page Header is 23 bytes, and the Page Footer is 4 bytes.

A page is divided into horizontal stripes of width 64 pixels each. So for example, an A4 portrait page at 600dpi horizontal and 300dpi vertical contains 54 stripes.

The math is like this: 300x 11.7inches = 3508, and 3508 /64 = 54.8 stripes, and remember an area of about 0.5cm at the top and the bottom is not printable.

Similar consideration applies for the horizontal and vertical pixel counts. They seems to be just the measurement of the paper size concerned minus the unprintable side margin.

The page header details is in table 4.

Byte 1 indicates the paper type and seems to be resolution independent (i.e. the code for A4 paper is fixed, regardless of printing at 600x300 or 1200x600). It seems to follow some kind of convention, so it might have been taken from an ISO specification table for paper sizes or something.

The others are all resolution dependent. See table 5 for how the paper size and pixel count bytes are used.

The horizontal pixel code at 0x0c, 0x0d is pixel count divided by 8 and then rounded up to multiple of 4 of 0x14, 0x15 ; They seems to be simple pixel counts, most significant byte first.

2.2.1 Custom Paper Size

Here is Custom Paper size (4.32 inch x 6.78 inch) compared with A4 (8.26 inch x 11.69 inch):

```
0e 40 02 54 00 00   00 00 0d 50 12 98 00 36  ff 00 01 ff fe 00 00 00   00
ff 40 01 2c 00 00   00 00 07 8d 09 56 00 1f  ff 00 01 ff fe 00 6d 00   ac
```

That seems to be the custom paper size in mm (4.32 inch = 109mm, 6.78 inch = 172mm).

According to the windows GUI interface, custom paper size can only be within the parameters shown in Table 6. The minima are probably governed by physical distance between rollers, etc with the printer, while the max are by physical contraints of the paper tray and paper path.

2.3 Page Footer

See table 7.

Byte 3	Byte 4	Description
0x00	0x00	300 x 300 (300dpi)
0x00	0x01	600 x 300 (600dpi Class)
0x01	0x00	600 x 600 (600dpi)
0x01	0x01	1200 x 600 (1200dpi Class)

Table 3: Job Header resolution usage

Byte	Off set	Off-set from job beginning	Description	value
1	0x00	0x0a		0x02
2	0x01	0x0b		0x00
3	0x02	0x0a	paper size code	
4	0x03	0x0b	unknown	0x40
5,6	0x04, 0x05	0x0c, 0x0d,	horizontal pixel count (/8, rounded up *4)	
7,8,9,10		0x0e,0f, 10, 11	unknown	0x00
11,12		0x12, 13	vertical pixel count	
13,14		0x14,15	horizontal pixel count	
15		0x16	unknown	0x00
16		0x17	Stripe count per page	
17		0x18	Tray selection: MP = 0, auto = 0xff (default)	0x00, 0xff
18		0x19	unknown	0x00
19		0x1a	Number of copies: 1 (default)	0x01, etc
20		0x1b	unknown	ff
21		0x1c	Avoid Page Error: 0xfe = off (default), 0xff = on	0xfe,0xff
22-25		0x1d to 0x20	Only used for Custom paper size	00 00 00 00

Table 4: Page header details

TODO: I believe the printer only needs the paper size selection (byte 1), horizontal pixel count /8 *4, and the stripe count. I wonder what happens if the printer encounter inconsistent parameters. (i.e. when byte 3,4 don't agree with byte 11,12, or byte 9,10 don't agree with byte 14.

2.4 Job Footer

See table 7.

TODO: I believe the printer only needs the paper size selection (byte 1), horizontal pixel count /8 *4, and the stripe count. I wonder what happens if the printer encounter inconsistent parameters. (i.e. when byte 3,4 don't agree with byte 11,12, or byte 9,10 don't agree with byte 14.

2.5 Stripe Header

Note - I always write my 1's and 0's in decoding order i.e. LSB first - this is unusual.

The 7-byte Stripe Header is simply `04 00 01 00` followed by the stripe byte count, most singnificant byte first.

An empty stripe is 64 groups of `101110 0000000` [1], which is 104 bytes long; so a blank A4 page (the smallest A4 print job) at 600dpi x 300 dpi is 10 (job header) + 23 (page header) + 54 x (7 + 104) +4 = 6031 byte long. The stripe header for a blank stripe is `04 00 01 00 00 00 68` (0x68 = 104).

