Crudetype
An Adaptable Device Driver

R.M. Damerell
Royal Holloway and Bedford College

Introduction

The purpose of this program is to provide a framework for users to write TEX device drivers for a variety of 'crude' devices. Roughly speaking, 'crude' means any printer that cannot print the fonts that METAFONT generates. This would include daisy-wheels and most impact dot-matrix printers. Considered as output printers for TEX, such devices usually have some of the following defects:

1. Coarse resolution.
2. Restricted character set.
3. Some printers cannot do reverse line feeds; some can, and tear the paper.
4. Slow interface between CPU and printer.

Although such printers cannot do justice to TEX output, drivers for them are still needed. Some users cannot afford high quality printers. Some can only afford to use them for final output; so they need to make proofs on a cheaper printer. Also, anybody who has a high quality printer may well need to refer to various WEB files while writing a driver for it. These can become illegible in critical places. Figure 1 gives a sample from DVItype.

Using the basic (line printer) version of Crudetype, we can get a copy of these formulae which is at least legible, even though the result may not be at all pleasant to look at. A further difficulty with conventional drivers is that most of these use the algorithm 'paint a page of pixels, send it down the line'. This places a heavy load on both the host computer and the link to the printer. Of course, one can try to reduce this load by various optimisations, (e.g. by writing critical bits of code in machine language) but this makes the program non-portable, and often introduces bugs. Crudetype is written entirely in PASCAL, without any attempt at optimisation. When compiled on a VAX 780 with the NO-OPTIMISE, CHECK and DEBUG qualifiers it runs at about 2–3 seconds a page. These times are highly variable, and the VMS optimiser reduces them by about 10–15%.

Printers vary enormously both in their capabilities and in the commands that drive them. The behaviour of Crudetype is controlled by a large number of constants, which supposedly describe how the target printer does things. This does have the disadvantage that the user must compile a separate copy of the program for each different printer, and also devise some way to ensure that he uses the right version for the intended printer. But the only alternative seemed to be that Crudetype should read and parse a file describing the printer and this appeared to be unbearably messy. Ideally, these constants should be so designed that:

1. Any decent printer can be driven by assigning the right values to these constants and recompiling.
2. If the printer is properly documented, it should be immediately obvious what are the correct values for all these constants.

At present I do not have enough experience of different printers to come near this ideal. In particular, some printers can download characters. The problems of writing a program to support this facility in proper generality are horrible and ghastly. I have not made any serious attempt yet to tackle them. There are just a few places where a hook appears, and I hope eventually to attach actual routines for downloading.

Some of the more obvious problems of downloading are: when can you download? any time? start of page? or only at start of document? Can you load one character, or must you load a whole font at a time? How much memory does the printer provide for downloading? How efficiently does it use its memory? What does it do when it runs out? Can you clear out old fonts to make more space? What is the format of a download command? What parameters does it need, in what order, with what punctuation? In what order must pixels be sent? Should they be compressed, and how?

Implementation

This program was originally based on D.E.Knuth's program DVItype, but so many changes were needed for various reasons that there is not much of the original code left. The original version of Crudetype was aimed at a line printer (because everybody has

```
A \\{fix\_word} whose respective bytes are $(a,b,c,d)$ represents the number
$$x=\left\{\vcenter{\halign{$#$,\hfil\qquad&if $##$\hfil\cr
b\cdot2^{-4}+c\cdot2^{-12}+d\cdot2^{-20}&a=0;\cr
-16+b\cdot2^{-4}+c\cdot2^{-12}+d\cdot2^{-20}&a=255.\cr}}\right.$$
```

Figure 1. DVItype output can become illegible in critical places.

these), and was written on the VAX/VMS operating system. It is intended to be easily adaptable both to other systems and to other printers. So most of it is written in standard PASCAL. (It is not possible to tell exactly how much of it is standard, as we do not have a certified compiler.) But in some places, it is necessary to use extensions. In particular, `Crudetype` must read the font files, whose names are dynamically specified. That would be impossible in pure PASCAL.

