A $\TeX$ DVI Driver Family

Nelson H.F. Beebe
Center for Scientific Computation
220 South Physics Building
University of Utah
Salt Lake City, UT 84112
USA
Tel: (801) 581-5254

EMAIL: Beebe@Science.Utah.Edu (Internet)

15 April 1987 — Revision 2.07

Contents

1 Introduction **1**

2 The DVI Driver Software **2**

3 Outline of the DVI Drivers **4**

4 The Grubby Details **8**

4.1 `abortrun.h` 9
4.2 `actfact.h` 9
4.3 `alldone.h` 9
4.4 `bitmap.h` 9
4.5 `bopact.h` 9
4.6 `chargf.h` 9
4.7 `charpk.h` 11
4.8 `charpxl.h` 12
4.9 `clrbmap.h` 12
4.10 `clrrow.h` 13
4.11 `commands.h` 13
4.12 devinit() 13
4.13 devterm() 13
4.14 `dispchar.h` 13
4.15 `dumpchar.h` 14
4.16 `dvifile.h` 14
4.17 `dvihead.h` 14
4.18 `dviinit.h` 14
4.19 `dviterm.h` 15
4.20 `eopact.h` 15
4.21 `f20open.h` 15
4.22 `fatal.h` 16
4.23 `fillrect.h` 16
4.24 `findpost.h` 17
4.25 `fixpos.h` 17
4.26 `fontfile.h` 18
4.27 `fontsub.c` 19
4.28 `gblprocs.h` 21
4.29 `gblvars.h` 21
4.30 `gendefs.h` 21

4.31 `getbmap.h` 21
4.32 `getbytes.h` 22
4.33 `getfntdf.h` 22
4.34 `getpgtab.h` 22
4.35 `inch.h` 23
4.36 `initglob.h` 23
4.37 `keydef.h` 24
4.38 `keybrd.c` 24
4.39 `loadchar.h` 25
4.40 `loadbmap()` 26
4.41 `machdefs.h` 27
4.42 `main.h` 28
4.43 makechar() 28
4.44 `movedown.h` 28
4.45 `moveover.h` 28
4.46 `moveto.h` 29
4.47 newfont() 29
4.48 `nosignex.h` 29
4.49 `openfont.h` 29
4.50 `option.h` 30
4.51 `outrow.h` 31
4.52 prtbmap() 32
4.53 `prtpage.h` 33
4.54 `readfont.h` 34
4.55 `readgf.h` 34
4.56 `readpk.h` 34
4.57 `readpost.h` 34
4.58 `readpxl.h` 34
4.59 `reldfont.h` 35
4.60 `rulepxl.h` 36
4.61 `setchar.h` 36
4.62 `setfntnm.h` 36
4.63 `setrule.h` 36
4.64 setstr() 37
4.65 `signex.h` 38
4.66 `skgfspec.h` 38
4.67 `skipfont.h` 38
4.68 `skpkspec.h` 38
4.69 `special.h` 38

4.70 strchr.h 40
4.71 strcm2.h 41
4.72 strid2.h 41
4.73 strrchr.h 41
4.74 tctos.h 41
4.75 typedefs.h 42
4.76 usage.h 42
4.77 warning.h 42

5 DVI Driver Device Support **43**
5.1 dvialw.c 44
5.2 dvibit.c 48
5.3 dvican.c 50
5.4 dvigd.c 50
5.5 dviimp.c 51
5.6 dvijet.c 51
5.7 dvijep.c 51
5.8 dvil75.c 58
5.9 dvim72.c 58
5.10 dvimac.c 58
5.11 dvimpi.c 58
5.12 dvio72.c 59
5.13 dvioki.c 59
5.14 dviprx.c 60
5.15 dvitos.c 61

6 Setting up the Environment **62**
6.1 IBM PC Setup 62
6.2 TOPS-20 Setup 63
6.3 VMS Setup 65

7 Building the Drivers **70**

8 Programming Conventions **72**

9 Implementation Problems **76**
9.1 TOPS-20 76
9.2 Unix 4.1BSD 78
9.3 HPUX 79

9.4 IBM PC Lattice C 79
9.5 IBM PC Wizard C 80
9.6 IBM PC Microsoft C 80

10 Implementation Efficiency **81**

11 Implementation Summary **83**

1 Introduction

This document describes a family of driver programs which translate the TeX device-independent (DVI) output file to a device-specific file for a particular output device, such as a dot matrix printer, a laser printer, a CRT display, or a phototypesetter.

These programs grew out of a DVI driver for the BBN BitGraph terminal[1] developed by Mark Senn at Purdue University. This was further worked on at the University of Washington by Stephen Bechtolsheim, Bob Brown, Richard Furuta, Jim Schaad, and Robert Wells, with contributions for PostScript devices by Neal Holtz at Carleton University. Simon Barnes of Schlumberger Cambridge Research Ltd., and Robin Rohlicek at BBN provided useful additions to the BBN BitGraph driver which have been generalized and incorporated in Version 2.07.

In the winter of 1986/87, John Sauter adapted one of the dot matrix drivers to produce `dvil75` for the DEC LA75 printer. Lon Willett at Utah adapted a laser printer driver to produce `dviimp` for the Imagen laser printer family.

The transformation to about ten other device drivers, plus the massive code rearrangement for many new features as well as easy identification of host- and device-dependent sections, was carried out at the University of Utah by the present author. The code has undergone such extensive modification that there remains little in common with the original version, which no doubt has spawned other families of drivers at various institutions.

What makes this driver family unique is the author's commitment to its support on a wide variety of host architectures and operating systems, and to *its remaining entirely in the public domain.* Recipients are encouraged to install it, use it, improve, and in the spirit of the Emacs community, return the improvements to this author for further distribution. Don Knuth, Leslie Lamport, and Mike Spivak have given us TeX and METAFONT, LaTeX, and $\mathcal{AMS}$-TeX, which are enormous contributions to the world of typesetting, and we should emulate them in giving away our own work. Many of us work at universities and to a large extent are supported by public educational

[1] This fine terminal, which is almost unique in having support for variable sized downloaded fonts, is no longer made by BBN. It has been licensed by BBN to EastMark, Inc. (240 Smith Street, Lowell, MA 01851, Tel: (617)453-9150), who have improved the documentation and packaging and sell it as the GTR01 model. Unit list price is about \$2950 (spring 1986). The screen resolution is 1024×780 in a portrait orientation nicely matching the printed page.

funding; we owe them something back. Someone who commercializes these DVI drivers is on his own; I shall continue to give the family away to anyone who asks for it.

In January 1986 the availability of the family was announced on the `TeXHaX` bulletin board, and repeated again in the March 1987 issue of the TeX Users Group newsletter, TUGBoat. In the first two months after the announcement, about 60 copies were shipped by mail, and based on file reference counts, a roughly equal number of Internet ANONYMOUS FTP retrievals were done from sites unknown. An electronic mailing list is maintained to inform recipients of changes and updates.

2 The DVI Driver Software

TeX is written in a subset of ISO Standard "PASCAL which was chosen both to enhance its portability, and to facilitate translations to other languages. For example, it avoids nested procedures, and use of enumeration variables. This author is aware of at least five[2] translations of TeX to C, and one of METAFONT to Tal and another to Fortran. The TeX software system is actually written in the Web language, and translated into a Pascal program by Tangle and into a TeX document by Weave. Web supports change files which contain local modifications to the source code, so that all sites can work off a common master source file.

Despite its many fine features which support the development of reliable and correct software, regrettably, Pascal is completely unsuited for the writing of DVI driver programs. The two serious omissions in this area are the lack of bit operations, and random-access I/O. The former are critical for efficient support of low-cost dot matrix printers, and the latter for efficient access to the many files which the driver requires.

Niklaus Wirth's second popular language, Modula-2, remedies both these problems, but is not yet widely-enough available, and lacks a standard I/O library. Because of its strong similarity to Pascal, it should not be difficult to prepare a version of the Web system to support it.

Ada would also be a good possibility, but compilers again are either not available, or too expensive in many environments.

[2](1) old TeX78 port to 16-bit workstation; (2) D.R. Fuchs' MicroTeX (available from Addison-Wesley); (3) N. Naugle and T. Rokicki at Texas A&M; (4) P. Monardo's Common TeX at UC Berkeley; (5) Per Bergsten at Chalmers Institute of Technology, Göteborg, Sweden.

Since TEX now runs on many different computer systems, including the IBM PC and the Apple Macintosh, there is a need for DVI driver programs which can be used on all of them. There is really only one language choice, and that is the C language originally developed at Bell Laboratories for the support of the Unix operating system. It is even more widely available than Pascal, and it contains the necessary support for bit operations, random I/O, and hands-on access to files without having the operating system throwing in extra garbage which ruins them.

Because C is a "low-level" high-level language, it is close to common computer architectures, and in most implementations, compiles into reasonably efficient machine code, provided some care is taken at the source level. The major weaknesses of C are its general lack of type and bounds checking, and its lax syntax which allows programming errors to be accepted as legal code. Two particularly noxious examples of the syntax problems are illustrated by the following program fragments:

```
for (k = 1; k <= n; ++k);
    foo(k);                 /* foo is called once */

for (k = 1; k <= n; ++k)
    foo(k);                 /* foo is called n times */
```

The indentation here implies to the reader that `foo()` is controlled by the `for` loop.

```
if (avar = bvar);   /* this sets avar */
    foo();          /* foo is called once */

if (avar = bvar)    /* this sets avar */
    foo();          /* foo is called if bvar is non-zero */

if (avar == bvar);  /* this leaves avar intact */
    foo();          /* foo is called once */

if (avar == bvar)
       foo();       /* foo is called if avar equals bvar */
```

The indentation again implies that `foo()` is controlled by the `if` statement. You must look closely to spot the tiny differences which give the very different effects noted in the comments.

Although the coming ANSI C Standard extends the language to support type checking of function arguments, arrays carry no bounds information unless the programmer arranges to create composite data types or pass dimension variables, and the majority of the standard library routines that have array or pointer arguments simply assume that adequate space is available when they are stored into.

Two versions of Web in C have been reported in TUGBoat, [3,4]but I have not had an opportunity yet to obtain either of them. The extensive variable and code section cross-referencing provided automatically by Web are exceedingly useful features, and good grounds for Don Knuth's calling Web programming *literate programming*.[5] Without these cross-references, we have to make do with string search utilities like the Unix `grep` program.

Fortunately, the C preprocessor provides most of the facilities that Web change files do. The fact that all versions exist inside the source code encapsulated by `#if ...#endif` sequences is both an advantage and a disadvantage. One source file has all implementations contained in it, but it is rarely possible for an implementor to completely test all versions when modifications are made to the version-specific sections.

Nevertheless, the use of C for this driver family has proved quite satisfactory, and development work on the DEC TOPS-20 operating system is supported by the availability of Richard Stallman's wonderful Emacs from MIT, and many of the most useful Unix utilities implemented with the portable C compiler and library, PCC-20, ported to TOPS-20 by Jay Lepreau, at the University of Utah. In February, 1987, I installed the KCC-20 compiler which is under active maintenance and development at SRI-NIC, and the driver family has been implemented with it as well. It has proved to be a fine compiler, and is being updated to conform to the draft ANSI C Standard.

3 Outline of the DVI Drivers

The DVI drivers follow to some extent David Fuch's `dvitype` program, which defines the standard DVI file format and how to process it. In simple terms, the DVI file is a sequential 8-bit byte command stream produced by TeX as it

[3]Klaus Guntermann and Joachim Schrod, *WEB Adapted to C*, TUGBoat **7**, No. 3, 134–137 (1986).

[4]Silvio Levy, *WEB Adapted to C, Another Approach*, TUGBoat **8**, No. 1, 12–13 (1987).

[5]*Computer Journal*, **27**, 97-111 (1984)

processes a document, and consists mostly of compactly-encoded commands to move to a specific page position and typeset a character there. It is further embellished with version numbers, page pointers, checksums, font names, magic numbers to identify the file as a DVI file, and the like. The DVI driver must be able to read this file, processing all, or user-selected, pages, and produce an output file in a device-specific format which can produce typeset pages on some particular device. Although the output is device-specific, much of the processing done by the DVI driver is concerned with interpreting user run time command options, decoding the DVI commands, finding font files, and extracting character bitmaps from them. All of this part of the drivers is common across devices, and accounts for quite substantial code sharing.

Each driver consists of a single C source file, `dvixxx.c`, where `xxx` is a mnemonic abbreviation of the device, such as `alw` for Apple LaserWriter. This file contains C preprocessor `#include` directives to obtain a standard header file, `dvihead.h`, which in turn `#include`'s a header file, `machdefs.h`, containing machine constants, and another, `typedefs.h`, containing type definitions.

Originally, `dvihead.h` also contained the definitive revision history of the driver family in its extensive initial comments. This practice has been discontinued, and the maintenance history is now stored in a separate file, `00revhst.txt`. This avoids unnecessary recompilations when the Unix `make` utility is used to manage the software.

C has only a limited number of primitive data types, but it does have `struct` and `typedef` statements permitting the creation of new data types. These do not particularly enhance the type checking possibilities, at least not until the coming ANSI C Standard has been widely implemented, but they do inform the compiler of storage requirements, which permits efficient code generation. This is of particular importance on the segmented memory architectures of the Intel iAPX microprocessor family used in the IBM PC and clones, because there is an enormous execution-time penalty for processing objects bigger than the base architecture easily supports. Two examples of these are the need for subroutine calls to perform 32-bit integer arithmetic, and around 30 instructions to compute the address of an element in a vector larger than 64 Kbytes in extent, compared to one instruction when the vector is smaller than that size. Many compilers for these architectures have provided no support whatever for addressing objects larger than one segment.

Objects in the DVI file consist of one-, two-, and four-byte integers, both signed and unsigned, and character strings. It is important to identify the exact space requirements of these to obtain an efficient implementation. Consequently, `typedefs.h` defines new types called `BOOLEAN`, `BYTE`, `COORDINATE`, `INT8`, `INT16`, `INT32`, `UNSIGN16`, `UNSIGN32`, and `void`.

The `void` type is available only in some C implementations, although it will be in the ANSI C Standard; it is used to signify the absence of a value, particularly for function values which are discarded. All C procedures are in fact functions which may or may not return a value. A common programming error is to return inconsistent values from inside a function, or to return a value from one part, and none from another. One of the sloppy habits of many C programmers is to ignore a returned function value, particularly with library procedures, most of which return error indicators of some sort when they cannot return a valid pointer or primitive value. The type cast `(void)` in front of a function call then tells both the compiler and the reader that the function may in fact be returning something, but the result is intentionally being discarded.

The `COORDINATE` type is used for device coordinates (usually dots or pixels), and it is defined because the precision required may depend on the output device and on the size of the available integer types provided by the compiler.

C is quite permissive about type conversion by assignment; almost any object can be assigned to another and the compiler will accept it and generate code for it. One of the commonest misuses of this feature is the assignment of pointers to integers, and vice versa. While this may be reasonable on some architectures, there is no good reason to assume that both are interchangeable. Indeed, on segmented memory architectures, such as the widely-used Intel iAPX family mentioned earlier, this is definitely not the case. An integer may be a 16-bit quantity, while a pointer may be a composite 32-bit (or larger) value made up of segment number, memory offset in the segment, access rights, and type information. Another common error is the passing of integers of different sizes through the same argument of a function. On the PDP-11 and VAX architectures where C was first implemented, this may work, but it definitely does not on the Intel and Motorola processors. Consequently, in the DVI driver family, extensive use is made of type casts in assignments, expressions, and function arguments. Null pointers are always cast as such; they are *never* assumed to be zero.[6]

After `machdefs.h`, the next file included by `dvixxx.c` is `main.h`, which

[6] If you find such an instance, it is a bug, so please inform me of it so it can be fixed.

is the function `main()` required to prepare an executable C program. It immediately includes all the standard system header files which are required for use of system library functions, plus four additional header files, `commands.h`, `gblprocs.h`, `gblvars.h`, and `gendefs.h`. For drivers for interactive displays, it will also include `keydef.h`.

`commands.h` defines mnemonics for the commands in the DVI file, such as `BOP` (*Beginning Of Page*), `SET_RULE`, `FNT_DEF1` (*Font Definition*), and so on.

`gendefs.h` is a catch-all for general macro definitions and assorted constants.

`gblprocs.h` defines all of the global functions. With the availability of a compiler anticipating the coming ANSI Standard, the Microsoft Versions 3.0 and 4.0 compilers on the IBM PC, this file has been revised so that the data types of arguments are defined; this was not previously possible in C. The alternate declaration forms are selected under control of the preprocessor symbol `ANSI`, which is set in `machdefs.h`.

`gblvars.h` defines all the scalars, vectors, arrays, and structures which must be known globally. These are analogous to `EXTERNAL` variables in languages like PL/1. They represent one of the weakest areas in C, in that no control exists over which functions reference or modify them; the packages of Modula-2 and Ada were introduced specifically to deal with this problem. The names of these global variables reside in a single name space, so it is important that they be chosen carefully so as to be unlikely choices for local variables inside a function. Otherwise, an omitted type declaration in the function will use the global variable, producing a hard-to-find side effect (a fashionable word for *bug*).

`keydef.h` contains flag definitions and function declarations for the keyboard input package, `keybrd.c`.

There are presently about 70 files with extension `.h`, and 9 have now been presented. The others are functions, most of which are required by all the drivers. By inclusion of all 5 of the standard top-level header files at the beginning of each of the remaining ones, they could be turned into separately-compiled functions. However, since several of them contain operating-system or device-dependent code sections delimited by preprocessor directives, extreme care would be required in building executable versions of the drivers to avoid inclusion of object files compiled with the wrong code segments selected. We have found it more reliable to use the `#include` directive to insert each of these functions into the `dvixxx.c` modules so that a only a single file need be presented to the compiler and linker. The recompilation time has so far not proved to be objectionable,[7] while the possibly more economic alternative of separate compilation would be exceedingly likely to cause problems.

To summarize then, here is the general structure of the file nesting:

```
dvixxx.c --> dvihead.h   --> machdefs.h
                         --> typedefs.h

         --> main.h      --> commands.h
                         --> gendefs.h
                         --> gblprocs.h
                         --> gblvars.h
                         --> keydef.h

         --> other function .h files
```

There is *no* further nesting of files than this. Each of the functions may call other ones, but the `.h` files do not include any additional program text.

In some of the function files, preprocessor directives select minor variations in code. When significant differences would be required, then `dvixxx.c` does not use the `#include` mechanism, but instead has a private version of the function embedded in itself. All functions following `main()` are in alphabetical order, and are readily identifiable on the terminal screen or in a program listing by a standard function header.

4 The Grubby Details

We shall now descend into subsections to discuss each of these header files separately. Like their invocations in `dvixxx.c`, these too will be given in alphabetical order. Since C permits letter case distinctions, many of the names were historically chosen to be in mixed case for readability (e.g. `AbortRun()` instead of `abortrun()` or `abort_run()`). This proved to be a nuisance as the code has been moved from machine to machine, so I have applied the Unix `sed` utility to replace all mixed-case names with lower-case ones. For portability, function names must be unique in the first six characters, and file names in the first eight, so all function names have been shortened to eight or fewer characters.

[7] 7 to 10 minutes on a standard IBM PC XT (4.77MHz 8088 CPU) with the Microsoft Version 3 C compiler, and under two minutes on mainframes.

4.1 abortrun.h

abortrun() is called to terminate the DVI driver execution, both normally, and in the event of errors. It closes any open files, issues an error message if called with a non-zero completion code, and then returns to the operating system with the normal exit() call.

4.2 actfact.h

actfact() takes a font magnification factor and adjusts it to match the nearest one in an internal table of standard TeX font magnifications. That way, a user request for, say, magnification 1094, will get adjusted to the expected value, 1095. Correct adjustments to an integer value are critical, because font files contain this value as part of their names or directory path.

4.3 alldone.h

alldone() is the routine called for normal termination. It closes any error log file and copies it to stderr, tells the user if a log file was created, and then calls abortrun() to finish the job.

4.4 bitmap.h

This file is included by all the dvixxx.c files which create page bitmaps in memory for dot-matrix printers. It defines macros and functions for accessing the bitmap; this is necessitated by the constraints of segmented memory on the Intel iAPX architecture.

4.5 bopact.h

bopact() is called from prtpage() to handle beginning-of-page action for interactive displays. For dvibit, it writes the command and status window.

4.6 chargf.h

The functions chargf(), charpk(), and charpxl() are called from readfont(), as well as from setchar() in dvibit.c, and loadchar() in dvialw.c, dvican.c, dviimp.c, and dvijep.c. They have the job of extracting the raster description, and possibly, character metrics, of a single character from the corresponding .gf, .pk, or .pxl font file, and decoding it to produce a simple bitmap which can be easily painted into the page bitmap by `dispchar()`, or formatted for downloading to a more intelligent output device, such as the BBN BitGraph, PostScript printers, and the Hewlett-Packard LaserJet Plus.

For most devices, the character bitmap is quite small, and there would be no difficulty in storing the full bitmap. However, printers of much higher resolution are likely to become available over the next few years, while host computer memories will probably not increase substantially. For example, a 10pt character at 2400 dots/inch would require about 8 Kbytes for the entire image, but fewer than 25 bytes for a single row. A 72pt character, such as in the *aminch* font, would need about 430 Kbytes for the image, but only about 200 bytes for a row. Since all three font formats store character bitmaps in row order, it seemed expedient to have `chargf()`, `charpk()`, and `charpxl()` build up the character description a row at a time, calling another routine, `outrow()`, each time a row is completed. They then require no knowledge of what is to be done with the character bitmap, and no duplication of code for this is necessary, since it can be entirely hidden in `outrow()`.