The worst case scenario, 1200dpi horizontal with random noises, contains 1200 x 9 inches x 64 pixels $\approx$ 700,000 per stripe, or would take about 90,000 bytes to encode literally; so 3 bytes for byte count should be enough.

[1] The line termination code - more about this in the compression algorithm section.

0a	0c 0d	12 13 14 15	17	**Paper selection, off-set from job beginning**
0e	01 2c	0d 50 09 4c	36	A4
0f	00 d0	09 4c 06 70	26	A5
19	01 04	0b 78 08 02	2e	B5
1e	01 34	0c 80 09 92	32	LT
1f	00 c4	09 92 06 0e	27	HLT
20	01 34	10 04 09 92	41	LGL
21	01 04	0b ea 08 1b	30	EXE
22	01 34	0e d8 09 92	3c	GLG
23	01 20	0b ea 08 fc	30	GLT
25	01 2c	0e d6 09 4c	3c	F4
50	00 88	08 66 04 26	22	MON
51	00 90	0a be 04 71	2b	C10
5a	00 98	09 c2 04 af	28	DL
5b	00 e4	0a 2c 07 15	29	C5
5c	00 9c	07 15 04 de	1d	C6
63	01 f0	0b 24 0f 74	2d	IB5
	01 2c	0d 50 09 4c	36	A4 300
	02 54	0d 50 12 98	36	A4 600c
	02 54	1a a0 12 98	6b	A4 600
	04 a8	1a a0 25 30	6b	A4 1200c
	01 34	0c 80 09 92	32	LT 300
	02 68	0c 80 13 24	32	LT 600c
	02 68	19 00 13 24	64	LT 600
	04 cc	19 00 26 48	64	LT 1200c

Table 5: Paper size and resolution (300x300 unless otherwise stated)

Dimension	min (cm)	max (cm)	min (inch)	max (inch)
Width	9.01	21.59	3.55	8.50
Height	14.80	35.56	5.83	13.99

Table 6: Custom paper size limits

3 The Stripe compression algorithm

The Stripe Compression Algorithm has now been more clearly understood - this section is being rewritten - for the time being I'll just remove wrong details...

The strip content is a 16-bit bit-stream with the most significant byte first, so to understand it properly, one has to read the stream like this: {byte 2 bit 1, bit 2 ,. . . , bit 8 }, {byte 1 bit 1, bit 2 ,. . . , bit 8 }, {byte 4 bit 1, bit 2 ,. . . , bit 8 }, {byte 3 bit 1, bit 2 ,. . . , bit 8 }, etc. It is padded at the end to 16-bit boundary so the stripe byte count is always even.

For sparse stripes (i.e. strips that don't have a lot of inks), it starts with `10`, then a run-length code (described in the next section) for how many byte to copy from the previous row, then `01` followed 8-bit literal, or `00` followed by a 4-bit code of unknown usage[2].

Need to add the copy-last, copy-last-2, copy-last-3 description here

3.1 An example

An example of a shape I drew with `xfig` and printed out is shown in figure 2. The corresponding bit code for the first non-trivial strip (3rd strip) is shown in table 9, for A4 at 300 x 300 dpi. When xfig exports to

[2]The 4 bit code still seems to perform the function of one 8-bit literal - see example.

value	Description
03 00	Page ends

Table 7: The Structure of the Page Footer

value	Description
01 00	Job ends

Table 8: The Structure of the Job Footer

pdf it centres the shape so it has moved somewhat sideways towards the centre of the paper.

It is the 3rd stripe, 2x 64 + 48 = 176 pixels, or just over half an inch from the top of the printable area of a page. The first non-trivial line is Copy 108 x 8 pixels (about 3 inches), 1 literal black byte, copy that black byte 55 times, ie. 59x8 pixels (an inch and half), and half a literal black byte after. Thereafter, it is just copying 165/164/163 bytes and literal 1 byte.

What I don't understand is how the *unknown 1 byte* works. One would expect literal 1 byte (`01 00000001`) instead of the unknwn (`00 1000`), for example, in the middle of table 9, for the first change from literal to unknown. This example switches from literal to unknown after 8 lines, but I have seen at least a few cases of switching after 5 lines; and it doesn't seems to be triggered by line positioning (i.e. switch only at multiple of 8 lines) either.