`Crudetype` also uses non-standard code in order to talk to the user's terminal. It asks for the name of the `DVI` file, and for the first page and the number of pages to print. If an operating system forbids terminal interaction, the installer will have to find another way to give the program this information. As file handling is inevitably system-dependent, I have here allowed myself a lot of latitude in using VMS-specific procedures. If `Crudetype` cannot find a file, it will ask the user for another name. On the other hand, all files are read and written sequentially, and I have got rid of all uses of the default **case** statement. The intention is that all the system-dependent stuff goes near the top of the file, and all printer-dependent stuff at the end. Then with any luck you can merely concatenate Change files for the local system and the local printer, instead of having to merge them. All the code that is known to be non-standard has been carefully segregated from the rest of the program. It amounts to about 20 lines out of 750.

It is clearly impossible to predict what difficulties will appear in trying to install `Crudetype` on other systems. It would seem to be advisable to get the line printer version working before trying to adapt it for any other printer. To try to ease the process, I propose to distribute some test files with the program; each of these will come with the corresponding `DVI` file and (lineprinter) output file. I have also written a change file for a Phillips printer; but it should be understood that this file only works on a particular model of Phillips GP300, with a particular suite of resident fonts. It is only intended as a pattern to show what a printer change file should look like.

Translating the device-independent file

This part of `DVItype` is long and complicated, and I have tried to tidy it up, using ideas originally due to Prof. M.Doob. `DVItype` has seven functions for reading integers from the `DVI` file and two more for the `TFM` file. By passing suitable parameters, these have been reduced to three. `DVItype` processes each `DVI` command via a very big **case** statement. This can be rewritten in a much more readable form. To begin with, 192 of these cases are very similar, so let's get rid of them first:

```
⟨ Get DVI command and execute it 7 ⟩ ≡
com ← get_byte(dvi);
if com < 128 then
  begin set_character(com); move_right;
  end
else if (com ≥ 171) ∧ (com ≤ 234) then
    change_font(com − 171);
  else
```

See also section 8.

Now we come to the **case** statement proper. The macro *four_cases* generates 4 case labels, and generates a procedure call that reads a parameter from the `DVI` file and assigns it to *par*. A similar macro is needed for the movement commands; it has to construct a signed parameter.

```
⟨ Get DVI command and execute it 7 ⟩ +≡
case com of
  four_cases(128)(set_character(par);
      move_right; );
132: begin set_rule; move_right;
  end;
  four_cases(133)(set_character(par));
137: set_rule;
138: do_nothing;
139, 247, 248, 249: bad_dvi('byte:␣', com : 1,
      '␣out␣of␣context␣inside␣page');
140: end_page ← true;
141: push;
142: pop;
  move_cases(143)(h ← h + par);
147:   { W0 }
  h ← h + w;
  move_cases(148)(w ← par; h ← h + w);
  ⟨ about 15–20 lines omitted here 0 ⟩
end;
```

Because all the cases are thus collected together, it is now very easy to check that the **case** statement has 64 labels in the subranges 128 .. 170 and 235 .. 255. Therefore all 256 possible values of *com* produce a defined action; so we can correctly omit the default **case** statement. The resulting code is not quite as beautiful as this example suggests. When `Crudetype` is scanning through the file and looking for the first page to be printed, it must discard `DVI` parameters instead of using them, and so a second **if** statement and **case** statement are needed. But `DVItype` also has a second **case** statement (in function *first_par*), so I maintain that this presentation is still an improvement.

Coding schemes

A crude printer cannot possibly print the full range of characters that TEX uses. So `Crudetype` tries to map each character onto the nearest equivalent in the printer's fonts, if any tolerable mapping exists. The mapping is defined in an array called *codes*. Since all characters on most crude printers are the same size, we need one piece of data, not for each TEX font, but for each coding scheme. For each character c in a TEX font whose coding scheme has internal number s, $code[s,c]$ defines the corresponding printer character. Also, $known\_schemes[s]$ is a character string which usually contains the coding scheme of that TEX font.