The `.gf` font file format produced by the new METAFONT is reasonably complex, sharing much in design with `.dvi` files. Instead of being stored as a full bitmap, with each row rounded up to a multiple of 32 bits to achieve word alignment on 32-bit hosts, as they are in a `.pxl` file, character bitmaps in a `.gf` file are stored in a run-length encoded form with counts of white and black pixels in each row, and commands to skip repeated white rows. This gives a compression of only a few percent for normal size characters, but `.gf` font files for very large characters may be up to 30% smaller than the corresponding `.pxl` files.

The `.pxl` file format has a 2K-byte directory at the end containing all necessary character metrics, plus pointers to their raster descriptions which are scattered throughout the file. This means that a relatively small block of data is all that is initially required from the file, and random access can be later used to retrieve needed character bitmaps. The `.gf` file also has a directory at the end, but it contains only character escapements and widths, plus pointers to encoded character packets which are scattered throughout the file. The escapements and widths are required for updating coordinates when a character is set, but the offsets of the character bitmap from the character reference point, which are required in clipping decisions when a character is to be set, are stored in the character packets. Their availability is desirable, because off-page characters need never be set. This deficiency is unfortunate, because to retrieve them, it is necessary to read through the entire font file, causing unnecessary file page mapping. For many fonts used in a typical TEX document, only a very few of the 128 possible characters in the file will ever be used, so this design flaw in the .gf file format could in the worst case almost double the amount of font file input, compared to use of .pxl files. Although it would be possible to carry a flag for each character indicating whether or not the offsets were available, I decided not to do so, because this would be needed only for one of the three font file formats.

The code in chargf() follows that in GFtoPXL fairly closely, although some duplication of code fragments was permitted to avoid the goto statements present in the latter. Most of the approximately 340 lines of code in chargf() are part of a large switch statement inside a loop, dispatching on the value of the next input byte.

4.7 charpk.h

As discussed in the preceding section, charpk() is called to handle the retrieval and decoding of the raster description of a single character from the .pk form of a font file.

The .pk font file format was defined by Tom Rokicki, formerly at Texas A&M University, now at Stanford University. He has provided support programs GFtoPK, PKtoPX, and PXtoPK for converting between the various font file formats. Combined with the Stanford TEX Project's GFtoPXL, only PXLtoGF or PKtoGF yet need to be written to allow going from any one of the three font file formats to the other.

The .pk format is even more complex than the .gf format. Instead of run-length encoding character bitmaps by rows, it first recognizes repeated rows,[8] then treats the remaining bitmap as a single bitstring to be run-length compressed. In addition, run-length counts are based on 4-bit nybbles, rather than 8-bit bytes. For the standard fonts, the .pk files occupy slightly less than half the disk space of the .gf files. This will clearly be of importance on small machines, particularly personal computers.

The .pk font file lacks a character directory, so to retrieve character metrics, the entire font file must be mapped in from disk. Each character is contained in a single packet, beginning with its number and metrics, and followed by its packed raster encoding. A stored packet length allows skipping over the rasters on the first pass by readpk() as the metrics are collected.

[8] About 37% of the bitmap rows in the standard fonts required by plain.tex are repeated!

The code in `charpk()` follows that in `PKtoPX` fairly closely, although some duplication of code fragments was permitted to avoid the `goto` statements present in the latter. There are approximately 350 lines of code in `charpk()`, but the processing is more complex than that required in `chargf()`. Consequently, the file `charpk.h` departs from the convention of one function per file, and it contains three private functions, `get_bit()`, `get_nybble()`, and `pk_packed_num()`, plus several private globals known only within this file. For historical reasons, the C data type `static` when used for objects declared outside a function body, really means *private to this file*, and we therefore have introduced a preprocessor symbol `PRIVATE` for this purpose. A `typedef` cannot be used, because `static` is a type *modifier*, not a pure data type.

4.8 `charpxl.h`

`charpxl()` is quite short—about 55 lines. It retrieves a single character bitmap from a `.pxl` file, and sends each row to `outrow()` for processing. Since there is no raster decoding necessary, once argument checking has been done and the font file successfully positioned to read the character bitmap, only eight lines of code are needed to do the whole job. Before the support for `.pk` and `.gf` font files was added, this operation was performed in a half-dozen different functions. Now, it is totally isolated in this one routine.

4.9 `clrbmap.h`

`clrbmap()` has the rather simple job of clearing a bitmap array to zero for those devices, such as dot matrix printers, which actually require one. Although it is a simple loop in C, it has to be done for every page, and accounts for a surprisingly large portion of the run time when the driver execution is monitored with a program counter histogramming utility. For some machines, we have prepared a more efficient assembly language version, `zerom()`, which it calls instead. This can take advantage of basic hardware facilities like block move to do it as fast as the machine is capable of. For segmented memory bitmaps, where the bitmap is allocated as a vector of pointers to raster lines, rather than as one large array, special code is required to clear individual raster lines. The preprocessor symbol `SEGMEM` identifies these code sections. Details can be found later in the section on `getbmap()`.

4.10 clrrow.h

clrrow() is called from chargf(), charpk(), and charpxl() to clear the global array img_row[] before collection of each character bitmap row begins. Although the code is a single loop, it is required frequently enough that it warranted encapsulation in a separate function.

4.11 commands.h

This file contains symbolic definitions of all of the DVI commands which are required in the driver program. These are fixed with the freezing of TeX, but no program should ever contain magic constants where symbolic names would make the code more readable.

4.12 devinit()

devinit() has the job of carrying out any startup initializations required by the output device for the processing of a single DVI file. It is called from dvifile(). For most dot-matrix printers, there is little to do, but terminals and laser printers may require considerably more.

4.13 devterm()

devterm() is called from dvifile() to perform any device termination following the processing of a single DVI file. Comments made about devinit() apply to this function as well.

4.14 dispchar.h

dispchar() has the delicate job of painting the current character onto the display surface. For downloaded font devices, this job may involve little more than output of a few command bytes, once the character rasters have been sent to the device. For bitmapped devices, it is fairly tricky, since the character bitmap will rarely be aligned on a word boundary in the output bitmap, and the code must be both efficient, since it is done several thousand times per page, and independent of the host computer word size. It is further complicated by the fact that the right shift operator in C may or may not propagate a sign bit; the unsigned integer type attribute which should suppress this propagation is not available in all compiler implementations.

The preprocessor symbol `ARITHRSHIFT` is used to select the appropriate code section.

Although the body of this function is less than 60 lines long, it is about the most complex function in the entire driver family, and also the one likely to require the most execution time. If code optimization is worthwhile, it surely will be here.

For segmented memory machines, the bitmap is allocated as a vector of pointers to raster lines, rather than as one large array. Code which accesses the bitmap must take care to not step pointers across raster lines. The `BITMAP(y,x)` macro otherwise hides the bitmap addressing. Since `dispchar()` fills in raster line order anyway, there is no penalty for this. Further details may be found in a later section on `getbmap()`.

4.15 dumpchar.h

This is a debugging routine used to dump character raster representations to the `stderr` file. It is not normally required or included in the drivers.

4.16 dvifile.h

`dvifile()` is called from `main()` once for each DVI file to be processed. It calls `dviinit()` and `devinit()` for initializations, `readpost()` to read in the DVI postamble, then loops calling `prtpage()` for each page to be printed, and finally calls `devterm()` and `dviterm()` for termination chores.

4.17 dvihead.h

This file is the only non-function header file included directly by the `dvixxx.c` files. It is the repository of old revision and development history of the family (current history is stored in `00revhst.txt` instead), and does nothing more than possibly define a debug symbol, and then include `machdefs.h` and `typedefs.h`, plus some system header files. The rest are included in `machdefs.h` and `main.h`.

4.18 dviinit.h

`dviinit()` is called from `dvifile()` for the initial DVI file processing. After defining some global variables, it does a simple parse on the DVI file name specified on the command line to identify the file's directory, if any, and its name and extension. The path and name are saved in global variables, and a name for the output file is constructed. The DVI file and output file are opened, and the magic identifying bytes in the DVI file are read to make sure this is really a proper DVI file. If it is not, `fatal()` is called to terminate the job. Otherwise, control returns to its caller.

If opening of the output and log files fails, then the file path of the DVI file is discarded, and these files are opened in the current directory. This allows processing of DVI files which reside in directories for which the user does not have write access.

4.19 `dviterm.h`

`dviterm()` is called from `dvifile()` to complete processing of the current DVI file. It closes all open files and frees all dynamically-allocated font memory.

4.20 `eopact.h`

`eopact()` is called from `prtpage()` to handle end-of-page action on interactive displays. It currently supports only the BBN BitGraph, but should be straightforward to adapt to new displays. It consists of a single command loop which runs as long as keyboard input is available. All input commands are processed, but the display is not updated until there is no more input. The keyboard input support package in `keybrd.c` supports this.

4.21 `f20open.h`

This section has a little story behind it, so in case you are interested, read the next four paragraphs; otherwise, skip to the last paragraph.

Most modern computer architectures are based on memory built up of 8-bit chunks, called *bytes*, which are then assembled into larger units like *half words*, *words*, *long words*, and so on. The 8-bit byte size is convenient in that it is large enough to contain both the 7-bit ASCII and ISO character sets, and the 8-bit EBCDIC, and extended ASCII and ISO sets.

The DEC-20 architecture dates from the 1960's when word-addressed memory was common, and like the original IBM 70x and 70xx machines, its word size is 36 bits. With early BCD, CDC, and Fieldata 6-bit character sets, this was not an unreasonable size, and the fact that six such characters fit exactly into a word is still with us in Fortran's six-character name limits and the size of external names on many operating systems. On many of these machines, character access required awkward masking and shifting operations, one exception being the Univac 1100 machines which have quarter-word instructions, and therefore use 9-bit ASCII.

The DEC-20 is unusual in that it supports any byte size from 1 to 36 bits with equal ease.[9] Conventionally, text files are stored in 7-bit ASCII, five per word, leaving one bit left over in the low-order position of the word.[10] When 8-bit bytes are stored, they could be packed in groups of six bytes in a word pair, as Univac does, but on the DEC-20, they are generally placed in four-byte groups in the high-order 32 bits of a word, with the four low-order bits zero, where they are amenable to efficient access with the byte instructions.

Unfortunately, the standard `fopen()` function does not know about byte sizes; it just assumes text bytes are wanted and opens the file in 7-bit mode. The DVI file, however, has 8-bit bytes, so on the DEC-20, we have this special version, `f20open()`, which stoops to a lower level in the I/O library to twiddle the right flags to get the open done for 8-bit bytes.

Nobody outside the DEC-20 has to care about this routine, because the preprocessor symbol `BINARYOPEN` is set automatically in `machdefs.h` to either `f20open()` for the DEC-20, or `fopen()` for everyone else.

4.22 `fatal.h`

When necessary files cannot be found or read, or the disk storage fills up, execution must be terminated. This routine writes out an error message, resets the terminal characteristics if they were changed, tells the user if a log file was created, and then calls `abortrun()` to terminate the job. The name of the function issuing the call to `fatal()` is included as part of the message.

4.23 `fillrect.h`

When TeX sets a horizontal or vertical rule, the output device probably simulates this by drawing a black solid-filled rectangle. For typesetters, this may be an available primitive, and so needs only the output of a few command bytes. For most others, we have to actually fill in bits in the

[9] At DECUS meetings, you can see folks walking around with T-shirts reading "I don't care what they say, 36 bits are here to stay".

[10] That bit is normally zero; when it is not, it signifies that the word contains a line number instead of text.

bitmap. This job is only marginally easier than the one `dispchar()` has to do, and many of the same comments made in that section apply here.

There usually are not a great many rules to typeset in a typical document, so efficiency in `fillrect()` is of little concern. However, the new Bezier curve option supported by LaTeX draws lots of little rules (e.g. 2 dots by 2 dots on a laser printer), so you might want to give it some attention. When many small rules are set like this, it might be valuable to have special case code for them, particular for the dot-matrix printer drivers. Only this function needs to know if you do. `dvialw`, `dvican`, `dviimp`, and `dvijep` arrange to avoid sending rule sizes to the device when they have not changed from the last use, and this has helped to reduce the output volume considerably.

For segmented memory machines, the bitmap is allocated as a vector of pointers to raster lines, rather than as one large array. Code which accesses the bitmap must take care not to step pointers across raster lines. The `BITMAP(y,x)` macro otherwise hides the bitmap addressing. Since `fillrect()` fills in raster line order anyway, there is no penalty for this. More details about this will be found in the later section on `getbmap()`.

4.24 `findpost.h`

When TeX sets a document, it does not know until it gets done where all the pages are, and what fonts are needed. Consequently, the DVI file stores this information in a postamble section which the DVI driver probably wants to make use of. Because the postamble size is clearly variable, in its last eight bytes, the DVI file has a four-byte magic number that tells the driver that TeX actually finished writing the file, and a four-byte pointer to the beginning of the postamble. `findpost()` has the job of reading this little trailer, validating it, exiting via `fatal()` if it is bad, and otherwise using the random I/O positioning standardly available in C to set the file pointer at the beginning of the postamble.

4.25 `fixpos.h`

TeX works in tiny units which do not match any output device, but still let it position characters accurately for even the most exacting phototypesetters.

The font designer works in device units (dots or pixels), because high quality demands attention to device resolution, particularly for the common low- and medium-resolution devices such as dot matrix printers and laser printers. The font designer specifies character widths in such a way as to provide for the most pleasing juxtaposition of characters into words, and TeX follows this specification precisely.

The problem is, TeX does not know what the output device resolution will be (remember that DVI means *device independent*), so it may ask for a character to be positioned at what will probably not be an exact device coordinate. If the driver just rounds each precise TeX coordinate to the device coordinate system, the chances are that characters in a word may not be positioned relative to one another as the font designer intended.

The effect of incorrect rounding can be glaringly obvious—a line of 60 asterisks followed by a line of two asterisks separated by 58 spaces set in a fixed-width font may be off by as much as one character position on a 300-dot/inch laser printer.

Considerable thought has gone into this problem, and it was not until Version 2.06 of `dvitype` that the folks in the TeX project got it right (at least until we see an example to the contrary). `fixpos()` implements the algorithm directly from `dvitype`.

4.26 `fontfile.h`

Although most of the code in the DVI drivers is completely portable across C implementations, operating system features like file names obviously are not. In particular, the drivers need to be able to take a font name, like *cmr10*, and a font magnification, say 1500 (normal 10pt type on a 300-dot/inch laser printer), and from them, construct a name of the file containing the character raster descriptions. On the DEC-20, this file would usually be called `texfonts:cmr10.1500pxl`. Under Unix, it might be `/usr/lib/tex-/fonts/1500/cmr10.pxl`. On the IBM PC, it could be `d:\tex\fonts-\1500\cmr10.pxl`.

Three different font file formats are now supported, and `fontfile()` has the job of figuring out just what the file names should be, and returning a list of them. For example, it might return for *Computer Modern 10pt* the names `texfonts:cmr10.300pk`, `texfonts:cmr10.300gf`, and `texfonts:-cmr10.1500pxl`. Note that the naming convention for `.gf` and `.pk` files has changed from that used for `.pxl` files—standard magnification is now 200 instead of 1000.

The directory paths supplied are determined by the preprocessor symbol `TEXFONTS` defined in `machdefs.h`. In order not to permanently bind directory paths into the compiled programs, the default directory names for these files may be overridden at run time by `initglob()` with values of environment variables.

4.27 fontsub.c

Sometimes, a DVI file references fonts which are not available in the requested magnification, or perhaps even in the entire family. Calling `fatal()` if this happens is hardly user-friendly—it may be that only one character in a fifty-page document belongs to this missing font.

The DVI drivers therefore offer the user the possibility of providing a file of font substitutions which can be tried in turn until one is found. If none can be, then the drivers consult a built-in table[11] of standard magnifications to find the nearest one in the requested family; surely a character from *cmr10* at 1440 magnification instead of 1500 magnification is better than none at all! If even that fails, then execution continues with the missing character metrics all assumed to be zero; with luck, this will affect only a few lines in the output.

Presumably, the `.tfm` file was available to TeX when it created the DVI file, so `fontsub()` could be extended to try to obtain the information from that file instead, and perhaps recover. A second possibility would be to supply some arbitrary, but reasonable, character width, and just set blank space for the missing characters. Neither of these has yet been implemented.

A message is issued to the log file if a substitution is made according to the requests in the substitution file, or if a neighboring magnification was chosen in the same family, so the user can still find out it happened.

The format of the font substitution file, and the mechanism for finding it, has been chosen to simplify its use. First of all, an explicit font substitution file can be specified by a driver command-line option at run time. If this is not done, then the name of the DVI file is used to construct a default name with a different extension. For example, on Unix, the DVI file `/usr/jones/book/hornets.dvi` would select a substitution file `/usr/jones/book/hornets.sub`. This gives a document-specific choice of substitution files.

If this one is not found, then a standard name, such as `texfonts.sub`, is looked for in the same directory. In our example, this would be `/usr/jones-/book/texfonts.sub`. This second choice is therefore a directory-specific

[11] This table is defined by `gblvars.h`. It contains values for devices of resolution 144, 200, and 300 dots/inch; if the family is extended to higher-resolution devices, this table should be extended.

solution.

If this file is not found either, then as a last resort, `fontsub()` looks for a system-wide file, usually stored in the same directory TeX gets its input files from. The example file would therefore be something like `/usr/lib/texinputs/texfonts.sub`.

The format of the substitution file is simple to prepare, and can be parsed (almost) unambiguously by the standard C input routine, `sscanf()`. It is also operating-system independent, so that substitution files can be moved from machine to machine if necessary.

Font substitution lines have the forms:

```
% comment
oldname.oldmag  ->      subname.submag  % comment
oldname oldmag  ->      subname submag  % comment
oldname         ->      subname         % comment
```

The first two forms request substitution of a particular font and magnification. The third form substitutes an entire font family; the closest available magnification to the required one will be used. Any dots in the non-comment portion will be converted to spaces, and therefore, cannot be part of a name field. Examples are:

```
% These provide replacements for some LaTeX invisible fonts:
iamr10 1500     ->      amr10 1500      % comment
iamr10.1500     ->      amr10.1500      % comment
iamssb8         ->      amssb8          % comment
```

The first matching substitution will be selected, so magnification-specific substitutions should be given first, before family substitutions.

Comments are introduced by percent and continue to end-of-line, just as for TeX. One whitespace character is equivalent to any amount of whitespace. Whitespace and comments are optional.

Anything which follows the substitution font will be ignored, so a line like

```
iamr10 1500     ->      amr10 1500 long live TeX!   % comment
```

will be parsed successfully. It is not *de rigueur* in computer science to accept sloppy parsing, but in this case, a simple workable solution was provided by a standard library facility, `sscanf()`, with one line of code to call it; doing a better job would involve at least a few hundred lines of code.

4.28 gblprocs.h

This file contains definitions of all of the functions referenced by the DVI driver family. `main()` includes it. For compilers which support the additional function argument type syntax of the coming ANSI C Standard, it has been updated to include these declarations. They are selected by a new preprocessor symbol, `ANSI`, defined in `machdefs.h`, in order to preserve compatibility with existing compilers.

4.29 gblvars.h

This file contains definitions of all the global scalars, vectors, arrays, and structures required by the drivers. This is also where the font magnification table used by `fontsub()` is defined.

4.30 gendefs.h

This file is included by `main.h` and serves to collect assorted macro and constant definitions. It should normally not require modification on any new host computer.

4.31 getbmap.h

Some output devices require the construction of a bitmap containing the page image. Originally, the bitmap was simply statically allocated as a single global array. However, the segmented memories of the Intel iAPX microprocessors pose a problem, because the bitmap is almost always larger than one memory segment. If all compilers supported accessing of objects larger than one segment, this problem could be ignored at the expense of efficiency (recall that a large array indexing operation takes about 30 iAPX instructions). None of the four C compilers I have tried permit compile-time allocation of objects larger than one segment. Three of them provide alternate memory allocation library functions which can be used at run time to create large objects, but unfortunately, Microsoft Version 3.0 permits *no* addressing of large objects, even when the large memory model compile option is selected; Version 4.0 remedies this problem.

After giving the matter some thought, I decided to implement the bitmap as a run-time allocated object, which on large machines is a single array, and on the Intel family, is a vector of large-memory pointers to raster lines. The allocation is performed by `getbmap()`. These two different representations pose a usage problem, because addressing of a particular word in a given raster line is different in the two cases. The bitmap is accessed frequently, so function call overhead to get at it cannot be tolerated. Fortunately, the C preprocessor comes to the rescue, and the `BITMAP(y,x)` macro has been defined to return the address of the requisite word with in-line code. This can be assigned to a scalar pointer variable, which is then incremented as the scan steps through the raster line. The only precaution necessary here is to avoid stepping the pointer from one raster line to another, since this is certain to fail with the vector-of-pointers representation. The only function which needed to be changed for this purpose is `clrbmap()`. Use of the scalar pointer variable reduces the bitmap addressing overhead to once per raster line, which is acceptable.

The preprocessor symbol `SEGMEM` is used to select appropriate code fragments in `clrbmap()`, `getbmap()`, `machdefs.h`, `main()`, and several of the `dvixxx.c` files.

Changes were also necessary in `makechar()` and in some of the `prtbmap()` variants which appear in several of the `dvixxx.c` files to handle vectors of raster pointers, rather than scalar pointers, but this again has been hidden from the code by appropriate preprocessor macro definitions.

4.32 `getbytes.h`

This function retrieves the next n bytes from a file (usually the DVI file), returning them in a character array. It is short enough, and used often enough, that it could profitably be redefined as a macro for in-line expansion in some future version of the code.

4.33 `getfntdf.h`

This function uses `readfont()` to collect all the font definitions stored in the DVI file postamble. It is called from `readpost()` when font preloading has been requested.