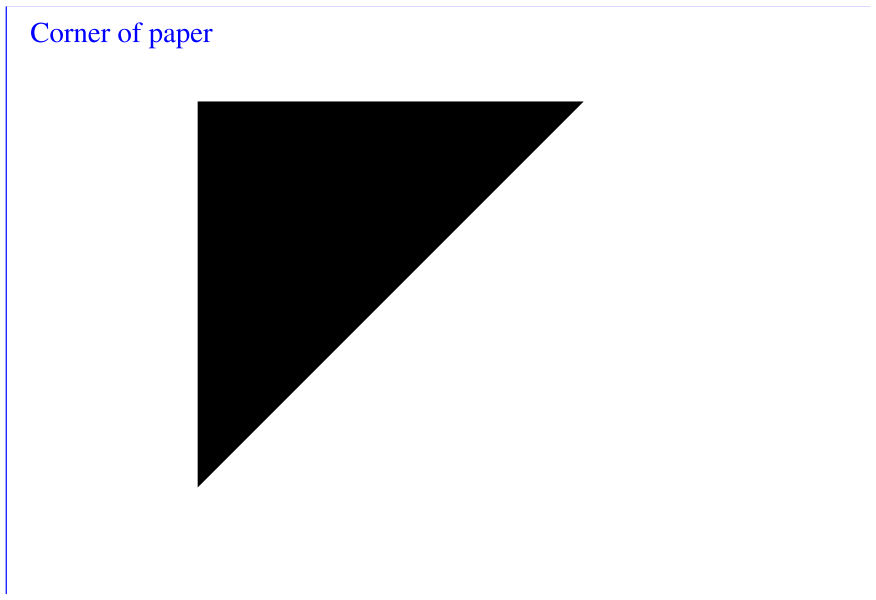

Figure 2: An example of a test figure

Needs to talk about the copy-left code ...

3.2 Other snipplets of information

I have a couple of small programs for bit-disassembling strips and strip-assembling from bits with the double-byte swapping and padding to 16-bit boundary included. So I can disassemble, modify, assemble, cat ¿ /dev/lp0 to see what it looks like.

bit code	**Description**
`10 1110 0000000` x 48	line end x 48
`10 1110 0011011 00 1110 01 11111111`	copy 108, unknown 1, literal 1
110 1110 1110110 01 00000001 10 1110 0000000	copy-left 55, literal 1, line end
`10 1110 1111111 0110010 01 00000000 10 1110 0000000`	copy 165, literal 1, line end
`10 1110 1111111 1010010 01 01111111 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 01 00111111 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 01 00011111 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 01 00001111 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 01 00000111 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 01 00000011 10 1110 0000000`	copy 164, literal 1, line end
`10 1110 1111111 1010010 00 1000 10 1110 0000000`	copy 164, unknown 1, line end
`10 1110 1111111 1010010 00 0100 10 1110 0000000`	copy 164, unknown 1, line end
`10 1110 1111111 0010010 00 1100 10 1110 0000000`	copy 163, unknown 1, line end
`10 1110 1111111 0010010 00 0010 10 1110 0000000`	copy 163, unknown 1, line end
`10 1110 1111111 0010010 00 1010 10 1110 0000000`	copy 163, unknown 1, line end
`10 1110 1111111 0010010 00 0110 10 1110 0000000`	copy 163, unknown 1, line end
`10 1110 1111111 0010010 00 1110 10 1110 0000000`	copy 163, unknown 1, line end
`10 1110 1111111 0010010 00 0001 10 1110 0000000`	copy 163, unknown 1, line end
`000000000`	padding

Table 9: Bit code for the top part of the triangular shape

3.3 The Run Length Encoding

Each of the strip content is some kind of run-length encoding of the difference to the previous row, for all 64 rows it contains. The run-length count (for how many byte to copy from the previous row, or how many black/inverted byte to follow) is encoded as follows:

count	**bit pattern**
1	0
2	10
3	1100
4	1101
5	11110
6	111110
7	111111
$8 \leq x < 128$	1110 ⟨*7-bit*⟩
$128 \leq x < 256$	1110 ⟨*7-bit*⟩ ⟨*7-bit*⟩
$256 \leq x < 384$	1110 ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩
$384 \leq x < 512$	1110 ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩
$512 \leq x < 640$	1110 ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩ ⟨*7-bit*⟩

Table 10: The Run-Length Code

Since the first 7-bit code can never be below 7, a 7-bit code of zero is used for line termination. i.e. the bit code `10 1110 0000000` seems to be a line termination code.