So when a font is read in, we try to determine which of the *known_schemes* it belongs to. If the printer is not absolutely crude, then it might have italic or bold fonts. Then we might want a coding scheme to correspond to a single TEX font. So first we look at the font name and see if that matches any of the *known_schemes*. But if the printer is fixed-width, then all fonts of the same face are the same size, so we drop the font size digits off the end of the name. If the font name is not in *known_schemes*, then we try again with the scheme given in the `TFM` file. If that fails, then the font is deemed to be unprintable and we do not load it.

Several crude printers (e.g. daisy-wheels) have only a limited set of characters, which cannot be extended. Sometimes you can generate more characters by overstriking. `Crudetype` can be programmed to do this, by placing suitable entries into a table called *ligatures*. The name is chosen by analogy with the *lig_kern* programs in `TFM` files, but the data is completely different. When one TEX character maps onto several printer characters, we call the image a 'multi-character' command.

Getting data into the *codes* array is clearly a very long and error-prone job, so special procedures were written to reduce this. First suppose that a run of consecutive characters in some TEX font maps onto consecutive characters in a printer font. The procedure *alphabet* will enter the whole run at one go. For example, to set up the AMTEX fonts (nearly ASCII) for a line printer, do:

```
known_schemes[1] ←
    'TEX␣EXTENDED␣ASCII␣␣␣␣␣␣␣␣␣␣␣␣';
alphabet(32,95,1,1,32);
```

The Standard requires that the coding scheme name be padded to the declared length. The parameters of *alphabet* are, in order, first character of TEX font, length of run, internal number of TEX font, printer font, and corresponding character of printer font.

Clearly, *alphabet* will only cover a very small part of the problem. The next procedure called *row* enters data into a subset of the *codes* array corresponding to a single row of a TEX font. In the standard font tables, row number m is the subrange $8m \,..\, 8m+7$ of a font. It is hoped that when the calls of this procedure are written out in a program, the result will be (just about) legible, whereas a string of statements like $codes[i,j].char \leftarrow 27$; is certainly not legible.

The main parameter of *row* is a character string that consists of 8 'character specifiers' separated by spaces. So a very simple call of *row* might be:

```
row('␣h␣i␣j␣k␣l␣m␣n␣o␣',2,13,1);
```

(As before, the string must be padded, but I have here removed the padding.) The numerical parameters are: TEX scheme, row, printer font. So this call of *row* will generate row 13 of TEX scheme 2, (TEX TEXT) which happens to be the same as the corresponding row of ASCII. In practice, we would never use *row* for such a simple purpose, because we would use *alphabet*.

There are several escape sequences that need to go into the row specifier string. Since all the PLAIN.TEX coding schemes (except the math extension one) have the upper case Roman characters in their ASCII positions, these characters will surely be inserted into *codes* by the *alphabet* procedure. So they are available as flag characters. But the brackets are also used as flags, as they are so much more intelligible than anything else. Some characters have most undesirable effects when used in `WEB` strings. So we make upper-case letters stand for them. 'A' generates an at-sign, 'Q' a single quote, and so on. 'L' says that the next character must be used literally. 'C' means that the next character must be replaced by the corresponding (ASCII) control character, and there are some further simple escape sequences.

Now things start to get rather complicated. *row* can also be made to generate multi-character commands, by bracketing several character specifiers together. Square brackets mean that the characters inside are to be overstruck, round brackets mean they are to be typed horizontally, and angle brackets mean that they are to be typed vertically above each other.