4.34 `getpgtab.h`

This function uses the *last page* pointer stored in the DVI file postamble to follow the chain of pointers to page beginnings right back to the first page set by TeX. It uses the random I/O facilities of C to accomplish this. The page table constructed contains byte position pointers to each of the pages for the output document; it is used in `main()` to find and process the pages requested by the user. TeX stores ten counters for each page; currently, only the first of these is collected and saved. This is conventionally the page number.

4.35 inch.h

In order to support run-time option values with attached units, the function `inch()` is provided. It converts a string containing a fractional number followed by a two-letter unit designator to a floating-point return value in inches. My apologies for the name of this function to the rest of the world which uses the metric system—our paper sizes are still in inches. The units recognized are given in the following table:

value	units implied
#.##bp	big point (1in = 72bp)
#.##cc	cicero (1cc = 12dd)
#.##cm	centimeter
#.##dd	didot point (1157dd = 1238pt)
#.##in	inch (1in = 2.54cm)
#.##mm	millimeter (10mm = 1cm)
#.##pc	pica (1pc = 12pt)
#.##pt	point (72.27pt = 1in)
#.##sp	scaled point (65536sp = 1pt)

A leading plus or minus sign is optional, and the letter case of the dimension name is ignored. No space is permitted between the number and the dimension.

4.36 initglob.h

`initglob()` is called once at the beginning of `main()` to handle all initializations of global variables. Although C permits declaration-time variable initializations by constants, this is not always convenient when machine-dependent actions, such as construction of word-size dependent bit masks, are required. `initglob()` uses the standard library function, `getenv()`, to obtain any run-time definitions of the environment variables `DVIHELP`, `FONTLIST`, `TEXINPUTS` and `TEXFONTS` which are to override the internal defaults. See the sections on `fontfile()`, `fontsub()`, `machdefs.h`, and `usage()` for further remarks.

The driver family will accept `.gf`, `.pk`, and `.pxl` font file variants, but not all sites may be expected to have all of these. A default font file type search path is therefore established internally, with font types in the order `.pk`, `.gf`, and `.pxl`, which corresponds to increasing file size, and places new METAFONT fonts before old METAFONT fonts. This search path can be overridden at run time by definition of an alternate path in the `FONTLIST` environment variable (logical name in DEC operating systems). Because operating systems may impose a particular syntax on environment variables, `initglob()` will ignore letter case and any other characters except the three strings `GF`, `PK`, and `PXL`; only the order, and presence or absence, of these in the definition is significant. Acceptable strings equivalent to the default search path could therefore be `PKGFPXL`, `pk/gf/pxl`, `pk GF pxl`, `pk-gf.pxl`, and so on.

There is absolutely no harm in using the default value of `FONTLIST`, even if some of the font file types are not available, since all that will happen is that a file open will fail and another will be tried. Redefinition of `FONTLIST` to avoid open attempts on files which are known not to exist is only for efficiency. It is of course also useful during debugging to be able to force the use of a particular font type.

4.37 `keydef.h`

`keydef.h` contains flag definitions and function declarations for the keyboard input package, `keybrd.c`. It is included by `main.h` only for those devices which need it.

4.38 `keybrd.c`

In order to support better control of screen display for interactive DVI drivers, I developed a set of keyboard input primitives defined in `keybrd.c`. These are analogous to C's standard input functions, but guarantee (if the host supports it) immediate access to keyboard characters without blocking in a wait for a carriage return, and also provide a function for testing if input is available. The code necessary to do this is ugly on all the operating systems supported for this DVI driver family, and is best hidden away in a separate module.

Because it is likely to be useful in other applications, this package must be separately compiled and linked with each interactive DVI driver, instead of being included in the source with a preprocessor directive.

The extensive initial comments in `keybrd.c` describe the package in detail. The header file `keydef.h` must be included in modules which use the package in order to obtain flag definitions and function declarations.

4.39 loadchar.h

As new characters are referenced by the DVI file, the driver must keep track of them, retrieving character metrics and bitmaps from the font files. With the exception of the main body font in a document, only a few characters from each font are likely to be used, so it would make no sense whatever to load an entire font in just because one character from it was used. Instead, space for character definitions is allocated dynamically, and a reference count for each character is maintained. When space is no longer available, which is discovered when `malloc()` begins to return null pointers, garbage collection is in order. This is a topic of continuing interest in computer science, particularly in the Lisp language family.

David Fuchs and Don Knuth wrote an interesting Stanford report in 1981 entitled *Optimal Prepaging and Font Caching*[12] which treats the problem from the point of view of a TeX DVI driver. They point out that in this case, the job is somewhat easier, because it is possible for the driver to look ahead to see what characters are needed next, rather than using reference counts which reflect past history of character usage. One of the surprising results they found was that with proper implementation of their algorithm, relatively little storage is actually needed on the output device to keep it always supplied with the needed fonts.

The scheme used in our DVI drivers is more modest. No lookahead is done, so the reference counts can be used to identify characters, and even entire fonts, which are candidates for storage reclamation. Actually, most implementations of the driver family so far have been on hosts with a large address space, and we have been able to get by without even having to do garbage collection. Nevertheless, the reference counts are there for use, or some ambitious implementor can go and study *Optimal Prepaging and Font Caching* to figure out how to do the job right.

`loadchar.h` is the version used for the bitmapped output devices; it retrieves character rasters and leaves them in a dynamically-allocated memory block located by a pointer in the `char_entry` structure. Private versions of `loadchar()` for downloaded font devices send the rasters directly to the device, rather than preserving them in memory. In either case, `loadchar()` does not itself actually retrieve the rasters from the font file;

[12] Later published in *ACM Transactions on Programming Languages and Systems*, **7**, No. 1, 62-79 (1985).

it calls `charxx()` to do the job. `charxx()` is declared in the `font_entry` structure as

```
void (*charxx)();
```

and was stored in the font structure by `readfont()` with a statement like

```
fontptr->charxx = (void(*)())charpk;
```

which saves the address of the function required to read the character rasters. It is more convenient, and more efficient, to do it this way, than to use the font type to index a `switch` statement every time one of these functions must be called. `loadchar()` should be able to call it by

```
(void)(*fontptr->charxx)(c,outrow);
```

since the pointer operator, `->`, has higher precedence than the indirection operator, `*`. In fact, Kernighan and Pike, in their book *The Unix Programming Environment*, use precisely this construction on p. 250. They also use the more complex, but equivalent, variant

```
(void)(*(void (*)())(*fontptr->charxx))(c,outrow);
```

on p. 264. These work as expected with the Lattice, Microsoft, and Wizard C compilers on the IBM PC, but on both PCC-20 and 4.1BSD `cc`, these generate a jump to the word *containing* the address of the function, rather than to the function itself! It took some experimentation to find that a workaround was possible:

```
void (*charyy)();
...
charyy = fontptr->charxx;
(void)(*charyy)(c,outrow);
```

This results in correct code with all compilers used so far for the DVI driver family, so `loadchar()` adopts it as the standard way to do things, in the expectation that the bug is likely to be found in other Unix compilers derived from the Portable C compiler. I have not had time to dig into the compiler sources to track down the source of the problem.

4.40 `loadbmap()`

`loadbmap()` is used to download large characters as raster bitmaps, instead of as downloaded fonts. It is used in those drivers which have a limitation on downloaded font sizes. Private definitions exist in `dvican.c` and `dvijep.c`.

4.41 machdefs.h

This file is included by dvihead.h, which in turn is included by dvixxx.c. It contains almost all of the machine-specific definitions; the only other ones are already bracketed by preprocessor conditional directives using symbols defined in this module. Unless you are developing a new implementation, *it should not be necessary to change more than this single file* when you install any one of the dozen or so drivers in the family.

Comments in this file describe each of the flags, and they will not be repeated here. The important things to change are

- the operating system flag, currently one of OS_PCDOS, OS_TOPS20, OS_UNIX, or OS_VAXVMS;
- the implementation flag, either none at all, or one of ATT, BSD41, BSD42, HPUX, PCC_20, KCC_20, IBM_PC_LATTICE, IBM_PC_MICROSOFT, or IBM_PC_WIZARD;
- the default file pathnames, FONTPATH and SUBPATH;
- the default environment variables, TEXFONTS and TEXINPUTS.

The last two of these are already encapsulated in code selected by the current OS_xxxx flag, but they may differ from site to site, even with the same operating system and compilers. In order to minimize the dependence of the programs on specific file path names, in most implementations, the environment variables (logical names in DEC operating systems) are tested at run time in initglob(), and any values they have replace the default file paths. Thus, even though a driver compiled on one IBM PC might have been built with the assumption that the fonts are found in the directory d:-\tex\fonts, it still may be usable on another system where the fonts have been installed in the directory c:\usr\pctex\texfonts. All the user need do is define the name TEXFONTS before running TeX:

```
SET TEXFONTS=C:\USR\PCTEX\TEXFONTS\
```

Note the final backslash—fontfile() will assume it can simply prefix the definition of TEXFONTS to a file name.

There are several other parameters defined in machdefs.h; the best thing to do is examine each of them in turn, and change them to suit the local environment, before compiling any of the drivers.

4.42 `main.h`

This is the main program for all of the DVI drivers. It had become large enough to cause problems with some compilers, so most of the code has been split out into seven other functions. It contains `#include`'s for `commands.h`, `gendefs.h`, `gblprocs.h`, and `gblvars.h`, as well as for several standard system header files. Following this, it defines the `BITMAP(y,x)` macro for bitmap addressing

The code body then checks the argument count, and calls `usage()` and `exit()` if none are given, and then goes on to call `initglob()` for initialization, `option()` to process run-time options, `dvifile()` once for each command-line argument which is not an option, and finally terminates with a call to `alldone()`.

4.43 makechar()

For those dot-matrix printers which require bitmap encodings of vertical bit columns, it has proved convenient to hide the messiness of this in a separate function, `makechar()`, which is defined in several of the `dvixxx.c` files. Because it must deal with bits in several raster lines, this function is sensitive to the bitmap representation, as discussed in the section on `getbmap()`.

4.44 `movedown.h`

This function is called in response to a vertical movement command from the DVI file. It updates the current vertical page position, `v`, in TEX units, and computes a new rounded position, `vv`, in device units, using `fixpos()` to get it right. Efficiency buffs might want to turn this into a preprocessor macro, since it is used often.

4.45 `moveover.h`

This function is called in response to a horizontal movement command from the DVI file. It updates the current horizontal page position, `h`, in TEX units, and computes a new rounded position, `hh`, in device units, using `fixpos()` to get it right. Like `movedown()`, it is often used, and is a candidate for macro metamorphosis.

4.46 `moveto.h`

This function should really be a macro; it just records the new current point (in device coordinates) in the global variables (`xcp`,`ycp`).

4.47 newfont()

When `readgf()`, `readpk()`, and `readpxl()` have retrieved necessary font information, they call `newfont()`, currently defined only in `dvialw.c`, `dvibit.c`, `dvican.c`, `dviimp.c`, and `dvijep.c`. It has the job of outputting whatever is necessary to the device to define the use of a new font.

4.48 `nosignex.h`

In the interests of compactness of the DVI file, when TEX writes an integer, it usually tries to use the fewest number of bytes possible. The integer may be a signed or an unsigned value. This function collects the next n bytes from the file and returns them as a single n-byte *unsigned* integer value, where n is in $1 \ldots 4$. `nosignex()`, like its twin `signex()`, is heavily used, and is a good candidate for special loopless code and in-line expansion.

4.49 `openfont.h`

Whenever a font switch is called for by the DVI file, it may be necessary to find and open a new font file; `openfont()` is called to handle the operation. It uses `fontfile()` to obtain lists of possible file names, and if none of these can be opened, it will first ask `fontsub()` for a possible font substitution, and if that too fails, it will try neighboring members in the magnification family.

In typical TEX documents, font switching occurs *often* (three times in this very sentence). On all operating systems, file opening is a relatively expensive operation,[13] and the drivers therefore take care to avoid unnecessary openings.

The number of open files allowed is usually dependent on fixed tables in the operating system, which in practice are much too small. You have to set special parameters in your `config.sys` file to get more than the default of

[13] On a large university IBM mainframe where I did my graduate work, each file opening cost us \$0.50, except at night, when it dropped to \$0.25. I'm glad those days are gone! I probably spent more money doing that, than I did for the IBM PC on which I'm now writing this document.

8 open files with PC-DOS, and 5 of these are already used by the operating system; 20 is the absolute limit. TOPS-20 allows more than 70 open files per process, but PCC-20 follows the Unix practice of limiting the number to 20, at least 3 of which are in use when the process begins. Also, buffer space is required for each open file, and since PCC-20 does not support TOPS-20 extended addressing, one tends to run out of memory for buffers before one runs out of file handles for new file openings. KCC-20 permits 32 open files, and its support for extended addressing eliminates the buffer space problem.

The number of open files that the drivers will use is determined by the constant `MAXOPEN` set in `machdefs.h`, and the ones currently open are maintained in a cache. Each open request for a file in the cache increments an open reference count. When the `MAXOPEN` limit is reached, the font file which has the lowest open reference count is closed, but any characters cached from it are retained in memory for further use. The new file is then opened and a cache entry created for it.

The font cache search is a simple sequential search; since `MAXOPEN` in practice is quite small (5 to 15), there would likely be little benefit in implementing a more sophisticated associative cache table using hashing or binary search. However, it should be considered if an implementation is made which permits much larger numbers of open files; all that needs to be changed is the first `for` loop in `openfont()`. TeX is prepared to have up to 64 active fonts, so `MAXOPEN` could potentially be as large as that.

4.50 option.h

All run-time command-line option parsing is handled by this function. After `initglob()` has done global initializations, `option()` is called once for each option found in the `argv[]` array passed to `main()` by the C run-time library startup routine. `option()` is just a big `switch` statement with cases selected by the second character of its string argument. A bad argument results in a call to `usage()` followed by a termination call to `fatal()`.

All of the variables which can be set by run-time options are global, so `option()` only has a single argument. New options can be easily added merely by introducing a new global variable in `gblvars.h`, a new `case` pair in the `switch` statement in `option()`, and code selected by the option wherever else it is required.

The available options will not be repeated here; they are completely described in `dvihead.h`, and in the separate help file, `dviman.ltx`. That help file also exists in a more useful format, `dviman.texinfo`, which can be converted by Gnu Emacs to a `.info` file.

The option parsing is simple, involving little more than noting the presence of an argument, or collecting its string or integer value, or calling `inch()` to convert a number followed by a unit specifier. Similar code is present in almost all C programs. The one difference here is that *option letter case is not significant*.

In my view, one of the design flaws of Unix and C was to make letter case significant in names. This means that every n-character identifier has 2^n different variants, all pronounced the same. The code therefore cannot pass the "telephone test" (dictate your favorite program over the telephone to a friend without excessive spelling of words).[14]

All user command-line input to the DVI drivers is letter-case *insensitive*, except possibly for file names, which must conform to the host operating system conventions. Internally, compiled-in file names are represented in lower case letters, since that works for both Unix and most case-insensitive operating systems. The Unix environment variables are in upper-case, since that is conventional, though not required, in Unix, similar to the common practice of C programmers of specifying variables with lower-case names, and preprocessor constants with upper-case names.

This practice, incidentally, is *rigorously* adhered to in the DVI driver family. When you see a name entirely in upper-case in the source code, you can be sure it is a preprocessor constant, and not a variable or function name. The reverse is also true. No names are spelled in mixed letter case.[15]

4.51 `outrow.h`

Once a single row of a character bitmap has been retrieved, decoded, and stored in the global array `img_row[]`, `outrow()` is called from `chargf()`, `charpk()`, and `charpxl()`, to prepare it for output. The public version in `outrow.h` is used by most of the family; it merely copies the row from its global storage location to the dynamically-allocated raster storage for that character. Private versions of `outrow()` in `dvialw.c`, `dvibit.c`, `dvican.c`,

[14] The situation is not unlike Japanese, for which the 600 or so commonest Kanji (Chinese) characters used in written Japanese share only about 25 different pronunciations, resulting in the necessity of speakers writing characters in the air or on their palms to distinguish which one is really meant when the choice is not clear from context. This homonymic situation also poses a problem for phonetic Roman character computer and word processor input schemes with oriental languages; see the January 1985 issue of *IEEE Computer* for several interesting articles.

[15] Except TeX, when for lack of it, you have to write it `TeX`!

`dviimp.c`, and `dvijep.c`, handle the formatting and output of the row for their particular device.

4.52 prtbmap()

This function is not stored in a separate include file, because it is sufficiently different from device to device to warrant inclusion inside the `dvixxx.c` routines as a private function. Nevertheless, it is an important one, because it has the job of formatting a completed bitmap for output to the device.

The bitmap cleared by `clrbmap()`, and written by `dispchar()`, is conceptually a simple rectangular array of bits, consecutive bits in memory corresponding to consecutive dots on the output rasters. For addressing efficiency, however, each scanline begins on a new word or byte boundary, depending on the host architecture, and if the host has segmented memory, it may be stored as a separately-allocated raster lines, as described in the earlier section on `getbmap()`. This format allows efficient operation by `dispchar()`, which can do its important job oblivious to the peculiarities of output device expectations.

`prtbmap()` must take this rectangular array of bits and reformat it for the output device. For some devices, like the Hewlett-Packard LaserJet, this is straightforward —only some escape sequences at the start and end of each raster line have to be attached, but otherwise, the raster bytes can be dumped directly to the output file.

The Printronix printers are almost straightforward—they want horizontal raster lines encoded six bits per character; the peculiarity is that the bits must be reversed in the byte! It looks like some designer got his wires crossed. This unpleasantness is most easily handled by table lookup, and that is just what `prtbmap()` in `dviprx.c` does.

Other printers are more obnoxious—they want 6, 7, 8, or 24 bits in a *vertical* column to be reordered into consecutive bytes, because then the print head pins can be matched to the input bits and fired or not. For those that place fewer than 8 bits in a byte, most do so by biasing by 64 or 128, which can be done by a logical *or* operation. The DEC LA family printers use a bias of 63, which requires a slower addition for each output byte. Printer designers have apparently not understood that input buffers of horizontal raster lines would be so much easier for the user, and also offer better possibilities for run-length compression of rasters.

This routine is probably the most tedious to code in the drivers, and it must be redone for each new bitmap format. It typically requires 50 to 100 lines of code (excluding translation tables). It is a fairly time-consuming operation, so the code goes to some length to be efficient, using tricks like scanning over zero words in a raster line, trimming empty top and bottom raster lines before starting the encoding, and unrolling inner loops. The code is also likely to be dependent on host word size, and for a few lines in the inner loops, portability may well be sacrificed to efficiency.

Some of the dot matrix printers support run-length compression of rasters, wherein consecutive identical bytes are collapsed into a count field and a single data byte. This is a little tricky to get right, and is less effective with the vertical encodings than it would be with horizontal encodings, but it usually does reduce the output file size somewhat. The disappointing feature is that the printer is likely to run *slower*, rather than faster, because of inefficient decoding of the compression. If you have such a printer, try it both ways before you decide which method to use.

4.53 `prtpage.h`

`prtpage()` is one of the largest functions in the family. Its job is to typeset a single page selected by its caller. It has nearly 400 lines of code, but is really just a single loop containing a giant `switch` statement which selects code fragments corresponding to each one of the 256 different byte codes which can be found in a DVI file. The loop exits when an end-of-page command is reached.

For interactive displays (e.g. `dvibit`), it also calls the end-of-page routine, `eopact()`, whenever keyboard input is available. This permits instant response of the screen display to user commands to reposition the display, go to a new page, or change the display magnification; no time is wasted finishing the page display if it is just going to be changed anyway.

Fortunately, the first 128 DVI codes are just ordinary character typesetting requests, and another 64 are font change commands, so 3/4 of the cases can be handled by four lines of code.

Almost all of the DVI driver run time is spent in `prtpage()` and the functions it calls, but because its structure is so simple, there is little one can do to improve it, short of in-line expansion of code for some of the more commonly-used command codes.

Implementation of the *Optimal Prepaging and Font Caching* algorithm discussed earlier in the `loadchar.h` section would probably involve converting this into a two-pass operation, one to collect commands and font requirements for a single page, caching the commands in memory to reduce

I/O requirements, followed by a similar loop to process them, with font preloading happening at appropriate times as determined by the Fuchs-Knuth algorithm.

4.54 `readfont.h`

This function is called by `getfntdf()` to read in a single font definition sequence in the DVI file postamble, or the first time a font definition is encountered, if preloading has been suppressed. It initializes the `font_entry` structure with parameters from the DVI file, then calls `reldfont()` to direct the loading of additional information from the font file.

4.55 `readgf.h`

`readgf()` is called from `readfont()` to initialize all font parameters for a `.gf` font file, and retrieve the character metrics. As noted in the earlier section on `chargf()`, the entire file must be read to collect all of the metrics. For the several drivers currently supporting downloaded fonts, `newfont()` is called to do whatever is required to start a new font.

4.56 `readpk.h`

`readpk()` is called from `readfont()` to initialize all font parameters for a `.pk` font file, and retrieve the character metrics. Just as for `readgf()`, the entire file must be read to collect all of the metrics, and for the drivers currently supporting downloaded fonts, `newfont()` must be called to do whatever is required to start a new font.

4.57 `readpost.h`

This function is called near the start of `main()` after command-line options have been processed. It coordinates the reading and checking of the DVI file postamble, using function `findpost()`, calls `getfntdf()` if font preloading has been requested, and finally calls `getpgtab()` to retrieve the document page table.

4.58 `readpxl.h`

`readpxl()` is called from `readfont()` to initialize all font parameters for a `.pxl` font file, and retrieve the character metrics. Since these are all available in a 2-Kbyte directory at the end of the file, minimal file activity is needed to retrieve them. `newfont()` is then called to do whatever is required to start a new font.

4.59 reldfont.h

This function was split out of `readfont.h` to separate the retrieval of font information from the DVI and font files, in order to support dynamic unloading of fonts required by the zoom option in `dvibit`. It is called from `readfont.h` and from the private function `unload_fonts()` in `eopact.h`. `openfont()` is used to find and open the font file, and then `getfntdf()` dynamically allocates a font entry containing basic character metrics for all characters in the font, but none of the potentially voluminous raster information. That portion is left in the font file for later retrieval and memory caching only when it is really needed.