4 Patents and Intellectual Property Right Issues

I have to say in advance that, I have every intention of honouring the IPR of Epson; I know it takes years of work of many talented individuals to do the work they do. I would be happy if they provide a close-source

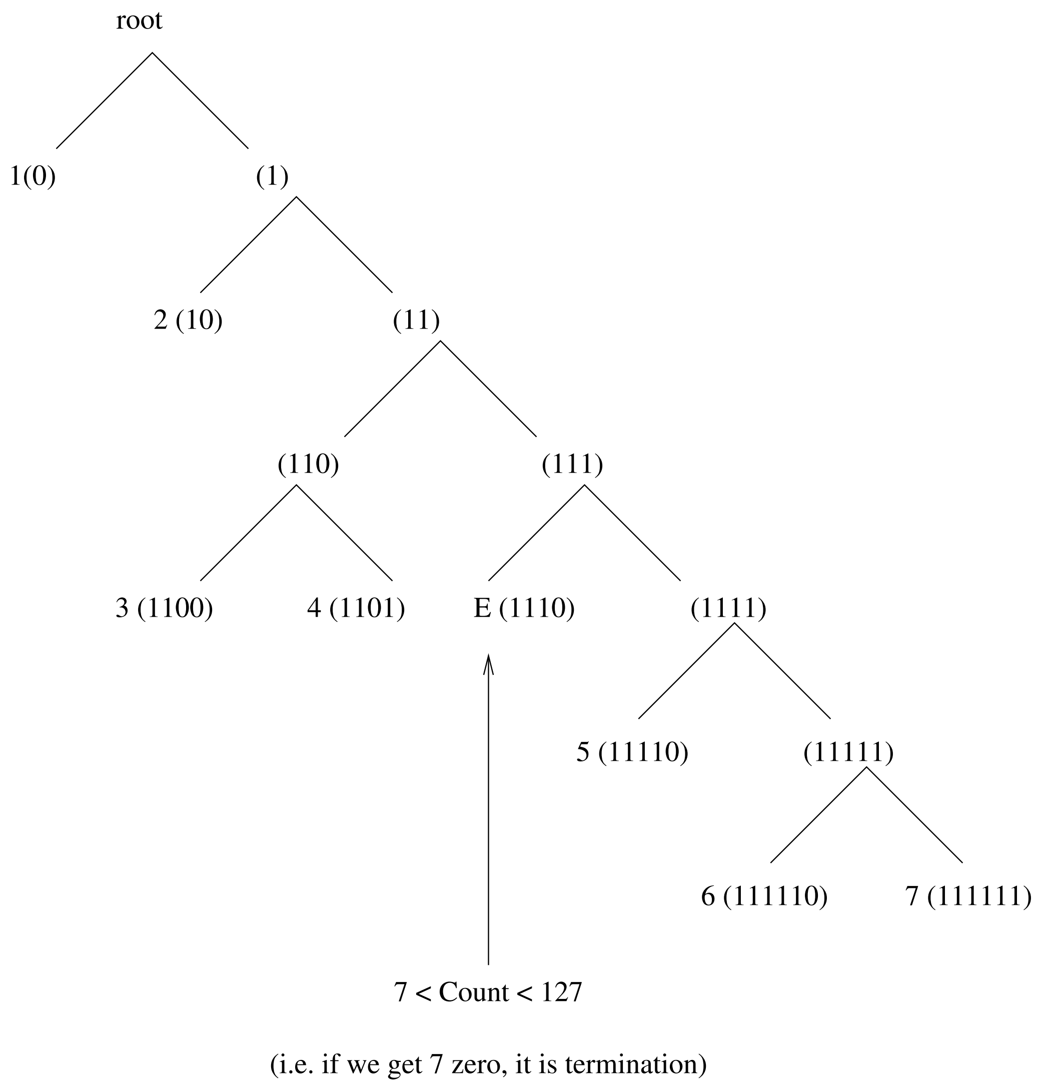

Figure 3: The decision tree for the Run Length Code

driver, but I am stuck with a printer that I bought which I can't use under linux. I want to have my consumer rights of being able to use it to print documents from the OS of my choice. I was trained as a theorectical research physicist and my primary document preparation system is LaTeX based under linux/unix, and I don't touch MS Office if I can avoid it. In fact I have not *ever* used Word 7 and Word 8 for any document for which I am the starting author! Although to use the printer, I can generate a postscript file, reboot my dual-boot box to windows and print it via win32 ghostscript's GDI driver, it is too painful for every document I want to print. Hence this effort.

Epson Seiko has a patent application family which applies to the US, Europe and most of the civilized world which details a more primitive version of the compression algorithm. Therefore, the compression algorithm should not be used for any other purpose than for interacting with the Epson EPL printers.