So to generate a Macsyma style summation sign, which looks something like this:

```
====
\
 >
/
====
```

we have to insert the following mess into the row specifier string:

```
<UUUU[====]\\[SL>]/[====]>
```

The 'UUUU' is needed to get correct vertical alignment. The 'L' is needed to prevent the following > being taken as an angle bracket. In order to keep track of what is happening and to provide some diagnostics, *row* has to impose some rather arbitrary rules of syntax. One of these is that character specifiers may not contain spaces. The 'S' is an escape for a space, and it is needed here to push the > one step right into its proper position.

Movements

This section considers the problem of deciding where each character has to be printed on the printer's page. This is by far and away the most difficult (and unsatisfactory) part of `Crudetype`. The current version is not a properly designed algorithm; it is merely a bodge, obtained by a lot of trial and error. It does seem to give tolerable results on `WEB` files, lineprinter, and VMS. We use these variables:

h is 'TEX's cursor'. It gives the 'exact' horizontal position (in `DVI` units) generated by `DVI` commands. This is always updated exactly as in `DVItype`.

hh is the 'printer's cursor'. It marks the position (in the printer's units) where the next character will be set.

The obvious method is to multiply *h* by a factor *h_conv* and round to nearest integer. This gives extremely bad results, because the characters in TEX fonts vary in width, while many crude printers have fixed-width characters. If *h_conv* is too large, then you get spaces in the middle of words. If *h_conv* is too small, then successive characters in a word get printed on top of each other. With an intermediate value of *h_conv*, you get both effects at once; in other words, the characters in TEX fonts vary so much in width that the 'too large' and 'too small' values of *h_conv* overlap. A great deal of jiggery-pokery is then needed to get a tolerable result (well, sometimes!). It is obvious that as soon as we begin to tamper with the exact rounding algorithm, *h* and *hh* will start to drift apart, so we must try to bring them together again. We want all the characters in each word to come together, and we want the accumulated drift to appear in spaces between the words.

So a second attempt to evaluate *hh* is as follows. On a crude printer, all simple characters have the same width (w, say), and usually $w = 1$. But multiple characters have different widths. So one of the things *row* must do when assembling a multi-character is to replace w by the correct value. Rules are in effect multiple characters. After we set each character, we increase *h* by the width and *hh* by w. Then we record the new value of *h* as *last_h*, and ignore all further changes in *h* until another character (or rule) is due to be set. (This 'lazy evaluation' on *hh* is not mere sloth but an essential part of the process). If $h - last\_h$ is small, we leave *hh* fixed. If $h - last\_h$ seems large enough to be a space between words, then we force *hh* to increase. If $h - last\_h$ is really large, we replace *hh* (provisionally) by:

$$new\_hh \leftarrow round(h * h\_conv);$$

This second attempt works a lot of the time on plain text, but often fails when TEX makes a large backspace. In fact TEX seems nearly always to do large backspaces by *pop* rather than an explicit move left. TEX often expresses boxes by a sequence like this:

```
PUSH Move right -----> [set characters] POP
  ↑                    ↑                ↑
```

followed by a move either to one of the positions marked by the arrows, or close by. I try to deal with this by dropping markers at each of the arrowed positions. The right hand arrow is marked by *last_h*. The left hand arrow has just been popped off the stack; since the stack is realised as an array, it will be 'just above' the top of the stack. The centre arrow will be marked by *left_h*, which is defined as the value of *h* just before setting the first character after the latest *push*. Each marker has a corresponding value of *hh* attached. Suppose that we are about to set a character, and $h - last\_h$ is large and negative. Then we compare the current value of *h* with all the markers. Let m be the closest of these, and *mm* the corresponding rounded value. Then we re-round *new_hh* to force it to lie on the 'correct' side of *mm*. This seems to work fairly often, but it does sometimes slip.