If font preloading has been suppressed (this is a run-time option), then `getfntdf()` will not be invoked at startup for all fonts found in the postamble, but only dynamically as each font is required. Preloading is forced if selected pages are printed, or pages are printed in reverse order (e.g. Canon LBP-CX laser print engines without additional paper handling mechanisms). This is necessary because TEX outputs a font definition command the first time a font is used, and then never again until the postamble is written. The code in `newfont()` ensures that the device receives the proper font definition sequences, and must be executed before any character in the font can be used.

With the addition in late April, 1986, of support for the new `.gf` and `.pk` font file formats, a considerable revision of font handling became necessary, and resulted in the introduction of 11 new functions: `chargf()`, `charpk()`, `charpxl()`, `clrrow()`, `outrow()`, `readgf()`, `readpk()`, `readpxl()`, `skgfspec()`, `skpkspec()`, and `strid2()`. The goal in this was to make the source of font data known to only a small number of routines. After opening a font file successfully, `reldfont()` examines the bytes at the start of the file which uniquely identify which of the three font file types it is. It saves the font type and character retrieval function (`chargf()`, `charpk()`, or `charpxl()`) in the `font_entry` structure, then calls `readgf()`, `readpk()`, or `readpxl()` to actually read the character metric data from the file.

Should a new font file format be introduced, all that is necessary to support it is to add code to `reldfont()` to recognize it, and provide analogues of `chargf()` and `readgf()` to retrieve character rasters and metrics.

4.60 `rulepxl.h`

This short function converts a dimension of a horizontal or vertical rule measured in high-precision TEX units to output device units. It is a good candidate for macro replacement.

4.61 `setchar.h`

Apart from positioning commands, *Set Character* commands make up the bulk of the commands in the DVI file. `setchar()` is called for each of them.[16]

First of all, it determines whether or not the character is visible on the current page. If it is, then it calls `moveto()` to update the current position, and then `dispchar()` to paint the character.

Then, depending on the particular DVI file command that invoked it, it may update the current horizontal position in both TEX coordinates and device units. `fixpos()` again is used to do it right.

4.62 `setfntnm.h`

There are 64 DVI file commands which request font changes, and `setfntnm()` is called for each of them. It defines the font to be used for future printing until another *Set Font* command is encountered.

Because actual font definitions are handled by another DVI command,[17] when this function is invoked, the font is already expected to have been loaded into the cache. A sequential search[18] of the cache is made to find the font pointer for the requested font; if it is not found, `fatal()` is called to terminate the job.

4.63 `setrule.h`

When TEX sets a horizontal or vertical rule, it generates a DVI command which results in `setrule()` being called. It in turn calls `fillrect()`, then updates the horizontal positions, `h` and `hh`, relying again on `fixpos()` for help.

[16] Except in some drivers; see `setstr()` in a later section.
[17] See `getfntdf()`.
[18] See the remarks in the subsection on `openfont()`.

4.64 setstr()

TEX does not insert glue between letters of a word, only between words. Consequently, words appear in a DVI file as consecutive *Set Character* requests, possibly interspersed by *Set Font* commands. For devices on which characters must be set by the DVI driver, such as dot matrix devices, there is no advantage to be taken of this. For devices with downloaded fonts but relatively low resolution, such as the BBN BitGraph, not much can be done either, because character widths must be rounded to pixel boundaries, and within three or four characters, the cumulative string position would be incorrect.

For downloaded characters in Adobe PostScript, it is possible to define precise widths which are not rounded to pixel boundaries. Several characters can be therefore be set by a single command, reducing the output volume considerably. PostScript has been licensed by several manufacturers, but the most widely available product using it at present is the Apple LaserWriter, about which we shall later have more to say.

`setstr()` is currently used in the `dvialw`, `dvican`, `dviimp`, and `dvijep` drivers. It functions similarly to `setchar()`, except that it loops collecting consecutive characters to be set, then outputs them all as a single string. The string is interrupted and a positioning command is issued if `fixpos()` produces a different horizontal position than the device would otherwise have from its character width information. Similarly, if a character is encountered which is too large to be definable as a single character on the device, the string is interrupted, and the large character is sent as a raster image. To conserve font memory on the Apple LaserWriter, "large" characters[19] are downloaded each time they are required, bracketed by PostScript *save* and *restore* commands which cause them to be discarded after use.

Because it is quite likely that a character encountered will not yet have been downloaded, the initial version just completed the current string, downloaded the new character definition by a call to `loadchar()`, and resumed the string collection. This entails some extra output overhead, so to reduce it, the latest version has two sets of loops. The first collects the characters into an internal array, and downloads any new ones. The second loop then processes the characters from the array, outputting them as a single string. There is a little extra run-time overhead because of the second loop, but it is minor compared to the reduction in the output volume.

[19] The size limit can be set by a run-time option.

4.65 `signex.h`

This is the twin to the function `nosignex()` discussed earlier. `signex()` collects the next n bytes from the file and returns them as a single n-byte *signed* integer value, where n is in $1 \ldots 4$. Negative values are represented in *two's complement* form; implementors on one's-complement machines (CDC, Univac) must revise the code here.

4.66 `skgfspec.h`

`skgfspec()` is called by `chargf()` to skip over any METAFONT *numspecial* and *special* commands embedded in the font file. These are not currently used in the *Computer Modern* fonts.

4.67 `skipfont.h`

If fonts have been preloaded, then when `prtpage()` encounters a *Define Font* command, the font definition which immediately follows in the DVI file must be discarded. This function performs that task. If font preloading has not been selected, `prtpage()` calls `readfont()` instead.

4.68 `skpkspec.h`

`skpkspec()` is called by `charpk()` to skip over any METAFONT *numspecial* and *special* commands embedded in the font file. These are not currently used in the *Computer Modern* fonts, but `GFtoPK` will copy any it finds in the input `.gf` file to the output `.pk` file.

4.69 `special.h`

The TEX `\special` command provides a means for inserting arbitrary text into the DVI file. The intention is to allow the DVI driver to provide support for features, like insertion of bitmap images, which TEX cannot easily or efficiently offer. Such strings necessarily depend on the implementation, operating system, and output device, and therefore detract from document portability. Nevertheless, the feature can be exceedingly useful. The TEX User's Group has twice had sessions on the subject of `\special` commands with a view to coming to some consensus for what forms the command string should take, but no definite proposal has yet been put forward.

When prtpage() meets a DVI file *XXXn* command, which is what TeX produces in response to a \special, it collects the following string and passes it to special(), which has the job of interpreting it. For most dot matrix printers, providing useful support for insertion of bitmap images is at best awkward, because the usually bizarre formatting required by the device would require reconstruction of a bitmap array from the encoding, then re-encoding it for output at the current position. However, it would not be difficult to add support for vector graphics files, such as the ubiquitous Tektronix format which is both compact and relatively easy to decode. The vectors could be suitably scaled and translated, then written directly into the bitmap which dispchar() has been updating. So far, for these devices, special() just raises a warning message containing its argument string, and then returns without further action.

For the more intelligent laser printers, particularly those using the PostScript language, considerably more is possible, and the implementations of special() are still in their infancy. In dvialw.c for the Apple LaserWriter, I have brazenly copied the format used by David Kellerman and Barry Smith in their driver for the Imagen laser printers. The TeX \special command is expected to look like one of the following:

```
\special{overlay filename}             % absolute positioning

\special{include filename}             % relative positioning

\special{insert filename}              % relative positioning
```

In the first case, the PostScript file to be included will be mapped onto the page at precisely the coordinates it specifies. In the other two cases, the upper-left corner of the bounding box will be placed at the current point. The PostScript file must then contain (usually near the start) a comment of the form

```
%%BoundingBox: llx lly urx ury
```

specifying the bounding box lower-left and upper-right coordinates in standard PostScript units (big points, 1/72 inch). Alternatively, if the comment

```
%%BoundingBox: (atend)
```

is found in the file, the last 1000 characters of the file will be searched to find a comment of the form:

```
%%BoundingBox: llx lly urx ury
```

If the PostScript file cannot be opened, or the `\special` command string cannot be recognized, or for relative positioning, the bounding box cannot be determined, a warning message is issued and the `\special` command is ignored.

Otherwise, the section of the PostScript file between the comment lines

```
%begin(plot)
%end(plot)
```

is copied to the output file surrounded by

```
save
% revert to standard 1/72 inch units
300 72 div 300 72 div scale
% if relative positioning
(xcp(in 1/72in)-llx) (ycp(in 1/72in)-ury) translate
...PostScript file contents...
restore
```

This format nicely supports PostScript output from my <PLOT79 > graphics system and has received considerable use locally.

As we get more experience with PostScript, `special()` will no doubt be expanded to support fancier operations, like insertion of MacPaint and MacDraw images from the Apple Macintosh.

Adobe Systems is aware of this problem of incorporating PostScript files in other PostScript documents, and in December, 1986, published a short document *Encapsulated PostScript File Format, EPSF Version 1.2* describing conventions for doing this cleanly. Future changes to `special()` in `dvialw.c` should adopt these.

4.70 `strchr.h`

C is a sparse language, and contains no built-in functions or I/O facilities. The inventors of C defined an extensive set of library functions, but implementors since have not always adhered to the original library conventions. Besides introducing new names for the same functions, implementors have sometimes committed the even worse sin of defining the same functions, but with changed function return types. This is certainly a detriment to portability. The string support functions are one area where significant differences exist. I have therefore chosen to implement some functions which are defined in the coming ANSI Standard C library.

`strchr()` searches a string for a single character, returning a pointer to the *first* matching character, or a null pointer if the character is not found. It is permissible to search for the `NUL` character which terminates the string; the returned value minus the address of the string is the length of the string, excluding the `NUL` terminator.[20]

4.71 `strcm2.h`

`strcm2()` compares two strings, ignoring case differences, and returns a negative number if the first is less than the second, zero if they match, and otherwise a positive number. It is analogous to the usual `strcmp()` function which does not ignore case differences. Microsoft C on the IBM PC calls this function `strcmpl()`, but this violates our rule of uniqueness in the first six characters of a function name. The October 1986 draft of the proposed ANSI C Standard has no equivalent function.

4.72 `strid2.h`

`strid2()` searches for the first occurrence of a substring in another string, returning -1 if it is not found, or else its index 0, 1, ... if it is found. Letter case is *ignored*. It is used by `initglob()` in the parsing of the `FONTLIST` environment variable. The October 1986 draft of the proposed ANSI C Standard has no equivalent function.

4.73 `strrchr.h`

`strrchr()` is the counterpart to `strchr()`. It searches a string for a single character, returning a pointer to the *last* matching character. It is permissible to search for the `NUL` terminator.

4.74 `tctos.h`

`tctos()` converts a global list of TEX page counters, `tex_counter[]`, to a character string, with counters separated by dots, and trailing zeroes dropped. It is used by the `dvixxx.c` programs to format them for printing.

[20] Pointer subtraction in C is defined to return the number of objects between the two addresses, not the number of memory units.

4.75 `typedefs.h`

This file is included by `dvihead.h` after it includes `machdefs.h`. As noted earlier, it contains `typedef` declarations for `BOOLEAN`, `BYTE`, `COORDINATE`, `INT8`, `INT16`, `INT32`, `UNSIGN16`, `UNSIGN32`, and `void`.

If the implementation supports the `void` type, then the `typedef` for it is suppressed.

The symbols `TRUE` and `FALSE` are also defined for use with objects of type `BOOLEAN`.

Dennis Ritchie's original definition of C did not clearly indicate whether `unsigned` is a *distinct* integer type, or a *modifier* of the integer types `char`, `short`, `int`, and `long`. Reasonable compilers implement it as a modifier, since one can quite reasonably require unsigned values in a variety of sizes, particularly in systems programming and graphics programming, where the integers being manipulated are really bit patterns, not numbers.

Unfortunately, some compilers have chosen to make `unsigned` a distinct type meaning `unsigned int`; the Lattice C compiler on the IBM PC is one such example. This complicates programming, and special attention must be given to those parts of the code which use the `UNSIGN16` and `UNSIGN32` types. The most important example is the bitmap array used for the dot matrix printer drivers.

4.76 `usage.h`

`usage` is called from `main()` whenever `dvixxx` is invoked without arguments, and from `option()` in response to a bad argument. On the file defined by its single argument, it displays a short message illustrating the expected command line, followed by a suggestion of where to find detailed documentation (set by the preprocessor constant `HELPCMD` defined in `machdefs.h`, and possibly overridden at run time by the environment variable `DVIHELP`), then returns to its caller.

4.77 `warning.h`

When situations arise which warrant the issuing of a warning message, such as a font substitution performed by `fontsub()`, `warning()` is called. If no log file is open, it tries to open one. If this succeeds, the message is written to the log file, possibly adorned by additional text. The name of the function issuing the call to `warning()` is included as part of the message.

For example, the TOPS-20 operating system uses the convention that text written to the primary output file (terminal, or batch job log file) which begins with a percent sign is a warning, and text which begins with a query is an error message. The batch controller recognizes these and raises an error flag if a query is written first. These two characters are usually sufficiently rare in text files that they make convenient search targets when one is examining a log file with a text editor or string search utility. Consequently, when the symbols `OS_TOPS20` or `OS_VAXVMS` are defined, `warning()` and `fatal()` prefix these characters to the message.

5 DVI Driver Device Support

The preceding sections have given a comprehensive survey of the structure of the DVI drivers. In the following subsections, we shall briefly discuss each of the currently-supported devices, mostly in alphabetical order by filename.

If you intend to implement a driver for a new device, you can profit by choosing the existing driver which most closely resembles your target device, then modifying it appropriately. Please send your implementation back to this author for inclusion in the collection.

You can quickly get an idea of just which functions are likely to require modification by using a string search utility to find processor conditionals "`#if`" in the source files. A listing will not be given here, because it would be difficult to keep up-to-date. You can similarly find operating system dependencies by searching for the string "`OS_`", and implementation dependencies by searching for any of the flags discussed earlier in the section on `machdefs.h`.

Unix makes it easy to combine programs so that the output of one becomes the input of the other, with both running simultaneously. Consequently, many Unix programs are designed to read their input from `stdin` and write their normal output to `stdout`, and error messages to `stderr`. These three files are automatically open when a process begins to execute. PCC-20 supports this model as well. I therefore had to make a design choice of whether the DVI drivers would adopt this model or not. In a pure Unix environment, this would be quite satisfactory, and conform well to the notion of a *filter* program. Unix users can quite reasonably expect to issue a command like

```
% tex myfile | dvixxx | xxxspool
```

to have the file `myfile.tex` processed by TeX, translated by `dvixxx`, and spooled to the output device by `xxxspool`, all in one concurrent step.

I chose not to use this model, because on the DEC-20, where all my development work is done, the output of the `dvixxx` program may not be a standard text file. Since PCC-20 supports only 7-bit I/O for `stdin`, `stdout`, and `stderr`, while the output may require an 8-bit file, it is necessary to create an intermediate disk file. The convention adopted is that the suffix of the driver name becomes a suffix of the DVI file name. Thus, on the DEC-20 or under Unix, `myfile.tex` is transformed by TeX to `myfile.dvi`, then by `dvixxx` to `myfile.dvi-xxx`. For operating systems which do not permit long file extensions, the driver may output it as `myfile.xxx`. This has proved to be a wise choice even in the 8-bit world, where record-structured file systems like VAX VMS conflict with the byte-stream output of `stdout`.

5.1 `dvialw.c`

This driver for the Apple LaserWriter is the most recent addition to the family. It was developed starting from the BBN BitGraph driver, `dvibit`, since this was the only other driver with downloaded font support at the time.

Adobe PostScript is a complete programming language designed for the support of intelligent printer drivers. Like the popular[21] Forth language, it is written in reverse Polish notation, which some claim is easier to parse because operands can be simply stacked until an operation is found, which then pops its operands off the stack, performs its job, and places the result, if any, back on the stack.

Reverse Polish notation is certainly not easy to read without extensive practice. For example, to set a string at a particular page position, one writes

```
250 300 moveto
(Hello there) show
```

This example shows the main problem with PostScript — it is verbose. The compact Tektronix vector format would encode the first command in 5 or fewer characters, instead of 14. Perhaps someday there will be efficient PostScript compilers. In the meantime, one can only try to reduce the PostScript volume by definition of compact macros; `dvialw` produces

[21] in some circles

```
(Hello there)250 300 S
```

for the above. A space after the right parenthesis is discarded, and the two command words are collapsed into a single letter.

This verbosity drastically reduces the potential performance of the Apple LaserWriter. The Canon print engine is capable of printing 8 pages per minute, and the Imagen 8/300 which uses it too will output TeX at 7 pages per minute. The best we have been able to do on the Apple LaserWriter is about 2 per minute, and that represents an average serial transmission rate of about 750 characters per second, close to the limit of 960 at the 9600 baud limit. The printer may perhaps run faster with the 29 Kbaud AppleTalk connection.

PostScript timing experiments showed that the Apple LaserWriter performs about one arithmetic operation per millisecond, and further experiments revealed the fact that the arithmetic conforms to a single-precision subset of the IEEE floating-point format. This is not surprising, because Jerome Coonan, one of the principal designers of the IEEE standard, works for Apple and did the software for the floating-point emulator for the Apple Macintosh. The printer and the computer both use the Motorola 68000 microprocessor, and neither have floating-point hardware. Rumors of an advance to the faster 68020 CPU and 68881 hardware floating-point chip with the newly released Enhanced Apple LaserWriter were wrong; it only has more internal fonts and more memory.

I bought an Apple LaserWriter in June, 1985, and although I was able to prepare a device driver for my <PLOT79 > graphics systems and have it operational in under a day, the TeX DVI driver has proved much more difficult. This is due to two reasons. First of all, the documentation and examples in the *PostScript User's Manual* (September 1984 printing) for downloaded fonts are incorrect; the examples simply do not work at all. Second, the PostScript Version 23.0 interpreter has two serious bugs.

One of these is a flow-control bug which is documented in the Apple *Inside the LaserWriter* manual—the printer on occasion will send out an `XOFF` character to stop host transmission, then never send the `XON` character to resume it. Eventually, the printer times out and aborts the job after issuing an error message. Initially, this seemed to happen on almost every page, and it took several weeks of experimentation during the summer before I was able to write a transfer program which reduced the incidence. In December, 1985, I obtained a proposed fix for this bug from ARPANET correspondence, and it seems to have helped, although the problem still does arise.

The second bug has to do with virtual memory management, which comes into play primarily with downloaded fonts, which neither the Apple Macintosh nor graphics applications make use of. At random intervals, the printer issues a *VM error* message and flushes the job. It is usually completely irreproducible; the same document will raise the error twice, and print on the third try. The longer the printer has been running, the slower it seems to get, until cycling the power reboots it. We have often found it necessary to do this several times a day, hardly a satisfactory situation for a printer which is expected to be a spooled output device.

Because of the flow control bug, and because the printer sends back all kinds of messages, such as *%%[PrinterError: out of paper]%%*, it is necessary to have an intelligent file copy utility for it. Some people have gone to the trouble of handling `XON` and `XOFF` characters themselves, establishing a real-time interrupt when an `XOFF` is received, and then resuming output either after receipt of an `XON`, or when the timer interrupts. My copy utility, `lw78.c`, lets the operating system handle flow control, but carries on a dialog with the printer before starting each job to make sure it is idle. It periodically monitors the printer input stream to see if any message is waiting, and if so, collects it, decodes it, and attempts to handle the commoner ones. Status messages do not always arrive complete, perhaps because of data loss at the serial interface on the host computer. Consequently, the copy utility cannot just collect two percents and an open bracket to start a message, then gobble characters until a close bracket and two more percents terminate it, because this might block the process completely. Instead, it alternates periods of sleep and tests for input data until about ten seconds have elapsed without receiving a complete message. At that point, the message is assumed to be garbled and a flag is set which causes later input attempts to flush data until a new start of message is recognized. This is clearly nothing but hackery; I am not proud of it, and it wasted weeks of work which could have been much more profitably spent.

The PostScript font mechanism is complex and difficult to understand, and much better documentation and many examples are badly needed in the manuals. It centers around the idea that in the current font, for each character there is a collection of numbers defining the bounding box and displacement to the next character, plus a procedure which is invoked to draw the character whenever the character appears in a *Set String* command. This is a powerful mechanism which is also fundamental to the Emacs text editor. The character procedure is free to utilize the entire language to create a picture at the current point. The built-in fonts use proprietary outline encoding mechanisms which provide arbitrary scaling. For TEX, we just send character bitmaps down encoded as hexadecimal strings.

After all of the work on `dvialw` had been completed, Addison-Wesley published the *PostScript Language Reference Manual* and the *PostScript Language Tutorial and Cookbook* in the spring of 1986. These offer much improved documentation, and remove my criticisms of that aspect. The font examples in the *Tutorial and Cookbook* now work as advertised, and the problems with the Apple LaserWriter are discussed in Appendix D of the *Language Reference Manual*. These two books are essential for any PostScript programmer.

To facilitate a certain amount of extensibility of the TEX PostScript driver, macro references, rather than raw PostScript, are written to the output file. The macros are defined in a standard file named `dvialw.ps` which is searched for in the current file directory, then in a standard system directory defined by the preprocessor symbol `TEXINPUTS`. Like the font substitution file, the definition of this symbol may be overridden by the user at run time, allowing personal customizations. The standard `dvialw.ps` file which goes with this driver is heavily commented to assist customization. The comments, and excess whitespace, are all stripped by function `cppsfile()` during the copy to the beginning of the output file, so they never have to be transferred to the printer. The driver is careful never to produce any unnecessary whitespace, in order to reduce the output volume. Even so, on large documents, the PostScript file is usually 4 to 6 times larger than the DVI file, and on short ones, up to 20 times longer.