4.1 The MS Windows driver

Under win98, there is a dll `eptcmpa0.dll` which has the following symbols (and corresponding clearly identifiable routine sections):

EPCompressBitsImage
EPCompressGlyph
EPCompressImage
EPExpandBitsRLE
EPExpandRLE
EPGetCompressBitsBufferSize
EPGetCompressBufferSize

Under Win2k, the `eptminb7.dll` contains these strings (but no identifiable starts and ends of routines):

EPCompressImage
EPGetCompressBufferSize

I had a quick look at the assembler dump of these (there is a few Win32 PE disasemblers which run under linux); they seem to have what I want, but bit-suffling in assembler is quite complicated and essentially pure bit-manipulation maths and one needs a lot of patience to read them...

I don't have any deep knowledge about MS windows programming (I am all unix based, almost exclusively). But maybe this information is useful for somebody who understands windows dll's know about calling conventions and the win32 printing sub system, etc (e.g. the wine people...). Both of the windows drivers come with about 30 dll's, so at least this narrows it down to one to save some investigative work.

5 Misc

5.1 On-line Resources

Most of these pages are in Japanese (I am Chinese, so I can read a good deal of written Japanese...), and they may be out-dated. I thought it would be useful to patch ghostscript with some extra drivers specific to the Japan locale (Epson being Japanese and what not), but none of it worked (in mid-2000, after I got the printer and before I took a job where MS windows is used mostly).

The first one is in English, and the official Epson printer support for linux and is probably most useful, and some Epson employees see to have hanging around the forum, so posting to the forum there might get some attention... although they have explicitly say that the 5700L, 5800L, 5900L is windows and Mac only, I suppose if I provide this much detail here, it might pressure them into giving me some actual help.

- EPSON KOWA CORPORATION - linux driver forum `http://www.epkowa.co.jp/english/linux_e/linux.html`

- How to add printer device to gs http://www.ee.t.u-tokyo.ac.jp/~mita/FreeBSD/gsprinter.html
- Ghostscript 6.01 and GSview 2.9 J http://auemath.aichi-edu.ac.jp/~khotta/ghost/index.html
- gdevepag ver.3 http://www.humblesoft.com/gdevepag.html
- Software Archive http://www.tcp-ip.or.jp/~tagawa/archive/index.html
- Norihito Ohmori's WWW page http://www.bukka.p.chiba-u.ac.jp/~ohmori/
- gdevmd2k http://plaza26.mbn.or.jp/~higamasa/gdevmd2k/
- Ghostscript drivers http://unicorn.p.chiba-u.ac.jp/~ohmori/gs/
- Ghostscript drivers http://www.bukka.p.chiba-u.ac.jp/~ohmori/gs/
- Ghostscript driver for LIPS & ESC/Page & NPDL http://www.bukka.p.chiba-u.ac.jp/~ohmori/gs/Gdevlips.htm
- Fujitsu FMLBP 2xx driver for Ghostscript http://www1.freeweb.ne.jp/~nakayama/gdevfmlbp-120.html
- gswin5.50j information http://itohws03.ee.noda.sut.ac.jp/~matsuda/gswinj/gswin5j.html
- FORMPRINT for Linux(ESC/Page) http://www.vector.co.jp/soft/unix/hardware/se116245.html

5.2 The USB interface

There is work to be done in this area (need volunteers).

This document details what is going through the parallel port. In fact, a spool file generated by the windows driver can be send by linux and print successfully like this (by the root user):

```
cat sp00001.prn > /dev/lp0
```

However, sending the code through `/dev/usblp0` doesn't work. There are various documentation on the net (just search for "Epson printer" and "USB" and "linux") which says that for Epson printers which has a USB interface, one has to put an ESP/Page Job Language header before the job like this (in hex) to enable the USB interface: Tried it already, and it won't work.

```
00 00 00
1b 01 40 45 4a 4c 20 31 32 38 34 2e 34 0a
40 45 4a 4c 20 20 20 20 20 0a
```

(This is actually the hex code for "EJL 1284.4 @EJL" with some extra null bytes, line feeds, carriage returns, spaces, etc).

It should be quite simple to find out how the USB interface of the EPL5700L works - just use a USB snoop utility (c.f. the linux usb support page) to have a look at what goes through while a print job is going through a win32 host to the printer connected via USB.

EPL5900L known to work uni-directionally via the /dev/usb/lp0 device. No change needed for EPL5900L.

TODO: USB work is under way and this section is getting very out-dated.