Setting the vertical position on a crude printer is also very hairy. TEX expects subscripts to be much smaller than the main line, so it drops them by only a very small amount. On a crude printer, the subscript has to be the same size, and the

drop would normally get rounded to zero; it must be forced to be nonzero. When characters are underlined, TEX drops by a comparatively large amount, while the printer's underscore must be printed on the main line. So if v is the 'exact' vertical position and vv the rounded position, vv cannot be any monotonic function of v. What I have done is to declare a separate variable *rule_vv*, used only for vertical rules. As with horizontal moves, any large vertical move sets both vvs equal to the rounded value of v.

Sorting the page

Although 'crude' printers differ very much in their capacities, one thing they nearly all have in common is that they cannot feed the paper backwards. Some printers cannot backfeed at all; some tear the paper, and others let the paper slip and so lose position. Therefore it seems to be essential to process each page as follows: first copy the page into a suitable structure, then sort it by vertical and horizontal position, then print it.

The choice of method for sorting gave a lot of trouble. First I wrote the data onto a file and used the VMS library routines, but that had to be abandoned as not portable. Then I wrote a merge sort procedure which was amazingly slow. I believe this was because the files were being held on disc, and the disc transfers were slow. Shell sort was fast but not stable; I eventually chose a merge sort from 'Algorithms', by B.Sedgewick (Addison-Wesley, 1983). The algorithm is: chop the list in half, call *sort* recursively on each half, then merge the sorted halves.

The type of object that this algorithm sorts is a linked list. This could be represented either by a big array or by dynamic storage. Neither is ideal, because the size of a page is unknown, so whatever size you declare for an array is bound to be either too big or too small; and some PASCALs apparently do not implement pointers. So I have expressed everything in terms of certain macros, defined in the system dependent part of the program. Then `Crudetype` can be switched from heap to array merely by redefining these. For example, if p is logically a pointer, then the thing it points to will be defined as follows:

define *image*(#) ≡ #↑

when using dynamic store, and as:

define *image*(#) ≡ *pool*[*p*]

when using an array, here called *pool*. This illustrates one of the most valuable features of the `WEB` system: `WEB` not only makes it much easier to write programs, but it allows one to make complicated and far-reaching changes with much less difficulty than anybody could reasonably expect.

Known defects (July, 1986)

First it must be emphasised that this is an experimental version, offered 'as is' with no guarantee of performance. I do not have the time or manpower or machines to run adequate tests. Bug reports would be welcome, but the likeliest response will be something like "yes this is a bug and I do not know how to fix it; meanwhile, you have the source." Bug fixes are more welcome! That said, the principal known defects are:

1. Bad line breaks. When passed through `WEAVE` and TEX, `Crudetype` contains lots of these. It is of course perfectly possible to suppress bad breaks by inserting lots of *no_break* or *force_break* tokens into the `WEB` file, but I think it would be completely foreign to the spirit of TEX to do this. There ought to be a better way, and I hope somebody will tell me what it is.

2. Horizontal positioning. As explained above, this problem is very difficult, and I have only been able to produce a bodge that works sometimes. It is bound to fail sooner or later.

3. Downloading. `Crudetype` cannot support printers that can download characters. This is most unfortunate because it very severely limits the range of printers for which `Crudetype` is useful.

4. Accents. At least one make of printer does not provide accents, but accented letters. To print ü, you must send something like this:

```
<ESC>[2w      (that means Select German)
]             (an umlauted u)
<ESC>[10w     (Select ASCII).
```

It would be perfectly possible to make *row* generate this, but the real difficulty is that there is no obvious way for `Crudetype` to determine what character has been accented. A similar difficulty would arise with underlined characters, if one wanted to use `Crudetype` as a previewer (say, on a VT-100). An underlined character will generate character, backspace, underscore, and the underscore erases the character. Another problem that will make it difficult to use `Crudetype` to preview is that in order to conform to the Standard, it reads the `DVI` file sequentially. But any decent previewer must surely provide a Go to Page n command. So although the program makes some vague references to the alleged possibility that it might be used with a VDU, this is not yet really practical.