`dvialw.c` supports the TEX `\special` command to a limited extent; see the section on function `special()` for details.

While it is in principle possible for `dvialw.ps` to support PostScript built-in fonts, this would require preparation of TEX font metric (`.tfm`) files for them. TEX needs this information to use a font. At present, there is insufficient font metric information available for the built-in fonts, and the character repertoire is inadequate for mathematical typesetting. Some members of the Stanford TEX project have been consulting with Adobe, so we may hope that this situation will improve.

For non-TEX printing on the Apple LaserWriter, we have used a utility program I wrote, `lptops.c`, to convert line printer files to PostScript. It has options to access all of the built-in fonts at arbitrary sizes, as well as support for multi-column printing, page numbering, overstriking, margin setting, multiple copies, page outlining, and manual paper feed selection.

The help file, `lptops.hlp`, describes these further.

The Enhanced Apple LaserWriter was announced in February 1986, at a list price of $7995. Ordinary LaserWriter's can be upgraded to it for about $1000. This new model contains version 38.0 of the PostScript interpreter, and offers several new features:

- many internal performance improvements;
- expanded communications options, including baud rates to 57.6 Kbaud and hardware flow control; this is important for the IBM PC, which without additional software does not support software flow control;
- several new language features;
- correction of nearly all the known problems in the Apple LaserWriter, including the flow-control bug;
- new resident fonts: AvantGarde, Bookman, Helvetica-Narrow, NewCenturySchoolBook, Palatino, ZapfChancery, and ZapfDingbats;
- additional memory

One of the important performance improvements is the overlap of page output with input processing; previously, the PostScript interpreter waited up to 6 seconds per page before resuming input processing. Initial user experience with this printer is reported to be quite favorable; I personally have not yet had any.

Adobe has recently made available accurate font metric and ligature information for the resident fonts. Kerning specifications are not yet available, but have been promised. Unfortunately, this information is not available in the new PostScript books, but font metric files are distributed with Adobe's TranScript software for Unix systems, and probably also with Pipeline Associates' devps competing software. This should make it possible to augment `dvialw` to support use of resident PostScript fonts with TEX. This will be even more important for the Mergenthaler Linotype Linotronic 300, which is the first phototypesetter to use PostScript.

5.2 `dvibit.c`

This driver is the first in the family, but has undergone the same extensive modifications as the rest of the members. It is the only one which provides a form of TEX preview mechanism, but we unfortunately have only one BBN

BitGraph terminal on campus, so it has not received the use I would have liked to see. With laser printers increasingly available, the need has declined. Although the display speed is satisfactory, the resolution is coarse (1024×768), and small characters are often indistinguishable. TEX previewers on the IBM PC have an even more severe limitation of half this resolution.

In order to reduce the number of fonts required, the default size with Mark Senn's original version has been changed to 603 magnification. This is in the standard magnification family for 300-dot/inch devices (magstep -5). Since we have METAFONT on the DEC-20, production of arbitrary sizes for all of the TEX fonts has been trivial; only disk space requirements limit our enthusiasm.

In order to write commands to the BitGraph, it is necessary to be able to send ASCII `ESC` characters, and to make the display easily controllable, character-at-a-time input without echo is desirable. Under Unix systems, this is straightforward—you just put the terminal in *half-cooked* or *raw* mode, and twiddle a couple of `ioctl()` flags. On the DEC-20, it is quite possible, but somewhat convoluted, so there are large code sections under control of the `OS_TOPS20` flag. When the driver exits, care has to be taken to reset the terminal characteristics properly.

With Version 2.07 of the family, a major change was made in `dvibit`. Keyboard input with `getc()` has been replaced with a more powerful package, `keybrd.c`, described earlier. This package supports immediate return of characters as they are typed, and also provides a primitive for checking if input is available, without blocking like `getc()` normally does. This makes it possible for the main command loop in `prtpage.h` to check for keyboard input before processing each DVI command, which in turn avoids unnecessary screen redisplay, and facilitates rapid skimming through a document, as well as rapid page repositioning.

In addition, the font handling mechanism was revised with the introduction of `reldfont.h`, so that `unload_fonts()` in `eopact.h` can dynamically change the font magnification to support a zoom command.

Drawing of the command and status window is now done at page beginning by `bopact.h`, and the display has been made larger and more informative.

The size limit at which downloaded fonts are abandoned in favor of sending large character bitmaps each time they are needed has been substantially increased, so that any reasonable document and magnification should be able to use downloaded fonts exclusively, giving much faster screen display.

The display now works correctly with both black and white screen back-grounds (they are selectable in terminal setup mode); previously, large characters were displayed incorrectly on a black background.

5.3 `dvican.c`

This driver is for the Canon LBP-8 A2 laser printer, which uses Canon's own controller with their LBP-CX engine. The LBP-8 A2 supports 300 dot/inch raster graphics, vector graphics, and, according to its documentation, downloaded fonts.

After seeing this printer offered at under $3000 from a local vendor, I obtained one on trial in late December, 1986, expecting to have a graphics interface and a DVI driver completed before the New Year. To make a long story short, in despair, it was returned to the dealer in March, 1987.

The documentation for the printer is abysmal, and it was only through the help of a local laser printer development firm that I got sufficient detail on the printer to write any software at all for it. Eventually, I obtained a Subsystem Manual from Canon headquarters in New York, but this seemed to only confirm that what `dvican` is doing is correct according to the documentation. Downloaded fonts just do not work properly, and to get any TeX output, it is necessary to send raster bitmaps for each character, which drastically increases the output file size, and slows the printer to about 1 page every 3 or 4 minutes.

Correspondence with a site in Holland that had obtained another DVI driver for this printer from Japan resulted in their sending me a hexadecimal dump of the DVI output for the standard TeX sample file, `story.tex`. To my surprise, it too handles characters as bitmaps, instead of using downloaded fonts. This tends to confirm my experience.

I have spent a great many frustrating hours with this printer; if anyone reading this has used the printer successfully with downloaded fonts, I would like to hear about it. Otherwise, I am inclined to blacklist it, and exclude it from further consideration.

5.4 `dvigd.c`

This driver was adapted for the *Golden Dawn Golden Laser 100* printer developed by a local Salt Lake City firm. This printer has not reached the commercial market, so the driver will not be of interest outside our local installation.

5.5 dviimp.c

This driver for the Imagen laser printer family was produced by Lon Willett at Utah, starting from `dvijep.c`, using the Stanford University Imagen driver in Web as a guide. It has been working quite satisfactorily with both our Imagen 8/300 and 3320 printers on DEC-20 and VAX VMS. The major work left to do is the addition of support for `\special` commands.

5.6 dvijet.c

The Hewlett-Packard LaserJet printer has taken the low-cost page printer market by storm, and was claimed in one trade magazine survey to be the largest selling printer of all time. Like the Apple LaserWriter and the Imagen 8/300, and several others on the market, it uses the 8 page/minute Canon print engine. It appeared on the market in the late summer of 1984 at a list price of $3495, and I got one in late November. There is a built-in Courier font, and additional fonts are provided by plug-in cartridges. There is no downloadable font capability, but fortunately, there is support for bitmap display, so it is treated very much like a dot matrix printer by the `dvijet.c` TeX driver. It has only about 55 Kbytes of raster memory, far less than the 900 Kbytes needed for the printable portion of a standard American page. Fortunately, it is possible to request the printing of a bitmap at 300, 150, 100, or 75 dots/inch, and the latter size is close to a full page. Each bit is imaged as a square 1 to 4 dots on edge for these resolutions.

In the fall of 1985, the LaserJet Plus was announced at a list price of $3995 and the regular LaserJet was reduced to $3495. The latter can be upgraded to the Plus, which has quite decent support for downloaded fonts and larger bitmaps (but still limited—to 512 Kbytes). In late March, 1986, an Enhanced LaserJet Plus, the HP 2686d, was announced. It has a list price of $4995 and offers two 250-sheet paper trays and in-order output stacking, but apparently still lacks sufficient memory for a full page bitmap.

This printer has been aggressively marketed, and universities have seen discounts of 30% or more. It could well become the most common TeX output device.

5.7 dvijep.c

I was able to secure the loan of a Hewlett-Packard LaserJet Plus printer in late March, 1986, for the development of a DVI driver, and `dvijep` is the result of that work. I lost about a day's work fiddling with trying to get downloaded characters to work, because of unclear documentation in the reference manual, but apart from that, conversion of `dvialw` to `dvijep` took less than a day. `dvialw` was chosen as the starting point, rather than `dvijet`, because it supports downloadable characters, and the translation of PostScript sequences required by `dvialw` to HP printer sequences seemed straightforward. Some of the improvements made in `dvijep` for output compression, including not outputting unchanged rule sizes, and remembering the current string y coordinate in a global variable for `setstr()`, have been carried back into `dvialw`.

The printer language used to control the Hewlett-Packard printers consists of escape sequences, most of which require a three-character prefix, and a single upper-case letter terminator, possibly separated by a digit string. For example, to move to a particular (x,y) coordinate position, one sends something like

```
<ESC>p*1234X
<ESC>p*3214Y
```

and to draw a solid-filled rule 200 dots wide by 50 dots high, with its upper-left corner at the current point, one sends

```
<ESC>*c200A
<ESC>*c50B
<ESC>*c0P
```

Characters outside of escape sequences are simply set in the current font, so in these examples, the sequences would be run together, for otherwise the newlines between them would generate blank lines in the output. Signed numbers can be specified for coordinates to get relative moves, instead of absolute moves.

Consecutive control sequences with common three-character prefixes can be collapsed by omitting the prefix on all but the first, and changing all but the last terminator to lower-case. In addition, if a digit string is zero, it may be omitted entirely. Thus, the above examples can be reduced to

```
<ESC>p*1234x3214Y
```

and

```
<ESC>*c200a50bP
```

a savings of almost half. `dvijep` takes great care to use these compact forms in order to decrease the output size, and the reduction has proved quite significant.

To avoid scattering these control sequences throughout the various `.h` files, they are entirely encapsulated in preprocessor macros defined at the beginning of `dvijep.c`. This proved valuable in making optimizations of them.

There is absolutely no error indication if a control sequence is bad—it is simply discarded by the printer. In the case of downloaded fonts, this means that characters whose downloaded description is incorrect, or garbled, will either be missing from the output, or will appear, but in a different font. This features makes debugging at times frustrating, since the printer cannot tell you what is wrong. It would be nice to have a mode wherein you could ask it to *print* error messages on the output page. Except for font downloading, most of the sequences are simple, and cause little difficulty.

Communicating with the printer requires nothing more than obeying its `XON/XOFF` flow-control protocol, so standard operating system `copy` commands usually suffice, and the pain and suffering I went through with `lw78` for the Apple LaserWriter did not have to be repeated.

Character raster descriptions are sent as 8-bit binary data, and therefore require half the output volume that PostScript needs. With some large test DVI files, the `dvijep` output is about 30% shorter than the `dvialw` output, and timing tests with 9600-baud communication show that the LaserJet Plus will output TEX at 3 to 3.5 pages per minute. Once again, the serial communication seems to be the limiting factor, and it may be that with 19200 baud, or with the Centronix parallel interface, it may be able to produce 6 or 7 per minute. The Canon LBP-CX engine is capable of 8 pages per minute.

Only the dummy `special()` function is presently used in `dvijep`. Support for insertion of raster images should not be difficult to add. As with the regular LaserJet, images can be displayed at 75, 100, 150, or 300 dots/inch, and the extra memory of the Plus permits a full page at 150 dots/inch, or a 5-inch square at 300 dots/inch.

Besides having support for larger bitmaps and downloaded fonts, the Plus adds one useful extension over the regular LASERJET. All coordinates can be specified in dots, as well as the decipoints used by the latter. This makes it easy to position characters precisely, without worrying about conversions of units of 1/720 inch to ones of 1/300 inch.

The printer permits origin resetting, but only to positions on the page, because positive coordinates are required. Experimenting with the output positioning showed that the upper-left page corner was actually at (0.1in, -0.07in) on the printer I have. Consequently, two new preprocessor symbols, `XORIGIN` and `YORIGIN`, have been introduced in `dvijep` which are used in `dviinit()` to bias the left and top margin values specified by default, or by command-line options.

Some remarks about the downloaded font capability are in order here. The Plus identifies fonts by numbers in 0 ... 32767, and fonts may be defined in any order and with any font numbers, provided that no more than 32 are resident in the printer at one time, and further, that no more than 16 are used on one page. It is not at all obvious why this page limitation should exist. Code has been inserted in `readfont()` to check for the possibility of more than 32 being defined; if this happens, it issues a warning indicating that the current page may be incomplete, and then deletes the last 16 and frees their font space. So far, this message has not been issued. No check is made to see if more than 16 fonts are in use on a single page, but this would not be hard to add. I am inclined to leave the changes until, and if, a problem arises from this limitation.

So far, only one document has been encountered locally which reached this limit. The printer then imaged some characters as blanks, and used the wrong font for others. The author decided to accept the printer limitation, and revised his document to reduce the number of fonts per page.

Fonts can be of two types—7-bit fonts, allowing characters 33 ... 127 to be downloaded, and 8-bit fonts, allowing characters 33 ... 127 and 160 ... 255 to be defined. The latter require double the internal storage. The size limitation on the characters is the same in both—255 dots high or wide; only the number of characters is different. TEX fonts use all characters in 0 ... 127,[22] so in order to keep the font mapping one-to-one, a macro `MAPCHAR` is defined which remaps TEX characters in 0 ... 32 onto 160 ... 192, so that this peculiarity is hidden from the code in `loadchar()`, `setchar()`, and `setstr()`.

The descriptions of the *Create Font* command in the *Hewlett-Packard LaserJet Printer Technical Reference Manual* caused me considerably confusion and the loss of time noted earlier. The discussion and examples are oriented toward using the command to link cartridge-resident fonts with a user-defined numbered font, and the command has fields for *font type*, *symbol set*, *pitch*, *style*, *stroke weight*, and *typeface*, which are attributes settable

[22]The DVI file allows 0 ... 255.

with other printer commands which the LaserJet somehow (i.e. completely non-intuitively) relates to particular fonts. This is the only way fonts can be selected on the regular LaserJet, and it has previously cost me a good deal of frustration, since it is impossible to just tell it "give me 10pt Courier bold". When my tests at downloading a simple character failed, I assumed that the problem was with the values I had chosen for all of these attributes, and I spent several hours experimenting with them, each time getting blank output pages.

The problem turned out to be none of these parameters, and their values are apparently arbitrary when one is downloading character descriptions. The *Create Font* command has four 16-bit fields called *cell width*, *cell height*, *baseline position*, and *font height*. Since the *Download Character* command defines its own character metrics, I had arbitrarily given these fields zero values. Later experiments with these revealed that *font height* is arbitrary, but the other three must be defined large enough to encompass the biggest enclosing box of any downloaded character. Downloaded characters with smaller metrics will print, but the larger ones are just discarded without error indication. The solution therefore was to have `readfont()` compute the requisite values for these from the metrics it retrieves from the font file, and output these in the *Create Font* command. I presume that the *cell height* and *cell width* parameters are used by the Plus for internal memory allocation, although there is no reason why it could not do this on receipt of the sizes specified in the *Download Character* commands. With sophisticated fonts used in typesetting, there is no reason whatever to assume any commonality of character sizes within a font, and neither is there any reason to store more than just the minimal bitmap for each character.

The *Download Character* instruction contains some fixed magic bytes, followed by *left offset*, *top offset*, *character width*, *character height*, *delta* x, and finally the binary character bitmap as a byte stream, each raster line being rounded up to the next higher byte boundary. The *left offset* is the distance from the character reference point (the same one TeX calls the character position) to the leftmost edge of the character bitmap; this is just the negative of the `xoffp` parameter in the TeX font file. The *top offset* is the distance from the reference point to the top of the character pattern, and is identical to the `yoffp` parameter in the font file. Although both of these are specified as 16-bit numbers in the *Download Character* instruction, they are apparently stored internally as one-byte signed values, because they are restricted to the range -128 ... 127. `loadchar()` enforces these limits when it issues the download sequence, and `readfont()` prints a warning message when it finds any character which violates this restriction. Eventually, such characters should be sent as an ordinary raster image instead. The *delta x* value is four times the (possibly fractional) dot spacing to the next character reference point, and therefore is equivalent, except for scale and precision, to the TeX `tfm` character width. At present, no advantage is taken in `setstr()` of the higher precision of this spacing parameter; when `fixpos()` changes the horizontal position, a new string is started.

The LaserJet supports landscape page orientation as well as portrait orientation, but unlike PostScript which requires only a very simple transformation matrix to get landscape mode, on the Plus, font definitions must be completely changed, and raster images (including character bitmaps) sent with vertical rasters, since these will be parallel to the engine scanning direction. This is a considerable inconvenience, with all the complications and host overhead noted earlier in the section on `makechar()`, so `dvijet` does not yet offer this option.

A final remark should be made here about the compactness of the output file. The *Create Font* and *Download Character* commands are encoded about as compactly as could be expected, although support for run-length encoding of raster images would be valuable. Once most of the required characters have been downloaded in the early pages of the document, the remainder of the file looks something like the following, taken from an actual test file:

```
<ESC>*p323x1584Yplicit            <--new line
<ESC>*p434Xexcitation
<ESC>*p631Xof
<ESC>*p683Xeac
<ESC>*p740Xh
<ESC>*p782Xquasi-normal
<ESC>*p1035Xmo
<ESC>*p1092Xde.
<ESC>*p1178XP
<ESC>*p1206Xo
<ESC>*p1226Xw
<ESC>*p1255Xer-la
<ESC>*p1335Xw
<ESC>*p1382Xtails
<ESC>*p1477Xpro
<ESC>*p1538Xduced
```

```
<ESC>*p1663Xb
<ESC>*p1686Xy
<ESC>*p1725Xthe
<ESC>*p1802Xv
<ESC>*p1823Xarious
<ESC>*p1949Xsources
<ESC>*p323x1634Yare                 <--new line
<ESC>*p394Xdiscussed.
<ESC>*p596XIn
<ESC>*p649Xparticular,
```

Newlines have been inserted after each escape sequence to make them readable. Otherwise, the entire output file is one single stream of 8-bit bytes. Note that at the beginning of each new line, marked by the arrows in the example, there is a y coordinate specified, but for the remainder of the line, only x coordinates are given. When words are broken, it is usually an indication that `fixpos()` found a position discrepancy, so `setstr()` began a new output string. It is evident that in these cases, and between words, there is a typical overhead of eight extra characters. In fact, for this example, there are 314 total characters, of which 208 are in escape sequences, and the other 106 are to be displayed on the typeset page. This means that about 2/3 of the file consists of overhead characters, and if this proportion could be reduced, the total savings, and speed-up in printing, would be substantial.

Some repositioning is of course necessary at the start of each word, and using relative coordinates would reduce each sequence by one character, giving 16 fewer characters in this 16-word example, which is not a large saving.

It would therefore appear valuable to define some of the currently unused font positions 193 ... 255 in each font as spaces of various sizes, then output the minimal number of these to obtain the required spacing. Since the interword spacing required is rarely more than 50 dots, or 200 quarter dots in the units defined for the *delta x* character parameter, if ten space characters of widths of 1, 2, 4, ..., 1024 quarter dots were defined, then by simple bit shifting of the required horizontal space value, we could output the required space with probably an average of three, instead of the eight characters in the above example. Care will of course have to be given to getting this precisely correct, because the danger of relative spacing is always that the small errors can accumulate to large ones, while using absolute coordinates for each word will repair small errors made in individual character positions.

5.8 `dvil75.c`

This driver is for the DEC LA75 dot matrix printer in 144-dot/inch mode. It was contributed by John Sauter, and has not been tested locally.

5.9 `dvim72.c`

This driver is for the Apple ImageWriter dot matrix printer in 72-dot/inch mode. This printer has the same command set as the popular Epson printers, so the driver should work with both. Because these are low-cost printers, paper registration is poor, and high quality cannot be expected. Each pass of the print head may lay down 8 raster lines, but when the next group of 8 is printed after scrolling the paper, a thin white space is commonly seen. Expect times of 2 to 6 minutes per page.

5.10 `dvimac.c`

This driver is for the Apple ImageWriter dot matrix printer in 144-dot/inch mode. Except for code changes to double the printer bitmap size and to support the encoding of the higher resolution, it is identical to `dvim72`. The higher resolution is achieved by encoding a 16-bit raster column into two 8-bit bytes, odd rows into the first byte, and even into the second. All odd bytes are then sent to the printer and printed, the carriage scrolls the paper up one dot, and then the even bytes are sent and printed. This suffers from even worse registration problems than the 72-dot/inch mode, and since it requires four times the data volume, it is hardly worthwhile. It may prove to be of better quality with the ImageWriter Plus, but I have not tried it.

5.11 `dvimpi.c`

`dvimpi` is the DVI translator for the portable MPI Sprinter dot matrix printer, which was made here in Salt Lake City. Despite considerable popularity, low cost, and typewriter-like portability of the Sprinter, MPI went out of business in late 1985.

The driver can be compiled for either 72- or 144-dot/inch printing, according to the preprocessor variable `HIRES` in `dvimpi.c`. The bitmap encoding is a 6-bit column per character, and in 144-dot/inch mode, printing takes four passes with intervening half-dot right and down motions.

The Sprinter has no reset command, so it is not possible to restore it to a guaranteed state on completion.

5.12 dvio72.c

`dvio72` is for the Okidata Pacemark 2410 dot matrix printer at 72 dots/inch. Along with the more expensive Printronix printers, this is one of the sturdiest dot matrix printers on the market, and has been quite popular as a replacement for 200 to 600 line/minute band and drum printers.

The bitmap encoding puts a 7-bit column into an 8-bit byte, but no bias is applied, so all 128 ASCII byte values are possible graphics data. This poses a problem discussed in the next paragraph, and necessitates a second pass by function `outline()` through the encoded bitmap data before it can be output. There is some problem in getting the print head to return to the same left margin position after each pass of the heads across the page, so the driver sends several carriage-return characters at the end of each line.

The printer has a serious design flaw in that it lacks a command to reset to power-on state. This is serious, because it uses `ETX` to enter graphics mode, `ETX STX` to exit graphics mode, and `ETX ETX` to mean `ETX` as graphics data. This means that a single `ETX` sent from the host may cause it to enter graphics mode, or generate an error if it is already in graphics mode. Similarly, an `ETX STX` will enter graphics mode if it is not already in it, instead of exiting graphics mode. We therefore turn on graphics mode for each line, then turn it off again. That way, there may be some remote hope of resynchronizing if we lose an `ETX` due to line errors along the way.

From an efficiency standpoint, it is regrettable that it has no run-length encoding of graphics data. The output quality and speed are otherwise quite reasonable as medium-cost dot matrix printers go.

5.13 dvioki.c

`dvioki` supports the Okidata Pacemark 2410 in 144-dot/inch mode, but is otherwise very similar to `dvio72`. This printer does not require awkward encoding of raster lines for high-resolution mode like the Apple ImageWriter and MPI Sprinter do, and printing is not done by half-line overlap, so the quality is higher. The printer takes care of whatever bit rearrangements are necessary; the bitmap is just four times as big.

5.14 dviprx.c

`dviprx` is the second member of this family, and has formed the basis for all the other dot matrix printer development. The Printronix printer family is one of the sturdiest on the market. It has been emulated by C-Itoh, and remarketed by many computer manufacturers under their own names. It has been on the market for at least ten years, and one outside my door has been in regular use since 1978. List price, depending on speeds of 200 to 600 lines/minute, ranges from $3000 to $7000.

Almost all the other dot matrix printers on the market have a moving head of vertically-aligned pins which impact the ribbon to print a dot. One firing of all the pins in the head prints one column of dots, and this operation must be repeated from 72 to 180 times per inch, depending on the printer resolution, with incremental head movement each time. Uniform print registration on successive passes is directly related to the cost of the printer, and most of the low-cost ones do a poor job.

The Printronix mechanism has a shuttle of 44 horizontal hammer banks (in the 600 model; the number is model-dependent). Oscillation of the shuttle carries each hammer bank across only 3 character positions to obtain standard 132 character/line printing, with a resolution of 60 dots/inch. In graphics mode, one line of data bytes with 6 data bits per character, biased by 64, produces one raster line. The paper then moves up 1/72 inch, and the next raster line is printed. Since much less mechanical movement is involved, speed and print quality are much better than with the other types of printers. The primary complaint is that it results in a non-unit aspect ratio of 60/72 = 5/6 which TeX users must put up with. I have not felt it worthwhile to ask METAFONT to generate special fonts to compensate for this. As noted earlier in the section on `prtbmap()`, the bit encoding is backwards, so a table-lookup translation of each output byte is necessary.

A modification of the Printronix mechanism for 100-dot/inch printing with unit aspect ratio was licensed by Trilog and marketed. One model could support color printing with a special multicolor ribbon, using three complete passes over the page. Unfortunately, the higher resolution came with a considerably higher price and speed of under 200 lines/minute, so I do not believe this was a great market success.

5.15 `dvitos.c`

Several Japanese printer companies, including Toshiba, NEC, Fujitsu, and Epson, and one American company, AMT, have developed 24-pin dot matrix printers capable of 180-dot/inch printing. The impetus for this came from the need to support the Japanese Kanji, Katakana, and Hierogana symbol sets. The JISCII (Japanese Industrial Standard Code for Information Interchange) character set is based on a 128 by 128 character matrix, containing ISO/ASCII in the first row, Greek and Cyrillic in the next rows, and about 6000 characters divided into two levels according to frequency of use in the remaining rows. Two ASCII characters then can effectively represent one Japanese character. These characters can be quite complex, but the Japan Standards Institute has developed a collection of about 10000 characters which can all be represented in a 24 by 24 matrix. This size led to the pin configuration of these high-resolution dot matrix printers.

As a side effect, 180 dots/inch is a great improvement over the resolution of most other dot matrix printers, and produces print quality which is not far from that of daisy-wheel printers, but at higher speed and with more versatility.

These printers hit the market in late 1983, with the TOSHIBA P-1351 being one of the first. At that time, electrostatic printers were still expensive, and printed on unpleasant paper, and laser printers cost \$20,000 and up. Although the existence of a new low-cost print engine with a disposable drum cartridge from Canon was rumored, there was nothing else on the market of comparable resolution, so I bought a P-1351 about February 1984 for \$1950 (after academic discount).

The Hewlett-Packard LaserJet was announced only three months later, and I was able to get one delivered in late November. It appeared then that the superior speed (8 pages/minute versus perhaps a half-page/minute), print quality, and quietness of the laser printer would be attractive, and despite a cost of at least \$1500 more, the business community at least might buy it. The enormous market success of low-cost laser printers has proven this view correct, and the 24-pin printers clearly have an extremely limited market life.

Nevertheless, if you happen to own a Toshiba P-1351, `dvitos` will produce TeX output on it for you.

The initial comments in `dvitos.c` give some performance statistics. At 180 dots/inch, a reasonable size page requires over 200 Kbytes for the bitmap. The P-1351 supports run-length encoding which can reduce the output volume somewhat, but in practice, it runs 2 to 3 times slower, and produces terrible print quality due to inaccurate positioning of the print head during horizontal skips. I do not know whether this is a mechanical problem with my particular printer, or symptomatic of all Toshiba printers in this class.

The bitmap encoding puts a 24-bit column into 6 bits each of 4 successive bytes, biased by 64. About 120 lines of code are required in `prtbmap()` to handle this. The printer has the additional restriction that in any two adjacent bit columns, no two horizontally-adjacent bits can be set. Rather than deal with this complication during the encoding, `prtbmap()` ignores the rule and hands each byte stream representing 24 raster lines off to `outline()`, which makes a second pass through it, turning off adjacent bits, before sending it to the output file.

Run-length encoding is selectable by a run-time option, and is handled by `outline()` in a further 55 lines of code. The printer supports only encoding of successive whitespace, and since at least 12 consecutive 24-dot columns must be white for the encoding to save space, the chances of this happening are not large. Horizontal raster line encoding of both blank and white would do much better. The space reduction on two test files was less than 30%, with print times of 1 to 6 minutes per page.

At hardly double the cost, laser printers give much higher quality, silent printing, up to 40 times the speed, and twice the resolution, so these dot-matrix printers do not fare well as TEX output devices.

6 Setting up the Environment

Someone setting up TEX for the first time is likely to find it difficult to get an overview of just how things should be set up. A TEX distribution tape is likely to fill about 30Mb of disk space with thousands of files in dozens of directories, and in order for the software to work properly and find its way around the file system, some naming conventions are necessary. To illustrate this, the layout we use on the IBM PC , TOPS-20, and VAX VMS, is sketched in the next three subsections.

6.1 IBM PC Setup

On my IBM PC, I maintain TEX and the DVI driver family on two dismountable 10Mb Bernouilli disk cartridges. For Personal TEX's PCTEX, the directory layout looks as follows:

```
D:\PCTEX              ! TeX.EXE, LaTeX.EXE, et al
D:\PCTEX\TEXFMTS      ! TeX .FMT files
D:\PCTEX\TEXTFMS      ! TeX .TFM files
D:\PCTEX\TEXINPUT     ! TeX input files
```

For MicroTEX, the structure is similar, and I chose to store the font files in its directory tree:

```
D:\TEX                ! TeX.EXE, LaTeX.EXE, et al
D:\TEX\INPUTS         ! TeX input files
D:\TEX\FONTS\180      ! *.pk font files
D:\TEX\FONTS\240      ! *.pk font files
...
D:\TEX\FONTS\1548     ! *.pk font files
```

The DVI family files reside in the directories

```
D:\DVI
D:\DVI\DOC
```

The environment variables required by the DVI family are the ones chosen by default in `machdefs.h`. They could be set in a startup file (typically `AUTOEXEC.BAT`):

```
set FONTPATH  d:\tex\fonts\
set PATH      c:\bin;d:\dvi;d:\pctex
set SUBPATH   d:\tex\inputs\
set TEXFONTS  d:\tex\fonts\
set TEXINPUTS d:\tex\inputs\
```

The PC-DOS `PATH` variable should include all the directories that contain executable programs you require. Although you can copy programs into a common directory, I find it preferable to keep them in the directories where they originate, and just have a long `PATH` search list.

If you use `dvialw`, its macro file, `dvialw.ps`, should be copied into the directory defined by the `TEXINPUTS` variable.

6.2 TOPS-20 Setup

The main TEX directory structure is identical to that maintained by the Stanford TEX Project, but we have numerous additional local directories.

In SYSTEM:n-m-CONFIG.CMD (where system-wide logical names are conventionally defined, and n and m are major and minor Monitor release numbers), we have

```
define DVISPOOL:   tex:nospool.exe        !Dummy .DVI file spooler

define TEX:        aps:<tex>              ! *.EXE files for TeX,
                                          ! TeXware and DVI drivers
define TEXFORMATS: aps:<tex.formats>      ! TeX and LaTeX *.FMT files

define TEXFONTS:   aps:<tex.cm>,-         ! CM fonts in .TFM and .PK form
                   scrtch:<tex.pxl>,-     ! AM fonts in .PXL form
                   scrtch:<tex.fonts>     ! AM*.TFM files

define TEXINPUTS:  aps:<tex.inputs>,-
                   aps:<tex.latex>,-
                   aps:<tex.amstex>

define TEXSPOOL:   tex:nospool.exe        !Dummy .DVI file spooler
```

Users can individually redefine TEXSPOOL: to be TEX:DVIxxx.EXE for their preferred DVI file translator, and DVISPOOL: to be TEX:IMSPOOL.EXE (Imagen) or TEX:PSSPOOL.EXE (PostScript).

Both laser printers are driven by spooler jobs running under SYSJOB (started by @DO OPR:--START-IM-SPOOL and @DO OPR:--START-PS-SPOOL); these use PCL command procedures to poll the spooler directory contents, print any files found there, and then sleep awhile. This approach is considerably simpler than modifying LPTSPL, although is does have a certain amount of polling overhead which can be reduced by making the sleep time longer. We have not found it objectionable in many months of use.

Here is what the main part of our TeX directory structure looks like:

```
APS:<TEX>
APS:<TEX.AMSFONTS>
APS:<TEX.AMSTEX>
APS:<TEX.BIBTEX>
APS:<TEX.CM>
APS:<TEX.DVI>
APS:<TEX.DVI.DOC>
APS:<TEX.FORMATS>
APS:<TEX.INPUTS>
APS:<TEX.LATEX>
```

```
APS:<TEX.LATEX.LOCAL-GUIDE>
APS:<TEX.MF>
APS:<TEX.PUB>
APS:<TEX.WEB>
SCRTCH:<TEX.CM>
SCRTCH:<TEX.DOC>
SCRTCH:<TEX.FONTS>
SCRTCH:<TEX.MF>
SCRTCH:<TEX.PXL>
SCRTCH:<TEX.PXLHEX>
SCRTCH:<TEX.WEB>
```

6.3 VMS Setup

Our VAX VMS directory structure follows the Kellerman and Smith distribution, again with additional directories.

The DVI driver family needs to have foreign command symbols created for each `.exe` file. For example, you might put lines like this

```
$ DVIALW :== $SYS$TEX:[TEX]DVIALW.EXE
```

in a command file referenced in `SYS$MANAGER:SYLOGIN.COM`.

Here is a display of the important command files, a directory tree, and the contents of two of the font directories.

```
type lib:[tex]define.com

$ ! SYS$LIBROOT:[TEX]DEFINE.COM, Thu Dec 18 11:03:33 1986, Edit by BEEBE (at VAX
-VMS)
$ ! Make define/system after installation complete
$ defsys :== define/nolog/system
$
$ ! New version on PS:
$ defsys sys$tex        sys$libroot:
$
$ ! Put [.amstex] and [.latex] first to override old stuff in [.inputs]
$ defsys tex_inputs     sys$tex:[tex.latex], -
                        sys$tex:[tex.amstex], -
                        sys$tex:[tex.inputs]
$ defsys tex_fonts      sys$tex:[tex.cm]
$ defsys tex_formats    sys$tex:[tex.formats]
$ defsys tex_output     texput

type lib:[imprint]*.com
```

SYS$LIBROOT:[IMPRINT]INSTALL.COM;6

```
$ !=====================================================================
$ ! Install IMPRINT spooler on system (do only once)
$ !=====================================================================
$ define imp sys$libroot:[imprint]
$ set command   /tables=sys$library:dcltables -
                /output=sys$library:dcltables -
                imp:imprint.cld
$ set process/privilege=cmkrnl
$ library/replace/help sys$help:helplib imp:imprint.hlp
$ run sys$system:install
sys$library:dcltables/replace
/exit
$ define/system imp_formats     sys$libroot:[imprint]
$ define/system imp_fonts       sys$libroot:[imprint.fonts]
```

SYS$LIBROOT:[IMPRINT]SHUTIMAGEN.COM;2

```
$ !=====================================================================
$ ! Shutdown the Imagen 8/300 queue
$ !=====================================================================
$ set noon
$ stop /next /queue sys$imagen
$ delete/queue sys$imagen
$ set device /nospooled sys$imagen
```

SYS$LIBROOT:[IMPRINT]STARTIMAGEN.COM;6

```
$ !=====================================================================
$ ! Start IMPRINT spooler
$ !=====================================================================
$ define/system sys$imagen      txa0:
$ define/system imp_formats     sys$libroot:[imprint]
$ define/system imp_fonts       sys$libroot:[imprint.fonts]
$ reply/enable=printer
$ set term sys$imagen -
        /permanent -
        /nowrap -
        /speed=19200
$ set device/spooled sys$imagen
$ initialize /queue -
        /term -
        /proc=imprint -
        /default=(feed,flag) -
        /start -
```

```
        sys$imagen


$ cd lib:[imprint]
$ @mgr:dirtree
Output File: tt:

   Your Directory Structure.     One moment please...

        IMPRINT
                ^-----FONTS
                        ^-----1200
                        ^-----1440

$ cd lib:[tex]
$ @mgr:dirtree
Output File: tt:

   Your Directory Structure.     One moment please...

        TEX
                ^-----AM
                        ^-----1000
                        ^-----1095
                        ^-----1200
                        ^-----1315
                        ^-----1440
                        ^-----1500
                        ^-----1643
                        ^-----1728
                        ^-----1800
                        ^-----2074
                        ^-----2160
                        ^-----2400
                        ^-----2488
                        ^-----2592
                        ^-----2628
                        ^-----2880
                        ^-----2986
                        ^-----3110
                        ^-----3456
                        ^-----3732
                        ^-----4479
                ^-----AMSTEX
                ^-----BIBTEX
                ^-----CM
                        ^-----100
```

```
                        ^-----1075
                        ^-----121
                        ^-----1290
                        ^-----145
                        ^-----1548
                        ^-----174
                        ^-----208
                        ^-----240
                        ^-----250
                        ^-----274
                        ^-----300
                        ^-----329
                        ^-----360
                        ^-----432
                        ^-----518
                        ^-----622
                        ^-----746
                        ^-----896
                ^-----DOC
                ^-----FORMATS
                ^-----HP
                ^-----INPUTS
                ^-----LATEX
                        ^-----LTXGDE
                        ^-----ROCHESTER
                ^-----SVI
                ^-----TEX
                ^-----TEXWARE
                ^-----TRIP
                ^-----WEB

$ dir tex_fonts

Directory SYS$LIBROOT:[TEX.CM]

00HEXBIN.COM;3      00HEXCHK.COM;2      00RENAME.COM;2      00TEST.COM;2
100.DIR;1           1075.DIR;1          121.DIR;1           1290.DIR;1
145.DIR;1           1548.DIR;1          174.DIR;1           208.DIR;1
240.DIR;1           250.DIR;1           274.DIR;1           300.DIR;1
329.DIR;1           360.DIR;1           432.DIR;1           518.DIR;1
622.DIR;1           746.DIR;1           896.DIR;1           AMR10.TFM;1
CIRCLE10.TFM;1      CIRCLEW10.TFM;1     CMB10.TFM;1         CMBSY10.TFM;1
CMBX10.TFM;1        CMBX12.TFM;1        CMBX5.TFM;1         CMBX6.TFM;1
CMBX7.TFM;1         CMBX8.TFM;1         CMBX9.TFM;1         CMBXSL10.TFM;1
CMBXTI10.TFM;1      CMCSC10.TFM;1       CMDUNH10.TFM;1      CMDUNH10.TFMHEX;1
CMEX10.TFM;1        CMFF10.TFM;1        CMFIB8.TFM;1        CMINCH.TFM;1
CMMI10.TFM;1        CMMI12.TFM;1        CMMI5.TFM;1         CMMI6.TFM;1
```

```
CMMI7.TFM;1         CMMI8.TFM;1         CMMI9.TFM;1         CMMIB10.TFM;1
CMR10.TFM;1         CMR12.TFM;1         CMR17.TFM;1         CMR5.TFM;1
CMR6.TFM;1          CMR7.TFM;1          CMR8.TFM;1          CMR9.TFM;1
CMSL10.TFM;1        CMSL12.TFM;1        CMSL8.TFM;1         CMSL9.TFM;1
CMSLTT10.TFM;1      CMSS10.TFM;1        CMSS12.TFM;1        CMSS17.TFM;1
CMSS8.TFM;1         CMSS9.TFM;1         CMSSBX10.TFM;1      CMSSDC10.TFM;1
CMSSI10.TFM;1       CMSSI12.TFM;1       CMSSI17.TFM;1       CMSSI8.TFM;1
CMSSI9.TFM;1        CMSSQ8.TFM;1        CMSSQI8.TFM;1       CMSY10.TFM;1
CMSY5.TFM;1         CMSY6.TFM;1         CMSY7.TFM;1         CMSY8.TFM;1
CMSY9.TFM;1         CMTCSC10.TFM;1      CMTEX10.TFM;1       CMTEX8.TFM;1
CMTEX9.TFM;1        CMTI10.TFM;1        CMTI12.TFM;1        CMTI7.TFM;1
CMTI8.TFM;1         CMTI9.TFM;1         CMTT10.TFM;1        CMTT12.TFM;1
CMTT8.TFM;1         CMTT9.TFM;1         CMU10.TFM;1         CMVTT10.TFM;1
DOC.TFM;1           DUMMY.TFM;1         FONTTEST.COM;2      FONTTEST.LOG;1
FONTTEST.TEX;1      ICMEX10.TFM;1       ICMMI8.TFM;1        ICMSY8.TFM;1
ICMTT8.TFM;1        ILASY8.TFM;1        ILCMSS8.TFM;1       ILCMSSB8.TFM;1
ILCMSSI8.TFM;1      LASY10.TFM;1        LASY5.TFM;1         LASY6.TFM;1
LASY7.TFM;1         LASY8.TFM;1         LASY9.TFM;1         LASYB10.TFM;1
LCMSS8.TFM;1        LCMSSB8.TFM;2       LCMSSI8.TFM;1       LINE10.TFM;1
LINEW10.TFM;1       LOGO10.TFM;1        LOGO8.TFM;1         LOGO9.TFM;1
LOGOBF10.TFM;1      LOGOSL10.TFM;1      MANFNT.TFM;1        TESTFONT.COM;11
TESTFONT.LIS;1

Total of 133 files.

$ dir sys$tex:[tex.cm.300]

Directory SYS$LIBROOT:[TEX.CM.300]

CIRCLE10.PK;2       CIRCLEW10.PK;2      CMB10.PK;2          CMBSY10.PK;2
CMBX10.PK;4         CMBX12.PK;2         CMBX5.PK;2          CMBX6.PK;2
CMBX7.PK;2          CMBX8.PK;2          CMBX9.PK;2          CMBXSL10.PK;2
CMBXTI10.PK;2       CMCSC10.PK;2        CMDUNH10.PK;2       CMEX10.PK;2
CMFF10.PK;2         CMFIB8.PK;2         CMINCH.PK;2         CMMI10.PK;2
CMMI12.PK;2         CMMI5.PK;2          CMMI6.PK;2          CMMI7.PK;2
CMMI8.PK;2          CMMI9.PK;2          CMMIB10.PK;2        CMR10.300GFHEX;1
CMR10.GF;1          CMR10.PK;3          CMR12.PK;2          CMR17.PK;2
CMR5.PK;2           CMR6.PK;2           CMR7.PK;2           CMR8.PK;2
CMR9.PK;2           CMSL10.PK;3         CMSL12.PK;2         CMSL8.PK;2
CMSL9.PK;2          CMSLTT10.PK;2       CMSS10.PK;2         CMSS12.PK;2
CMSS17.PK;2         CMSS8.PK;2          CMSS9.PK;2          CMSSBX10.PK;2
CMSSDC10.PK;2       CMSSI10.PK;2        CMSSI12.PK;2        CMSSI17.PK;2
CMSSI8.PK;2         CMSSI9.PK;2         CMSSQ8.PK;2         CMSSQI8.PK;2
CMSY10.PK;2         CMSY5.PK;2          CMSY6.PK;2          CMSY7.PK;2
CMSY8.PK;2          CMSY9.PK;2          CMTCSC10.PK;2       CMTEX10.PK;2
CMTEX8.PK;2         CMTEX9.PK;2         CMTI10.PK;2         CMTI12.PK;2
CMTI7.PK;2          CMTI8.PK;2          CMTI9.PK;2          CMTT10.PK;2
```

```
CMTT12.PK;2         CMTT8.PK;2          CMTT9.PK;2          CMU10.PK;2
CMVTT10.PK;2        LASY10.PK;2         LASY5.PK;4          LASY6.PK;2
LASY7.PK;2          LASY8.PK;2          LASY9.PK;2          LASYB10.PK;2
LINE10.PK;1         LINEW10.PK;1        LOGO10.PK;1         LOGO8.PK;1
LOGO9.PK;1          LOGOBF10.PK;1       LOGOSL10.PK;1       MANFNT.PK;1

Total of 92 files.
```

7 Building the Drivers

With Version 2.07 of the DVI drivers, a public-domain implementation of the Unix `make` utility is included with the distribution. This version of `make` runs on all the operating systems currently supported for the DVI driver family, plus some others.

In my view, `make` ranks right next to compilers in importance. Unix users should usually be introduced to it at an early stage, but people unfamiliar with Unix may be unaware of it.

The essential idea behind `make` is that a user should not have to issue routine compile and link commands to build software, but instead, should be able to describe in a `makefile` dependencies of files on other files, and rules for producing one file from another file, or group of files. For example, an object files depends on its corresponding source file, since if the source file is changed, the object file must be replaced by one from a fresh source compilation. The object file also depends on all the header files included by the source file, since if these are changed, recompilation is also in general necessary. However, a source file does *not* depend on its included files, since if they are changed, no changes to the source file are required.

Supplementing the file dependency lists in the `makefile` are a set of rules for converting one file type to another. The commoner ones are built-in to `make` itself, but additional ones (or replacements for built-in rules) can be provided in the `makefile`, both as defaults, and as specific rules to be used for certain files.

By default, `make` finds the first target file in the `makefile`, then executes all the necessary rules to bring it up to date. Thus, as Stu Feldman, the author of the original `make` says,

> It is usually easier to type the make command than to issue even one of the needed operations, so the typical cycle of program development operations becomes
>
> think - edit - make - test . . .

`makefile`'s in general contain multiple targets, and one or more of these can be selected by name when `make` is run. This makes it possible to store the commands for making all of the files in a given directory in a single file. For software stored in a directory tree, the `makefile` at the top level can run `make` in each of its subdirectories, and these in turn can run `make` in their subdirectories, so a single top-level `make` command can build an entire software system, like the complete Unix operating system.

`makefile`'s have been prepared for each of the host computers supported in the DVI family, and they are given filenames `makefile.xxx`, where `xxx` is the host name. You can just copy the desired one into `makefile`, and type `make` to build all of the drivers.

The leading comments in each `makefile` list all the targets that can be made by executing `make`, and should be consulted for details.

Here is the `makefile` which typesets this document:

```
#-----------------------------------------------------------------------
# Usage:
#
#       make dviman
# or
#       make dvidriver
# or
#       make clean
#
# [17-Apr-87]
#-----------------------------------------------------------------------

LATEX = LaTeX \batchmode \input

CP = copy

.SUFFIXES:

.SUFFIXES:      .dvi .ltx .ids .idx

.idx.ids:
        $(CP) $*.idx $*.id
        texidx $*.id
        $(RM) $*.id.*

dviman: dviman.ltx
        $(LATEX) dviman.ltx

dvidriver:      dvidriver.ltx dvidriver.sty
        $(LATEX) dvidriver.ltx
```

```
        $(RM) dvidriver.dvi.*
        make dvidriver.ids
        $(LATEX) dvidriver.ltx
        make dvidriver.ids

dvidriver.ids:  dvidriver.idx

clean:
        $(RM) *.dvi*.*
        $(RM) *.lst.*
        $(RM) *.aux.*
```

The first `.SUFFIXES` clears the default file suffix list, and the second installs four suffixes in it. The rule for converting the output `.idx` file from LaTeX into the sorted input `.ids` file is given following the line beginning `.idx.ids`. It tells `make` to copy the `.idx` file to one with extension `.id`, run the indexing program, and then delete the temporary copy. This rule is used when the target `dvidriver.ids` is needed, as happens when the rules for `dvidriver` are executed.

By default, `make` will process the `dviman.ltx` file, since that target occurs first. The string `$(LATEX)` requests a macro substitution of the symbol `LATEX` which is defined earlier to run LaTeX in batch mode.

The `clean` target is conventional in `makefile`'s; it contains commands to delete unnecessary files in the directory, and is usually executed when work is completed.

8 Programming Conventions

I am quite particular about readability and appearance of computer programs, because I spend much of my time reading and writing them. The previous sections have mentioned certain programming conventions, but it is useful to summarize them here, in the hope that they may prove a guide to implementors. Some of these rules are designed to facilitate code portability across compiler implementations and operating systems. They are usually not written down in books, though *A C Reference Manual*, Prentice-Hall(1985), by Samuel Harbison and Guy Steele, Jr., is a valuable work which every serious C programmer should study and use. A second edition incorporating much of the proposed draft ANSI C Standard was published in early 1987,

- Compound statements begin and end on separate lines, with opening and closing braces in the same column, and on their own lines. This makes visual brace matching much easier than it is with the common practice of placing the opening brace at the end of a previous line.

- Indentation nesting increases 4 spaces for each level. Spaces and tabs may be used for indentation, and tabs fill up to the next column which is a multiple of 8.

- Only one statement appears per line, with preprocessor macro definitions exempted from the rule, since a definition is required to fit on one line, even if it expands into several C statements.

- Lines must never exceed 80 characters in length, in order that the software can be distributed as 80-character line images on magnetic tape. It is expected that software transported in this manner will be blank-trimmed on reading from the tape. The standard C backslash-newline sequence can be used to break a long line; continuation occurs in column 1 of the next line if a string constant has been broken, and otherwise 4 characters right of the leftmost character of the first line of the continued statement.

- Type declarations should declare only one variable per statement, and that statement should have a brief comment describing the variable.

- Single statements controlled by a C conditional, looping, or switch statement appear on separate lines, and are indented 4 characters to the right of the controlling statement. Complicated statements in this context should be converted to brace-delimited compound statements to enhance readability.

- Preprocessor symbols are entirely in upper-case letters. Variables are entirely in lower-case letters.

- If definition of a macro for some code sequence simplifies it, do so.

- Labels and `goto` statements are not to be used.

- Preprocessor `#if` constant expressions should be avoided if possible. If you require expressions, do not use the operators `!`, `||`, and `&&`. This rule is occasioned by a Lattice C misfeature.

- The only C preprocessor statements to be used are `#include`, `#define`, `#undef`, `#if`, `#else`, and `#endif`. In particular, the useful, but commonly absent, `#elif` may not be used, nor may `#if defined(var)`. `#ifdef`, `#ifndef`, and `#undef` are discouraged. In the driver family, they are restricted to `machdefs.h` where they are used to eliminate compiler "macro redefinition" warning messages. When `#undef` is used, the programmer must be aware that this may not make the symbol undefined; only its most recent definition may disappear, with a previous definition now visible. Lattice C uses this scheme. Fortunately, the draft ANSI C Standard forbids this interpretation.

- Preprocessor statements must begin in column 1, and have no whitespace between the `#` and the following keyword.

- Tricks, like empty comments, for preprocessor symbol concatenation are forbidden; until the ANSI C Standard becomes a reality and is widely implemented, there is no way to do this on many implementations.

- Substitution of preprocessor `#define` arguments in quoted strings is forbidden, and no `#define` argument name may appear as a character string inside a quoted string.

- Type casts must be used for all assignments where the data types of the left and right sides differ, and for function arguments which do not match formal parameters.

- Function values which are discarded must be cast as `(void)`, unless the function has already been declared that way.

- All functions should be declared before the `main()` function.

- All functions must be *explicitly* typed, with `void` used for those which do not return a value.

- The `main()` function should be given first, followed by function definitions in alphabetical order.

- Alphabetical ordering, with letter case ignored, should be adhered to in macro, variable, and function declarations, and in long sequences of independent assignment statements.

- Equivalence of pointers and integers must not be assumed; type casts must be employed on assignment.
- The standard `NULL` pointer declared in `stdio.h` should be cast to the requisite type and compared against, rather than assuming null pointers are zero. That is, write `while (s != (char *)NULL)`, rather than `while (s)`.
- Use preprocessor symbols instead of magic constants. `if (c == ESC)` is more readable than `if (c == 27)`.
- If a function returns a yes/no or true/false value, declare it as `BOOLEAN` and use the symbols `TRUE` and `FALSE` instead of constants 1 and 0.
- Avoid the generic `int` type when a more restricted type could be used instead, or when the result might require more than 16-bit precision.
- Follow the conventions for visible function headers used in the DVI driver family.
- C has at least 28 precedence levels in expressions, which is far too many for a programmer to remember accurately. Use parentheses instead to say what you mean, and don't be afraid to use whitespace to improve readability.
- Use long variable names freely, but keep external symbols unique in their first *six* characters. File names, variable names, and preprocessor variables should be unique in their first *eight* characters.
- Avoid features of C which are not widely implemented. These include the `enum` type (preprocessor constants can simulate this), and functions returning `struct`'s (return pointers to `struct`'s instead).
- Functions with variable numbers of arguments are *not* to be used.
- If you have to resort to machine-dependent coding, use the C preprocessor to select it.
- When possible, use compiler options and tools like the Unix `lint` program to help track down potential problems in code.
- Finally, use comments regularly to document what the code is doing. A recent letter to the editor of the *Communications of the ACM* reported

that fewer than 5% of the lines of code in the entire Unix system are commented! C is most definitely not a self-documenting language; in fact, Philippe Kahn, the developer of the fine Turbo Pascal system on the IBM PC, calls C a *write only* language.

9 Implementation Problems

We have in several earlier sections remarked on problems encountered with particular operating systems and compilers. These are summarized in the following subsections. After insertion of included files by the C Preprocessor, these drivers each amount to between 6500 and 8000 lines of code. Since this is contained in a single file, it tends to tax the IBM PC compilers severely.

9.1 TOPS-20

Text files on TOPS-20 are stored as simple byte streams as they are in Unix, except that `CR LF` terminates a line, instead of just `LF`. The C run-time library will take care of translation between these two, but in order to be able to suppress the translation to compress PostScript output, which does not require the `CR`, text files are opened in binary mode, and the `NEWLINE()` preprocessor macro is used to output the desired end-of-line sequence.

ASCII is a 7-bit character set, and TOPS-20 normally stores text files as five 7-bit characters in each 36-bit word, with the rightmost (low-order) bit set to zero. In order to obtain 8-bit I/O, it is necessary to use `open()` instead of `fopen()`, so this is encapsulated in the function `f20open()`. The code references this only through the preprocessor macro `BINARYOPEN`. With a little more thought in the implementation of the I/O library, it should have been possible to extend the arguments to `fopen()` to achieve this, so that `stdin` and `stdout` could be chosen to be 8-bit streams if desired. KCC-20 accepts an extended mode argument to `fopen()` to obtain binary files of particular byte sizes, but it does not permit binary I/O on the files `stderr`, `stdin`, and `stdout`.

External names in TOPS-20 are normally limited to 6 upper-case characters. Although the linker supports long case-insensitive external names, and both the Pascal and Algol compilers use them, PCC-20 does not. Consequently, C programmers must take care to avoid long external names which are not unique in the first 6 characters, ignoring case. Neither does PCC-20 provide support for extended addressing in a 23-bit address space, instead of an 18-bit address space, like Fortran and Pascal do. The compiler does not store function traceback information, or generate a symbol table for local variables, nor does it employ the linker argument type blocks which have proved enormously useful in catching argument type and count mismatches at link time with other languages. This makes debugging tedious, and I admit to having become fairly adapt at visually uncompiling assembly code back into C inside the DDT debugger.

By default, the compiler splits the 18-bit address space into two, half for code and half for data. With the large bitmaps required by some of the drivers, this provided insufficient space for data, so I created some alternate library routines which increase the data space to 3/4 of the address space.

PCC-20 does not permit members of different structures to be given the same names; this was a minor annoyance in the definition of the `font_entry` and `font_list` structures which both contain the same object, but are forced to name it differently.

The ugliest piece of code occasioned by PCC-20's deficiencies is the saving and restoring of the terminal state in functions `initterm()` and `rsetterm()` in `dvibit.c`.

The `void` data type is not recognized by PCC-20, but is so useful that I simply inserted a statement `typedef int void;` in `stdio.h`, making it available to everyone.

As noted in the section on `loadchar.h`, the compiler did not correctly handle a function reference of the type

```
(void)(*fontptr->charxx)(c,outrow);
```

It also generated incorrect code for functions returning a `struct`, but I was able to fix that problem in the compiler. It does not apply to the DVI driver family, but the Unix AWK utility makes heavy use of this feature.

The C preprocessor had a symbol table for only 1000 symbols, which is much too low for a file the size of `monsym.h`, which defines all the symbols for operating system calls. For the DVI driver family, I merely selected a subset of these and installed them in `machdefs.h`. However, for convenience in other programs, I rebuilt the preprocessor with a table size of 20000 to eliminate this problem.

The maximum `switch` table size was increased from 250 to 513. This makes it possible to compile `switch` statements indexed by 8-bit and 9-bit bytes. This was essential to avoid ugly workarounds in `prtpage.h`.

Availability of the compiler and library source code has been essential in overcoming some of these problems, and in tracking down questionable behavior in the run-time library.

Adding support for KCC-20 was straightforward, and the compiler has proved to be reliable, and the library quite close to Unix. As of April, 1987, it does not implement the Berkeley terminal handling facility provided by `ioctl()` and `sgtty()`.

The DEC-20 JSYS interface provided by KCC-20 is similar to PCC-20's, but the symbol names are closer to the assembly language definitions. To reduce code duplication, one set of symbols is defined explicitly in terms of the others. Like PCC-20, a complete `monsym.h` proved to have too many symbols for KCC-20, so for the DVI driver family, the needed values are defined explicitly in `machdefs.h`. The KCC-20 developers have since linked an extended addressing version of the compiler which happily handles `monsym.h`, and we now use that version for systems programming.

One caveat for KCC-20 must be issued here. It stores character data as 9-bit values, and short integers as 18-bit values, both accessed with DEC-20 byte instructions. This produces compact data, but the current hardware microcode implementation of these instructions is slower than the fullword integer instructions, so the drivers compiled with KCC-20 run considerably slower than when compiled with PCC-20, despite the generally superior code generation of the former. The well-known `sieve` benchmark is about 4 times slower with byte instructions than with fullword integer instructions.

9.2 Unix 4.1BSD

Like PCC-20, `cc` under 4.1BSD Unix does not correctly handle the case of function reference to a structure member.

Filenames are restricted to 14 characters in length, so output file extensions are `.xxx` instead of `.dvi-xxx`.

Otherwise, the drivers compile without problems. We found that after completion of execution, and production of correct output, they die with a stack error in `exit()`. Attempts to debug this with `sdb` were unsuccessful, because the debugger would step right into the stack error and crash. When the drivers are run under control of the assembly-level debugger, `adb`, they complete normally!

In February, 1987, we traced this to a 4.1BSD Unix system bug. `fclose()` is called in `abortrun()` to close all font files, and it turned out that whenever one of the files passed to `fclose()` was already closed, the library would install a time bomb which blew up the code at exit. According to the manual page for `fclose()`, no action should be taken if the file is not already open. `abortrun()` has been revised to avoid calls to `fclose()` for such files, and the problem was been solved.

9.3 HPUX

No problems were encountered with HPUX which is based on AT&T System V Unix, with 4.2BSD extensions. The version I used ran on a Hewlett-Packard 9000/350 workstation. File names up to 1024 characters in length are supported, and up to 60 open files are permitted, so the only change was the addition of a preprocessor symbol `HPUX` in `machdefs.h` to select code defining the larger limits.

9.4 IBM PC Lattice C

I have both versions 2.15a and 3.0 of the Lattice C compiler running under PC-DOS, and have used version 2.15a extensively for graphics work. The major problem with this version is the limited overlap with the standard Unix run-time library. This has been largely remedied with version 3.0. However, both versions have proved singularly incapable of producing working versions of any of the drivers—the executable programs die with errors.

Version 2.15a implements `unsigned` as a type, meaning a 16-bit unsigned integer, rather than a type modifier. The original Kernighan and Ritchie definition is unclear, but the ANSI C standard committee has made it clear that it is a legal modifier on any integer data type. I expect that this would have posed a problem in the drivers, particularly those parts which twiddle bits in the character rasters and output bitmaps. Type `void` is not recognized, either, but that can be solved with a simple `typedef` in `typedefs.h`.

Version 3.0 raises a fatal error when `(void)` is used as a typecast on a function previously declared to be of type `void`; it does not do so if `void` is replaced by any other data type. This particular use occurs extensively throughout the drivers, so I have shelved further work with this version until Lattice fixes it. One attractive feature of version 3.0 is the addition of support for argument type checking according to the prescription in the coming ANSI C standard.

The Lattice compilers correctly handle large arrays, but in the interests of efficiency, the driver code selects a bitmap implementation using a vector of raster pointers for all IBM PC compilers.

9.5 IBM PC Wizard C

The Wizard C compiler was attractive because of its claim to a high degree of compatibility with the Unix run-time library.

The Wizard compiler correctly handles large arrays, but in the interests of efficiency, the driver code selects a bitmap implementation using a vector of raster pointers for all IBM PC compilers.

Unfortunately, it has been unable to compile any of the drivers without getting internal table overflows. Its diagnostics about variables possibly used before definition, or declared but not used, and about suspicious statements, like `if (a = b) foo();` have nevertheless been quite useful as an alternative to the Unix `lint` utility.

With Version 3.0 of the Wizard compiler, the drivers will compile, but they run incorrectly. For example, the trivial loop in `cppsfile()()` in `dvialw.c` which copies the macro header file to the output file returns random characters with `getc()` (as determined by setting an assembly level breakpoint immediately after the call to that function).

9.6 IBM PC Microsoft C

Version 3.0 of the Microsoft C compiler is claimed to be a completely new release, sharing a very high degree of compatibility with the Microsoft Xenix compiler, and the Unix run-time library. The compiler implements the ANSI argument type checking language extensions and will even generate a set of function type declarations from a source file.

The 600 or so pages of documentation is thorough and, at least in the library reference volume, quite readable. The volume of the use of the compiler could be improved—it is quite difficult to track down precisely what compiler options to use, because they are spread across dozens of pages of the manual. There is an on-line one-page summary available if you invoke the compiler with the help option.

Error diagnostics are about as good (or bad) as any other C compiler—generally just the line number and a short message. Unfortunately, the error recovery techniques are abysmal, and a single error is likely to propagate dozens, or even hundreds, of other errors, even across function boundaries, so it is frequently quite difficult to find the sick tree in the forest of output. Version 4.0 of the compiler does not have this problem.

Like most C compilers for the Intel microprocessor family, this compiler supports several memory models. However, its large memory model support, which is essential for this driver family, is fatally flawed. Unlike the Lattice and Wizard compilers, which provide alternatives to `malloc()` for the case of large memory allocations,[23] Microsoft provides no such function. Even worse, 32-bit pointers consisting of a 16-bit segment and 16-bit offset, are *by design handled incorrectly*. Arithmetic on such pointers operates only on the offset portion. This makes it impossible to reliably use the `++` or `--` operators on such a pointer, because it could result in segment wrap. Version 4.0 of the compiler introduced a *huge* memory model which does not have this limitation.

For these reasons, the bitmap representation as a vector of raster pointers had to be introduced, and a special function `normaddr()` defined in `bitmap.h` to allow arithmetic on pointers.

This design flaw in the compiler, which is quite intentional because it is documented in the manuals, is frankly unforgivable.

Perfectly decent code in `openfont()` generated a fatal compiler error, and a workaround under control of the preprocessor had to be installed.

Getting around these flaws has occasioned considerable extra work, but the compiler does succeed in producing compiled code, and Version 4.0 has provided a reliable implementation of the DVI driver family which has received heavy use locally. The one library bug that we have noted with Version 4.0 is incorrect handling of formats to `sscanf()`; for the page number range switch parsed in `option.h`, it is necessary to supply a trailing colon (i.e. use `-o7:9:` instead of `-o7:9`).

10 Implementation Efficiency

As the driver family has undergone continued development, virtually all of the code has been reexamined with a view to its run-time efficiency. Where reasonable, in-line code is generated by macros, pointer arithmetic is used in preference to array indexing, and heavily used variables are declared with the `register` attribute. Since not much computation is required in any of the processing, one might expect the drivers to be I/O bound, and this indeed proves to be the case.

As a sample of what might be expected, the `dvitype.web` file was se-

[23] Recall that `malloc()` is defined to have the memory size argument be an `int`, rather than a `long`, introducing a 32-Kbyte limitation when an `int` is only 16 bits. The draft ANSI C Standard defines the argument to `malloc()` to be of a new type, `size_t`, which is returned by the `sizeof` operator; this should provide a clean work-around.

lected for processing. This file will be available at every TEX site. Application of Weave generates a .TEX file which produces 53 printed pages, and the following run-time results for the PCC-20 implementation on our DEC-20/60 may be of interest.

Operation	CPU (sec)	Output Filesize (kbytes)
weave dvitype.web	22	149
tex dvitype.tex	127	239
dvialw dvitype (PXL fonts)	65	818
dvialw dvitype (GF fonts)	79	
dvialw dvitype (PK fonts)	70	817
dviprx dvitype (PXL fonts)	114	2608
dviprx dvitype (GF fonts)		
dviprx dvitype (PK fonts)	290	2379

The `dvialw` results are the average of 3 runs which had a variance of about 5 sec. Clearly, the penalty for the more complex `.gf` and `.pk` font formats is not large – about 20% for `.gf` fonts, and 8% for `.pk` fonts. Since the `.pk` fonts average about half the disk space of the `.gf` fonts, and `dvialw` runs faster with `.pk` fonts than with `.gf` fonts, there is no reason to keep `.gf` fonts on-line once they have been generated by METAFONT and converted by `GFtoPK`.

To further investigate the CPU usage inside the driver, I ran `dvialw` and `dviprx` under control of the run-time PC-histogram utility, `pclook`. This produces a histogram of where the program is spending its time, based on samples of the program counter taken every 25 msec. The total number of samples taken ranged from 1600 to 2000. In general, the time was distributed in small percentages throughout most of the functions in `dvialw`, but the following results are of interest:

dvialw test	`.pxl`	`.pk`	`.gf`
JSYS (TOPS-20 Monitor)	52%	55%	47%
C run-time library	15%	16%	16%
`printf`	11%	10%	11%
`dvialw` (all functions)	27%	25%	31%
`chargf`			3%
`charpk`		1%	
`charpxl`	0.2%		
`setstr`	10%	8%	8%

dviprx test	`.pxl`	`.pk`	`.gf`
JSYS (TOPS-20 Monitor)	48%	61%	
C run-time library	4%	21%	
`printf`	0.1%	1%	
`dviprx` (all functions)	46%	18%	
`chargf`			
`charpk`		0.1%	
`charpxl`	0.1%		
`dispchar`	17%	5%	
`fontsub`		2%	
`prtbmap`	17%	4%	
`setchar`	3%	1%	

The time for `printf` is included in the run-time library total, but for `dvialw` it represents the largest use of any library function. `dviprx` uses `putc()` for the output of its bitmap, rather than `printf()`.

The `charxx()` functions are the ones which actually read the font raster information and unpack it for output to the device. It is surprising that so little of the total time is spent doing this work.

`setstr()` accounts for the largest fraction of time of any function in `dvialw`; it loops reading characters from the DVI file, calling `charxx()` to output them. Similarly, `dispchar()` and `setchar()` in `dviprx` together account for 20% of the time, and an almost equal amount of time is spent by `prtbmap()` converting the page bitmap into a character stream for the printer.

From 50% to 70% of the total time is spent inside the run-times and the TOPS-20 operating system, putting it beyond control of the ordinary programmer. Since the major reason for calling the operating system and run-time library is for I/O, it is evident that host I/O efficiency is the dominant factor in the DVI translator, and few source code changes in DVI driver family could be made that would have any significant effect on the overall efficiency.

11 Implementation Summary

You will no doubt encounter problems I have not documented here if you implement this driver family under a new operating system or with a new compiler. As you make changes, remember that the goal is to reduce all future changes to settings of preprocessor variables in *one* file, `machdefs.h`.

Look there first, and make sure that the operating system and implementation symbols are properly defined for your system.

Find out whether your compiler generates sign-propagating arithmetic right shifts, or uses logical right shifts, and set `ARITHRSHIFT` accordingly. According to the proposed ANSI C Standard, right shifts of signed integers should propagate the sign bit, and right shifts of unsigned integers should propagate leading zero bits. This was implementation dependent in the original C definition. The file `OOARIT.C` can be compiled and run to test the behavior of shifts.

Check the file name, extension, and directory path specifications, and modify them if necessary, preferably under control of a preprocessor `#if` statement.

Finally, document all problems and send your comments back to me. You can complain all you like about something I have done, but if you do not tell me, I cannot fix it, or explain why I did it that way.

Index

#

#define, 74
#elif, 74
#else, 74
#endif, 4, 74
#if, 4, 43, 74, 84
#if defined(var), 74
#ifdef, 74
#ifndef, 74
#include, 5, 7–8, 28, 74
#undef, 74

+

++, 81

-

--, 81

.

.dvi, 10
.dvi-xxx, 78
.exe, 65
.gf, 9–12, 18, 23–24, 34–35, 38, 82–83
.h, 7–8, 53
.id, 72
.ids, 72
.idx, 72
.info, 31
.pk, 9, 11–12, 18, 23–24, 34–35, 38, 82–83
.pxl, 9–12, 18, 23–24, 34, 83
.tfm, 19, 47
.xxx, 78

/

/usr/jones/book/hornets.dvi, 19
/usr/jones/book/hornets.sub, 19
/usr/jones/book/texfonts.-sub, 19
/usr/lib/tex/fonts/1500-/cmr10.pxl, 18
/usr/lib/texinputs-/texfonts.sub, 20

@

@DO OPR:--START-IM-SPOOL, 64
@DO OPR:--START-PS-SPOOL, 64

0

00ARIT.C, 84
00revhst.txt, 5, 14

3

3320, 51

4

4.1BSD, iii, 26, 78
4.2BSD, 79

6

68020, 45
68881, 45

8

8/300, 45, 51

A

abort_run(), 8
AbortRun(), 8
abortrun(), 8–9, 16, 78–79

abortrun.h, i, 9
actfact(), 9
actfact.h, i, 9
Ada, 2, 7
adb, 78
Adobe, 37, 40, 44, 47–48
Algol, 76
alldone(), 9, 28
alldone.h, i, 9
alphabetical order, 8, 43, 74
alw, 5
ANONYMOUS FTP, 2
ANSI, 7, 21
Apple, 45
Apple ImageWriter, 58–59
Apple Macintosh, 3, 40, 45–46
AppleTalk, 45
argv[], 30
ARITHRSHIFT, 14, 84
AT&T System V, 79
ATT, 27
AUTOEXEC.BAT, 63

B

Barnes, Simon, 1
BBN BitGraph, 1, 10, 15, 37, 44, 49
Bechtolsheim, Stephen, 1
Bergsten, Per, 2
Bezier curve, 17
BINARYOPEN, 16, 76
BITMAP(y,x), 14, 17, 22, 28
bitmap.h, i, 9, 81
block move, 12
BOOLEAN, 6, 42, 75
BOP, 7
bopact(), 9
bopact.h, i, 9, 49
Brown, Bob, 1
BSD41, 27
BSD42, 27
BYTE, 6, 42

C

C-Itoh, 60
c:\usr\pctex\texfonts, 27
Canon, 35, 45, 50–51, 53, 61
case, 30
cc, 26, 78
CDC, 15, 38
char, 42
char_entry, 25
character painting, 13, 36
chargf(), 9–13, 31, 34–36, 38
chargf, 83
chargf.h, i, 9
charpk(), 9–13, 31, 35, 38
charpk, 83
charpk.h, i, 11–12
charpxl(), 9–10, 12–13, 31, 35
charpxl, 83
charpxl.h, i, 12
charxx(), 26, 83
Chinese characters, 31
clrbmap(), 12, 22, 32
clrbmap.h, i, 12
clrrow(), 13, 35
clrrow.h, i, 13
commands.h, i, 7, 13, 28
commercialization policy, 2
Computer Journal, 4
config.sys, 29
Coonan, Jerome, 45
COORDINATE, 6, 42
copy, 53
cppsfile(), 47
cppsfile()(), 80
CR, 76

Cyrillic, 61

D

`d:\tex\fonts\1500\cmr10.pxl`, 18
`d:\tex\fonts`, 27
DDT, 77
DEC LA75, 58
DEC-20, 15–16, 18, 44, 49, 51, 78
`devinit()`, 13–14
devps, 48
`devterm()`, 13–14
`dispchar()`, 10, 13–14, 17, 32, 36, 39, 83
`dispchar`, 83
`dispchar.h`, i, 13
`dumpchar.h`, i, 14
`dvialw`, 17, 37, 44, 47–48, 52–53, 63, 82–83
`dvialw.c`, iii, 9, 29, 31, 39–40, 44–45, 47, 80
`dvialw.ps`, 47, 63
`dvibit`, 9, 33, 35, 44, 49
`dvibit.c`, iii, 9, 29, 31, 48–49, 77
`dvican`, 17, 37, 50
`dvican.c`, iii, 9, 26, 29, 31, 50
`dvifile()`, 13–15, 28
`dvifile.h`, i, 14
`dvigd.c`, iii, 50
`dvihead.h`, i, 5, 14, 27, 30, 42
`DVIHELP`, 23, 42
`dviimp`, 1, 17, 37
`dviimp.c`, iii, 9, 29, 32, 51
`dviinit()`, 14, 54
`dviinit.h`, i, 14
`dvijep`, 17, 37, 51–54
`dvijep.c`, iii, 9, 26, 29, 32, 51, 53, 55, 57
`dvijet`, 52, 56
`dvijet.c`, iii, 51
`dvil75`, 1
`dvil75.c`, iii, 58
`dvim72`, 58
`dvim72.c`, iii, 58
`dvimac.c`, iii, 58
`dviman.ltx`, 30, 72
`dviman.texinfo`, 30
`dvimpi`, 58
`dvimpi.c`, iii, 58
`dvio72`, 59
`dvio72.c`, iii, 59
`dvioki`, 59
`dvioki.c`, iii, 59
`dviprx`, 60, 82–83
`dviprx.c`, iii, 32, 60
`DVISPOOL:`, 64
`dviterm()`, 14–15
`dviterm.h`, i, 15
`dvitos`, 61
`dvitos.c`, iii, 61
`dvitype`, 4, 18
`dvitype.web`, 82
`dvixxx`, 42, 44
`dvixxx.c`, 5–9, 14, 22, 27–28, 32, 41

E

EastMark, Inc., 1
efficiency, 12–13, 16–17, 21, 24, 28, 32–33, 59, 80–81
electronic mail, 2
Emacs, 1, 4, 46
Enhanced, 45, 48
Enhanced LaserJet Plus, 51
`enum`, 75
environment variable, 19, 23–24, 27, 31, 41–42
`eopact()`, 15, 33

eopact.h, i, 15, 35, 49
EPSF (*Encapsulated PostScript File Format*), 40
Epson, 58
ESC, 49
ETX, 59
exit(), 9, 28, 78
EXTERNAL, 7

F

f20open(), 16, 76
f20open.h, i, 15
FALSE, 42, 75
fatal(), 15–17, 19, 30, 36, 43
fatal.h, i, 16
fclose(), 78–79
Feldman, Stu, 70
fillrect(), 17, 36
fillrect.h, i, 16
findpost(), 17, 34
findpost.h, i, 17
fixpos(), 18, 28, 36–37, 56–57
fixpos.h, i, 17
FNT_DEF1, 7
font cache search, 30
font magnification, 9, 18–21, 29, 33, 49
font magnification table, 19
font name, 18
font substitution file, 19
font_entry, 26, 34–35, 77
font_list, 77
fontfile(), 18, 23, 27, 29
fontfile.h, i, 18
FONTLIST, 23–24, 41
FONTPATH, 27
fontsub(), 19–21, 23, 29, 42
fontsub, 83
fontsub.c, i, 19
foo(), 3
footnotes, 1–2, 4, 6, 8, 11, 16, 19, 25, 29, 31, 36–37, 41, 44, 54, 81
fopen(), 16, 76
for, 3, 30
Forth, 44
Fortran, 2, 15, 77
FTP, 2
Fuchs, David, 2, 4, 25
Fuchs-Knuth algorithm, 34
Furuta, Richard, 1

G

gblprocs.h, i, 7, 21, 28
gblvars.h, i, 7, 19, 21, 28, 30
gendefs.h, i, 7, 21, 28
get_bit(), 12
get_nybble(), 12
getbmap(), 12, 14, 17, 21–22, 28, 32
getbmap.h, ii, 21
getbytes.h, ii, 22
getc(), 49, 80
getenv(), 23
getfntdf(), 34–36
getfntdf.h, ii, 22
getpgtab(), 34
getpgtab.h, ii, 22
GF, 24
GFtoPK, 11, 38, 82
GFtoPXL, 11
global data, 7
Gnu Emacs, 31
goto, 11–12, 73
Greek, 61
grep, 4
GTR01, 1

H

h, 28, 36
HELPCMD, 42
Hewlett-Packard, 52
Hewlett-Packard 9000/350, 79
Hewlett-Packard LaserJet, 10, 32, 51, 61
hh, 28, 36
Hierogana characters, 61
HIRES, 58
Holtz, Neal, 1
HP, 52
HP 2686d, 51
HPUX, iii
HPUX, 27, 79
HPUX, 79

I

IBM, 29
IBM PC, iii, iv, 3, 5, 7–8, 18, 26–27, 29, 41–42, 49, 62, 76, 79–80
IBM_PC_LATTICE, 27
IBM_PC_MICROSOFT, 27
IBM_PC_WIZARD, 27
IEEE Computer, 31
IEEE floating-point, 45
if, 3
if (a = b) foo();, 80
if (c == 27), 75
if (c == ESC), 75
Imagen, 1, 39, 45, 51, 64
ImageWriter Plus, 58
img_row[], 13, 31
implementation flag, 27
inch(), 23, 31
inch.h, ii, 23
initglob(), 19, 23–24, 27–28, 30, 41
initglob.h, ii, 23
initterm(), 77
int, 42, 75, 81
INT16, 6, 42
INT32, 6, 42
INT8, 6, 42
Intel, 6, 21, 81
Intel iAPX, 5–6, 9, 21
ioctl(), 49, 78
ISO Standard Pascal, 2

J

Japanese characters, 31, 61
JISCII character set, 61
JSYS interface, 78

K

Kanji characters, 31, 61
Katakana characters, 61
KCC-20, 4, 30, 76, 78
KCC_20, 27
Kellerman, David, 39, 65
Kernighan, Brian, 26, 79
keyboard input, 7, 15, 24, 33, 49
keybrd.c, ii, 7, 15, 24, 49
keydef.h, ii, 7, 24
Knuth, Donald, 1, 4, 25

L

LA, 32
Lamport, Leslie, 1
LaserJet, 51, 53, 55–56
LaserJet Plus, 51, 53
LaserWriter, 48
Lattice, iv, 26, 42, 74, 79, 81
LBP-8 A2, 50
LBP-CX, 35, 50, 53
Lepreau, Jay, 4
letter case, 31

LF, 76
lint, 75, 80
loadbmap(), ii, 26
loadchar(), 9, 25–26, 37, 54–55
loadchar.h, ii, 25, 33, 77
log file, 9, 15–16, 19, 42–43
logical name, 24, 27, 64
long, 42, 81
lptops.c, 47
lptops.hlp, 48
LPTSPL, 64
lw78, 53
lw78.c, 46

M

m, 64
MacDraw, 40
machdefs.h, ii, 5–7, 14, 16, 18, 21–23, 27, 30, 42–43, 63, 74, 77–79
MacPaint, 40
macro definitions, 7
main(), 7–8, 14, 21–23, 30, 34, 42, 74
main.h, ii, 6, 14, 21, 24, 28
maintenance history, 5, 14
make, 5, 70–72
makechar(), 22, 28, 56
makefile, 70–72
makefile.xxx, 71
malloc(), 25, 81
MAPCHAR, 54
MAXOPEN, 30
Mergenthaler Linotype Linotronic 300, 48
Microsoft, iv, 7–8, 21, 26, 41, 80–81
Modula-2, 2, 7
Monardo, Pat, 2
monsym.h, 77–78
Motorola, 6
Motorola 68000, 45
movedown(), 28
movedown.h, ii, 28
moveover.h, ii, 28
moveto(), 36
moveto.h, ii, 29
MPI, 58
MPI Sprinter, 58–59
myfile.dvi, 44
myfile.dvi-xxx, 44
myfile.tex, 44
myfile.xxx, 44

N

n, 64
naming conventions, 8
Naugle, Norman, 2
nesting of files, 8
newfont(), 29, 34–35
NEWLINE(), 76
normaddr(), 81
nosignex(), 29, 38
nosignex.h, ii, 29
NUL, 41
NULL, 75

O

Okidata Pacemark 2410, 59
open(), 76
open file limit, 29
openfont(), 29–30, 35–36, 81
openfont.h, ii, 29
operating system flag, 27
option(), 28, 30, 42
option parsing, 30
option.h, ii, 30, 81
oriental languages, 31

OS_, 43
OS_PCDOS, 27
OS_TOPS20, 27, 43, 49
OS_UNIX, 27
OS_VAXVMS, 27, 43
OS_xxxx, 27
outline(), 59, 62
outrow(), 10, 12, 31, 35
outrow.h, ii, 31

P

P-1351, 61
page counters, 23, 41
Pascal, 2–3, 76–77
PATH, 63
PC-DOS, 30, 63, 79
PCC-20, 4, 26, 30, 43–44, 76–78, 82
PCC_20, 27
pclook, 82
PDP-11, 6
Pike, Rob, 26
Pipeline Associates, 48
PK, 24
pk GF pxl, 24
pk-gf.pxl, 24
pk/gf/pxl, 24
pk_packed_num(), 12
PKGFPXL, 24
PKtoGF, 11
PKtoPX, 11–12
PL/1, 7
plain.tex, 11
Plus, 10, 51, 53–56
Portable, 26
PostScript, 1, 10, 37, 39–40, 44–48, 52–53, 56, 64, 76
printf(), 83
printf, 83
Printronix, 32, 59–60
PRIVATE, 12
private modules, 12, 35
program counter, 82
programming conventions, 72
prtbmap(), 22, 32, 60, 62, 83
prtbmap, 83
prtpage(), 9, 14–15, 33, 38–39
prtpage.h, ii, 33, 49, 77
putc(), 83
PXL, 24
PXLtoGF, 11
PXtoPK, 11

R

readfont(), 9, 22, 26, 34, 38, 54–55
readfont.h, ii, 34–35
readgf(), 29, 34–36
readgf.h, ii, 34
readpk(), 11, 29, 34–35
readpk.h, ii, 34
readpost(), 14, 22
readpost.h, ii, 34
readpxl(), 29, 34–35
readpxl.h, ii, 34
reference count, 2, 25, 30
register, 82
reldfont(), 34–35
reldfont.h, ii, 35, 49
Ritchie, Dennis, 79
Rohlicek, Robin, 1
Rokicki, Tom, 2, 11
rsetterm(), 77
rulepxl.h, ii, 36
run-length encoding, 10–11, 32–33, 56, 59, 61–62

S

Sauter, John, 1
Schaad, Jim, 1
sdb, 78
sed, 8
SEGMEM, 12, 22
segmented memory, 6, 9, 12, 14, 17, 21, 32
Senn, Mark, 1
SET_RULE, 7
setchar(), 9, 36–37, 54, 83
setchar, 83
setchar.h, ii, 36
setfntnm(), 36
setfntnm.h, ii, 36
setrule(), 36
setrule.h, ii, 36
setstr(), 36–37, 52, 54, 56–57, 83
setstr, 83
sgtty(), 78
shifts, 84
short, 42
sieve, 78
signex(), 29, 38
signex.h, ii, 38
size_t, 81
sizeof, 81
skgfspec(), 35, 38
skgfspec.h, ii, 38
skipfont.h, ii, 38
skpkspec(), 35, 38
skpkspec.h, ii, 38
Smith, Barry, 39, 65
special(), 39–40, 47, 53
special.h, ii, 38–39
Spivak, Michael, 1
Sprinter, 58–59
sscanf(), 20, 81
Stallman, Richard, 4
static, 12
stderr, 9, 14, 43–44, 76
stdin, 43–44, 76
stdio.h, 75, 77
stdout, 43–44, 76
story.tex, 50
strchr(), 41
strchr.h, iii, 40
strcm2(), 41
strcm2.h, iii, 41
strcmp(), 41
strcmpl(), 41
strid2(), 35, 41
strid2.h, iii, 41
strrchr(), 41
strrchr.h, iii, 41
struct, 5, 75, 77
STX, 59
SUBPATH, 27
switch, 11, 26, 30, 33, 77
SYS$MANAGER:SYLOGIN.COM, 65
SYSJOB, 64
SYSTEM:n-m-CONFIG.CMD, 64

T

Tal, 2
Tangle, 2
tctos(), 41
tctos.h, iii, 41
Tektronix, 39, 44
telephone test, 31
TeX Users Group, 2
TEX:DVIxxx.EXE, 64
TEX:IMSPOOL.EXE, 64
TEX:PSSPOOL.EXE, 64
tex_counter[], 41
TEXFONTS, 18, 23, 27
texfonts.sub, 19
texfonts:cmr10.1500pxl, 18
texfonts:cmr10.300gf, 18

texfonts:cmr10.300pk, 18
texfonts:cmr10.1500pxl, 18
TeXHaX, 2
TEXINPUTS, 23, 27, 47, 63
TEXSPOOL:, 64
tfm, 56
to-do list
  Tektronix graphics file support, 39
  fast font cache search, 30, 36
  fonts for small space characters, 58
  functions into macros, 22, 28–29, 36
  handling missing fonts, 19
  optimal font handling, 25, 34
  track down compiler error, 26
TOPS-20, iii, 4, 30, 43, 62–63, 76–77, 83
Toshiba P-1351, 61
TranScript, 48
Trilog, 60
TRUE, 42, 75
TUGBoat, 2, 4
typedef, 5, 12, 42, 79
typedef int void;, 77
typedefs.h, iii, 5–6, 14, 42, 79

U
Univac, 16, 38
Univac 1100, 16
Unix, iii, 3–5, 8, 18–19, 26, 30–31, 43–44, 48–49, 70–71, 75–80
unload_fonts(), 35, 49
UNSIGN16, 6, 42
UNSIGN32, 6, 42
unsigned, 13, 42, 79
unsigned int, 42
usage(), 23, 28, 30
usage, 42
usage.h, iii, 42

V
v, 28
VAX, 6, 44, 51, 62, 65
VMS, iii, 44, 51, 62, 65, 67, 69
(void), 6
void, 6, 42, 74, 77, 79
vv, 28

W
warning(), 42–43
warning.h, iii, 42
Weave, 2, 82
Web, 2, 4, 51
Wells, Robert, 1
while (s != (char *)NULL), 75
while (s), 75
Willett, Lon, 1
Wirth, Niklaus, 2
Wizard, iv, 26, 80–81

X
xcp, 29
Xenix, 80
XOFF, 45–46, 53
xoffp, 55
XON, 45–46, 53
XORIGIN, 54
XT, 8
xxx, 5, 71
xxxspool, 44

Y
ycp, 29
yoffp, 55
YORIGIN, 54

Z

`zerom()`, 12
zoom command, 49